

# Advanced Java

---

## Agenda

- JPQL
- Primary Keys

## Spring Data/JPA

### HQL/JPQL queries (JPA)

- JPA queries target to JPA entities & fields (not RDBMS tables & columns).
- JPQL supports SELECT, UPDATE, DELETE, and INSERT (limited) queries.

### JPQL Query examples

- The keywords like SELECT, FROM, WHERE, ORDER BY, GROUP BY, INSERT, UPDATE are case-insensitive; but entity class names and property names are case-sensitive.
- SELECT
  - `SELECT b FROM Book b` // selects all rows mapped columns (@Column) from BOOKS table (given by @Table(name="BOOKS")).
  - `from Book b` // same as above
  - `from Book b where b.subject = :p_subject`
  - `from Book b order by b.price desc`
  - `select distinct b.subject from Book b`
  - `select b.subject, sum(b.price) from Book b group by b.subject`
  - `select new Book(b.id, b.name, b.price) from Book b`
- DELETE
  - `delete from Book b where b.subject = :p_subject`
- UPDATE
  - `update Book b set b.price = b.price + 50 WHERE b.subject = :p_subject`
- INSERT

- insert into Book(id, name, price) select id, name, price from old\_books

## JPQL Joins

- JPQL joins are little different than SQL joins. They can be used on association so that condition will be generated automatically.
  - SQL Joins
    - RDBMS Tables
    - SELECT e.ename, d.dname FROM emp e, dept d WHERE e.deptno = d.deptno; -- SQL'87
    - SELECT e.ename, d.dname FROM emp e INNER JOIN dept d ON e.deptno = d.deptno; -- SQL'93
    - Join conditions: ON t1.col1 = t2.col2
  - JPQL Joins
    - JPA Entities
    - Join conditions: Entity1 e1 JOIN e1.field e2 where "e1.field" represent relation (1-1, 1-n, n-1, n-n)
- Alternatively property of composite object can be given in select or may use traditional join syntax.

```
@Entity
public class Emp {
    @Id
    private int empno;
    private String ename;
    @ManyToOne
    @JoinColumn(name = "deptno")
    private Dept dept;
    // ...
}

@Entity
public class Dept {
    @Id
    private int deptno;
    private String dname;
    @OneToMany(mappedBy="dept")
    private List<Emp> empList;
```

```
// ...  
}
```

- SQL Join Syntax:

```
SELECT e.ename, d.dname FROM emp e INNER JOIN dept d ON e.deptno = d.deptno;
```

- JPQL Join syntax:
  - SELECT e.ename, d.dname FROM Emp e INNER JOIN e.dept d;
  - SELECT e.ename, e.dept.dname FROM Emp e;
  - SELECT e.ename, d.dname FROM Emp e, Dept d WHERE e.dept.deptno = d.deptno;
- Note that, SQL joins are executed automatically when JPA associations are used.
- MobileShop Example

```
class User {  
    @Id  
    private int id;  
    @Column(name="uname")  
    private String name;  
    @OneToMany(mappedBy="user")  
    private List<Order> orderList;  
}  
class Mobile {  
    @Id  
    private int id;  
    @Column(name="mname")  
    private String name;  
    // ...  
}  
class Order {  
    @Id
```

```

private String id;
@ManyToOne
@JoinColumn(name="uid")
private User user;
@ManyToOne
@JoinColumn(name="mid")
private Mobile mobile;
}

```

- JPQL Joins -- Repository interface -- @Query("...")
  - Get last orders date and user name.
    - SELECT o.ordDate, u.name FROM Order o INNER JOIN o.user u ORDER BY o.ordDate DESC LIMIT 1;
    - SELECT o.ordDate, o.user.name FROM Order o ORDER BY o.ordDate DESC LIMIT 1;
  - Get mobile name and mobile image.
    - SELECT m.name, m.image.image FROM Mobile m;
  - Get given users name and his last order date.
    - SELECT u.name, o.ordDate FROM User u INNER JOIN u.orderList o WHERE u.id=?1 ORDER BY o.ordDate DESC LIMIT 1;

### Spring Data Custom Queries

- For the complex requirements queries can be written using @Query.
  - @Query(value="jpql query")
  - @Query(value="sql query", nativeQuery=true)
- The DML queries need to be marked with @Modifying.
- Also DML query should be executed within @Transactional context directly or indirectly.

```

interface MobileDao extends JpaRepository<Mobile,Integer> {
    @Query("SELECT m.ram, COUNT(m.id) FROM Mobile m GROUP BY m.ram")
    List<Object[]> findMobileCountPerRAMSize();

    @Modifying // <-- Spring Boot/Data for DML operations
    @Query("UPDATE Mobile m SET m.price = m.price + ?1")

```

```
    long increaseMobilePriceByAmount(double amount);

    @Modifying
    @Query("DELETE FROM Mobile m WHERE m.company=?1")
    long deleteMobileByCompany(String company);
}
```

## Using NamedParameters

- Unnamed or Named parameters can be used with custom queries.
  - Unnamed parameters use JPA style ordinal numbers (1-based).
  - Named parameters use JPA/Hibernate style names (:name).

```
public interface OrderDetailDao extends JpaRepository<OrderDetail, Integer> {
    @Query("from OrderDetail od WHERE od.quantity * od.rate < ?1")
    List<OrderDetail> findByAmountLessThan(double amount);
    @Query("from OrderDetail od WHERE od.quantity * od.rate > :p_amount")
    List<OrderDetail> findByAmountGreaterThan(@Param("p_amount") double amount);
}
```

## @Embeddable

- TABLE: User table (id, name, password, enabled, email, mobile, address)

```
@Embeddable
class Contact {
    @Column
    private String email;
    @Column
    private String mobile;
}
```

```
@Column
private String address;
// ...
}
@Entity
class Customer {
    @Id
    @Column
    private int id;
    @Column
    private String name;
    @Column
    private String password;
    @Column
    private int enabled;
    @Embedded
    private Contact contact;
    // ...
}
```

## Primary Key

### Auto-generated Primary Key

- @GeneratedValue annotation is used to auto-generate primary key.
- This annotation is used with @Id column.
- There are different strategies for generating ids.
  - IDENTITY: RDBMS AUTO\_INCREMENT / IDENTITY
    - @GeneratedValue(strategy = GenerationType.IDENTITY)
    - MySQL 8: id will be AUTO\_INCREMENT by database.
  - SEQUENCE: RDBMS sequence using @SequenceGenerator

- @SequenceGenerator(name = "gen", sequenceName = "bookIdSeq", initialValue = 10, allocationSize = 2)
  - @GeneratedValue(generator = "gen", strategy = GenerationType.SEQUENCE)
  - MySQL 8: Emulated with table.
  - AUTO: Depends on database dialect.
    - @GeneratedValue(generator = "gen", strategy = GenerationType.AUTO)
    - MySQL 8: id will be taken from "next\_val" column of "gen" table (like emulated sequence).
  - TABLE: Dedicated table for PK generation
    - @TableGenerator(name = "gen", table = "shopIdsTable", initialValue = 10, allocationSize = 2, pkColumnName = "book")
    - @GeneratedValue(generator = "gen", strategy = GenerationType.TABLE)
- In SEQUENCE strategy to maintain different series for each entity type, use different sequence in database.

```
@Entity
public class Book {
    @SequenceGenerator(name = "gen", sequenceName = "bookIdSeq")
    @GeneratedValue(generator = "gen", strategy = GenerationType.SEQUENCE)
    @Id
    private int id;
}
```

```
@Entity
public class User {
    @SequenceGenerator(name = "gen", sequenceName = "userIdSeq")
    @GeneratedValue(generator = "gen", strategy = GenerationType.SEQUENCE)
    @Id
    private int id;
}
```

```
@Entity
public class Order {
    @SequenceGenerator(name = "gen", sequenceName = "orderIdSeq")
    @GeneratedValue(generator = "gen", strategy = GenerationType.SEQUENCE)
    @Id
    private int id;
}
```

- In TABLE strategy to maintain different series for each entity type single table is used.

```
public class Book {
    @TableGenerator(name = "gen", table = "shopIdsTable", pkColumnValue = "book")
    @GeneratedValue(generator = "gen", strategy = GenerationType.TABLE)
    @Id
    private int id;
}
```

```
public class User {
    @TableGenerator(name = "gen", table = "shopIdsTable", pkColumnValue = "user")
    @GeneratedValue(generator = "gen", strategy = GenerationType.TABLE)
    @Id
    private int id;
}
```

```
public class Order {
    @TableGenerator(name = "gen", table = "shopIdsTable", pkColumnValue = "order")
    @GeneratedValue(generator = "gen", strategy = GenerationType.TABLE)
    @Id
```

```
private int id;  
}
```

- The "shopIdsTable" will have three rows (i.e. book, user, and order) to maintain series of these three ids.

### Composite Primary Key

- JPA/Hibernate @Id is always serializable.
- In case of composite PK, create a new class for PK and mark as @Embeddable and Serializable.
- Create object of that class into entity class & mark it as @EmbeddedId.
- @Embeddable used as @Id class must implement equals() and hashCode().

```
// CREATE TABLE students(roll int, std int, name varchar(50), marks double, primary key(roll, std));  
@Entity @Table("students")  
class Student {  
    @EmbeddedId  
    private StdRoll id;  
    @Column  
    private String name;  
    @Column  
    private double marks;  
    // ...  
}  
  
@Embeddable  
class StdRoll implements Serializable {  
    @Column  
    private int std;  
    @Column  
    private int roll;  
}
```

```
// ...  
}
```

## Assignments

1. Create categories table (id, title, description). Example categories are Electronics, Fashion, Grocery. Create products table (id, name, price, category\_id). Create appropriate entity mappings. Create a web mvc application to show all categories. If any category is clicked, on next page display products for that category. Also provide button for creating new product. Create form to add new product. Do validation for new product.