**JPQL**

**(Java Persistence Query Language)**

**Introduction to JPQL**

**What is JPQL?**

Java Persistence Query Language (JPQL) is a query language used in Java Persistence API (JPA) to interact with relational databases using object-oriented queries. It is similar to SQL but operates on entity objects rather than database tables.

**Why JPQL?**

* Works with entity objects instead of raw database tables.
* Database-independent (portable across different databases).
* Provides better integration with JPA and Hibernate.
* Supports relationships between entities.

**JPQL vs SQL**

| **Feature** | **JPQL** | **SQL** |
| --- | --- | --- |
| Works with | Entity objects | Database tables |
| Case Sensitivity | Case-insensitive for keywords | Case-insensitive for keywords, but column names depend on DB |
| Joins | Uses entity relationships | Uses foreign key joins |
| Aggregation Functions | Supports SUM, COUNT, AVG, etc. | Supports SUM, COUNT, AVG, etc. |
| Portability | Database-independent | Database-dependent |

**Basic JPQL Syntax**

JPQL queries are structured similarly to SQL queries but use entity names and properties instead of table and column names.

**SELECT Query**

// Fetch all employees

@Query("SELECT e FROM Employee e")

List<Employee> getAllEmployees();

**WHERE Clause**

// Fetch employees by department

@Query("SELECT e FROM Employee e WHERE e.department = :dept")

List<Employee> getEmployeesByDepartment(@Param("dept") String department);

**ORDER BY Clause**

// Fetch employees sorted by name

@Query("SELECT e FROM Employee e ORDER BY e.name ASC")

List<Employee> getEmployeesSortedByName();

**LIKE Clause (Pattern Matching)**

// Fetch employees whose name starts with 'J'

@Query("SELECT e FROM Employee e WHERE e.name LIKE 'J%'")

List<Employee> getEmployeesWithNameStartingJ();

**JPQL Joins**

**1. Inner Join**

@Query("SELECT e FROM Employee e JOIN e.department d WHERE d.name = :deptName")

List<Employee> getEmployeesByDeptName(@Param("deptName") String deptName);

**2. Left Join**

@Query("SELECT e FROM Employee e LEFT JOIN e.projects p WHERE p IS NULL")

List<Employee> getEmployeesWithoutProjects();

**3. Fetch Join (Eager Loading)**

@Query("SELECT e FROM Employee e JOIN FETCH e.department")

List<Employee> getEmployeesWithDepartments();

**JPQL Aggregate Functions**

**COUNT**

@Query("SELECT COUNT(e) FROM Employee e")

Long getEmployeeCount();

**SUM**

@Query("SELECT SUM(e.salary) FROM Employee e")

Double getTotalSalary();

**AVG**

@Query("SELECT AVG(e.salary) FROM Employee e")

Double getAverageSalary();

**JPQL Named Queries**

Named queries are pre-defined JPQL queries stored in entity classes.

**Defining Named Query**

@Entity

@NamedQuery(name = "Employee.findByDepartment", query = "SELECT e FROM Employee e WHERE e.department = :dept")

public class Employee {

@Id

@GeneratedValue

private Long id;

private String name;

private String department;

}

**Using Named Query**

@Query(name = "Employee.findByDepartment")

List<Employee> findEmployeesByDepartment(@Param("dept") String dept);

**JPQL Native Queries**

If JPQL is not sufficient, we can use native SQL queries.

**Example**

@Query(value = "SELECT \* FROM employee WHERE department = ?1", nativeQuery = true)

List<Employee> findEmployeesByDeptNative(String dept);

**JPQL Pagination**

@Query("SELECT e FROM Employee e")

Page<Employee> findEmployees(Pageable pageable);

**JPQL Update and Delete Queries**

**UPDATE Query**

@Modifying

@Query("UPDATE Employee e SET e.salary = :salary WHERE e.id = :id")

void updateSalary(@Param("id") Long id, @Param("salary") Double salary);

**DELETE Query**

@Modifying

@Query("DELETE FROM Employee e WHERE e.id = :id")

void deleteEmployee(@Param("id") Long id);

**JPQL Best Practices**

1. **Use Named Queries** – Improves performance by pre-compiling queries.
2. **Use Fetch Joins** – Avoids LazyInitializationException by eagerly loading related entities.
3. **Avoid N+1 Problem** – Use JOIN FETCH instead of multiple separate queries.
4. **Parameter Binding** – Use named parameters instead of concatenating strings to prevent SQL injection.

**Conclusion**

* JPQL is an object-oriented query language for querying JPA entities.
* It provides flexibility and portability across databases.
* Understanding joins, aggregations, and named queries enhances query performance.
* Use best practices to write efficient and maintainable JPQL queries.

# **Lecture Notes on JPQL (Java Persistence Query Language)**

**Introduction to JPQL**

**What is JPQL?**

Java Persistence Query Language (JPQL) is a query language used in Java Persistence API (JPA) to interact with relational databases using object-oriented queries. It is similar to SQL but operates on entity objects rather than database tables.

**Why Use JPQL When JPA Provides Default Methods?**

JPA provides built-in CRUD methods via JpaRepository that allow basic operations like:

**Available Default Methods in JPA (JpaRepository)**

| **Method** | **Description** |
| --- | --- |
| save(T entity) | Saves or updates an entity |
| findById(ID id) | Finds an entity by primary key |
| findAll() | Retrieves all entities |
| delete(T entity) | Deletes an entity |
| deleteById(ID id) | Deletes an entity by ID |
| existsById(ID id) | Checks if an entity exists |
| count() | Returns the total number of entities |

These default methods are useful for simple database interactions. **However, they are limited when dealing with complex queries, filtering, or aggregations.**

**When Should You Use JPQL Instead of Default JPA Methods?**

1. **Custom Queries with Filters**: Fetching data based on specific conditions.
   * Example: Get employees with a salary greater than 50,000.
2. **Joins Between Entities**: Fetching related data efficiently.
   * Example: Get employees along with their department details.
3. **Aggregation Queries**: Performing operations like SUM, COUNT, or AVG.
   * Example: Calculate the average salary of employees.
4. **Sorting & Pagination**: Fetching sorted or paginated results.
   * Example: Get employees sorted by their joining date.
5. **Update and Delete Operations**: When updating/deleting data without retrieving it first.
   * Example: Increase salaries of all employees in a department by 10%.

**JPQL vs Default JPA Methods**

| **Feature** | **Default JPA Methods** | **JPQL** |
| --- | --- | --- |
| CRUD Operations | Yes | Yes |
| Filtering | Limited | Full control using WHERE |
| Joins | Not supported | Fully supported |
| Aggregations (SUM, AVG, COUNT) | Not supported | Fully supported |
| Bulk Updates/Deletes | Not supported | Fully supported |

**Basic JPQL Syntax**

**SELECT Query (Fetching Specific Data)**

// Fetch all employees

@Query("SELECT e FROM Employee e")

List<Employee> getAllEmployees();

**WHERE Clause (Filtering Data)**

// Fetch employees by department

@Query("SELECT e FROM Employee e WHERE e.department = :dept")

List<Employee> getEmployeesByDepartment(@Param("dept") String department);

**JOINs (Fetching Related Data)**

// Fetch employees with their department details

@Query("SELECT e FROM Employee e JOIN e.department d WHERE d.name = :deptName")

List<Employee> getEmployeesByDeptName(@Param("deptName") String deptName);

**Aggregation Query (SUM, COUNT, AVG)**

// Get total number of employees

@Query("SELECT COUNT(e) FROM Employee e")

Long getEmployeeCount();

**Real-World Example: Employee Management System**

**Scenario**

A company wants to manage employees and their departments using a Spring Boot application.

**Entity Class: Employee**

@Entity

public class Employee {

@Id

@GeneratedValue

private Long id;

private String name;

private String department;

private Double salary;

}

**Using Default JPA Methods**

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<Employee> findByDepartment(String department);

}

**Limitation**: We cannot perform advanced queries like aggregations or joins using default JPA methods.

**Using JPQL to Overcome Limitations**

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

@Query("SELECT e FROM Employee e WHERE e.salary > :minSalary")

List<Employee> findEmployeesWithHighSalary(@Param("minSalary") Double minSalary);

}

**Conclusion**

* **Default JPA methods** are useful for simple CRUD operations.
* **JPQL is needed** for complex queries, joins, aggregations, and batch updates.
* Understanding the differences helps in writing efficient and maintainable code.

**Q&A Session**

Encourage students to experiment with both approaches and discuss real-world use cases!

**Lecture Notes on JPQL (Java Persistence Query Language)**

**Introduction to JPQL**

**What is JPQL?**

Java Persistence Query Language (JPQL) is a query language used in Java Persistence API (JPA) to interact with relational databases using object-oriented queries. It is similar to SQL but operates on entity objects rather than database tables.

**Why Use JPQL When JPA Provides Default Methods?**

JPA provides built-in CRUD methods via JpaRepository that allow basic operations like:

**Available Default Methods in JPA (JpaRepository)**

| **Method** | **Description** |
| --- | --- |
| save(T entity) | Saves or updates an entity |
| findById(ID id) | Finds an entity by primary key |
| findAll() | Retrieves all entities |
| delete(T entity) | Deletes an entity |
| deleteById(ID id) | Deletes an entity by ID |
| existsById(ID id) | Checks if an entity exists |
| count() | Returns the total number of entities |

These default methods are useful for simple database interactions. **However, they are limited when dealing with complex queries, filtering, or aggregations.**

**When Should You Use JPQL Instead of Default JPA Methods?**

1. **Custom Queries with Filters**: Fetching data based on specific conditions.
   * Example: Get employees with a salary greater than 50,000.
2. **Joins Between Entities**: Fetching related data efficiently.
   * Example: Get employees along with their department details.
3. **Aggregation Queries**: Performing operations like SUM, COUNT, or AVG.
   * Example: Calculate the average salary of employees.
4. **Sorting & Pagination**: Fetching sorted or paginated results.
   * Example: Get employees sorted by their joining date.
5. **Update and Delete Operations**: When updating/deleting data without retrieving it first.
   * Example: Increase salaries of all employees in a department by 10%.

**JPQL vs Default JPA Methods**

| **Feature** | **Default JPA Methods** | **JPQL** |
| --- | --- | --- |
| CRUD Operations | Yes | Yes |
| Filtering | Limited | Full control using WHERE |
| Joins | Not supported | Fully supported |
| Aggregations (SUM, AVG, COUNT) | Not supported | Fully supported |
| Bulk Updates/Deletes | Not supported | Fully supported |

**Basic JPQL Syntax**

**SELECT Query (Fetching Specific Data)**

// Fetch all employees

@Query("SELECT e FROM Employee e")

List<Employee> getAllEmployees();

**WHERE Clause (Filtering Data)**

// Fetch employees by department

@Query("SELECT e FROM Employee e WHERE e.department = :dept")

List<Employee> getEmployeesByDepartment(@Param("dept") String department);

**JOINs (Fetching Related Data)**

// Fetch employees with their department details

@Query("SELECT e FROM Employee e JOIN e.department d WHERE d.name = :deptName")

List<Employee> getEmployeesByDeptName(@Param("deptName") String deptName);

**Aggregation Query (SUM, COUNT, AVG)**

// Get total number of employees

@Query("SELECT COUNT(e) FROM Employee e")

Long getEmployeeCount();

**Real-World Example: Employee Management System**

**Scenario**

A company wants to manage employees and their departments using a Spring Boot application.

**Entity Class: Employee**

@Entity

public class Employee {

@Id

@GeneratedValue

private Long id;

private String name;

private String department;

private Double salary;

}

**Using Default JPA Methods**

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<Employee> findByDepartment(String department);

}

**Limitation**: We cannot perform advanced queries like aggregations or joins using default JPA methods.

**Using JPQL to Overcome Limitations**

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

@Query("SELECT e FROM Employee e WHERE e.salary > :minSalary")

List<Employee> findEmployeesWithHighSalary(@Param("minSalary") Double minSalary);

}

**Conclusion**

* **Default JPA methods** are useful for simple CRUD operations.
* **JPQL is needed** for complex queries, joins, aggregations, and batch updates.
* Understanding the differences helps in writing efficient and maintainable code.

**Q&A Session**

Encourage students to experiment with both approaches and discuss real-world use cases!

**Lecture Notes on JPQL (Java Persistence Query Language)**

**Introduction to JPQL**

**What is JPQL?**

Java Persistence Query Language (JPQL) is a query language used in Java Persistence API (JPA) to interact with relational databases using object-oriented queries. It is similar to SQL but operates on entity objects rather than database tables.

**Why Use JPQL When JPA Provides Default Methods?**

JPA provides built-in CRUD methods via JpaRepository that allow basic operations like:

**Available Default Methods in JPA (JpaRepository)**

| **Method** | **Description** |
| --- | --- |
| save(T entity) | Saves or updates an entity |
| findById(ID id) | Finds an entity by primary key |
| findAll() | Retrieves all entities |
| delete(T entity) | Deletes an entity |
| deleteById(ID id) | Deletes an entity by ID |
| existsById(ID id) | Checks if an entity exists |
| count() | Returns the total number of entities |

These default methods are useful for simple database interactions. **However, they are limited when dealing with complex queries, filtering, or aggregations.**

**When Should You Use JPQL Instead of Default JPA Methods?**

1. **Custom Queries with Filters**: Fetching data based on specific conditions.
   * Example: Get employees with a salary greater than 50,000.
2. **Joins Between Entities**: Fetching related data efficiently.
   * Example: Get employees along with their department details.
3. **Aggregation Queries**: Performing operations like SUM, COUNT, or AVG.
   * Example: Calculate the average salary of employees.
4. **Sorting & Pagination**: Fetching sorted or paginated results.
   * Example: Get employees sorted by their joining date.
5. **Update and Delete Operations**: When updating/deleting data without retrieving it first.
   * Example: Increase salaries of all employees in a department by 10%.

**JPQL vs Default JPA Methods**

| **Feature** | **Default JPA Methods** | **JPQL** |
| --- | --- | --- |
| CRUD Operations | Yes | Yes |
| Filtering | Limited | Full control using WHERE |
| Joins | Not supported | Fully supported |
| Aggregations (SUM, AVG, COUNT) | Not supported | Fully supported |
| Bulk Updates/Deletes | Not supported | Fully supported |

**Basic JPQL Syntax**

**SELECT Query (Fetching Specific Data)**

// Fetch all employees

@Query("SELECT e FROM Employee e")

List<Employee> getAllEmployees();

**WHERE Clause (Filtering Data)**

// Fetch employees by department

@Query("SELECT e FROM Employee e WHERE e.department = :dept")

List<Employee> getEmployeesByDepartment(@Param("dept") String department);

**JOINs (Fetching Related Data)**

// Fetch employees with their department details

@Query("SELECT e FROM Employee e JOIN e.department d WHERE d.name = :deptName")

List<Employee> getEmployeesByDeptName(@Param("deptName") String deptName);

**Aggregation Query (SUM, COUNT, AVG)**

// Get total number of employees

@Query("SELECT COUNT(e) FROM Employee e")

Long getEmployeeCount();

**Real-World Example: Employee Management System**

**Scenario**

A company wants to manage employees and their departments using a Spring Boot application.

**Entity Class: Employee**

@Entity

public class Employee {

@Id

@GeneratedValue

private Long id;

private String name;

private String department;

private Double salary;

}

**Using Default JPA Methods**

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<Employee> findByDepartment(String department);

}

**Limitation**: We cannot perform advanced queries like aggregations or joins using default JPA methods.

**Using JPQL to Overcome Limitations**

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

@Query("SELECT e FROM Employee e WHERE e.salary > :minSalary")

List<Employee> findEmployeesWithHighSalary(@Param("minSalary") Double minSalary);

}

**Conclusion**

* **Default JPA methods** are useful for simple CRUD operations.
* **JPQL is needed** for complex queries, joins, aggregations, and batch updates.
* Understanding the differences helps in writing efficient and maintainable code.