DAM. UNIT 3. ACCESS USING OBJECT-RELATIONAL MAPPING (ORM). ACCESS TO RELATIONAL DATABASES USING HIBERNATE HQL

DAM. Acceso a Datos (ADA) (a distancia en inglés)

Unit 3. ACCESS USING OBJECT-RELATIONAL MAPPING (ORM)

Access to relational databases using Hibernate HQL

Abelardo Martínez

Year 2024-2025

1. Hibernate queries

Hibernate provides several options to deal with our database:

1. **Native SQL queries** (low level of abstraction)
   Use MAINLY native SQL language to set complex queries
2. **HQL Queries** (medium level of abstraction)
   MIX specific Hibernate query language and methods
3. **HQL Queries using criteria** (high level of abstraction)
   Use MAINLY specific methods (NO QUERY AT ALL!)

1.1. Native queries

Hibernate provides an option to execute native SQL queries through the use of SQLQuery object. Hibernate SQL Query is very handy when we have to execute database vendor specific queries that are not supported by Hibernate API. For normal scenarios, Hibernate SQL query is not the recommended approach because we lose benefits related to hibernate association and hibernate first level cache.

For Hibernate Native SQL Query, we use **Session.createSQLQuery(String query)** to create the SQLQuery object and execute it. The syntax is as follows:

```
query = session.createSQLQuery("select ... from …  where … order by ...”);
rows = query.list();
for(Object[] row : rows){ … }
```

Notice that list() method returns the List of Object array, we need to explicitly parse them to double, long etc.

Hibernate Native SQL Example

Detailed example: https://www.journaldev.com/3422/hibernate-native-sql-query-example

For instance, you can join two tables by using direct SQL queries:

```
query = session.createSQLQuery("select e.emp_id, emp_name, emp_salary,address_line1, city,
    zipcode from Employee e, Address a where a.emp_id=e.emp_id");
rows = query.list();
for(Object[] row : rows){
    Employee emp = new Employee();
    emp.setId(Long.parseLong(row[0].toString()));
    emp.setName(row[1].toString());
    emp.setSalary(Double.parseDouble(row[2].toString()));
    Address address = new Address();
    address.setAddressLine1(row[3].toString());
    address.setCity(row[4].toString());
    address.setZipcode(row[5].toString());
    emp.setAddress(address);
    System.out.println(emp);
```

```
}
```

We can also pass parameters to the Hibernate SQL queries, just like JDBC PreparedStatement. The parameters can be set using the name as well as index. For example:

```
query = session
        .createSQLQuery("select emp_id, emp_name, emp_salary from Employee where emp_id = ?");
List<Object[]> empData = query.setLong(0, 1L).list();
for (Object[] row : empData) {
    Employee emp = new Employee();
    emp.setId(Long.parseLong(row[0].toString()));
    emp.setName(row[1].toString());
    emp.setSalary(Double.parseDouble(row[2].toString()));
    System.out.println(emp);
}
```

That was a brief introduction of Hibernate Native SQL Query. Remember you should avoid using it unless you want to execute any database specific queries. For a deep view on those queries you can have a look here: https://docs.jboss.org/hibernate/core/3.3/reference/en/html/querysql.html

1.2. HQL queries

Hibernate supports an object-oriented query language called **HQL** (**H**ibernate **Q**uery **L**anguage). It is a language derived from SQL, with a strong object orientation. It provides great ease of use while allowing complex queries. The Hibernate Query Language (HQL) is the query language used by Hibernate to obtain objects from the database. Its main peculiarity is that the queries are performed on the java objects that form our business model, that is, the entities that are persisted in Hibernate. This means that HQL has the following characteristics:

- The data types are Java data types.
- Queries are independent of the database-specific SQL language.
- Queries are independent of the database table model.
- It is possible to deal with Java collections.
- It is possible to navigate between the different objects in the query itself.

Although you can use native SQL, it is recommended to use HQL whenever possible to avoid database portability hassles, and to take advantage of Hibernate's caching strategies. The syntax is as follows:

```
List items = session.createQuery("from … where …  order by ...").list();
for (Iterator iterator = items.iterator(); iterator.hasNext();) { … }
```

> Be aware of the keywords like SELECT, FROM, and WHERE, etc., are not case sensitive, but properties like table and column names are case sensitive in HQL.

Native HQL and SQL queries are represented by an instance of **org.hibernate.query**. To get a Query we must make use of the **createQuery** method of the **Session** class. Some of the main methods of Query are:

Method	Description
Iterator iterate()	Returns the result of the query in an Iterator object.
List list()	Returns the result of the query in a List.
Query setFetchSize (int size)	Sets the number of results to retrieve for each access to the database.
int executeUpdate()	Executes a modify or delete statement. Returns the number of affected rows.
String getQueryString()	Returns the query in a String.
Object uniqueResult()	Returns an object (when we know the query returns one) or null.
Query setCharacter(int pos, char value) Query setCharacter(String nameCol, char value)	Assigns the specified value to a char parameter.
Query setDate(int pos, Date value) Query setDate(String nameCol, Date value)	Assigns the specified value to a Date parameter.
Query setDouble(int pos, double value) Query setDouble(String nameCol, double value)	Assigns the specified value to a double parameter.
Query setInteger(int pos, int value) Query setInteger(String nameCol, int value)	Assigns the specified value to an integer parameter.
Query setString(int pos, String value) Query setString(String nameCol, String value)	Assigns the specified value to a string parameter.
Query setParameterList(String name, Collection values)	Assigns a collection of values to the parameter.
Query setParameter(String name, Object object) Query setParameter(int position, Object object)	Assigns a value to the specified parameter.

1.2.1. DQL clauses

There are many HQL clauses available to interact with relational databases. Data Query Language (DQL) based are listed below:

- FROM Clause
- SELECT Clause
- WHERE Clause
- ORDER BY Clause
- GROUP BY Clause

1) FROM Clause

To load a whole persistent object into memory, the FROM clause is used.

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

2) SELECT Clause

The SELECT clause is used when only a few attributes of an object are required rather than the entire object.

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

It is notable here that **Employee.firstName** is a property of Employee object rather than a field of the EMPLOYEE table.

3) WHERE Clause

Filtering records is done with the WHERE clause. It's used to retrieve only the records that meet a set of criteria.

```
String hql = "FROM Employee E WHERE E.id = 10";
Query query = session.createQuery(hql);
List results = query.list();
```

4) ORDER BY Clause

The ORDER BY clause is used to sort the results of an HQL query. Note that DESC will sort in descending order and ASC will sort in ascending order.

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas as follows:

```
String hql = "FROM Employee E WHERE E.id > 10 " +
             "ORDER BY E.firstName DESC, E.salary DESC ";
Query query = session.createQuery(hql);
List results = query.list();
```

5) GROUP BY Clause

This clause lets Hibernate pull information from the database and group it based on a value of an attribute and, typically, use the result to include an aggregate value.

```
String hql = "SELECT SUM(E.salary), E.firstName FROM Employee E " +
             "GROUP BY E.firstName";
Query query = session.createQuery(hql);
List results = query.list();
```

1.2.2. DML clauses

With the HQL language we can also perform Insert, Update and Delete operations in the classical SQL style. Data Manipulation Language (DML) based are listed below:

- UPDATE Clause
- DELETE Clause
- INSERT Clause

The syntax is as follows:

```
(UPDATE | DELETE) [FROM] EntityName [WHERE condition]
```

where:

- The FROM clause is optional.
- The WHERE clause is optional.

There can be only one entity mentioned in the FROM clause and it can have an alias. No association can be specified in an HQL bulk query. Subqueries can be used in the WHERE clause (subqueries can contain associations). To execute UPDATE or DELETE in HQL, we will use the **executeUpdate()** method which returns the number of entities affected by the operation.

The Insert sentence, on the other hand, has the following syntax:

```
INSERT INTO EntityName (property list) select_query
```

where:

- Only the INSERT INTO ... SELECT ... form is supported, not the INSERT INTO ... VALUES form, i.e. we can only insert data coming from another table.
- The list of properties is analogous to the list of columns in the SQL INSERT statement.
- The SELECT statement can be any valid HQL SELECT query.
- Make sure that the types returned by the select query match those expected by the INSERT.

Named Parameters

Hibernate supports the inclusion of parameters in queries in the same way JDBC did. We can refer to them by their position or their name. Parameter names in JDBC have the format **:name** in the query string. Hibernate numbers parameters starting from 0. To assign values to parameters we use the setXXX() methods seen in the previous table.

This makes writing queries that accept input from the user easy and you do not have to defend against SQL injection attacks. When using JDBC query parameters, any time you add, change or delete parts of the SQL statement, you need to update your Java code that sets its parameters because the parameters are indexed based on the order in which they appear in the statement.

Hibernate lets you provide names for the parameters in the HQL query, so you do not have to worry about accidentally moving parameters around in the query.

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id",10);
List results = query.list();
```

1) UPDATE Clause

The UPDATE clause is required to update the value of an attribute.

```
String hql = "UPDATE Employee set salary = :salary "  +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

2) DELETE Clause

It is required to delete a value of an attribute.

```
String hql = "DELETE FROM Employee "  +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
```

```
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

3) INSERT Clause

It is required to Insert values into the relation.

```
String hql = "INSERT INTO Employee(firstName, lastName, salary)"  +
             "SELECT firstName, lastName, salary FROM old_employee";
Query query = session.createQuery(hql);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

1.2.3. Aggregate Methods

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL, so you do not have to learn any specific Hibernate terminology. The difference is that in HQL, aggregate methods apply to the properties of persistent objects. The list of the available functions is shown below:

Function	Description
avg(property name)	The average of a property's value.
count(property name or *)	The number of times a property occurs in the results.
max(property name)	The maximum value of the property values.
min(property name)	The minimum value of the property values.
sum(property name)	The sum total of the property values.

1.2.4. Pagination using Query

For pagination using query we have two methods available for it, below table contains the methods and its description.

Method	Action performed
Query setMaxResults(int max)	Instructs Hibernate to get a specific number of items.
Query setFirstResult(int starting_no)	Takes an integer as an argument that represents the first row in your result set, beginning with row 0.

For example:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
query.setFirstResult(1);
query.setMaxResults(10);
List results = query.list();
```

1.2.5. HQL language overview

To sum up:

- HQL queries are not case sensitive, except for Java class and property names.
- The simplest clause that exists in Hibernate is FROM, which gets all instances of a class. For example "FROM Employee" gets all instances of the class Employee.
- The ORDER BY clause sorts the query results.
- The WHERE clause allows you to refine the list of instances returned.
- We can assign aliases to classes using the AS clause: FROM Employee AS em, or without using as: FROM Employee em.
- Several classes can appear in the FROM clause, which produces a Cartesian product or a Cross Join: FROM Employee AS em, Certificate AS cer.

To obtain certain properties (columns) in a query we use the SELECT clause: SELECT firstname, salary FROM Employee, obtains the attributes firstname and salary of the class Employee.

Queries can return multiple objects and/or properties such as an array of type Object[], a list, a class, etc.

- HQL supports grouping functions in a similar way to SQL:
  - avg(...), sum(...), min(...), max(...)
  - count(*)
  - count(...), count(distinct ...), count(all ...)
- Aliases may be used to name attributes and expressions. Arithmetic operators, concatenation operators and SQL functions recognised in the SELECT clause may be used.
- Expressions used in the WHERE clause include the following:
  - Mathematical operators: +, -, *, /
  - Binary comparison operators =, >=, <=, <>, !=, like
  - Logical operators and, or, not
  - Parentheses indicating grouping.
  - in, not in, between, is null, is empty, is not empty, member of, not member of...

- String concatenation: || or concat(... , ...)
- current_date(), current_time, current_timestamp(), etc.
- Any HQL defined function or operator.
- JDBC positional parameters
- Java constants

1.3. HQL criteria

Hibernate provides three different ways to retrieve data from a database. We have already discussed HQL and native SQL queries. Now we will discuss our third option i.e. **hibernate criteria queries**. The criteria query API lets us build nested, structured query expressions in Java, providing a compile-time syntax checking that is not possible with a query language like HQL or SQL. The Criteria API also includes **query by example (QBE)** functionality. This lets us supply example objects that contain the properties we would like to retrieve instead of having to step-by-step spell out the components of the query. It also includes projection and aggregation methods, including count().

> The Hibernate Criteria API had been deprecated back in Hibernate 5.x and these have been **removed in Hibernate 6.0**. Usually, all queries using the legacy API can be modeled with the **JPA Criteria API** that is still supported. In some cases it is necessary to use the Hibernate JPA Criteria extensions. https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial/
>
> Most of the resources you find surfing the Net will be about the deprecated version. You can use this ones to use the new approach: https://www.baeldung.com/hibernate-criteria-queries

Hibernate-specific extensions to the Criteria API

JPA specification defines the string-based JPQL query language and Hibernate extends it to support things like database-specific functions, window functions, and set-based operations. Since version 6, Hibernate has done the same for JPA's Criteria API. Hibernate 6 handles this by providing the ***HibernateCriteriaBuilder*** interface, which extends JPA's CriteriaBuilder interface, and by adding a method to its proprietary Session interface to get a HibernateCriteriaBuilder instance.

The syntax is as follows:

```
CriteriaBuilder cb = session.getCriteriaBuilder();

CriteriaQuery<Item> cr = cb.createQuery(Item.class);
cr.select(root).where(cb.gt(root.get("field"), 10000)); // "field" must have same type as the
other value
cr.orderBy(cb.desc(root.get("field")));
```

```
Query<Item> query = session.createQuery(cr);
List<Item> items = query.getResultList();
for (Iterator iterator = items.iterator(); iterator.hasNext();) { … }
```

The above query is a simple demonstration of how to get all the items. Let's see it step by step:

- Create an instance of CriteriaBuilder by calling the getCriteriaBuilder() method
- Create an instance of CriteriaQuery by calling the CriteriaBuilder createQuery() method
- Create an instance of Query by calling the Session createQuery() method
- Call the getResultList() method of the query object, which gives us the results

The CriteriaBuilder can be used to restrict query results based on specific conditions, by using CriteriaQuery where() method and providing Expressions created by CriteriaBuilder. Let's see some examples of commonly used Expressions.

1.3.1. Comparison operators

Let's see some examples of commonly used Expressions.

In order to get items having a price of more than 1000:

```
cr.select(root).where(cb.gt(root.get("itemPrice"), 1000));
```

Next, getting items having itemPrice less than 1000:

```
cr.select(root).where(cb.lt(root.get("itemPrice"), 1000));
```

Items having itemName contain Chair:

```
cr.select(root).where(cb.like(root.get("itemName"), "%chair%"));
```

Records having itemPrice between 100 and 200:

```
cr.select(root).where(cb.between(root.get("itemPrice"), 100, 200));
```

Items having itemName in Skate Board, Paint and Glue:

```
cr.select(root).where(root.get("itemName").in("Skate Board", "Paint", "Glue"));
```

1.3.2. Property NULL/NOT NULL

To check if the given property is null:

```
cr.select(root).where(cb.isNull(root.get("itemDescription")));
```

To check if the given property is not null:

```
cr.select(root).where(cb.isNotNull(root.get("itemDescription")));
```

We can also use the methods isEmpty() and isNotEmpty() to test if a List within a class is empty or not.

1.3.3. Logical operators and char expressions

Additionally, we can combine two or more of the above comparisons. The Criteria API allows us to easily chain expressions:

```
Predicate[] predicates = new Predicate[2];
predicates[0] = cb.isNull(root.get("itemDescription"));
predicates[1] = cb.like(root.get("itemName"), "chair%");
cr.select(root).where(predicates);
```

To add two expressions with logical operations:

```
Predicate greaterThanPrice = cb.gt(root.get("itemPrice"), 1000);
Predicate chairItems = cb.like(root.get("itemName"), "Chair%");
```

Items with the above-defined conditions joined with Logical OR:

```
cr.select(root).where(cb.or(greaterThanPrice, chairItems));
```

To get items matching with the above-defined conditions joined with Logical AND:

```
cr.select(root).where(cb.and(greaterThanPrice, chairItems));
```

1.3.4. Sorting

In the following example, we order the list in ascending order of the name and then in descending order of the price:

```
cr.orderBy(
cb.asc(root.get("itemName")),
cb.desc(root.get("itemPrice")));
```

1.3.5. Aggregate functions

Get row count:

```
CriteriaQuery<Long> cr = cb.createQuery(Long.class);
Root<Item> root = cr.from(Item.class);
cr.select(cb.count(root));
Query<Long> query = session.createQuery(cr);
List<Long> itemProjected = query.getResultList();
```

The following is an example of aggregate function for average:

```
CriteriaQuery<Double> cr = cb.createQuery(Double.class);
Root<Item> root = cr.from(Item.class);
cr.select(cb.avg(root.get("itemPrice")));
Query<Double> query = session.createQuery(cr);
List avgItemPriceList = query.getResultList();
```

Other useful aggregate methods are sum(), max(), min(), count(), etc.

1.3.6. CriteriaUpdate

CriteriaUpdate has a **set() method** that can be used to provide new values for database records:

```
CriteriaUpdate<Item> criteriaUpdate = cb.createCriteriaUpdate(Item.class);
Root<Item> root = criteriaUpdate.from(Item.class);
criteriaUpdate.set("itemPrice", newPrice);
criteriaUpdate.where(cb.equal(root.get("itemPrice"), oldPrice));

Transaction transaction = session.beginTransaction();
session.createQuery(criteriaUpdate).executeUpdate();
transaction.commit();
```

In the above snippet, we create an instance of CriteriaUpdate<Item> from the CriteriaBuilder and use its set() method to provide new values for the itemPrice. In order to update multiple properties, we just need to call the set() method multiple times.

1.3.7. CriteriaDelete

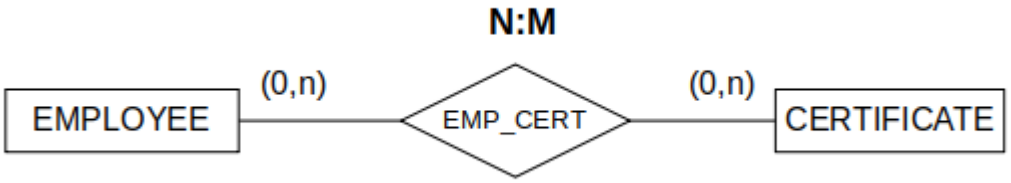
CriteriaDelete enables a delete operation using the Criteria API. We just need to create an instance of CriteriaDelete and use the where() method to apply restrictions:

```
CriteriaDelete<Item> criteriaDelete = cb.createCriteriaDelete(Item.class);
Root<Item> root = criteriaDelete.from(Item.class);
criteriaDelete.where(cb.greaterThan(root.get("itemPrice"), targetPrice));

Transaction transaction = session.beginTransaction();
session.createQuery(criteriaDelete).executeUpdate();
transaction.commit();
```

2. Setting up the project and the database

2.1. The (MySQL) database

We will use the same database as seen in the ORM mapping with XML.



We create the database and the user:

```
CREATE DATABASE IF NOT EXISTS ADAU3DBExample CHARACTER SET utf8mb4 COLLATE utf8mb4_es_0900_ai_ci;

CREATE USER mavenuser@localhost IDENTIFIED BY 'ada0486';
GRANT ALL PRIVILEGES ON ADAU3DBExample.* to mavenuser@localhost;

USE ADAU3DBExample;
```

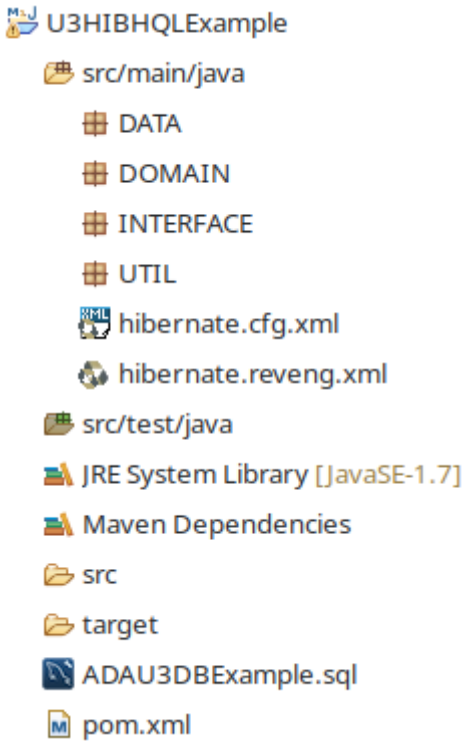
We create the structure:

```
CREATE TABLE Employee (
empID       INTEGER PRIMARY KEY AUTO_INCREMENT,
firstname   VARCHAR(20),
lastname    VARCHAR(20),
salary      FLOAT
);

CREATE TABLE Certificate (
certID      INTEGER PRIMARY KEY AUTO_INCREMENT,
certname    VARCHAR(30)
);

CREATE TABLE EmpCert (
employeeID     INTEGER,
certificateID  INTEGER,
```

```
PRIMARY KEY (employeeID, certificateID),
CONSTRAINT emp_id_fk FOREIGN KEY (employeeID) REFERENCES Employee(empID),
CONSTRAINT cer_id_fk FOREIGN KEY (certificateID) REFERENCES Certificate(certID)
);
```

2.2. The (Java-Maven) project

We will replicate the previous project with annotations to use HQL.



3. Upgrading example with HQL

We will now implement methods to use HQL queries.

3.1. POJO. Class EmployeeDAO

We'll create a new method to list only the employees earning greater than 10000, order by salary descending. It's as easy as adding the condition using HQL, almost identical (in this case) to standard SQL:

```
//Object name constant
static final String OBJEMPNAME = "Employee";
static final String OBJCERTNAME = "Certificate";

/* READ all the employees earning > 10000. STANDARD VERSION */
    public void listRichEmployees() {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibernate session
factory
        Transaction txDB = null; //database transaction

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            List<Employee> listEmployees = hibSession.createQuery("FROM " + OBJEMPNAME + " WHERE
salary > 10000 ORDER BY salary DESC", Employee.class).list();
            if (listEmployees.isEmpty())
                System.out.println("******** No items found");
            else
                System.out.println("\n***** Start listing ...\n");
                for (Iterator<Employee> itEmployee = listEmployees.iterator();
itEmployee.hasNext();) {
                    Employee objEmployee = (Employee) itEmployee.next();
                    System.out.print(OBJEMPNAME + " first Name: " + objEmployee.getFirstname() +
" | ");
                    System.out.print("Last name: " + objEmployee.getLastname() + " | ");
                    System.out.println("Salary: " + objEmployee.getSalary());
                    Set<Certificate> setCertificates = objEmployee.getCertificates();
                    for (Iterator<Certificate> itCertificate = setCertificates.iterator();
itCertificate.hasNext();) {
                        Certificate objCertificate = (Certificate) itCertificate.next();
                        System.out.println(OBJCERTNAME + " name: " + objCertificate.getCertname()
+ '\n');
                    }
                }
                txDB.commit(); //ends transaction
```

```
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

3.2. Interface layer

Now we modify the main programme to add more employees and apply the new query.

```
package INTERFACE;

import java.util.HashSet;

import DATA.*;
import DOMAIN.*;
import UTIL.*;

/**
 * ============================================================================
 * Interface layer. Program to manage Employees and Certificates
 * @author Abelardo Martínez. Based and modified from Sergio Badal
 * ============================================================================
 */

/*
 * For further information see this tutorial:
 * https://www.tutorialspoint.com/hibernate/hibernate_examples.htm
 */

public class TestHibernateMySQL {

    /*
     * -----------------
     * MAIN PROGRAMME
     * -----------------
     */
    public static void main(String[] stArgs) {

        //Create new objects DAO for CRUD operations
        EmployeeDAO objEmployeeDAO = new EmployeeDAO();
        CertificateDAO objCertificateDAO = new CertificateDAO();

        //TRUNCATE TABLES. Delete all records from the tables
        objEmployeeDAO.deleteAllItems();
        objCertificateDAO.deleteAllItems();
```

```
        // Add certificates in the database
        Certificate objCert1 = objCertificateDAO.addCertificate("MBA");
        Certificate objCert2 = objCertificateDAO.addCertificate("PMP");

        //Set of certificates
        HashSet<Certificate> hsetCertificates = new HashSet<Certificate>();
        hsetCertificates.add(objCert1);
        hsetCertificates.add(objCert2);

        // Add employees in the database
        Employee objEmp1 = objEmployeeDAO.addEmployee("Alfred", "Vincent", 4000,
hsetCertificates);
        Employee objEmp2 = objEmployeeDAO.addEmployee("John", "Gordon", 3000, hsetCertificates);
        Employee objEmp3 = objEmployeeDAO.addEmployee("Edgard", "Codd", 20000, hsetCertificates);
        Employee objEmp4 = objEmployeeDAO.addEmployee("Joseph", "Smith", 5000, hsetCertificates);
        Employee objEmp5 = objEmployeeDAO.addEmployee("Mary", "Jones", 15000, hsetCertificates);

        // List down all the employees and their certificates
        objEmployeeDAO.listEmployees();
        //List down all the rich employees. Standard way (HQL no criteria)
        objEmployeeDAO.listRichEmployees();

        //Close global hibernate session factory
        HibernateUtil.shutdownSessionFactory();
    }

}
```

4. Upgrading example with HQL criteria

We will now implement methods to use HQL criteria queries.

4.1. POJO. Class EmployeeDAO

We'll create a new method to list only the employees earning greater than 10000, order by salary descending. We translate the method into HQL criteria:

```
/* READ all the employees earning > 10000. HQL CRITERIA VERSION */
    public void listRichEmployeesHQLC() {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //hibernate session factory
        Transaction txDB = null; //database transaction
        System.out.println("\n***** Listing rich employees using HQL criteria...\n");

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            // 1. Create a CriteriaBuilder instance by calling the
            // Session.getCriteriaBuilder() method.
            //Hibernate Extensions to the API
            //https://thorben-janssen.com/hibernate-specific-extensions-to-the-criteria-api/
            //HibernateCriteriaBuilder hcbCritBuilder =
hibSession.unwrap(Session.class).getCriteriaBuilder();
            //HibernateCriteriaBuilder hcbCritBuilder = (HibernateCriteriaBuilder)
hibSession.getCriteriaBuilder();
            CriteriaBuilder crbCritBuilder = hibSession.getCriteriaBuilder();
            // 2. Create a query object by creating an instance of the CriteriaQuery interface.
            //Hibernate Extensions to the API
            //https://thorben-janssen.com/hibernate-specific-extensions-to-the-criteria-api/
            CriteriaQuery<Employee> crqHQL = crbCritBuilder.createQuery(Employee.class);
            // 3. Set the query Root by calling the from() method on the CriteriaQuery
            // object to define a range variable in FROM clause.
            //Hibernate Extensions to the API
            //https://thorben-janssen.com/hibernate-specific-extensions-to-the-criteria-api/
            Root<Employee> rootEmployee = crqHQL.from(Employee.class);
            // 4. Specify what the type of the query result will be by calling the select()
            // method of the CriteriaQuery object.
            crqHQL.select(rootEmployee).where(crbCritBuilder.gt(rootEmployee.get("salary"),
10000));
            crqHQL.orderBy(crbCritBuilder.desc(rootEmployee.get("salary")));
            // 5. Prepare the query for execution by creating a org.hibernate.query.Query
            // instance by calling the Session.createQuery() method, specifying the type of
            // the query result.
            Query<Employee> qryHQL = hibSession.createQuery(crqHQL);
```

```
            // 6. Execute the query by calling the getResultList() or getSingleResult()
            // method on the org.hibernate.query.Query object.
            List<Employee> listEmployees = qryHQL.getResultList();

            if (listEmployees.isEmpty())
                System.out.println("******** No items found");
            else
                System.out.println("\n***** Start listing ...\n");
                for (Iterator<Employee> itEmployee = listEmployees.iterator();
itEmployee.hasNext();) {
                    Employee objEmployee = (Employee) itEmployee.next();
                    System.out.print(OBJEMPNAME + " first Name: " + objEmployee.getFirstname() +
" | ");
                    System.out.print("Last name: " + objEmployee.getLastname() + " | ");
                    System.out.println("Salary: " + objEmployee.getSalary());
                    Set<Certificate> setCertificates = objEmployee.getCertificates();
                    for (Iterator<Certificate> itCertificate = setCertificates.iterator();
itCertificate.hasNext();) {
                        Certificate objCertificate = (Certificate) itCertificate.next();
                        System.out.println(OBJCERTNAME + " name: " + objCertificate.getCertname()
+ '\n');
                    }
            }
            txDB.commit(); //ends transaction
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
    }
```

4.2. Interface layer

Now we modify the main programme to apply the new query after the previous standard HQL.

```
package INTERFACE;

import java.util.HashSet;

import DATA.*;
import DOMAIN.*;
import UTIL.*;

/**
 * =============================================================================
 * Interface layer. Program to manage Employees and Certificates
 * @author Abelardo Martínez. Based and modified from Sergio Badal
 * =============================================================================
 */

/*
 * For further information see this tutorial:
 * https://www.tutorialspoint.com/hibernate/hibernate_examples.htm
 */

public class TestHibernateMySQL {

    /*
     * -----------------
     * MAIN PROGRAMME
     * -----------------
     */
    public static void main(String[] stArgs) {

        //Create new objects DAO for CRUD operations
        EmployeeDAO objEmployeeDAO = new EmployeeDAO();
        CertificateDAO objCertificateDAO = new CertificateDAO();

        //TRUNCATE TABLES. Delete all records from the tables
        objEmployeeDAO.deleteAllItems();
        objCertificateDAO.deleteAllItems();
```

```
        // Add certificates in the database
        Certificate objCert1 = objCertificateDAO.addCertificate("MBA");
        Certificate objCert2 = objCertificateDAO.addCertificate("PMP");

        //Set of certificates
        HashSet<Certificate> hsetCertificates = new HashSet<Certificate>();
        hsetCertificates.add(objCert1);
        hsetCertificates.add(objCert2);

        // Add employees in the database
        Employee objEmp1 = objEmployeeDAO.addEmployee("Alfred", "Vincent", 4000,
hsetCertificates);
        Employee objEmp2 = objEmployeeDAO.addEmployee("John", "Gordon", 3000, hsetCertificates);
        Employee objEmp3 = objEmployeeDAO.addEmployee("Edgard", "Codd", 20000, hsetCertificates);
        Employee objEmp4 = objEmployeeDAO.addEmployee("Joseph", "Smith", 5000, hsetCertificates);
        Employee objEmp5 = objEmployeeDAO.addEmployee("Mary", "Jones", 15000, hsetCertificates);

        // List down all the employees and their certificates
        objEmployeeDAO.listEmployees();
        //List down all the rich employees. Standard way (HQL no criteria)
        objEmployeeDAO.listRichEmployees();
        //List down all the rich employees. New way (HQL using criteria)
        objEmployeeDAO.listRichEmployeesHQLC();

        //Close global hibernate session factory
        HibernateUtil.shutdownSessionFactory();
    }

}
```

5. Bibliography

Sources

- Josep Cañellas Bornas, Isidre Guixà Miranda. Accés a dades. Desenvolupament d'aplicacions multiplataforma. Creative Commons. Departament d'Ensenyament, Institut Obert de Catalunya. Dipòsit legal: B. 29430-2013. https://ioc.xtec.cat/educacio/recursos
- Alberto Oliva Molina. Acceso a datos. UD 3. Herramientas de mapeo objeto relacional (ORM). IES Tubalcaín. Tarazona (Zaragoza, España).
- Hibernate ORM Documentation - 6.5. https://hibernate.org/orm/documentation/6.5/
- Hibernate Native SQL Queries with CRUD Operations. https://www.javaguides.net/2018/11/guide-to-hibernate-native-sql-queries.html
- A Guide to Hibernate Query Language. https://docs.jboss.org/hibernate/orm/6.3/querylanguage/html_single/Hibernate_Query_Language.html
- Geekforgeeks. Hibernate – Query Language. https://www.geeksforgeeks.org/hibernate-query-language/
- Cursohibernate. http://www.cursohibernate.es/doku.php?id=unidades:05_hibernate_query_language:02_hql
- Tutorialspoint. Hibernate - Query Language. https://www.tutorialspoint.com/hibernate/hibernate_query_language.htm
- HowToDoInJava. Guide to Hibernate Criteria Queries. https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial/
- JavaTPoint. HCQL (Hibernate Criteria Query Language). https://www.javatpoint.com/hcql
- Baeldung. JPA Criteria Queries. https://www.baeldung.com/hibernate-criteria-queries
- Oracle. How to Write Doc Comments for the Javadoc Tool. https://www.oracle.com/es/technical-resources/articles/java/javadoc-tool.html