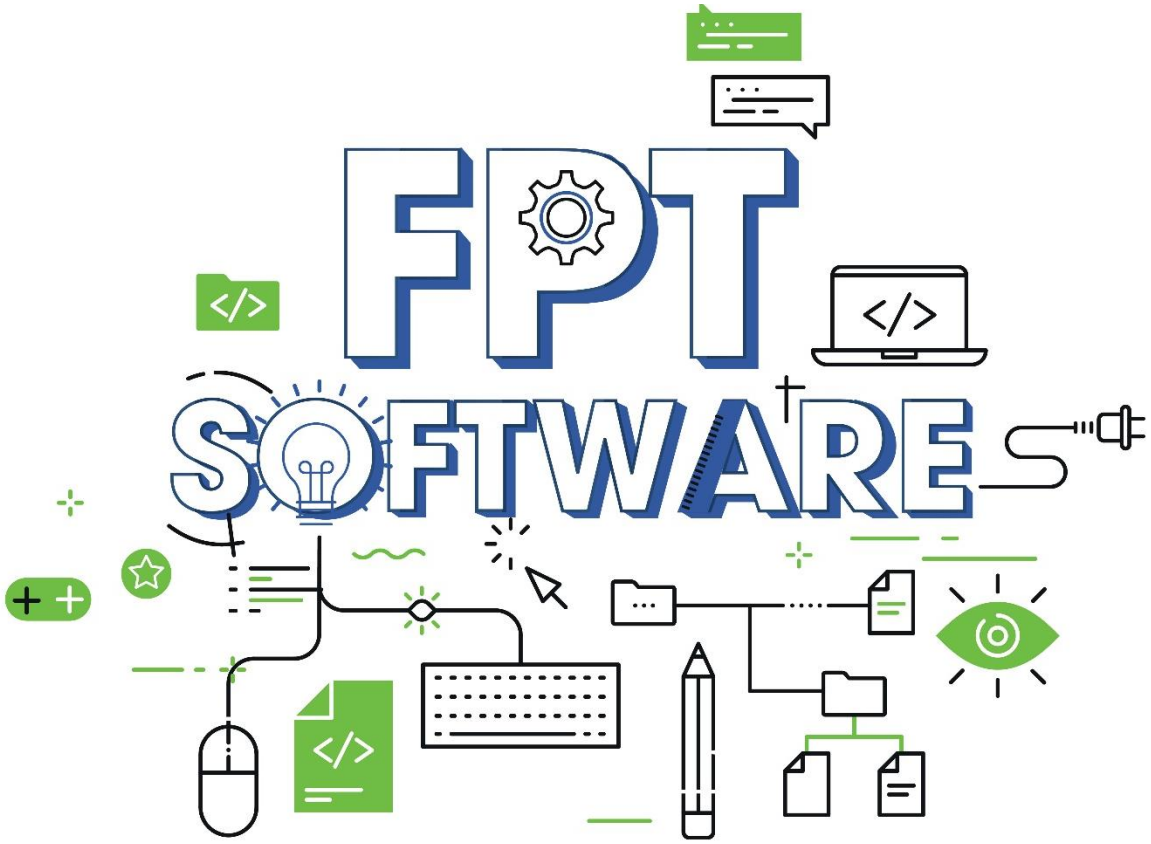
Hibernate Queries

Design by: DieuNT1



Agenda

1. Queries Introduction
2. Native Query
3. Hibernate Query Language
4. Hibernate Named Query
5. Q&A

Lesson Objectives

1. Understand the queries be used in Hibernate.
2. Understand the Native Query and @NamedNativeQuery.
3. Able to use Hibernate Query Language.
4. Able to use Hibernate Named Query and Named Stored Procedure

Section 01

Queries Introduction

Native Query

- You may also express queries in the native SQL dialect of your database.
  - ✓ This is useful if you want to *utilize database specific features* such as query hints or the CONNECT BY option in Oracle.
  - ✓ It also provides a clean migration path from a direct SQL/JDBC based application to Hibernate.
  - ✓ Note that Hibernate allows you to specify handwritten SQL (including stored procedures) for all **create**, **update**, **delete**, and **load** operations.

Hibernate Native Query

Hibernate Query Language

- The Hibernate Query Language (HQL) and Java Persistence Query Language (JPQL) are both object model focused query languages similar in nature to SQL.
  - ✓ JPQL is a heavily-inspired-by subset of HQL.
  - ✓ A JPQL query is always a valid HQL query, however the reverse is not true.
  - ✓ Both HQL and JPQL are non-type-safe ways to perform query operations. Criteria queries offer a type-safe approach to querying.

Hibernate Named Query

- A named query is a ***JPQL*** or ***Navite SQL*** expression with a predefined unchangeable query string.
  - ✓ You can define a named query either in *hibernate mapping file* or in *an entity class*.
  - ✓ The named queries in hibernate is a technique to group the HQL statements in a single location, and lately refer them by some name whenever the need to use them.
  - ✓ It helps largely in code cleanup because these HQL statements are no longer scattered in whole code.

Section 02

NATIVE Query

Native Query

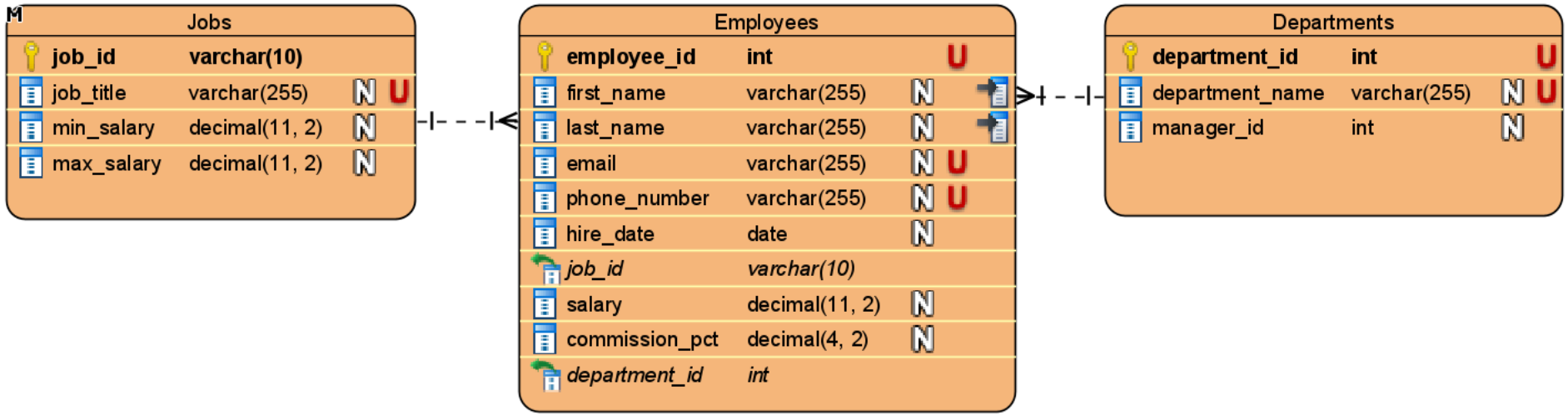
- Hibernate allows us to execute the native SQL queries for all create, update, delete and retrieve operations.
- Hibernate SQL query is not the recommended approach because we loose benefits related to hibernate association and *hibernate first level cache*.
- Query object:
  - ✓ **Syntax** to create the Query object and execute it:

```
Query<Employees> query = session
                        .createNativeQuery(String query, Class resultClass);
```

  - ✓ **SQLQuery Methods:**
    - **List** list() method: returns the list of Object array, we need to explicitly parse them to double, long etc.
    - addEntity() and addJoin() methods to fetch the data from associated table using tables join

Context Buildup

- The following diagram illustrates relationship of the tables used in examples of this section:



See details of these entity classes in Hibernate mapping of the previous Unit!!

Native Query

- **Example 1**: SELECT Statement

```
@Override
public List<Jobs> findAll() throws Exception {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {

            Query<Jobs> query = session.createNativeQuery(
                                "SELECT * FROM dbo.Jobs", Jobs.class);

             return query.list();
        }
}
```

- **Results:**

```
[Jobs [jobId=J01, jobTitle=Java Dev1, minSalary=1000.0, maxSalary=2000.0],
Jobs [jobId=J02, jobTitle=Java Dev2, minSalary=1200.0, maxSalary=2200.0],
Jobs [jobId=J03, jobTitle=Java Dev3, minSalary=1400.0, maxSalary=3200.0]]
```

Native Query

- **Example 2**: native query with the conditions/parameters

```
public List<Jobs> findByNameAndSalary(String title, double salary) throws Exception {
         try (Session session = HibernateUtil.getSessionFactory().openSession()) {

            Query query = session.createNativeQuery(
                   "SELECT * FROM dbo.Jobs j WHERE j.job_title LIKE :title "
                   + "AND j.min_salary <= :salary AND j.max_salary >= :salary", Jobs.class);

            query.setParameter("title", "%" + title + "%");
            query.setParameter("salary", salary);

            return query.list();

        }
}
```

- **Results:**

```
[Jobs [jobId=J01, jobTitle=Java Dev1, minSalary=1000.0, maxSalary=2000.0]]
```

Native Query

- **Example 3**: addEntity(), addJoin()

```
@Override
    public List<Object[]> findAll() throws Exception {
         try (Session session = HibernateUtil.getSessionFactory().openSession()) {
            Query query = session.createNativeQuery(
                    "SELECT j.*, e.* FROM dbo.Jobs j JOIN dbo.Employees e "
                    + "ON j.job_id = e.job_id")
                    .addEntity("j", Jobs.class).addJoin("e", "j.employees");

            List<Object[]> jobs = query.list();

            return jobs;

        }
}
```

Native Query

- **Results:**

```
@Test
void testFindAll() throws Exception {
          List<Object[]> jobs = jobDao.findAll();

          for (Object[] object : jobs) {
             Jobs job = (Jobs) object[0];
             System.out.println(job);

             for (Employees employee : job.getEmployees()) {
                 System.out.println(employee);
             }
          }
}
```

```
Jobs [jobId=J01, jobTitle=Java Dev1, minSalary=1000.0, maxSalary=2000.0]
Employees     [employeeId=5, first_name=Nguyen, last_name=Minh Thanh, email=thanh@fsoft.com.vn, phoneNumber=0988777111, hireDate=1999-01-01, salary=1000.0, commissionPct=1.1]
Employees     [employeeId=1, first_name=Nguyen, last_name=Quang Anh, email=anhnd22@fsoft.com.vn, phoneNumber=0988777666, hireDate=2019-01-01, salary=1000.0, commissionPct=1.1]
              Jobs [jobId=J01, jobTitle=Java Dev1, minSalary=1000.0, maxSalary=2000.0]
Employees     [employeeId=5, first_name=Nguyen, last_name=Minh Thanh, email=thanh@fsoft.com.vn, phoneNumber=0988777111, hireDate=1999-01-01, salary=1000.0, commissionPct=1.1]
Employees     [employeeId=1, first_name=Nguyen, last_name=Quang Anh, email=anhnd22@fsoft.com.vn, phoneNumber=0988777666, hireDate=2019-01-01, salary=1000.0, commissionPct=1.1]
              Jobs [jobId=J02, jobTitle=Java Dev2, minSalary=1200.0, maxSalary=2200.0]
Employees     [employeeId=7, first_name=Hoang, last_name=Van Liem, email=Liem@fsoft.com.vn, phoneNumber=0988777112, hireDate=1999-01-01, salary=1000.0, commissionPct=1.1]
```

Native Query

- Using @**NamedNativeQuery** and @**NamedNativeQueries** Annotations.
- **Syntax**:

```
@Entity
@Table(indexes = {@Index(columnList = "first_name, last_name", name = "IDX_EMP_NAME") })
@NamedNativeQueries({
        @NamedNativeQuery(name = "FIND_EMP_BY_JOB", query = "SELECT e.* "
                + "FROM dbo.Employees e JOIN dbo.Jobs j ON e.job_id = j.job_id "
                + "AND j.job_id LIKE :jobTitle", resultClass = Employees.class),
        @NamedNativeQuery(name = "EMP_FIND_ALL", query = "SELECT * FROM dbo.Employees",
                                                resultClass = Employees.class)
        @NamedNativeQuery(name = "COUNT_EMP",
                                query = "SELECT AVG(e.salary) FROM dbo.Employees e WHERE e.job_id = :jobId")})
public class Employees {


}
```

Native Query

- The session.createNamedQuery(String name) method:

```
@Override
    public List<Employees> findByJob(String jobTile) {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {

            Query<Employees> query = session.createNamedQuery("FIND_EMP_BY_JOB");

            query.setParameter("jobTitle", "%" + jobTile + "%");
            return query.list();

        }
    }
```

Native Query

- The `query.getSingleResult()` method:

```
@Override
    public double countByJob(String jobId) {
        try (Session session = HibernateUtil.getSessionFactory().openSession()) {

            Query query = session.createNamedQuery("COUNT_EMP");
            query.setParameter("jobId", jobId);

            return (double) query.getSingleResult();

        }
}
```

Section 03

Hibernate Query Language

Introduction

HQL or **Hibernate Query Language** is the object-oriented query language of Hibernate Framework.

HQL is **very similar to SQL** except that we **use Objects instead of table** names, that makes it more close to object oriented programming.

Hibernate Query Language (HQL)

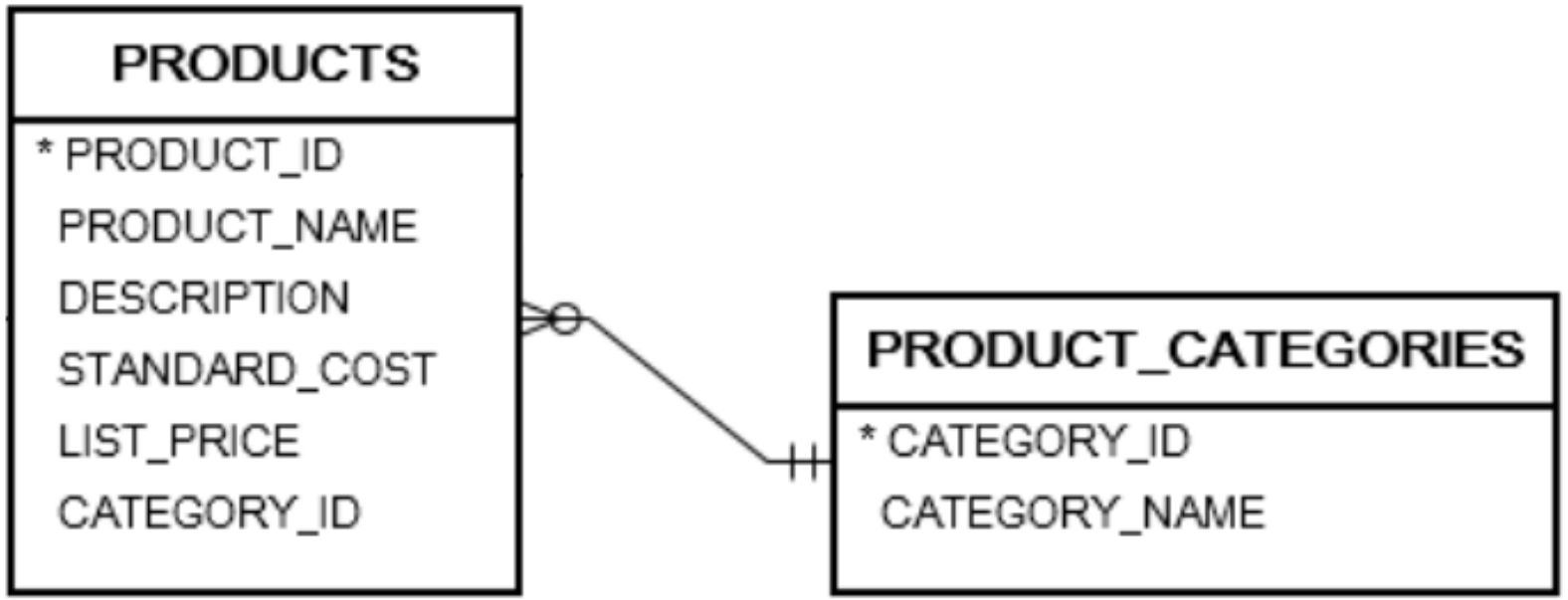
- Syntax is quite similar to database SQL language.
- Uses class name instead of table name, and property names instead of column name:
  - ✓ **SQL similarity:** HQL's syntax is very similar to standard SQL.
  - ✓ **Fully object-oriented:** HQL doesn't use real names of table and columns. It uses class and property names instead. HQL can understand inheritance, polymorphism and association.
  - ✓ **Case-insensitive for keywords:** Like SQL, keywords in HQL are case-insensitive. That means **SELECT**, **select** or **Select** are the same.
  - ✓ **Case-sensitive for Java classes and properties:** HQL considers case-sensitive names for Java classes and their properties, meaning **Person** and **person** are two different objects.

Hibernate Query Language (HQL)

- **HQL From**: HQL From is same as select clause in SQL, `from Employee` is same as `select * from Employee`. We can also create alias such as `from Employee emp` or `from Employee as emp`.
- **HQL Join** :
  - ✓ HQL supports **inner join**, **left outer join**, **right outer join** and **full join**.
  - ✓ For example: `SELECT e.name, a.city FROM Employee e INNER JOIN e.address a`. In this query, Employee class should have a variable named address.
- **Aggregate Functions**: HQL supports commonly used aggregate functions such as `count(*)`, `count(distinct x)`, `min()`, `max()`, `avg()` and `sum()`.
- **Expressions**: HQL supports arithmetic expressions (+, -, *, /), binary comparison operators (=, >=, <=, <>, !=, like), logical operations (and, or, not) etc.
- HQL also supports ordre by and group by clauses.
- HQL also supports sub-queries just like SQL queries.
- HQL supports DDL, DML and executing store procedures too.

Context Buildup

- The following diagram illustrates relationship of the tables used in examples of this section:



See details of these entity classes in Hibernate mapping of the previous Unit!!

Statement types

- HQL features four different kinds of statement:
  - ✓`select` queries,
  - ✓`update` statements,
  - ✓`delete` statements, and
  - ✓`insert` … `values` and `insert` … `select` statements.

Execute HQL in Hibernate

- HQL doesn't require a `select` clause, but JPQL does.
- Basically, it's fairly simple to execute HQL in Hibernate. Here are the steps:
  - ✓ Write your HQL:

```
String hql = "Your Query Goes Here";
```

  - ✓ Create a `Query` from the `Session`:

```
Query query = session.createQuery(hql, Class resultClass);
```

  - ✓ Execute the query: depending on the type of the query (listing or update), an appropriate method is used:
    - For a listing query (`SELECT`)

```
List listResult = query.list();
```

    - For an update query (`INSERT`, `UPDATE`, `DELETE`):

```
int rowsAffected = query.executeUpdate();
```

Execute HQL in Hibernate

- Set parameter before execute the query (if need):

```
query.setParameter(parameterName, value);
```

- Extract result returned from the query: depending of the type of the query, Hibernate returns different type of result set.
  - ✓ Select query on a mapped object returns a list of those objects.
  - ✓ Join query returns a list of arrays of Objects which are aggregate of columns of the joined tables. This also applies for queries using aggregate functions (count, sum, avg, etc).

Hibernate List Query Example

- The following code snippet executes a query that returns all `Category` objects:

```
String hql = "FROM Category";

Query<Categories> query = session.createQuery(hql, Categories.class);
List<Categories> categories = query.list();

for (Categories category : categories) {
      System.out.println(category.getCategoryName());
}
```

Hibernate Search Query Example

- The following statements execute a query that searches for all products in a category whose name in the `categoryName` parameter :

```
String hql = "FROM Products p WHERE p.category.categoryName = :categoryName";

Query<Products> query = session.createQuery(hql, Products.class);
query.setParameter("categoryName", categoryName);

List<Products> products = query.list();

for (Products product : products) {
        System.out.println(product.getProductName());
}
```

- Hibernate automatically generates JOIN query between the `Products` and `Categories` tables behind the scene.

```
select p1_0.product_id,p1_0.category_name,p1_0.descriptions,p1_0.list_price,p1_0.productName,p1_0.st
join Categories c1_0 on c1_0.categoryId=p1_0.category_name where c1_0.category_name=?
```

Hibernate Update/Delete Query Example

- The UPDATE/DELETE query is similar to SQL.
- ***MutationQuery*** **– Implementing modifying queries:** MutationQuery interface is much cleaner and easier to use than the Query interface by excluding all selection-specific methods.

```
Transaction transaction = session.beginTransaction();
MutationQuery query = session.createMutationQuery("UPDATE Products SET listPrice = :listPrice "
                                                  + "WHERE productId = :productId");

query.setParameter("listPrice", product.getListPrice());
query.setParameter("productId", product.getProductId());

int rowsAffected = query.executeUpdate();
if (rowsAffected > 0) {
        System.out.println("Updated " + rowsAffected + " rows.");
}

transaction.commit();
```

Hibernate Insert - Select Query Example

- HQL doesn't support regular INSERT statement (you know why - because the session.persist(Object) method does it perfectly).
- So we can only write INSERT ... SELECT query in HQL.
- The following code snippet executes a query that inserts all rows from Categories table to OldCategories table:

```
String hql = "INSERT INTO Categories (category_name)"
                + " SELECT category_name FROM OldCategories";

MutationQuery query = session.createMutationQuery(hql);

int rowsAffected = query.executeUpdate();

if (rowsAffected > 0) {
        System.out.println(rowsAffected + "(s) were inserted");
}
```

Hibernate Insert

- **Example**:

```
public boolean insert(Book book) {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        Transaction transaction = session.beginTransaction();

        MutationQuery query = session.createMutationQuery("INSERT INTO "
                + "Book(title, year, version) VALUES (?1,?2,?3)");

        query.setParameter(1, book.getTitle());
        query.setParameter(2, book.getYear());
        query.setParameter(3, book.getVersion());

        int result = query.executeUpdate();
        transaction.commit();

        return (result > 0);
    }
}
```

Hibernate Join Query Example

HQL supports the following join types (similar to SQL):

- **INNER JOIN** (can be abbreviated as **JOIN**).
- **LEFT OUTER JOIN** (can be abbreviated as **LEFT JOIN**).
- **RIGHT OUTER JOIN** (can be abbreviated as **RIGHT JOIN**).
- **FULL JOIN**

- For example, the following code snippet executes a query that retrieves results which is a join between two tables **Products** and **Categories**:

```
String hql = "FROM Products p JOIN p.category";

Query<Object[]> query = session.createQuery(hql, Object[].class);
List<Object[]> listResult = query.list();

for (Object[] aRow : listResult) {
        Products product = (Products) aRow[0];
        Categories category = (Categories) aRow[1];
        System.out.println(product.getProductName() + " - " + category.getCategoryName());
}
```

Hibernate Join Query Example

- Using the JOIN keyword in HQL is called **explicit join**.
- Note that a JOIN query returns a list of Object arrays, so we need to deal with the result set differently:

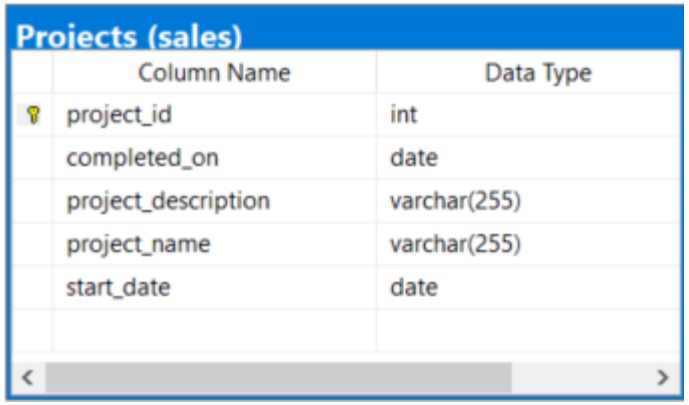
```
List<Object[]> listResult = query.list();
```

- HQL provides **with** keyword which can be used in case you want to supply extra join conditions. For example:

```
FROM Products p JOIN p.category WITH p.listPrice > 500
```

Hibernate Sort Query Example

- Sorting in HQL is very similar to SQL using ORDER BY clause follows by a sort direction ASC (ascending) or DESC (descending).



Projects (sales)

	Column Name	Data Type
🔑	project_id	int
	completed_on	date
	project_description	varchar(255)
	project_name	varchar(255)
	start_date	date

```
public List<Projects> searching(LocalDate startDate) throws Exception {
     try (Session session = HibernateUtil.getSessionFactory()
                                          .openSession()) {

     String hql = "FROM Projects WHERE startDate >= :startDate "
                  + "ORDER BY completedOn DESC";

     Query<Projects> query = session.createQuery(hql, Projects.class);

     query.setParameter("startDate", startDate);

     return query.list();

     }
}
```

Hibernate Group By Query Example

- Using GROUP BY clause in HQL is similar to SQL.
- The following query summarizes price of all products grouped by each category:

```
String hql = "SELECT p.category.categoryName, SUM(p.listPrice) "
                + "FROM Products p GROUP BY category";

Query<Object[]> query = session.createQuery(hql, Object[].class);

List<Object[]> listResult = query.list();

for (Object[] aRow : listResult) {
        String category = (String) aRow[0];
        Double sum = (Double) aRow[1];
        System.out.println(category + " - " + sum);
}
```

Hibernate Pagination Query Example

- To return a subset of a result set, the Query interface has two methods for limiting the result set:
  - ✓ setFirstResult(intfirstResult): sets the first row to retrieve.
  - ✓ setMaxResults(intmaxResults): sets the maximum number of rows to retrieve.
- For example, the following code snippet lists first 10 products:

```
Query query = session.createQuery("FROM Products", Products.class);

query.setFirstResult(0);
query.setMaxResults(10);

List<Product> listProducts = query.list();

for (Product aProduct : listProducts) {
    System.out.println(aProduct.getName() + "\t - " + aProduct.getPrice());
}
```

Using Aggregate Functions in Hibernate Query

- HQL supports the following aggregate functions:
  - ✓ avg(…), sum(…), min(…), max(…)
  - ✓ count(*)
  - ✓ count(…), count(distinct…), count(all…)

```
String hql = "SELECT COUNT(productName) FROM Products";


Query query = session.createQuery(hql, Products.class);
List listResult = query.list();
Number number = (Number) listResult.get(0);
System.out.println(number.intValue());
```

Named Query

- The hibernate named query is way to use any query by some **meaningful name**. It is like using *alias names*.
- So that application programmer need not to **scatter** queries to all the java code.
- There are two ways to define the named query in hibernate:
  - **by annotation**
  - by mapping file

Named Query

- Named Query by Annotation:
  - **@NameQueries:** is used to define the multiple named queries.
  - **@NameQuery:** is used to define the single named query.
  - **Syntax:**

```
@NamedQueries(
   {
      @NamedQuery(
         name = "findProductByName",
         query = “FROM Products p WHERE p.productName = :name"
      )
   }
)
public class Products {

}
```

@NamedStoredProcedureQuery

- Create a new user stored procedure:

```
CREATE PROC usp_FindProduct(
    @nameNVARCHAR(200),
    @beginFLOAT,
    @endFLOAT
)

AS
BEGIN
    SELECT p.*
    FROM dbo.Products p
    WHERE p.product_name LIKE '%'+@name+'%' AND
                (p.list_price BETWEEN @begin AND @end)

END
```

@NamedStoredProcedureQuery

- Declaring Named Stored Procedure:

```
@NamedStoredProcedureQueries({
@NamedStoredProcedureQuery(name = "findProduct", procedureName = "usp_FindProduct",
        resultClasses = {Products.class},
        parameters = {
            @StoredProcedureParameter(mode = ParameterMode.IN, type = String.class),
            @StoredProcedureParameter(mode = ParameterMode.IN, type = Double.class),
            @StoredProcedureParameter(mode = ParameterMode.IN, type = Double.class)
        }
)
})
public class Products {

}
```

@NamedStoredProcedureQuery

- Executing Stored Procedure:

```
@Override
public List<Products> find(String name, double begin, double end) {
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        StoredProcedureQuery procedureQuery =
                        session.createNamedStoredProcedureQuery("findProduct");

        procedureQuery.setParameter(1, name);
        procedureQuery.setParameter(2, begin);
        procedureQuery.setParameter(3, end);

        return procedureQuery.getResultList();
    }
}
```

Summary

- Queries Introduction
- Native Query
- Hibernate Query Language
- Hibernate Named Query

THANK YOU!