GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport
ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

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

Part 1. Access using Hibernate Classic

Acceso a Datos (ADA) (a distancia en inglés)
CFGS Desarrollo de Aplicaciones Multiplataforma (DAM)

Abelardo Martínez
Year 2024-2025

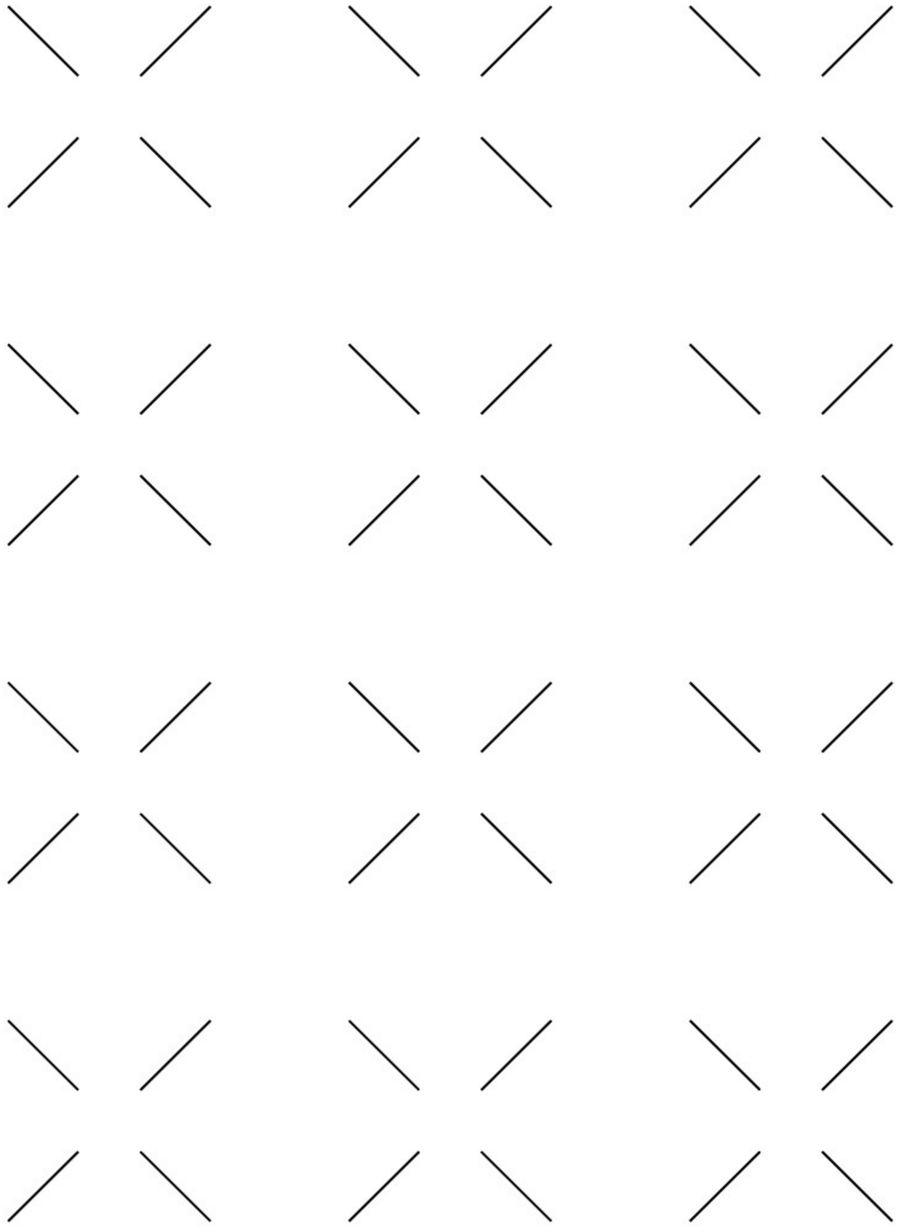
Credits

- Notes made by Abelardo Martínez.
- Based and modified from Sergio Badal (www.sergiobadal.com).
- The images and icons used are protected by the LGPL licence and have been obtained from:
  - https://commons.wikimedia.org/wiki/Crystal_Clear
  - https://www.openclipart.org

Contents

1. WHAT IS ORM?

1.1 Introduction



ORM (Object–relational mapping)

1) **DAO**. A data access object is a pattern that is often followed when an application needs to interact with some persistent data store (often a database). The DAO provides a series of operations to the rest of the application without the application needing to know the details of the data store.

2) **ORM**. An ORM usually describes a more robust library/API used to make interactions with a database.

In summary, a DAO is an object that abstracts the implementation of a persistent data store away from the application and allows for simple interaction with it.
An **ORM** is a robust library/API that provides a bunch of tools to save/retrieve an object directly to/from the database **without having to write your own SQL statements**.

Data Access Object (DAO) pattern is a structural pattern that allows us to isolate the application/business layer from the persistence layer using an abstract API.

Object–Relational Mapping (ORM) is a technique for converting data between incompatible type systems using object-oriented programming languages.

DAO and ORM

DAO. There are several ways to encapsulate database entities into classes (DAO). The two most used DAO over Java projects are:

- **POJO** ⟵
- **JavaBeans**

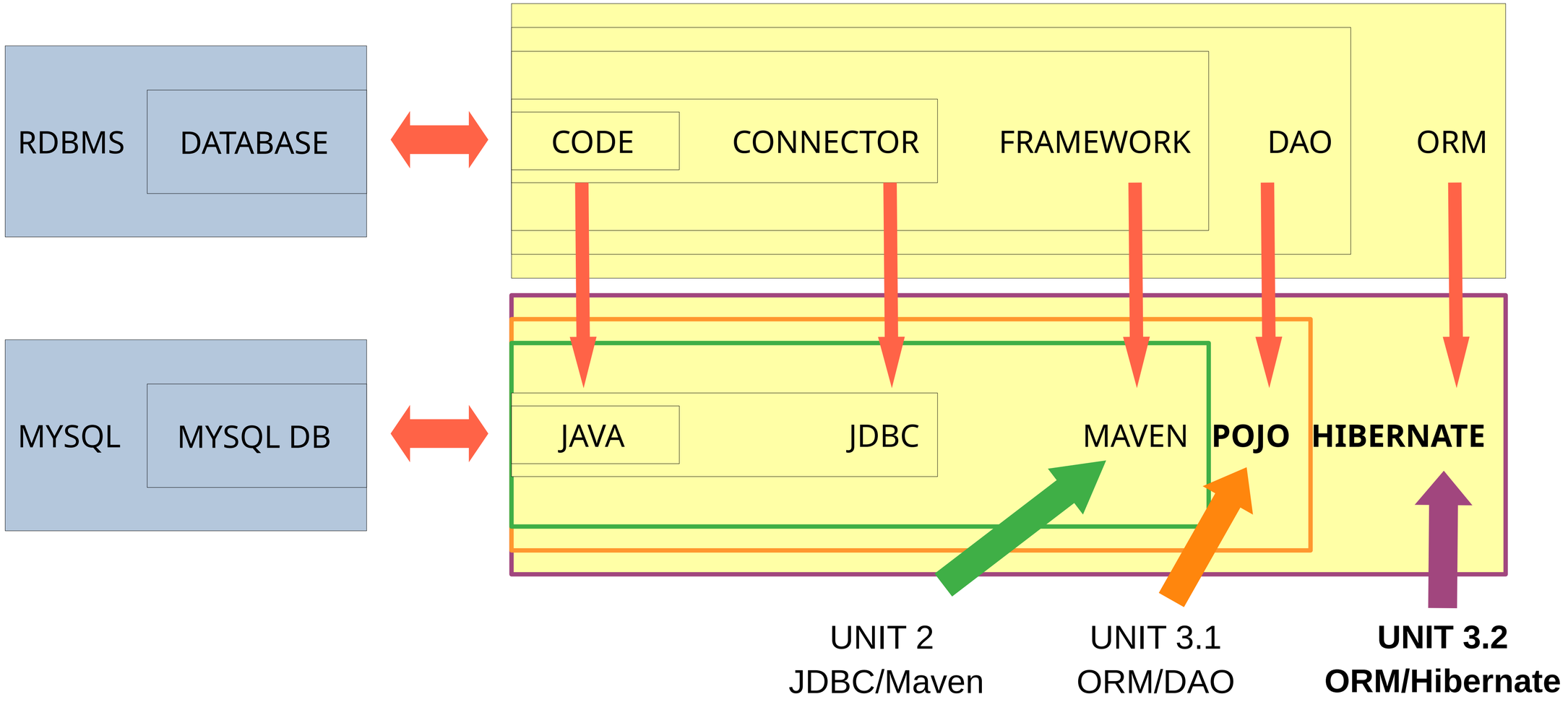
ORM. There are several ways to build the “bridge” between the database and the code (ORM). The most used ORM over Java projects are:

- **Hibernate** ⟵
- **Spring**

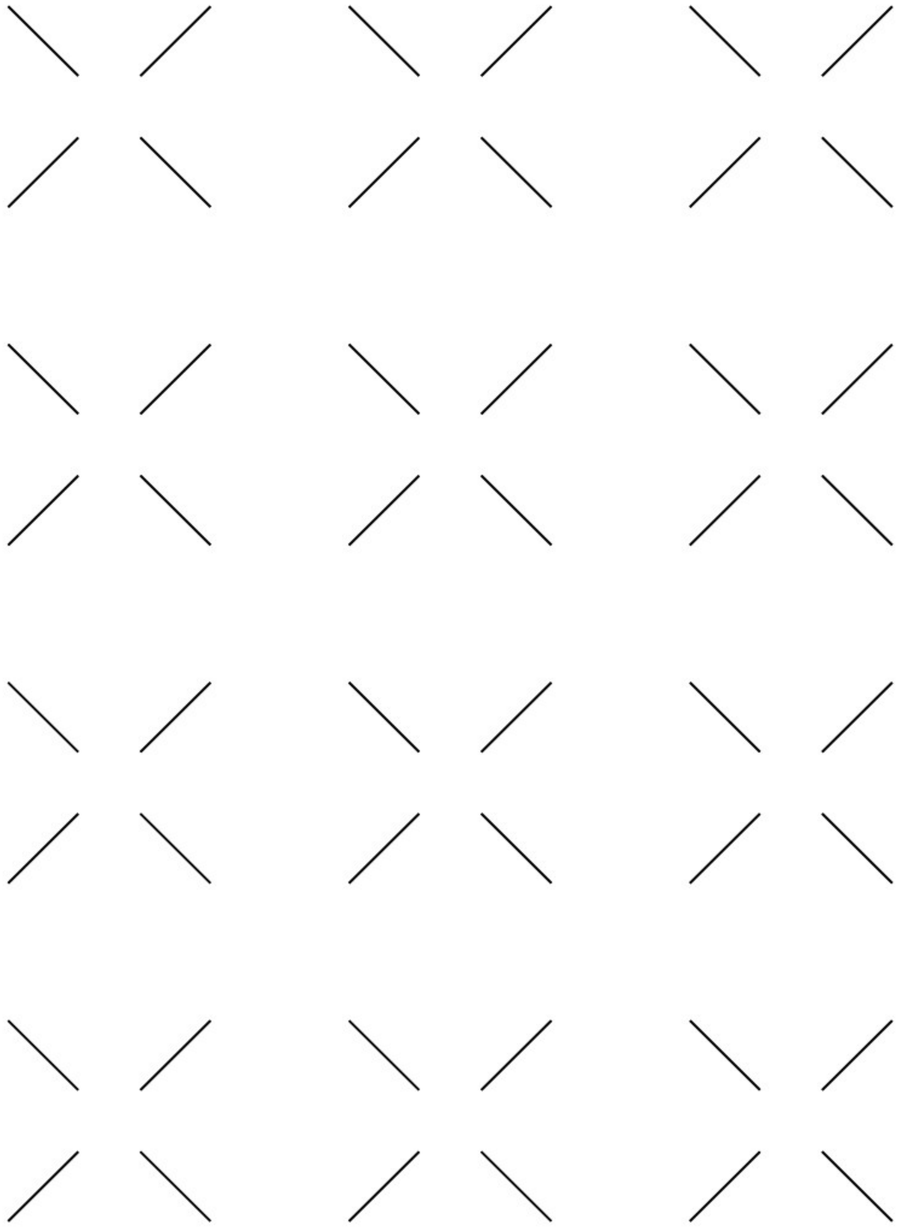
We will use those ones!

Encapsulation layers

What we are doing is connecting the **code** with the **database** by using layers, tools and frameworks to make it easier. We will use these ones:



1.2 POJO in Java



POJO in Java

POJO in Java stands for Plain **O**ld **J**ava **O**bject. It is an ordinary object, which is not bound by any special restriction. The POJO file does not require any special classpath. It increases the readability & re-usability of a Java program.

POJOs are now widely accepted due to their easy maintenance. They are easy to read and write.

A POJO class does not have any naming convention for properties and methods. It is not tied to any Java Framework; any Java Program can use it.

The term POJO was introduced by Martin Fowler (an American software developer) in 2000.

Features POJO encapsulation

Properties of POJO class:

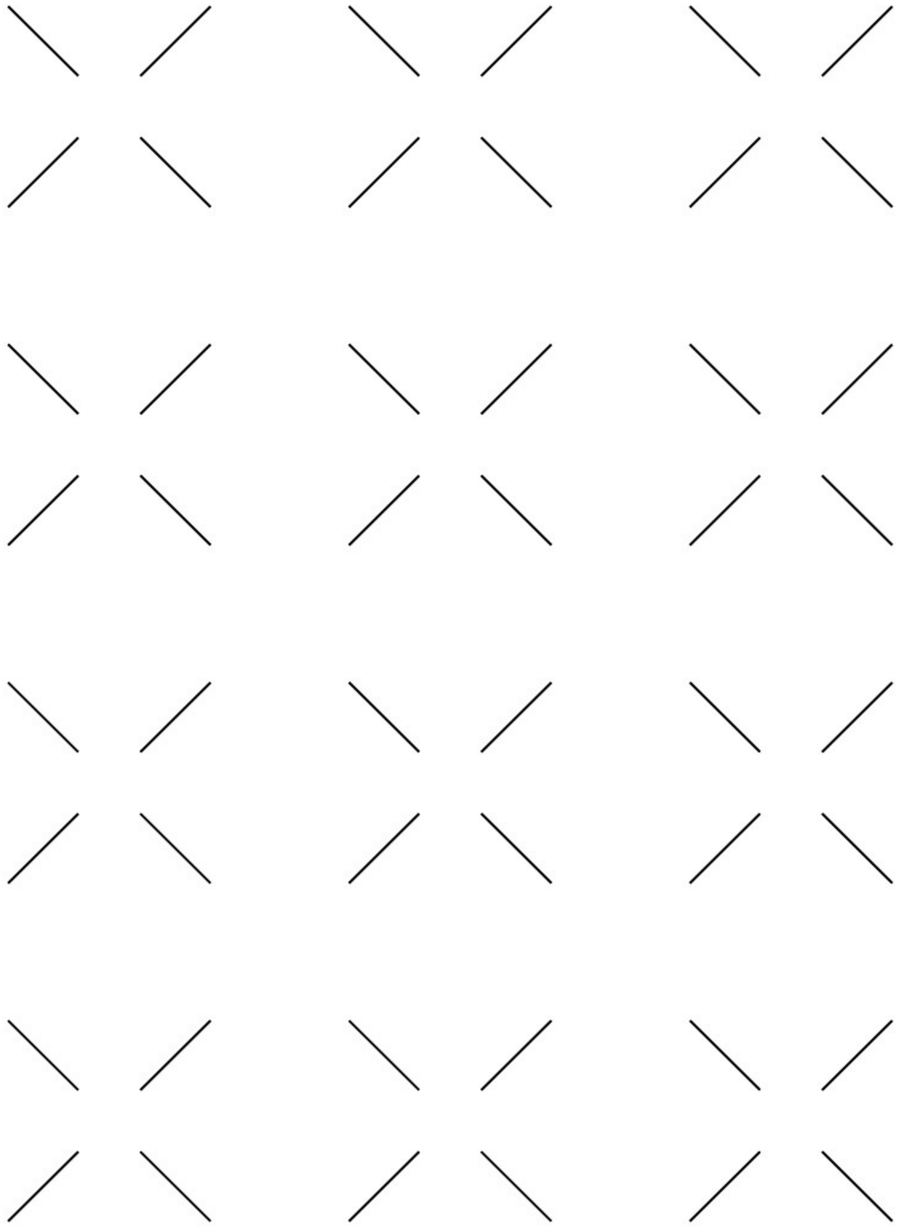
- The POJO class must be public.
- It must have a public default constructor.
- It may have the arguments constructor.
- All objects must have some public Getters and Setters to access the object values by other Java Programs.
- The object in the POJO Class can have any access such as private, public, protected. But, all instance variables should be private for improved security of the project.
- A POJO class should not extend predefined classes.
- It should not implement prespecified interfaces.
- It should not have any prespecified annotation.

For further information: https://www.javatpoint.com/pojo-in-java

Employee.java:

```
// POJO class Exmaple
package Jtp.PojoDemo;
public class Employee {
private String name;
private String id;
private double sal;
public String getName() {
   return name;
}
public void setName(String name) {
   this.name = name;
}
public String getId() {
   return id;
}
public void setId(String id) {
   this.id = id;
}
public double getSal() {
   return sal;
}
public void setSal(double sal) {
   this.sal = sal;
}
}
```

1.3 ORM. Hibernate



ORM. Hibernate

Hibernate is an Object-Relational Mapping (ORM) solution for JAVA. It is an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieves the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.

Features Hibernate encapsulation

Advantages

- Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.
- Provides simple APIs for storing and retrieving Java objects directly to and from the database.
- If there is a change in the database or in any table, then you need to change the XML file properties only.
- Abstracts away the unfamiliar SQL types and provides a way to work around familiar Java Objects.
- Hibernate does not require an application server to operate.
- Manipulates complex associations of objects of your database.
- Minimises database access with smart fetching strategies.
- Provides simple querying of data.

For further information:
https://www.tutorialspoint.com/hibernate/hibernate_overview.htm

Supported Databases

Hibernate supports almost all the major RDBMS:

- Oracle
- MySQL
- Microsoft SQL Server Database
- PostgreSQL
- HSQL Database Engine
- DB2/NT
- FrontBase
- Sybase SQL Server
- Informix Dynamic Server

Supported Technologies

Hibernate supports a variety of other technologies, including:

- XDoclet Spring
- J2EE
- Eclipse plug-ins
- Maven

2. ELEMENTS OF A HIBERNATE PROJECT

Elements of a Hibernate project

We’re now building a project from scratch to create a **CRUD** application to work with some tables using:

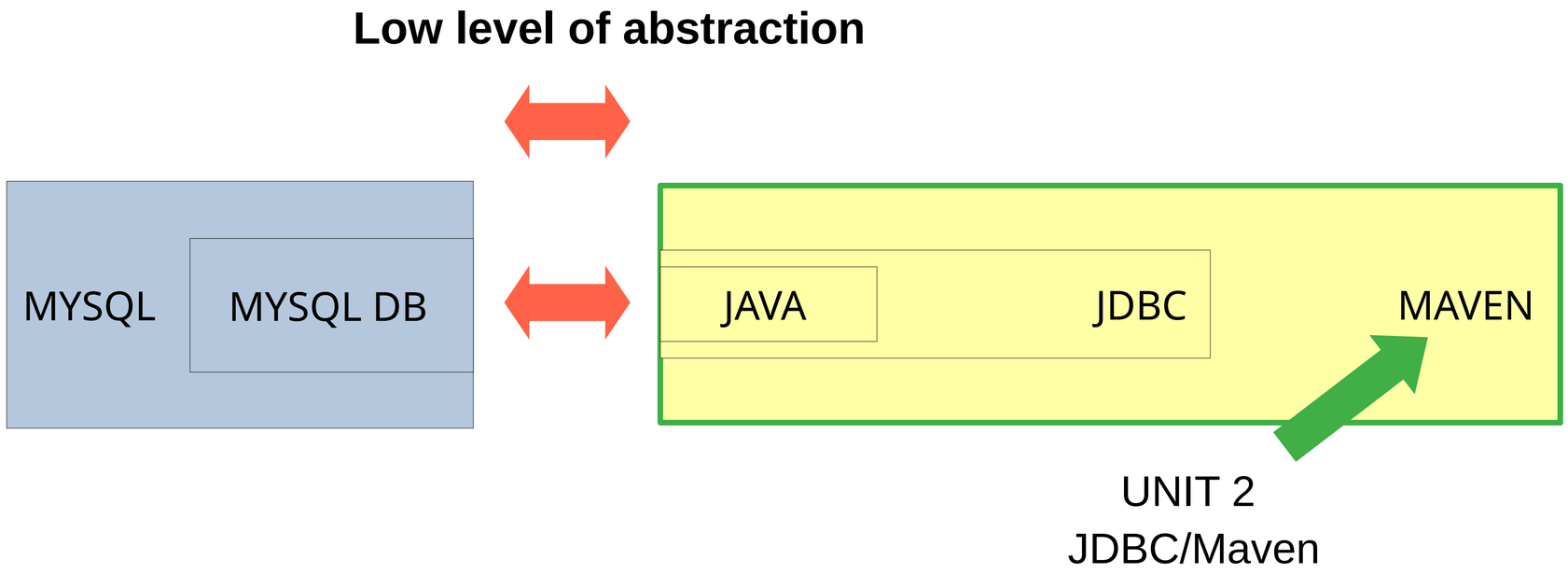
- **MySQL** as our **RDBMS**
- **JDBC** as our **connector**
- **Java** as our **language**
- **MAVEN** as our **framework**
- **POJO** as our **DAO pattern**
- **Hibernate** as our **ORM technique**

Firstly, let’s review what we’re trying to achieve

Encapsulation. Level 1

First thing we did (UNIT 2) was to access the database by using JDBC connector and Maven framework over Java language.

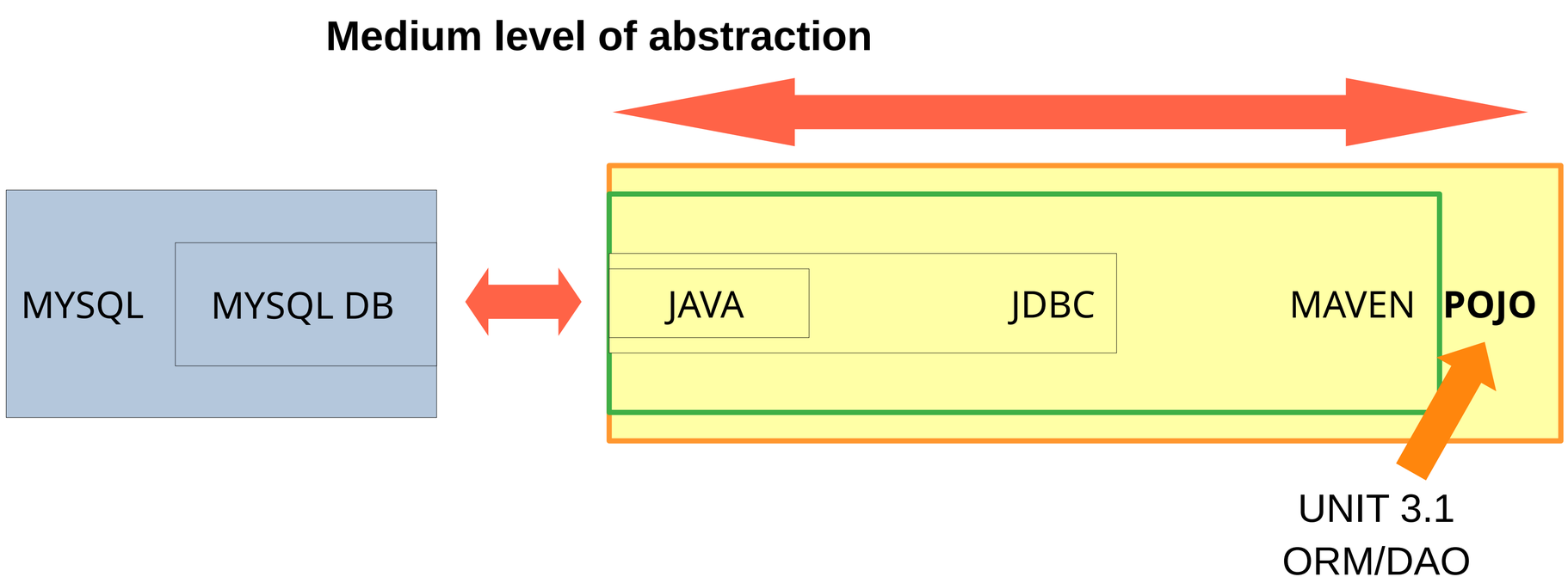
In that case, we had to use SQL code at the main class! The level of abstraction (the distance from the database to the code) was irrelevant.



Encapsulation. Level 2

Secondly, (UNIT 3 WEEK 1) we connected to the database by using JDBC connector, Maven framework and DAO objects using POJO patterns over Java language.

In that case, we moved the SQL code to a DAO class, introducing a separation layer between the database and the code. The level of abstraction was medium and still close to the design of the database.

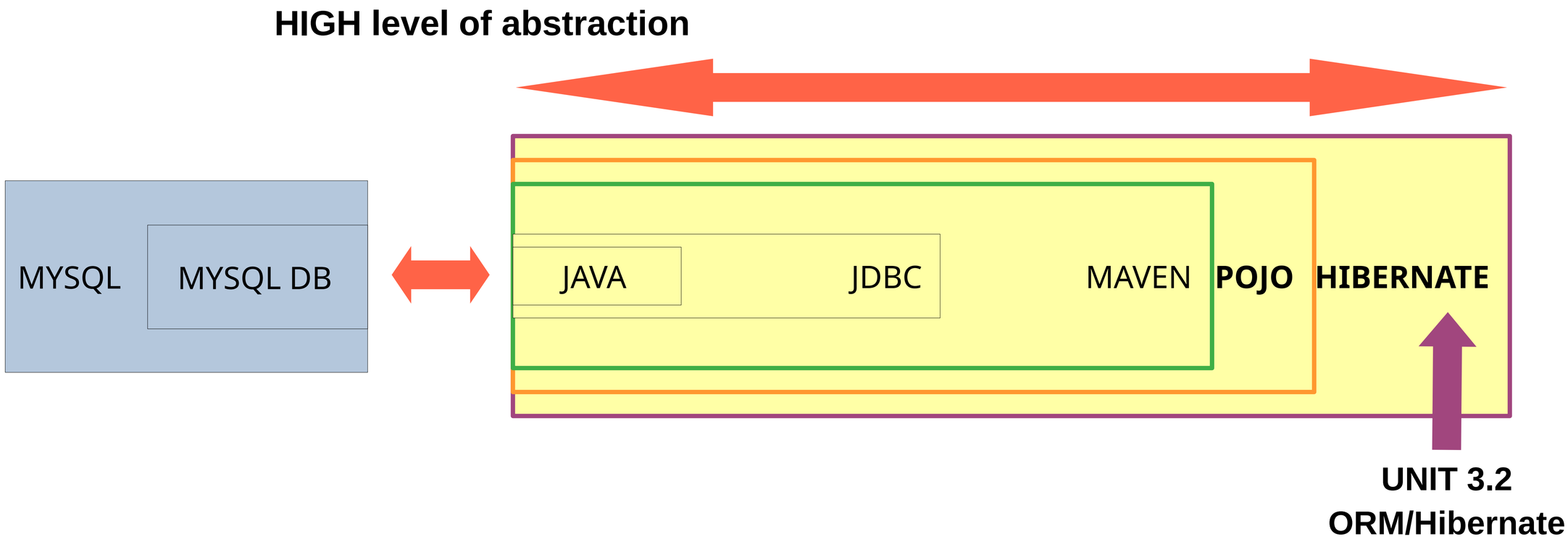


We didn't mention POJO last week but we used this pattern as you already should have guessed

Encapsulation. Level 3

Now, (UNIT 3 WEEK 2 and next) we will connect the database by using JDBC connector, Maven framework, DAO objects using POJO patterns, and ORM technique over Java language.

The SQL code will DISSAPEAR from our Java code and the level of abstraction will be maximum.



3. SETTING UP THE PROJECT & THE DATABASE

Example resources

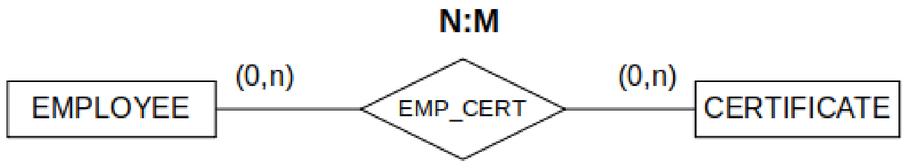
We’ll be presenting a solution made from these great tutorials:

- https://www.tutorialspoint.com/hibernate
- https://www.javawebtutor.com/articles/maven/maven_hibernate_example.php
- https://www.journaldev.com/2934/hibernate-many-to-many-mapping-join-tables

The (MySQL) database

For now, just create a simple database with these tables to represent a N:M relationship between EMPLOYEE and CERTIFICATE.

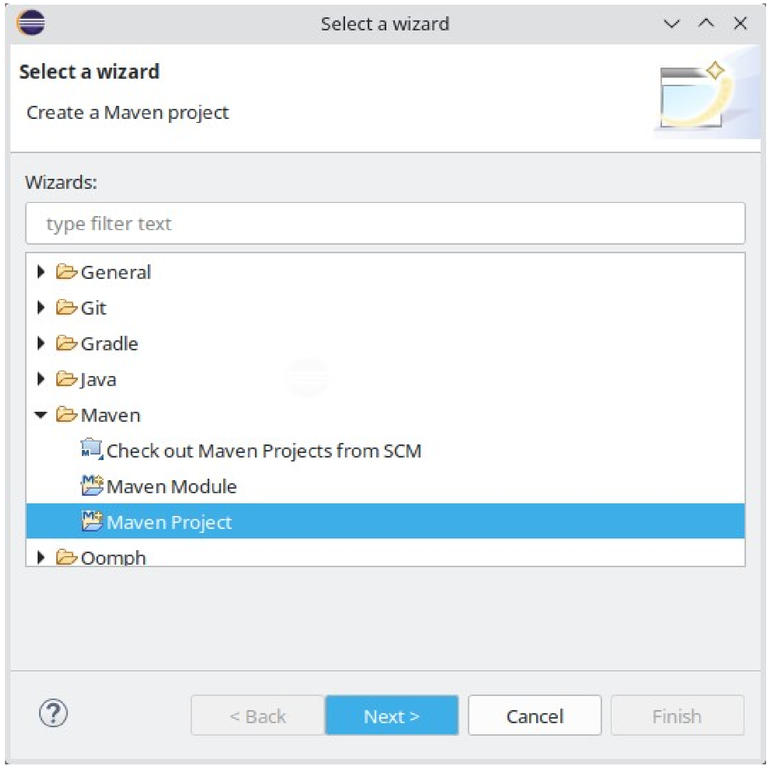
```
CREATE DATABASE ADAU3DBExample CHARACTER
SET utf8mb4 COLLATE utf8mb4_es_0900_ai_ci;

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

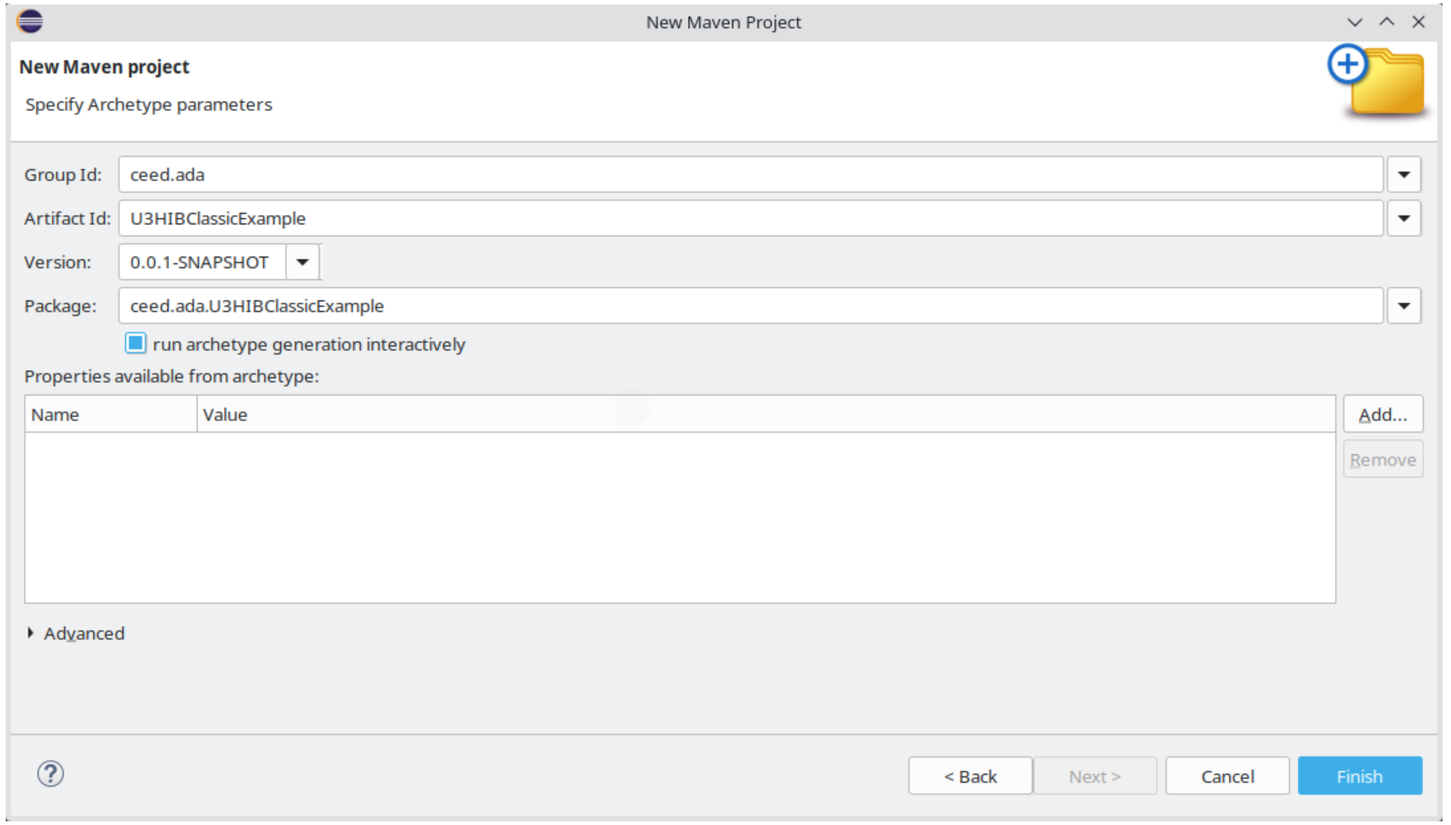



```
USE ADAU3DBExample;

CREATE TABLE Employee (
   empID       INTEGER NOT NULL AUTO_INCREMENT,
   firstname   VARCHAR(20),
   lastname    VARCHAR(20),
   salary      DOUBLE,
   CONSTRAINT emp_id_pk PRIMARY KEY (empID)
);

CREATE TABLE Certificate (
   certID      INTEGER NOT NULL AUTO_INCREMENT,
   certname    VARCHAR(30),
   CONSTRAINT cer_id_pk PRIMARY KEY (certID)
);

CREATE TABLE EmpCert (
   employeeID     INTEGER,
   certificateID  INTEGER,
   CONSTRAINT empcer_pk PRIMARY KEY (employeeID, certificateID),
   CONSTRAINT emp_id_fk FOREIGN KEY (employeeID) REFERENCES
Employee(empID),
   CONSTRAINT cer_id_fk FOREIGN KEY (certificateID) REFERENCES
Certificate(certID)
);
```

The (Java-Maven) project

- In ECLIPSE, create an empty Java Maven Project with these parameters as we saw at UNIT 2:





The (Database) connection

Add the dependencies to the Maven Project (pom.xml) for MySQL. Check **how to get your version** here: https://phoenixnap.com/kb/how-to-check-mysql-version

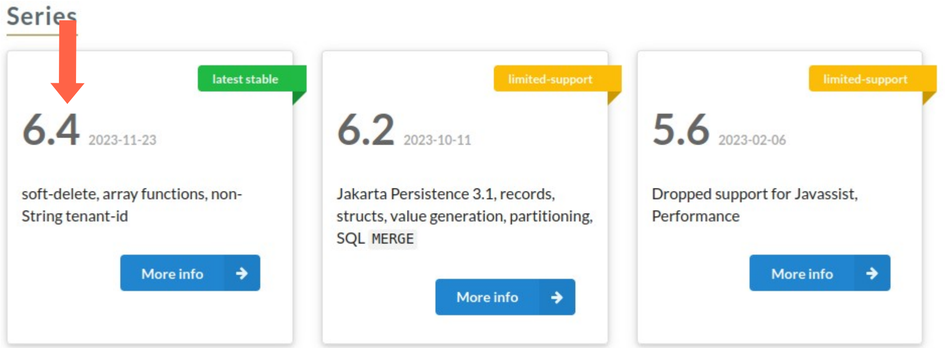
```
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc -->
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>8.0.33</version>
    </dependency>
  </dependencies>
```

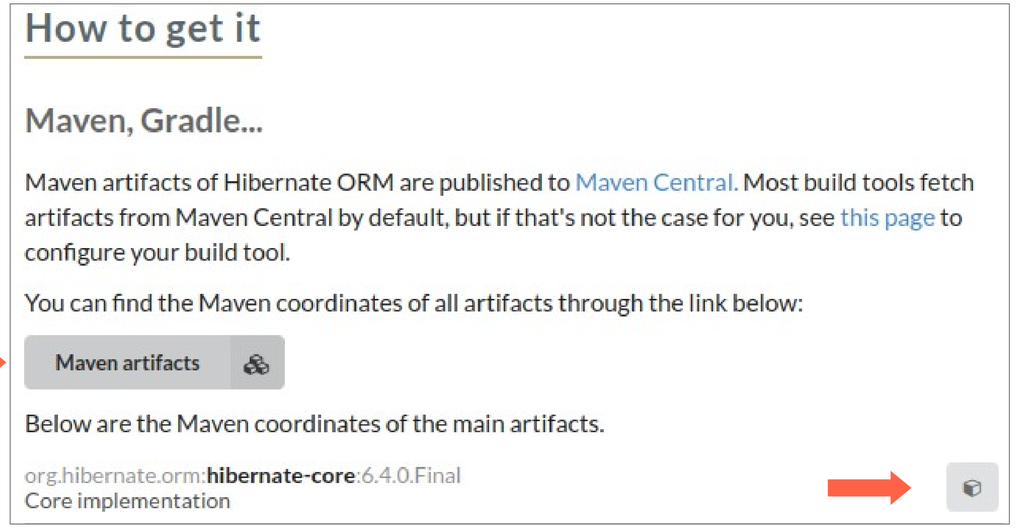
4. SETTING UP HIBERNATE

Hibernate dependencies

Follow these simple steps to download Hibernate and establish the dependencies:

- Go to https://hibernate.org/orm/releases/
- Click on “More info” on the last stable version
- Option 1:
  - Click on the link hibernate-core (x.x.x.Final)
- Option 2:
  - Click on “Maven artifacts” button
  - Click on the link hibernate-core (x.x.x.Final)
- Copy and paste the dependency to your POM
- Save and let Maven do the work!





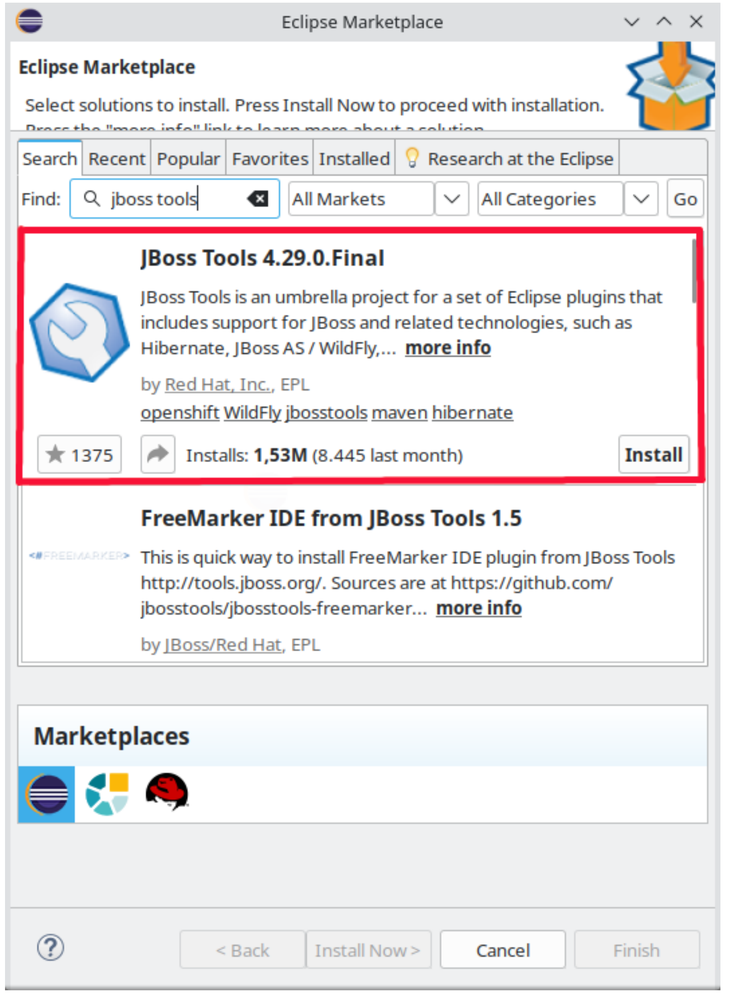

```
  <dependencies>
      [...]

   <!--
https://central.sonatype.com/artifact/org.hibernate.orm/h
ibernate-core/6.4.0.Final-->
   <dependency>
       <groupId>org.hibernate.orm</groupId>
       <artifactId>hibernate-core</artifactId>
       <version>6.4.0.Final</version>
   </dependency>
  </dependencies>
```

Set-up Hibernate automatically

Now we could set-up Hibernate in an automatic or manual way. If we choose to **set-up automatically**, we must install the **Jboss tools plugin** and follow the instructions explained in the extended materials.

Although it is most common to configure Hibernate automatically, the following slides will explain the manual procedure, as it helps to better understand the ORM mapping concepts.

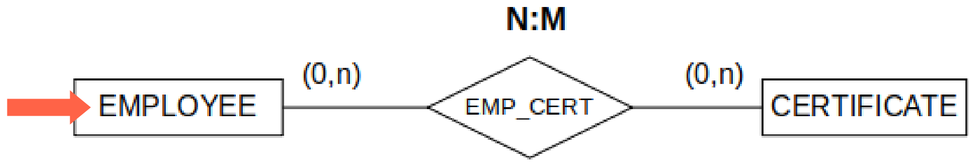
The extended materials detail how to perform the same process automatically using the “JBoss tools” plugin.

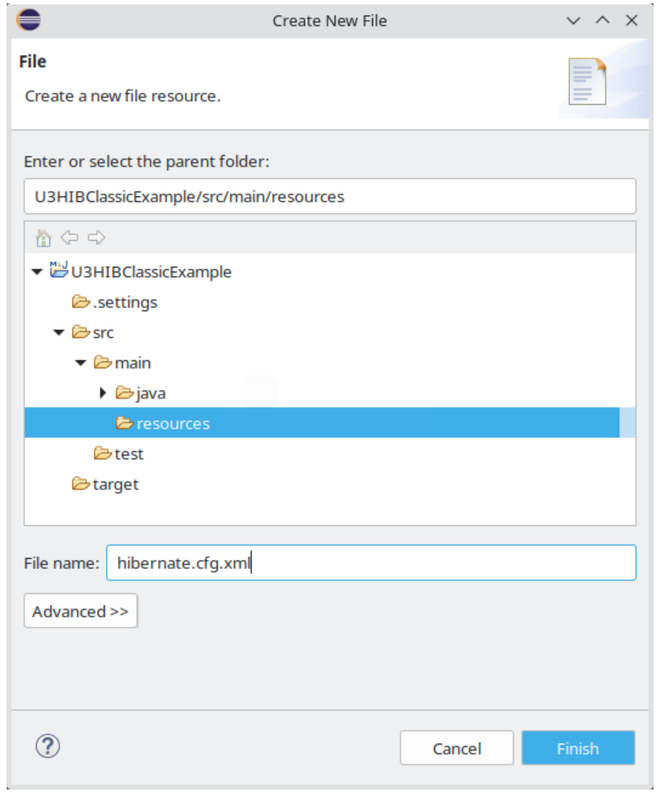


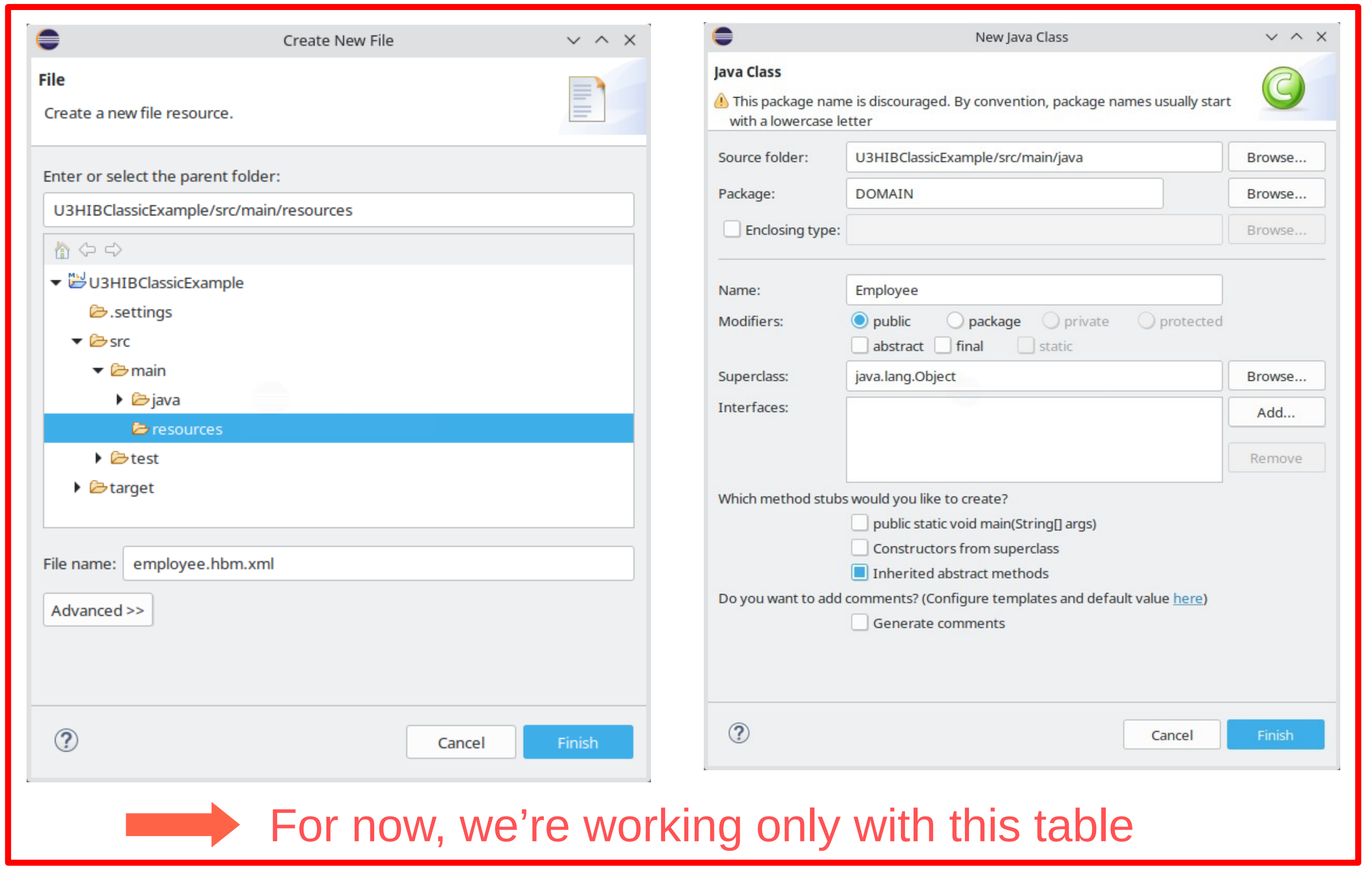
Set-up Hibernate manually

Now we need to create:

- **One generic config file**
- **One config file** + **one POJO class file** for **every table**



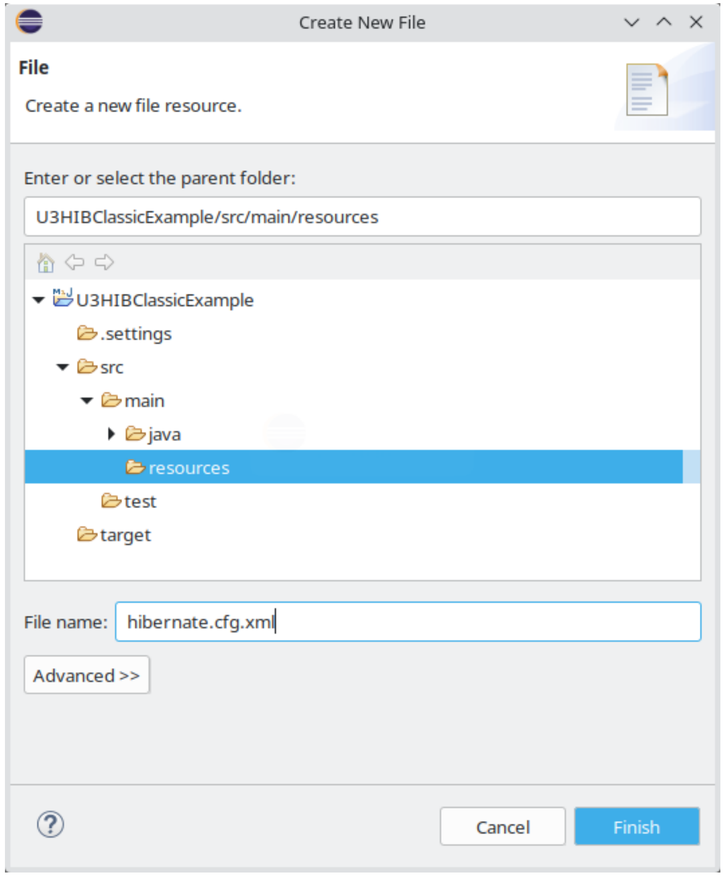




Generic config file

Follow these simple steps to set-up this 1st file:

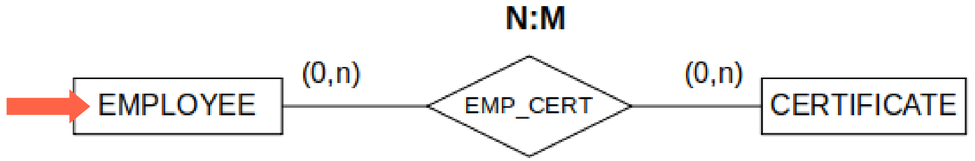
- Create a folder **resources** on **src/main**
- Create a file on **src/main/resources**
- Save it as **hibernate.cfg.xml**
- Copy and paste the code of next slide, changing the connection settings to the database:
  - Database name: **DBCertificates**
  - User: <your user> or **mavenuser**
  - Password: <your password> or **ada0486**



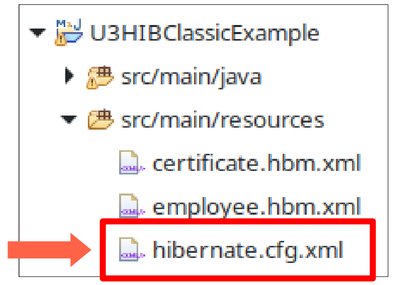
hibernate.cfg.xml

Generic config file

For now, we're working only with this table, but if we want to work with more tables, we will add more lines and more files: **one file for each table**




```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/ADAU3DBExample</property>
        <property name="hibernate.connection.username">mavenuser</property>
        <property name="hibernate.connection.password">ada0486</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
        <property name="show_sql">false</property>
        <property name="format_sql">true</property>
        <property name="hbm2ddl.auto">update</property>
        <mapping resource="employee.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

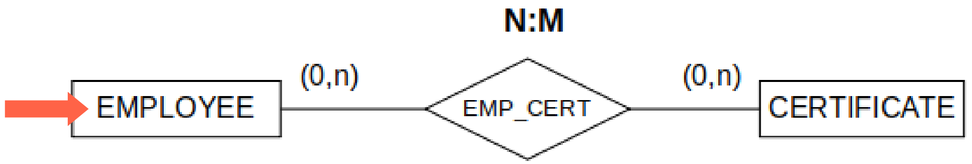


hibernate.cfg.xml

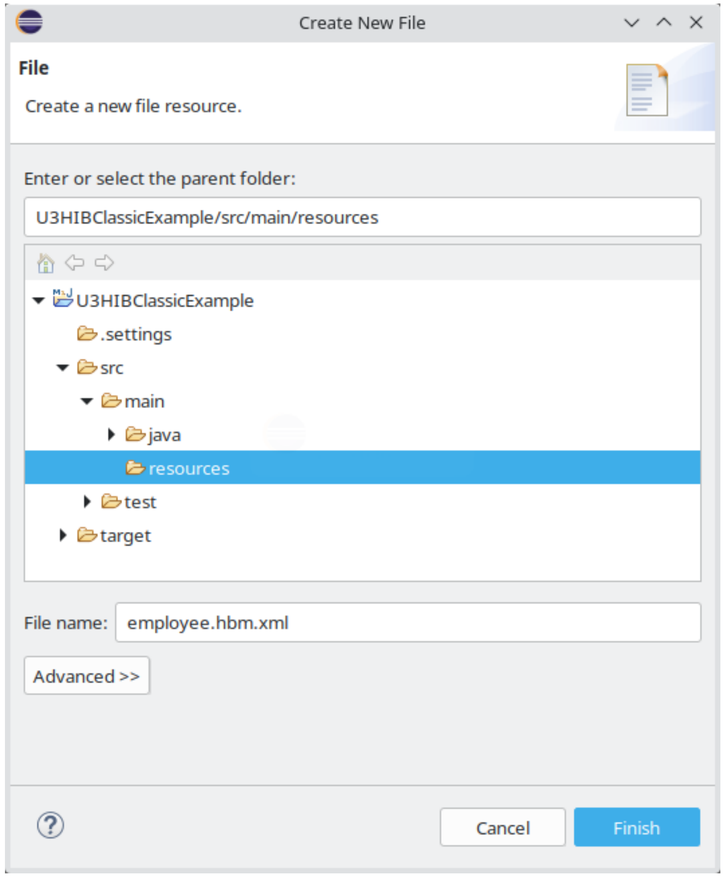
Table(s) config file

Follow these simple steps to set-up this file:

- Create a file on **src/main/resources**
- Save it as **employee.hbm.xml**
- Copy and paste the code of next slide, where you will be setting up how your table is represented in your class using DAO (via **POJO notation**).



For now, we're working only with this table



employee.hbm.xml

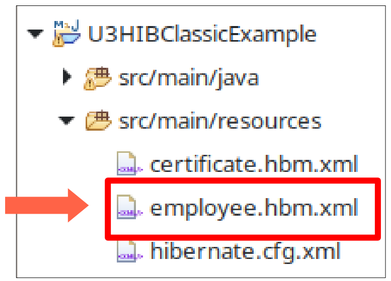
Table(s) config file

As you can see here, we are setting up this mapping file to work only with “**Employee**” table. Next step will be to create a java class (DOMAIN.Employee.java) **to map this table** according to these specifications.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name = "DOMAIN.Employee" table = "Employee">
      <meta attribute = "class-description">
         This class contains the employee detail.
      </meta>
      <id name = "iEmpID" type = "int" column = "empID">
         <generator class="native"/>
      </id>
      <property name = "stFirstName" column = "firstname" type = "string"/>
      <property name = "stLastName" column = "lastname" type = "string"/>
      <property name = "dSalary" column = "salary" type = "double"/>
   </class>
</hibernate-mapping>
```

employee.hbm.xml

The <**generator**> element within the id element is used to generate the primary key values automatically. The class attribute of the generator element is set to native to let hibernate pick up either identity, sequence or thread algorithm to create a primary key depending upon the capabilities of the underlying database.



POJO file (for every table)

Now we should create a java class (DOMAIN.Employee.java) to **map this table** according to the database specifications and POJO standards.

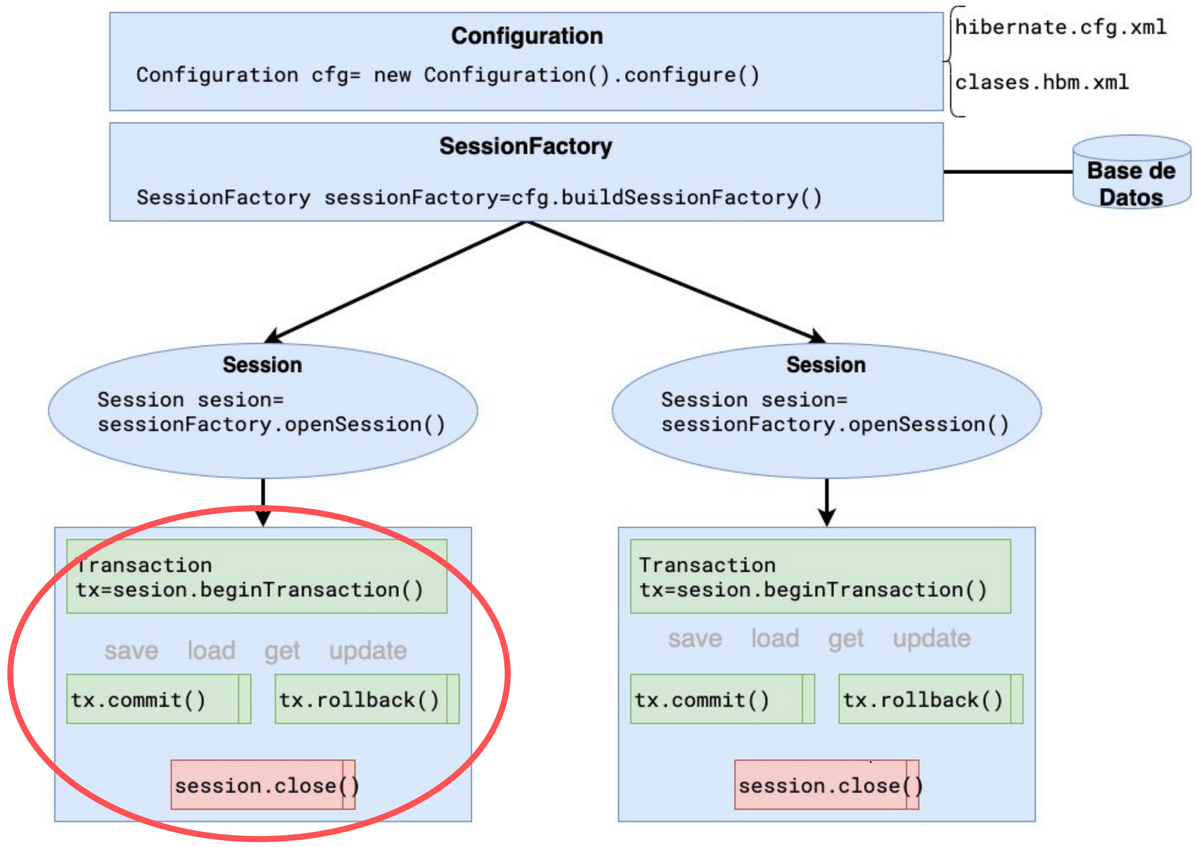
Be careful about the name of the variables. The getters and setters must follow the same criteria to allow Hibernate to find the appropriate methods: https://stackoverflow.com/questions/921239/hibernate-propertynotfoundexception-could-not-find-a-getter-for


```java
package DOMAIN;

import java.util.Set;

public class Employee {

     // ATTRIBUTES
    private int iEmpID;
    private String stFirstName;
    private String stLastName;
    private double dSalary;

     // METHODS
     //Empty constructor
    public Employee() {
    }

     // Constructor without ID. All fields, except primary key
    public Employee(String stFirstName, String stLastName, double dSalary) {
        this.stFirstName = stFirstName;
        this.stLastName = stLastName;
        this.dSalary = dSalary;
    }

     // GETTERS
    public int getiEmpID() {
        return iEmpID;
    }

    public String getstFirstName() {
        return stFirstName;
    }

    public String getstLastName() {
        return stLastName;
    }

    public double getdSalary() {
        return dSalary;
    }

     // SETTERS
    public void setiEmpID(int iempID) {
        this.iEmpID = iempID;
    }

    public void setstFirstName(String stFirstName) {
        this.stFirstName = stFirstName;
    }

    public void setstLastName(String stLastName) {
        this.stLastName = stLastName;
    }

    public void setdSalary(double dSalary) {
        this.dSalary = dSalary;
    }
}
```

5. HIBERNATE: SESSIONS

Run Hibernate via sessions

Now that we have set-up Hibernate, let's switch it on and make the final step. We'll be working with an object called FACTORY and several SESSIONS like this:



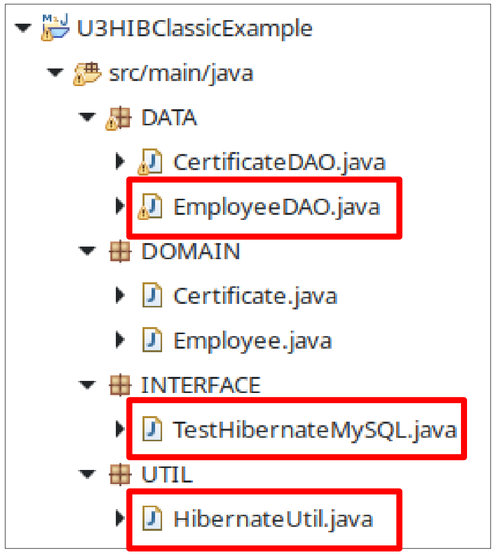

```java
// CONFIGURATION
Configuration = cfg = new Configuration().configure();

// FACTORY
SessionFactory sessionFactory = cfg.buildSessionFactory();

// SESSIONS
Session session = sessionFactory.openSession();
session.beginTransaction();
   Employee emp = new Employee();
   emp.setEmpId(1);
   emp.setEmpName("Sergio");
   …
session.persist(emp); // insert emp
session.getTransaction().commit();
session.close();
```

Let's put down this idea into three more files:

- **TestHibernateMySQL**.java (with main method)
- **HibernateUtil**.java (to wake up Hibernate)
- **EmployeeDAO**.java (with CRUD methods)

Three more files to go

1) DAO file with all the methods to interact with the database via Hibernate.
2) A new “test” class file with the main method to run our project.
3) **A class file to wake up the Hibernate process and put it to sleep**.

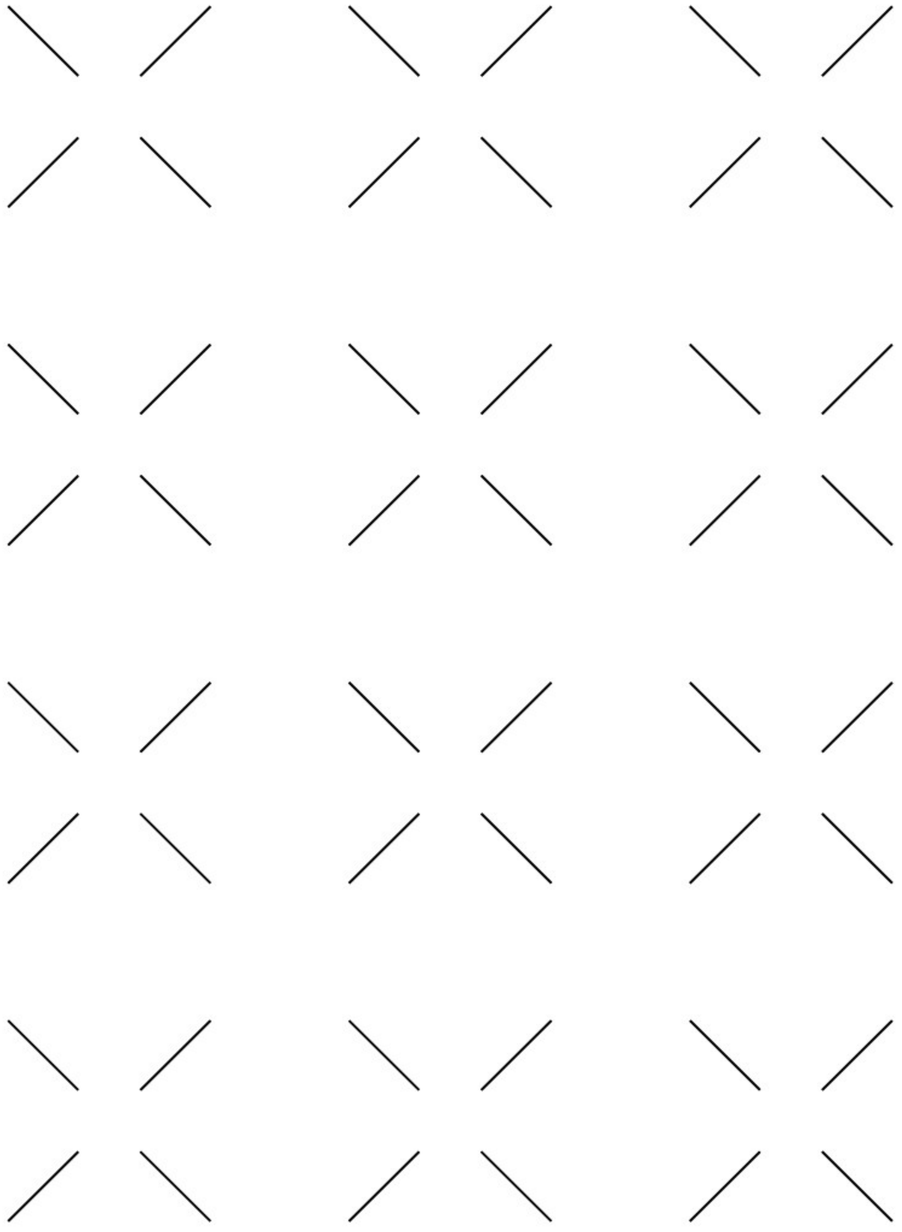


These are the files we need:

- Maven configuration files: **pom.xml**
- Hibernate configuration files: **hibernate.cfg.xml**
- A class file to switch on/off Hibernate: **HibernateUtil.java**
- A “dummy” interface class file with the main method: **TestHibernateMySQL.java**
- For every table/resource:
  - Data layer file: **EmployeeDAO.java**
  - Domain related file: **Employee.java**
  - Config table file: **employee.hbm.xml**

6. HIBERNATE: SINGLE-TABLE MAPPING

6.1 DAO file



DAO file. CRUD operations. Create

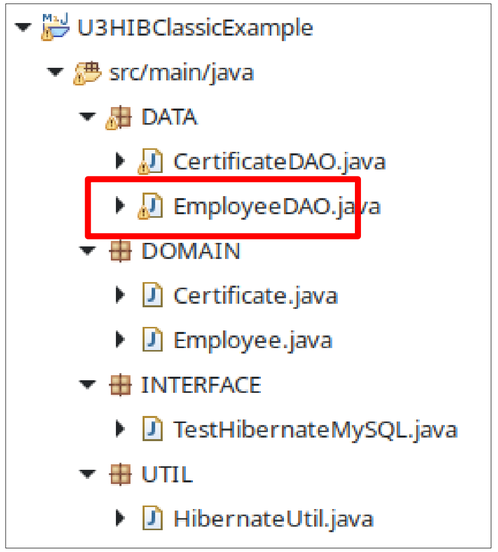
```java
package DATA;

import java.util.List;
import java.util.Set;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;

import DOMAIN.*;
import UTIL.*;

public class EmployeeDAO {
    /*
     * ------------------
     * INSERT
     * ------------------
     */
    /* Method to CREATE an employee in the database */
    public Employee addEmployee(String stFirstName, String stLastName, double dSalary) {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibernate session factory
        Transaction txDB = null; //database transaction
        Employee objEmployee = new Employee(stFirstName, stLastName, dSalary);

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            hibSession.persist(objEmployee);
            txDB.commit(); //ends transaction
            System.out.println("***** Item added.\n");
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
                hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
        return objEmployee;
    }
```

DAO file. CRUD operations. Read

```java
/*
 * ------------------
 * SELECT
 * ------------------
 */
/* Method to READ all the employees */
public void listEmployees( ) {

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

   try {
      txDB = hibSession.beginTransaction( ); //starts transaction
      //old createQuery method is deprecated, but still working in the latest version
      //https://www.roseindia.net/hibernate/hibernate5/hibernate-5-query-deprecated.shtml
      List<Employee> lstEmployees = hibSession.createQuery("FROM Employee", Employee.class).list( );
      if (lstEmployees.isEmpty( ))
         System.out.println("******** No items found");
      else
         System.out.println("\n***** Start listing ...\n");

      for (Iterator<Employee> itEmployee = lstEmployees.iterator( ); itEmployee.hasNext( );) {
         Employee objEmployee = (Employee) itEmployee.next( );
         System.out.print("First Name: " + objEmployee.getstFirstName( ) + " | ");
         System.out.print("Last Name: " + objEmployee.getstLastName( ) + " | ");
         System.out.println("Salary: " + objEmployee.getdSalary( ));
      }
      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
   }
}
```

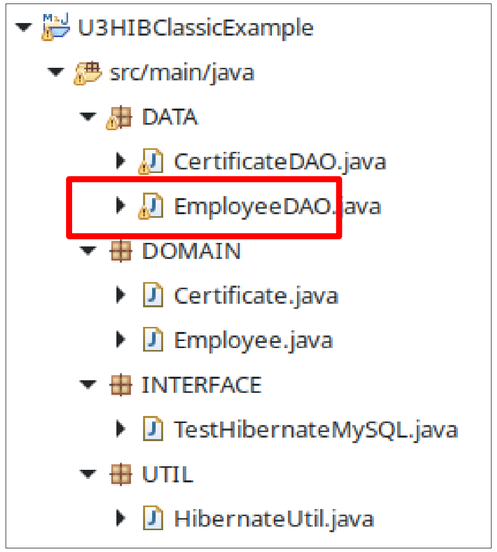
DAO file. CRUD operations. Update

```
   /*
    * ------------------
    * UPDATE
    * ------------------
    */
   /* Method to UPDATE salary for an employee */
   public void updateEmployee(int iEmpID, double dSalary) {

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

      try {
         txDB = hibSession.beginTransaction(); //starts transaction
         Employee objEmployee = (Employee) hibSession.get(Employee.class, iEmpID);
         objEmployee.setdSalary(dSalary);
         hibSession.merge(objEmployee);
         txDB.commit(); //ends transaction
         System.out.println("***** Item updated.\n");
      } catch (HibernateException hibe) {
         if (txDB != null)
            txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
      } finally {
         hibSession.close(); //close hibernate session
      }
   }
```

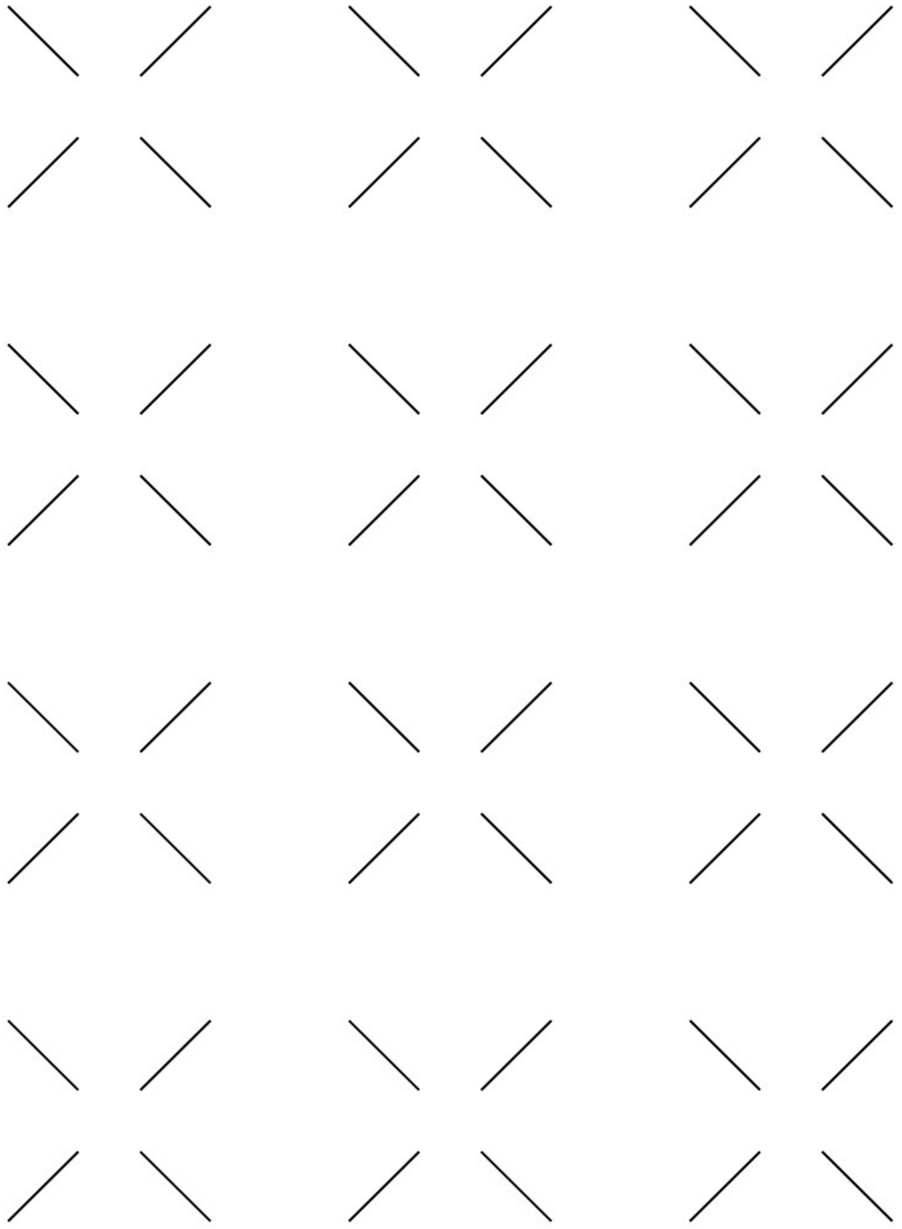


DAO file. CRUD operations. Delete

```
   /*
    * ------------------
    * DELETE
    * ------------------
    */
   /* Method to DELETE an employee from the records */
   public void deleteEmployee(int iEmpID) {

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

      try {
         txDB = hibSession.beginTransaction(); //starts transaction
         Employee objEmployee = (Employee) hibSession.get(Employee.class, iEmpID);
         hibSession.remove(objEmployee);
         txDB.commit(); //ends transaction
         System.out.println("***** Item deleted.\n");
      } catch (HibernateException hibe) {
         if (txDB != null)
            txDB.rollback(); //something went wrong, so rollback
            hibe.printStackTrace();
      } finally {
         hibSession.close(); //close hibernate session
      }
   }
```

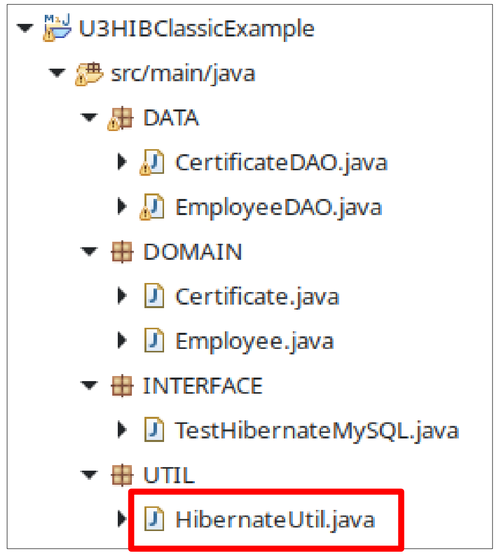


6.2 Util Hibernate



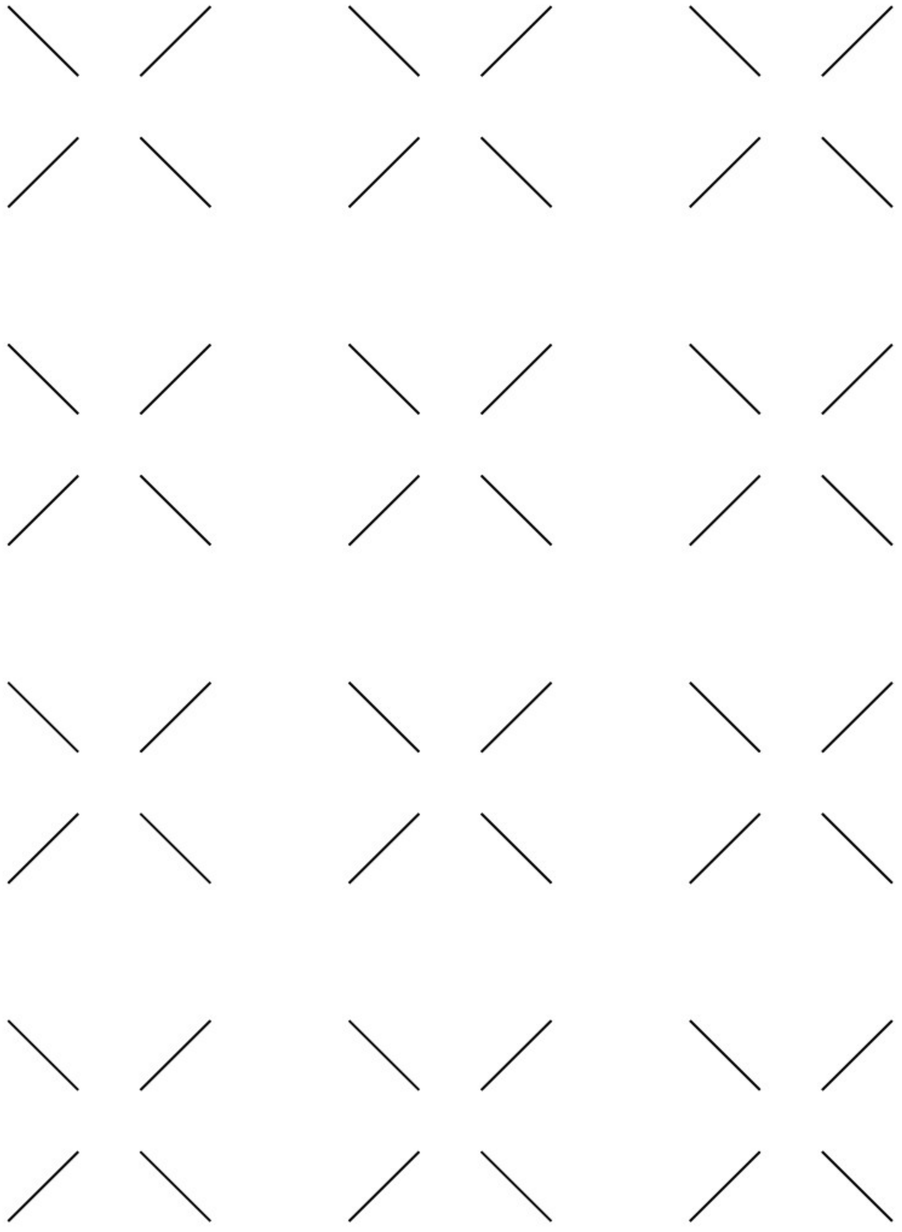
Util Hibernate

Finally, we need a **class to wake up the Hibernate process and put it to sleep**.



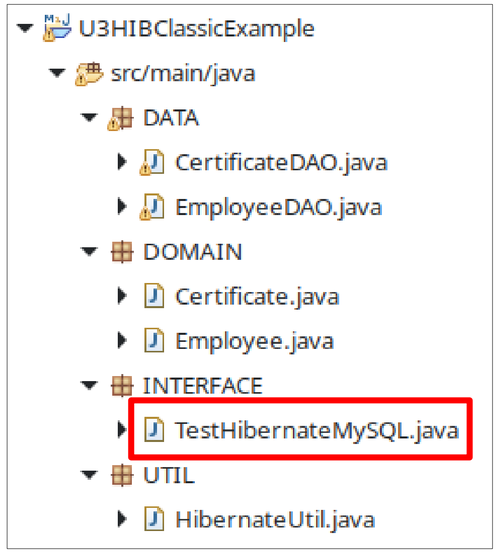

```java
package UTIL;

import java.util.logging.Level;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

   //Persistent session
   public static final SessionFactory SFACTORY = buildSessionFactory();


    // SESSION MANAGEMENT
   /*
    * Create new hibernate session
    */
   private static SessionFactory buildSessionFactory() {
      java.util.logging.Logger.getLogger("org.hibernate").setLevel(Level.OFF);
      try {
         // Create the SessionFactory from hibernate.cfg.xml
         return new Configuration().configure().buildSessionFactory();
      } catch (Throwable sfe) {
         // Make sure you log the exception, as it might be swallowed
         System.err.println("SessionFactory creation failed." + sfe);
         throw new ExceptionInInitializerError(sfe);
      }
   }

   /*
    * Close hibernate session
    */
   public static void shutdownSessionFactory() {
      // Close caches and connection pools
      getSessionFactory().close();
   }

   /*
    * Get method to obtain the session
    */
   public static SessionFactory getSessionFactory() {
      return SFACTORY;
   }
}
```

6.3 Test Hibernate



TestHibernate. Interface Layer

And the **main programme**:



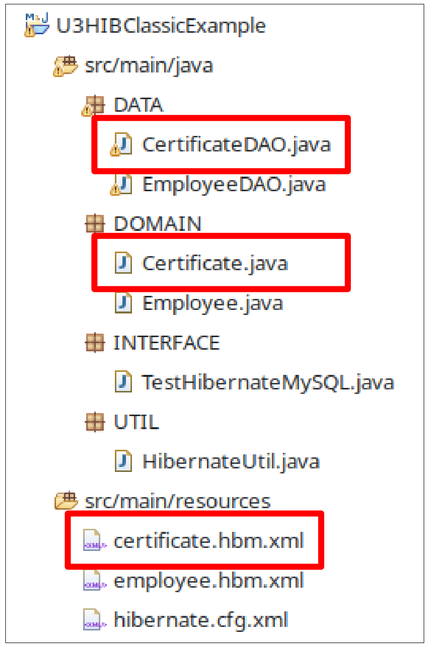

```java
import DATA.*;
import DOMAIN.*;
import UTIL.*;

public class TestHibernateMySQL {

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

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

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

      /* Add records in the database */
      Employee objEmp1 = objEmployeeDAO.addEmployee("Alfred", "Vincent", 4000);
      Employee objEmp2 = objEmployeeDAO.addEmployee("John", "Gordon", 3000);

      /* Update employee's salary field */
      objEmployeeDAO.updateEmployee(objEmp1.getiEmpID(), 5000);
      /* List down all the employees */
      objEmployeeDAO.listEmployees();
      /* Delete an employee from the database */
      objEmployeeDAO.deleteEmployee(objEmp2.getiEmpID());
      /* List down all the employees */
      objEmployeeDAO.listEmployees();

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

7. HIBERNATE: ADVANCED MAPPINGS

Adding more resources (tables)

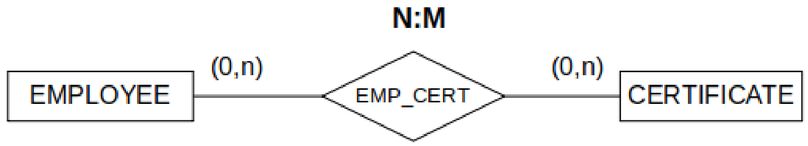
Once we have woken up the Hibernate process to work with a single table we can go further and add more resources following these easy steps:

1)Create its DAO file (**DATA/TableDAO.java**)
2)Create its POJO file (**DOMAIN/Table.java**)
3)Modify the config (mapping) file of the table related to (**table.hbm.xml**)
4)Add the resource to Hibernate config generic file (**hibernate.cfg.xml**)



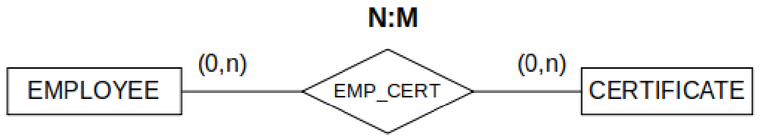
Note that placing files into packages inside src/main/java is TOTALLY OPTIONAL but helps us making it easier to undersand.

Have a look at the new files added to **handle this two entities (Employee and Certificate)**. Bear in mind Hibernate will handle the relations one-to-many, many-to-many, etc., so you only have to set-up the ENTITIES, **never middle tables such us EmpCert**.



Setting many-to-many (N:M)

To configure this type of relationship we need to make this workaround:

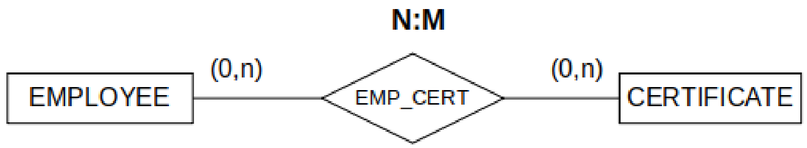


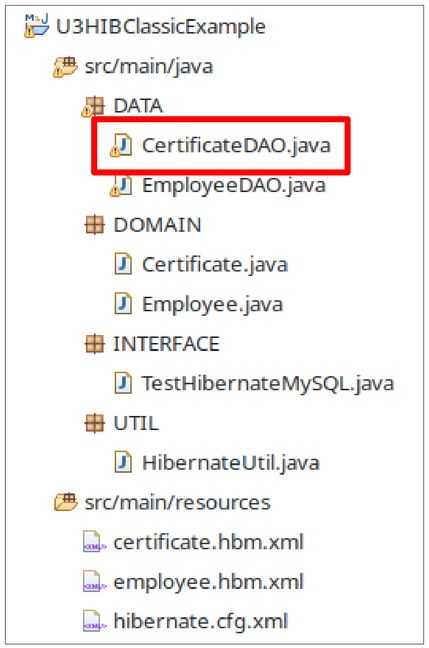
1)Create its DAO file (**DATA/CertificateDAO.java**)
2)Create its POJO file (**DOMAIN/Certificate.java**)
3)Create methods at the other side of the relationship **equals**() and **hashCode**() so that Java can determine whether any two elements/objects are identical.
    For instance, we can do so at right side: **DOMAIN/Certificate.java**
4)Create its config (mapping) file (**certificate.hbm.xml**)
5)Add the resource to Hibernate config generic file (**hibernate.cfg.xml**)
6)Create a new field at one side of the relationship (left side) and place a SET field there to store "**as many as items of the other side**" we need, with its setters and getters
    For instance, we can do so at left side: **DOMAIN/Employee.java**
7)Modify DAO file to insert a set of items of one side when adding items of the other and list all related items.
    For instance, we can do so at left side: **DATA/EmployeeDAO.java**
8)Add a new SET field mapping to the left side (**src/main/resources/Employee.hbm.xml**)

Setting many-to-many. Step 1

1) Create its DAO file (**DATA/CertificateDAO.java**)

All methods are almost identical to data.EmployeeDAO.java





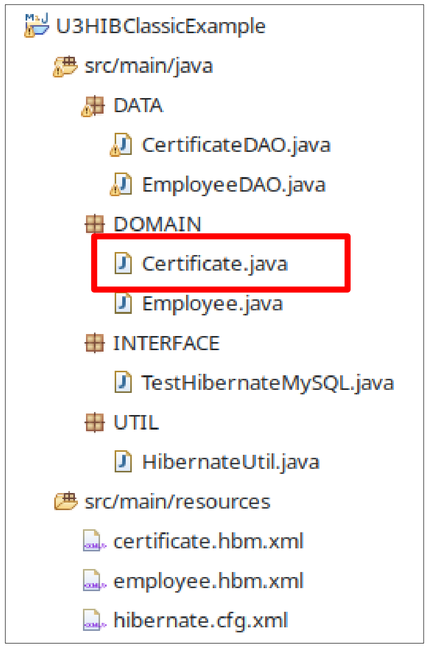

```java
import java.util.List;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;

import DOMAIN.*;
import UTIL.*;

public class CertificateDAO {

    // INSERT
   /* Method to CREATE a certificate in the database */
   public Certificate addCertificate(String stCertName) {
   }

    // SELECT
   /* Method to READ all the certificates */
   public void listCertificates( ) {
   }

    // UPDATE
   /* Method to UPDATE name for a certificate */
   public void updateCertificate(int iCertID, String stCertName) {
   }

    // DELETE
   /* Method to DELETE a certificate from the records */
   public void deleteCertificate(int iCertID) {
   }

   /* Method to DELETE all records */
   public void deleteAllItems( ) {
   }

}
```

Setting many-to-many. Step 2 & 3

2) Create its POJO file (**DOMAIN/Certificate.java**)

3) Create methods at the other side of the relationship **equals**() and **hashCode**() so that Java can determine whether any two elements/objects are identical.

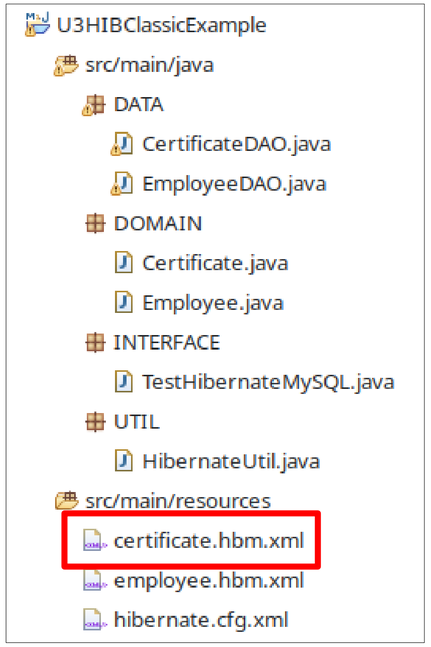
For instance, we can do so at right side: **DOMAIN/Certificate.java**




```java
package DOMAIN;

public class Certificate {

     // ATTRIBUTES
    private int iCertID;
    private String stCertName;

     // METHODS

     //Empty constructor
    public Certificate() {
    }
     //Constructor without ID. All fields, except primary key
    public Certificate(String stCertName) {
        this.stCertName = stCertName;
    }

     // GETTERS
    public int getiCertID() {
        return iCertID;
    }

    public String getstCertName() {
        return stCertName;
    }

     // SETTERS
    public void setiCertID(int iCertID) {
        this.iCertID = iCertID;
    }

    public void setstCertName(String stCertName) {
        this.stCertName = stCertName;
    }
    /*
     * Set MANY-TO-MANY relationship between Employee and Certificate
     * We choose right side (table Certificate)
     * Create methods equals() and hashCode() so that Java can determine whether any two
elements/objects are identical
     */
    public boolean equals(Object obj) {
        if (obj == null)
            return false;
        if (!this.getClass().equals(obj.getClass()))
            return false;

        Certificate objCert = (Certificate) obj;
        if ((this.iCertID == objCert.getiCertID()) &&
(this.iCertName.equals(objCert.getstCertName()))) {
            return true;
        }
        return false;
    }

    public int hashCode() {
        int iHash = 0;
        iHash = (iCertID + stCertName).hashCode();
        return iHash;
    }

}
```

Setting many-to-many. Step 4

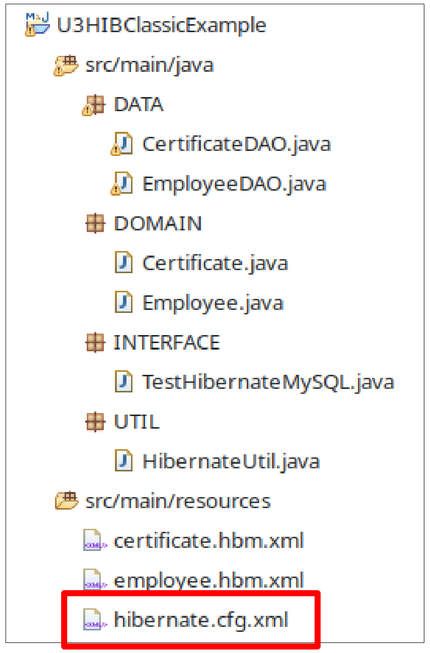
4) Create its config (mapping) file (**certificate.hbm.xml**)




```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name = "DOMAIN.Certificate" table = "Certificate">
      <meta attribute = "class-description">
         This class contains the certificate detail.
      </meta>
      <id name = "iCertID" type = "int" column = "certID">
         <generator class="native"/>
      </id>
      <property name = "stCertName" column = "certname" type = "string"/>
   </class>
</hibernate-mapping>
```

Setting many-to-many. Step 5

5) Add the resource to Hibernate config generic file (**hibernate.cfg.xml**)

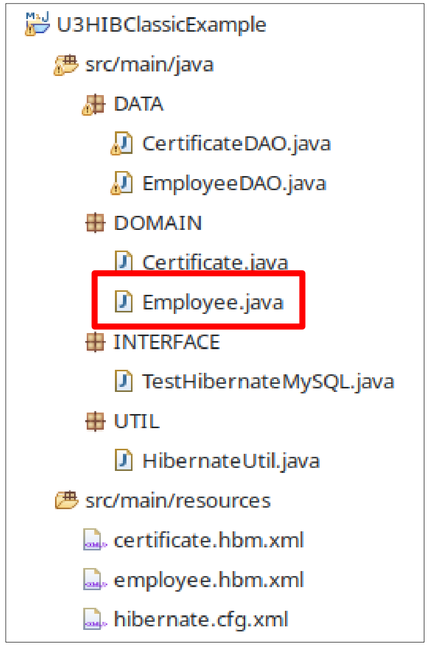



```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/DBCertificates</property>
        <property name="hibernate.connection.username">mavenuser</property>
        <property name="hibernate.connection.password">ada0486</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
        <property name="show_sql">false</property>
        <property name="format_sql">true</property>
        <property name="hbm2ddl.auto">update</property>
        <mapping resource="employee.hbm.xml" />
        <mapping resource="certificate.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Setting many-to-many. Step 6

6) Create a new field at one side of the relationship (left side) and place a SET field there to store “**as many as items of the other side**” we need, with its setters and getters

For instance, we can do so at left side: **DOMAIN/Employee.java**

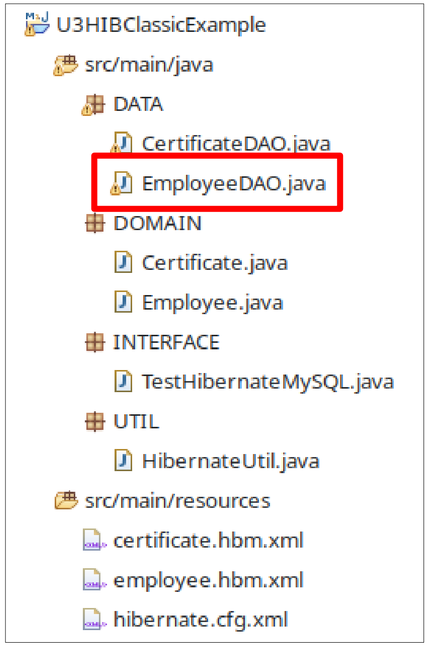



```java
package DOMAIN;

import java.util.Set;

public class Employee {

    // ATTRIBUTES
    private int iEmpID;
    private String stFirstName;
    private String stLastName;
    private double dSalary;
    //Set class in Java https://www.geeksforgeeks.org/set-in-java/
    private Set<Certificate> relCertificates; //relationship Employee-Certificate (N:M)

    // METHODS

    //Empty constructor
    public Employee() {
    }
    //Constructor without ID. All fields, except primary key
    public Employee(String stFirstName, String stLastName, double dSalary) {
        this.stFirstName = stFirstName;
        this.stLastName = stLastName;
        this.dSalary = dSalary;
    }
            [...]

    // GETTERS

            [...]

    public Set<Certificate> getrelCertificates() {
        return relCertificates;
    }

    // SETTERS

            [...]

    public void setrelCertificates(Set<Certificate> relCertificates) {
        this.relCertificates = relCertificates;
    }

}
```

Setting many-to-many. Step 7

7) **Modify DAO** file to insert a set of items of one side when adding items of the other and list all related items.

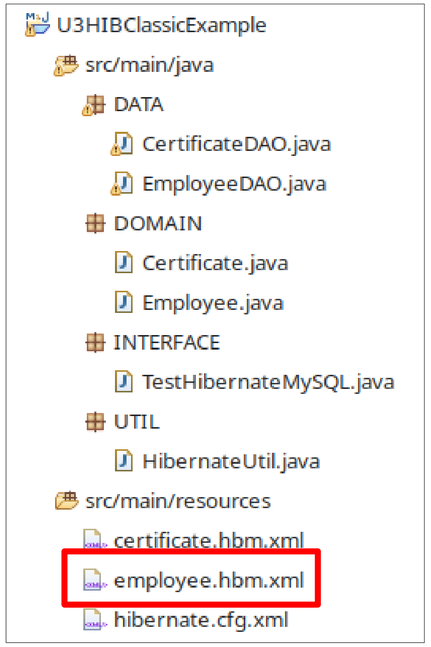
For instance, we can do so at left side: **DATA/EmployeeDAO.java**




```java
public class EmployeeDAO {

     // INSERT
    /* Method to CREATE an employee in the database */
    public Employee addEmployee(String stFirstName, String stLastName, double dSalary, Set<Certificate>
setCertificates) {

        Session hibSession = HibernateUtil.SFACTORY.openSession(); //open hibernate session factory
        Transaction txDB = null; //database transaction
        Employee objEmployee = new Employee(stFirstName, stLastName, dSalary);

        try {
            txDB = hibSession.beginTransaction(); //starts transaction
            objEmployee.setrelCertificates(setCertificates);
            hibSession.persist(objEmployee);
            txDB.commit(); //ends transaction
            System.out.println("***** Item added.\n");
        } catch (HibernateException hibe) {
            if (txDB != null)
                txDB.rollback(); //something went wrong, so rollback
                hibe.printStackTrace();
        } finally {
            hibSession.close(); //close hibernate session
        }
        return objEmployee;
    }

     // SELECT
    /* Method to READ all the employees */
    public void listEmployees( ) {

        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 Employee", 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("First Name: " + objEmployee.getstFirstName() + " | ");
                System.out.print("Last Name: " + objEmployee.getstLastName() + " | ");
                System.out.println("Salary: " + objEmployee.getdSalary());
                Set<Certificate> setCertificates = objEmployee.getrelCertificates();
                for (Iterator<Certificate> itCertificate = setCertificates.iterator(); itCertificate.hasNext();) {
                    Certificate objCertificate = (Certificate) itCertificate.next();
                    System.out.println("Certificate: " + objCertificate.getstCertName());
                }
            }
            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
        }
    }
```

Setting many-to-many. Step 8

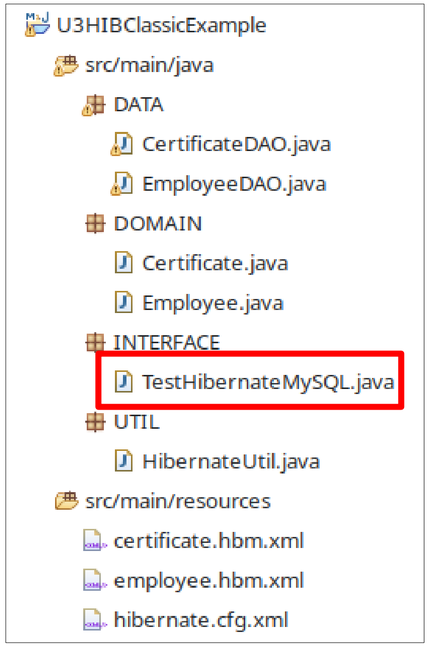
8) Add a new SET field mapping to the left side (**src/main/resources/Employee.hbm.xml**)




```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name = "DOMAIN.Employee" table = "Employee">
      <meta attribute = "class-description">
         This class contains the employee detail.
      </meta>
      <id name = "iEmpID" type = "int" column = "empID">
         <generator class="native"/>
      </id>
       <!--  https://www.tutorialspoint.com/hibernate/hibernate_many_to_many_mapping.htm -->
       <set name = "relCertificates" cascade="save-update" table="EmpCert">
         <key column = "employeeID"/>
         <many-to-many column = "certificateID" class="DOMAIN.Certificate"/>
      </set>

      <property name = "stFirstName" column = "firstname" type = "string"/>
      <property name = "stLastName" column = "lastname" type = "string"/>
      <property name = "dSalary" column = "salary" type = "double"/>
   </class>
</hibernate-mapping>
```

TestHibernate

And the **main programme**:




```java
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 records 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 records in the database */
      Employee objEmp1 = objEmployeeDAO.addEmployee("Alfred", "Vincent", 4000, hsetCertificates);
      Employee objEmp2 = objEmployeeDAO.addEmployee("John", "Gordon", 3000, hsetCertificates);

      /* Update employee's salary field */
      objEmployeeDAO.updateEmployee(objEmp1.getiEmpID(), 5000);
      /* List down all the employees */
      objEmployeeDAO.listEmployees();
      /* Delete an employee from the database */
      objEmployeeDAO.deleteEmployee(objEmp2.getiEmpID());
      /* List down all the certificates */
      objCertificateDAO.listCertificates();
      /* List down all the employees */
      objEmployeeDAO.listEmployees();

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

8. ACTIVITIES FOR NEXT WEEK

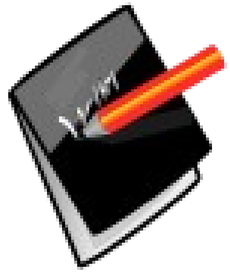
Proposed activities

Check the suggested exercises you will find at the “Aula Virtual”. **These activities are optional and non-assessable but** understanding these non-assessable activities is essential to solve the assessable task ahead.

Shortly you will find the proposed solutions.

9. BIBLIOGRAPHY

Resources



- Tutorialspoint. Hibernate tutorial. https://www.tutorialspoint.com/hibernate/index.htm
- An Introduction to Hibernate 6. https://docs.jboss.org/hibernate/orm/6.3/introduction/html_single/Hibernate_Introduction.html#queries
- Hibernate ORM 6.0.0.CR1 User Guide. https://docs.jboss.org/hibernate/orm/6.0/userguide/html_single/Hibernate_User_Guide.html#pc
- Hibernate: save, persist, update, merge, saveOrUpdate. https://docs.jboss.org/hibernate/orm/6.3/introduction/html_single/Hibernate_Introduction.html#queries
- 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).