# **Guided LAB 305.5.1B - Demonstration to ManyToOne and OneToMany Relationships and Mapping with IntelliJ Ultimate**

# 

## **Lab Overview:**

## In this lab, we will explore and demonstrate the **OnetoMany** relationship and the **ManytoOne** relationship in Hibernate. This demonstration consists of the following files:

## **Model classes: Department.java and Teacher.java**

## **Hibernate XML configuration file: hibernate.cfg.xml**

## **Run program: App.java**

## **Maven project: pom.xml**

## **Learning Objectives:**

By the end of this lab, learners will be able to:

* Describe the One-to-Many relationship in Hibernate.
* Describe the Many-to-One relationship in Hibernate.
* Utilize the One-to-Many and Many-to-One relationship in the Java applications.

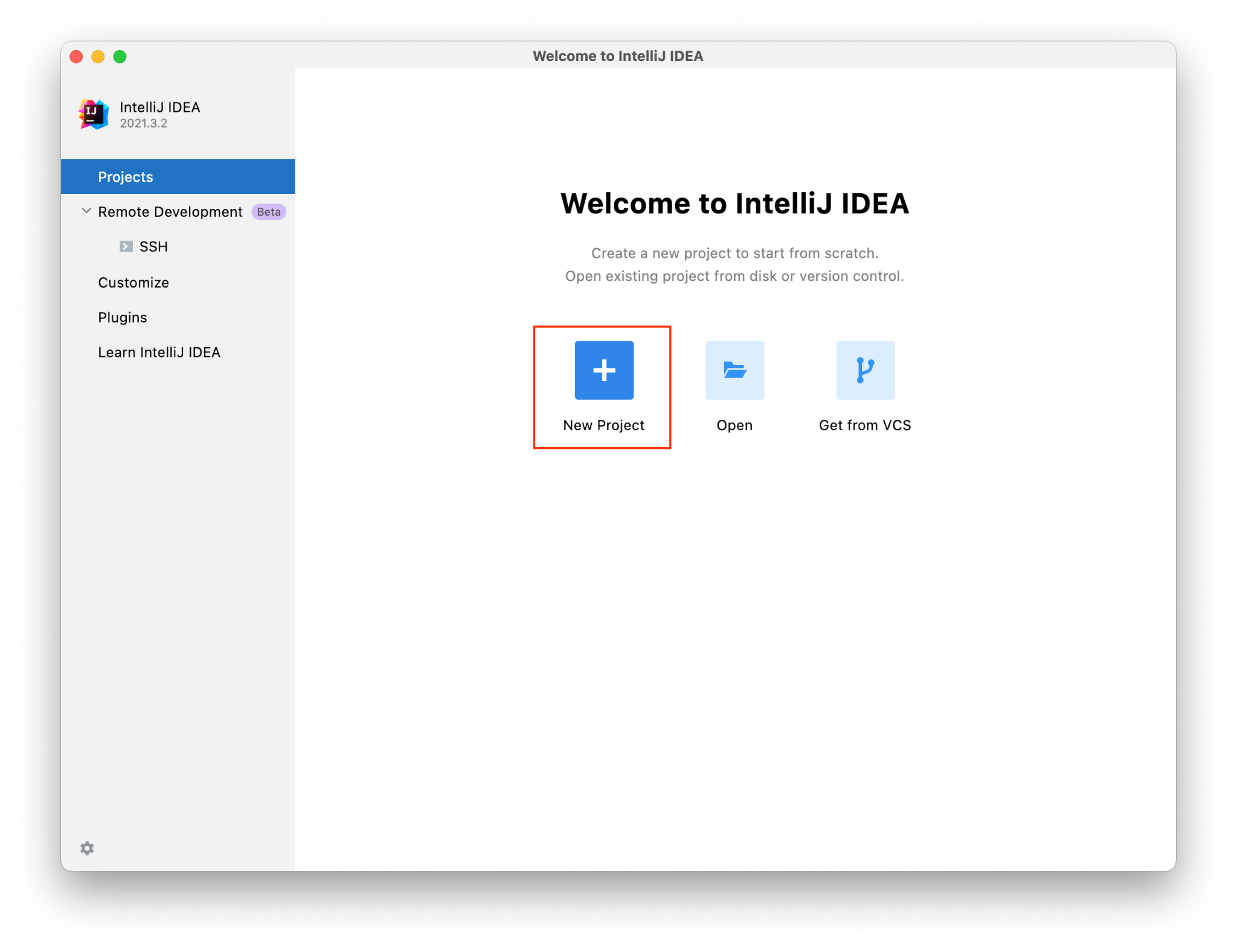
## **Example: @ManyToOne Relationship and Mapping**

**Scenario:**

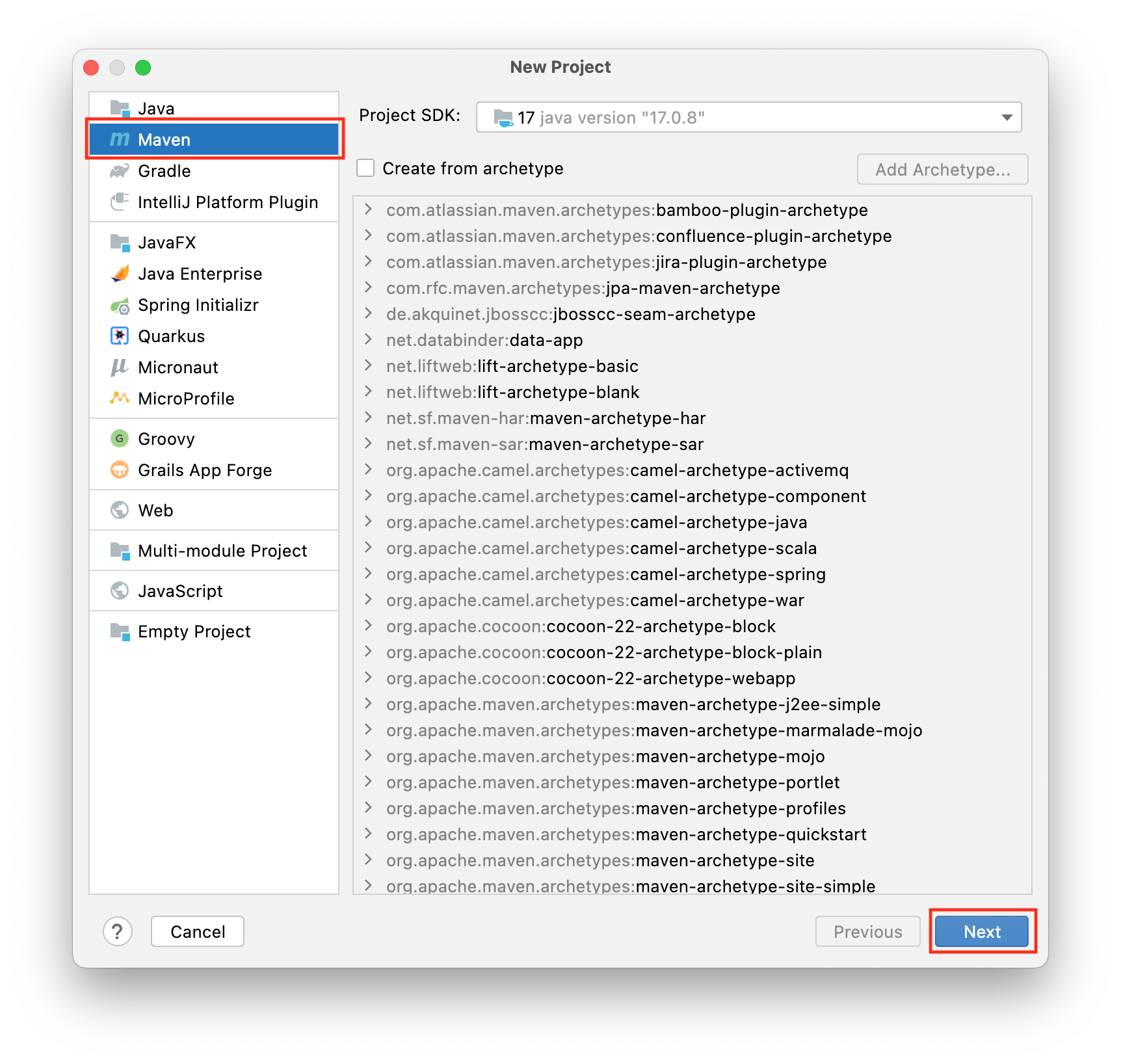
Let us consider an example of a relationship between a **Teacher** anda **department** entity. A Department consists of multiple Teachers, but each Teacher belongs to only one Department. That is a typical example of a **many-to-one** relationship or association. We will model this in a database where we need to store the [primary key](https://thorben-janssen.com/jpa-generate-primary-keys/) of the ***Department*** record as a foreign key in the ***Teacher*** table.

### **Step 1: Setup Java Maven Project and Add Jar Dependencies**

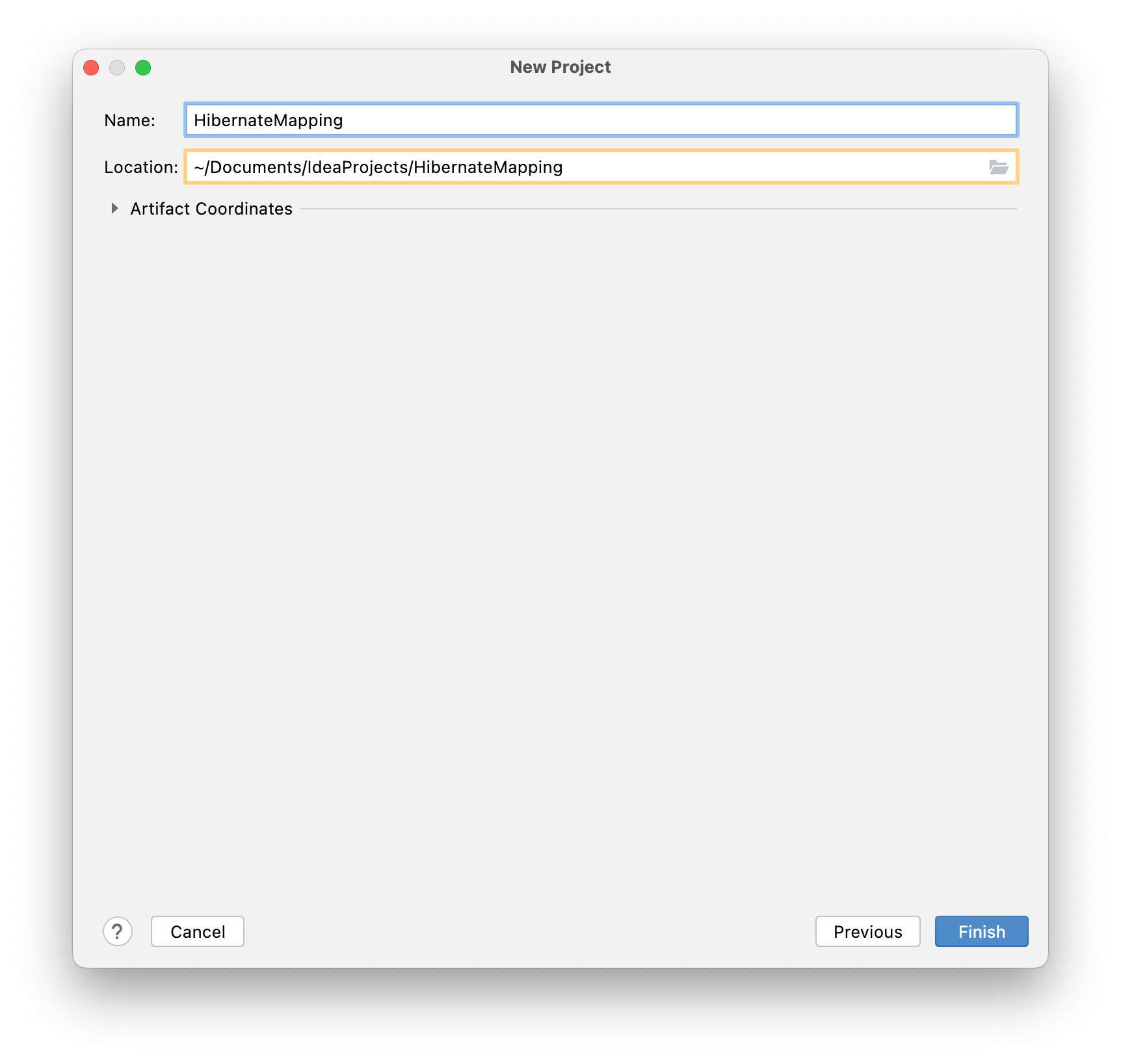
Open IntelliJ and click on the new project button .

******

Select Maven and click on Next.

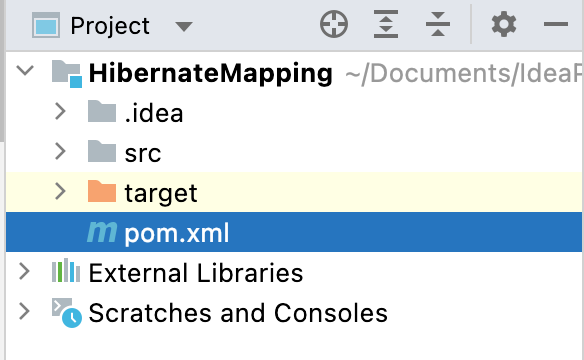


Name your project **HibernateMapping** and click Finish

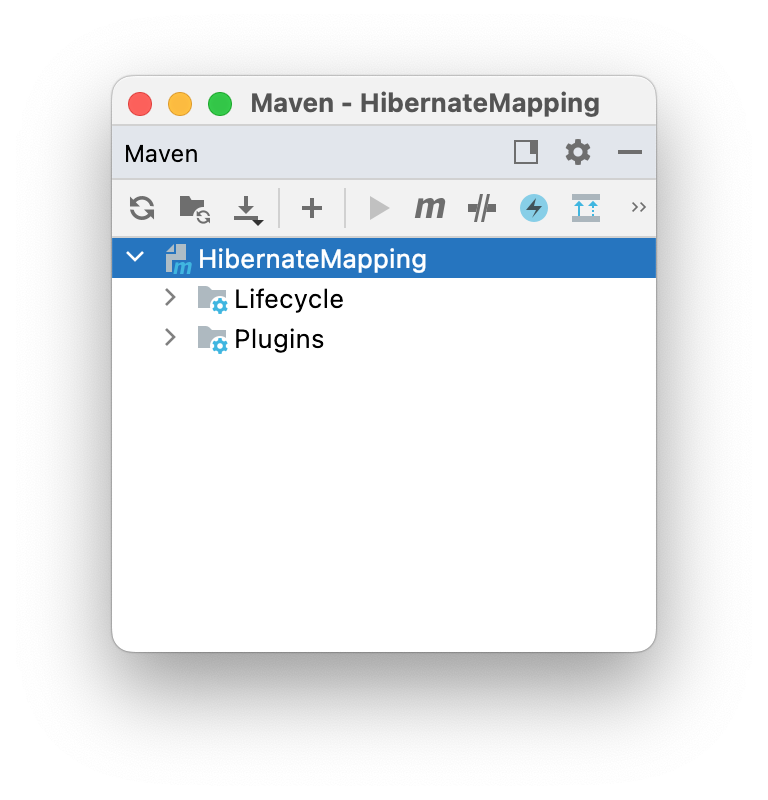


### **2) Add Jar dependencies, and configuration in the pom.xml file.**

Next, we need to add a couple of jar dependencies for Hibernate, JPA, and MySQL Connector. In the pom.xml file of our Maven Project, open the pom.xml file and insert the following code:

****

| | *<?*xml version="1.0" encoding="UTF-8"*?>*  <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>org.example</groupId>  <artifactId>HibernateMapping</artifactId>  <version>1.0-SNAPSHOT</version>  <properties>  <maven.compiler.source>17</maven.compiler.source>  <maven.compiler.target>17</maven.compiler.target>  </properties>  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.13.2</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.hibernate.orm</groupId>  <artifactId>hibernate-core</artifactId>  <version>6.2.6.Final</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>8.0.33</version>  </dependency>  </dependencies>  </project> | | --- |   You should see an **M** icon in the far right-hand corner as shown in the screenshot. Click on it to Load Maven Changes.      Or you can open the Maven tab in the right-hand corner and load the changes by clicking the on the icon in the left-hand corner. |
| --- | --- |



### **3). Create the Persistence Class (Model Class or Pojo).**

* Create a package “model” under the *src/main/Java/*
* Create an entity class named “Teacher.java” under the above package.
* Add the following code in the Teacher.java class:

| package model;  import jakarta.persistence.\*;  import java.io.Serial;  import java.io.Serializable;  @Entity  @Table  public class Teacher implements Serializable {  @Serial  private static final long *serialVersionUID* = 1L;  @Id  @GeneratedValue( strategy=GenerationType.*IDENTITY* )  private int teacherId;  private String salary;  private String TeacherName;  @ManyToOne  private Department department;  public Teacher(int teacherId, String salary, String teacherName) {  super();  this.teacherId = teacherId;  this.salary = salary;  TeacherName = teacherName; }  public Teacher() {}  public Teacher(String s, String mHaseeb, Department dept1) {  }  public Department getDep() {  return department; }  public void setDep(Department department) {  this.department = department;  }  public int getTeacherId() {  return teacherId;  }  public void setTeacherId(int teacherId) {  this.teacherId = teacherId;  }  public String getSalary() {  return salary;  }  public void setSalary(String salary) {  this.salary = salary;  }  public String getTeacherName() {  return TeacherName;  }  public void setTeacherName(String teacherName) {  TeacherName = teacherName; }  } |
| --- |

* Create a second entity class named **“Department.java”** under the above package.
  + ***src/main/Java/model/Department.java***
* Here is the initial code of the **Department.java** class:

In the **Teacher** class, we specified a ***ManyToOne*** relationship between the **Department** and **Teacher** entities, as one Department can have more than one Teacher.

| package org.example.model;  import jakarta.persistence.\*;  import java.io.Serial;  import java.io.Serializable;  @Entity  @Table  public class Department implements Serializable {  @Serial  private static final long *serialVersionUID* = 1L;  @Id  @GeneratedValue( strategy=GenerationType.*IDENTITY* )  private int deptId;  private String deptName;  public Department(int deptId, String deptName) {  super();  this.deptId = deptId;  this.deptName = deptName;  }  public Department() {}  public Department(String deptName) {  this.deptName = deptName;  }  public int getDeptId() {  return deptId;  }  public void setDeptId(int deptId) {  this.deptId = deptId;  }  public String getDeptName() {  return deptName;  }  public void setDeptName(String deptName) {  this.deptName = deptName;  }  } |
| --- |

### **Explanation:**

### The **Teacher** entity represents the many sides of the relationship, and the **Teacher** table contains the **foreign key** of the record in the **Department**table. As you can see in the above code snippet, we can model this association with an attribute/variable of type **department** and a **@ManyToOne** annotation.

### @ManyToOne private Department department;

### The “***Department department”*** attribute/variable models the association, and the annotation tells Hibernate how to map it to the database.

### That is all you need to do to model this association. By default, Hibernate generates the name of the foreign key column based on the name of the relationship mapping attribute and the name of the primary key attribute. In this example, Hibernate would use a column named ***department\_did*** to store the foreign key to the ***Teacher*** entity.

### If you want to use a different column, you need to define the foreign key column name with a **@JoinColumn** annotation. The example in the following code snippet tells Hibernate to use the column fk\_order to store the foreign key.

| @ManyToOne  @JoinColumn(name = "fk\_dep")  private Department department;  } |
| --- |

### **Step 3: Create the Hibernate Configuration File (hibernate.cfg.xml)**

Right-click on the resources folder **under the *src/main/*** and create a new file. Name the file “hibernate.cfg.xml” inside the resources folder. Add the following code below. You will have to enter your username and password to connect to your database. In this lab, we will use the **“usersDb”** database. If you prefer to use another database, change the database name in the code below.

hibernate.cfg.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>  *<!-- Drop and re-create the database on startup -->*  <property name="hibernate.hbm2ddl.auto"> create-drop </property>  *<!-- Database connection settings -->*  <property name="connection.driver\_class">com.mysql.cj.jdbc.Driver</property>  <property name="connection.url">jdbc:mysql://localhost:3306/usersDb</property>  <property name="connection.username">*<!--TODO Add username --><*/property>  <property name="connection.password">*<!--TODO Add password -->*</property>  *<!-- MySQL DB dialect -->*  <property name="dialect">org.hibernate.dialect.MySQLDialect</property>  *<!-- print all executed SQL on console -->*  <property name="hibernate.show\_sql" >true </property>  <property name="hibernate.format\_sql" >true </property>  *<!-- Mapping entity file →*  <mapping class="model.Teacher"/>  <mapping class="model.Department"/>  </session-factory>  </hibernate-configuration> |
| --- |

The **“teacherId”** value is auto generated, so the value does not have to be added. The session.persist() method is used to persist the value in the database, and once the value is saved, the id value (primary key) is returned. Once the object is saved, the transaction is committed. If an exception occurs, the transaction is rolled back. The transaction ends either through a commit or rollback action. Once the transaction ends, the session is closed.

**Step 4: App.java (main class)**

Add the following code to the App.java class:

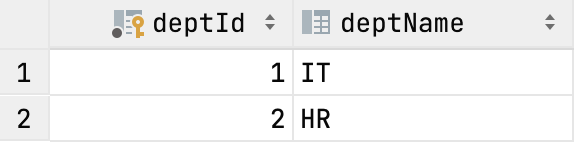
| package org.example;  import org.example.model.Department;  import org.example.model.Teacher;  import org.hibernate.Session;  import org.hibernate.SessionFactory;  import org.hibernate.Transaction;  import org.hibernate.cfg.Configuration;  public class App { public static void main(String[] args) {  *manyToOne();*  }  public static void manyToOne(){  SessionFactory factory = new Configuration().configure().buildSessionFactory();  Session session = factory.openSession();  Transaction transaction = session.beginTransaction();  *//creating departments*  Department dept1 = new Department("IT");  Department dept2 = new Department("HR");  *//creating teacher*  Teacher t1 = new Teacher("1000","MHaseeb",dept1);  Teacher t2 = new Teacher("2220","Shahparan",dept1);  Teacher t3 = new Teacher("3000","James",dept1);  Teacher t4 = new Teacher("40000","Joseph",dept2);  *//Storing Departments in database*  session.persist(dept1);  session.persist(dept2);  *//Storing teachers in database*  session.persist(t1);  session.persist(t2);  session.persist(t3);  session.persist(t4);  transaction.commit(); } } |
| --- |

### **Step 4: Run an Application**

Now run the App class to execute the created code.

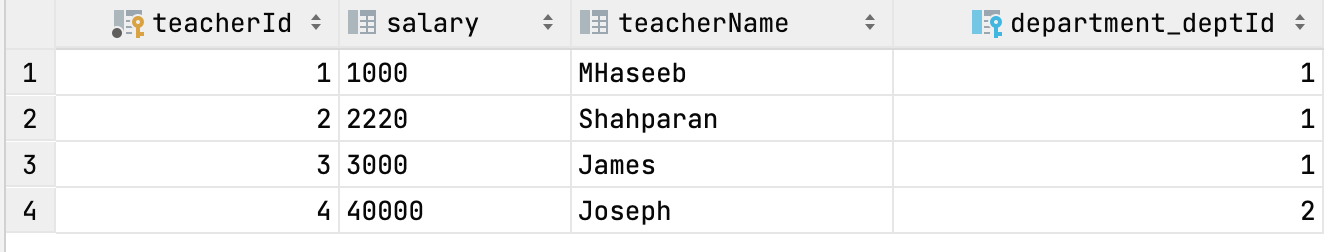
Once you execute your **App.java** class, the database schema will be created, and you will see the **Department table** and **Teacher table** created, along with the records in the ***usersDb*** database.

**Department Table**



* + DeptID is the Primary key in the Department Table

**Teacher Table**



* **DEPARTMENT\_deptId** in the Teacher Table is the foreign key (reference field) from the Department table, and teacherId is the Primary key.

## **Example: @OneToMany Relationship and Mapping Example**

In a **one-to-many** relationship, each row of one entity is referenced by many child records in another entity. The important thing is that children's records cannot have multiple parents. In a one-to-many relationship between Table A and Table B, each row in Table A is linked to 0, 1, or many rows in Table B.

Let us consider the above example. If **Teacher** and **Department** are in a reverse unidirectional manner, the relation is a many-to-one relation.

In this example, we will modify the above example, but you can also set up a new Hibernate project, and then apply a one-to-manyrelationship there.

### **Step 1: Teacher.java Class**

Below is the **Teacher** class with modifications:

| package model;  import jakarta.persistence.\*;  import model.Department;  import java.io.Serial;  import java.io.Serializable;  @Entity  @Table  public class Teacher implements Serializable {  @Serial  private static final long *serialVersionUID* = 1L;  @Id  @GeneratedValue( strategy=GenerationType.*IDENTITY* )  private int teacherId;  private String salary;  private String teacherName;  public Teacher( String salary, String teacherName) {  super();  this.salary = salary;  this.teacherName = teacherName; }  public Teacher() {}  public Teacher(String salary, String teacherName, Department department) {  this.salary = salary;  this.teacherName = teacherName;  }  public int getTeacherId() {  return teacherId;  }  public void setTeacherId(int teacherId) {  this.teacherId = teacherId;  }  public String getSalary() {  return salary;  }  public void setSalary(String salary) {  this.salary = salary;  }  public String getTeacherName() {  return teacherName;  }  public void setTeacherName(String teacherName) {  this.teacherName = teacherName; }  } |
| --- |

### **Step 2: Department.java Class**

One-to-many relationship mapping is not very common. In the following example, it only models the association on the **Department** entity and not on the **Teacher** entity. The basic mapping definition is very similar to the one-to-many association. It consists of the “**List teacherList”** attribute, which stores the associated entities, and a @OneToMany association, as shown in the following source code:

| package model;  import jakarta.persistence.\*;  import model.Teacher;  import java.io.Serial;  import java.io.Serializable;  import java.util.List;  @Entity  @Table  public class Department implements Serializable {  @Serial  private static final long *serialVersionUID* = 1L;  @Id  @GeneratedValue( strategy=GenerationType.*IDENTITY* )  private int deptId;  private String deptName;  public Department(int deptId, String deptName) {  super();  this.deptId = deptId;  this.deptName = deptName;  }  @OneToMany(targetEntity= Teacher.class, cascade = {CascadeType.*ALL*})  private List<Teacher> teacherList;  public List<Teacher> getTeacherList() {  return teacherList;  }  public void setTeacherList(List<Teacher> teacherList) {  this.teacherList = teacherList;  }  public Department() {}  public Department(String deptName) {  this.deptName = deptName;  }  public int getDeptId() {  return deptId;  }  public void setDeptId(int deptId) {  this.deptId = deptId;  }  public String getDeptName() {  return deptName;  }  public void setDeptName(String deptName) {  this.deptName = deptName;  }  } |
| --- |

**Explanation**:

We used the **‘List<E>’** datatype for the **Teachers** class. You can also use the **“Set<E>”** in the **Teacher** class so that every record is unique.

We wanted to save the mapped entity whenever the relationship owner entity is saved. To enable this, we had to use the **“CascadeType”** attribute.

Look at the bold line in the above source code for department.java. We used the @***onetoman***y annotation, along with two additional attributes:

‘targetEntity=Teacher.class' and “cascade=CascadeType.ALL”

Cascade=CascadeType.ALL: Any change that happens to the **teacher** must cascade to the **department** as well. If you save a **Teacher**, all associated accounts will also be saved into the database. If you delete a **Teacher**, all accounts associated with that **Teacher** will also be deleted.

targetEntity=Teacher.class: It is used to determine the entity class that is the target of the association.

targetEntity: Specify the entity class that is the target of the association. This is optional only if the collection property is defined using Java generics; otherwise, it must be specified.

### **Step 3: App.java class**

| import model.Department;  import model.Teacher;  import org.hibernate.Session;  import org.hibernate.SessionFactory;  import org.hibernate.Transaction;  import org.hibernate.cfg.Configuration;  import java.util.ArrayList;  import java.util.List;  public class App {  public static void main(String[] args) {  *oneToMany*();  }  public static void manyToOne(){  SessionFactory factory = new Configuration().configure().buildSessionFactory();  Session session = factory.openSession();  Transaction transaction = session.beginTransaction();  *//creating departments*  Department dept1 = new Department("IT");  Department dept2 = new Department("HR");  *//creating teacher*  Teacher t1 = new Teacher("1000","MHaseeb",dept1);  Teacher t2 = new Teacher("2220","Shahparan",dept1);  Teacher t3 = new Teacher("3000","James",dept1);  Teacher t4 = new Teacher("40000","Joseph",dept2);  *//Storing Departments in database*  session.persist(dept1);  session.persist(dept2);  *//Storing teachers in database*  session.persist(t1);  session.persist(t2);  session.persist(t3);  session.persist(t4);  transaction.commit(); }    public static void oneToMany(){  SessionFactory factory = new Configuration().configure().buildSessionFactory();  Session session = factory.openSession();  Transaction t = session.beginTransaction();  *//creating teacher*  Teacher t1 = new Teacher("1000","MHaseeb");  Teacher t2 = new Teacher("2220","Shahparan");  Teacher t3 = new Teacher("3000","James");  Teacher t4 = new Teacher("40000","Joseph");  Teacher t5 = new Teacher("200","Ali");  *//Add teacher entity object to the list*  ArrayList<Teacher> teachersList = new ArrayList<>();  teachersList.add(t1);  teachersList.add(t2);  teachersList.add(t3);  teachersList.add(t4);  teachersList.add(t5);  session.persist(t1);  session.persist(t2);  session.persist(t3);  session.persist(t4);  session.persist(t5);  *//Creating Department*  Department department = new Department();  department.setDeptName("Development");  department.setTeacherList(teachersList);  *//Storing Department*  session.persist(department);  t.commit(); }  } |
| --- |

### 

### 

### **Step 4: Run an Application**

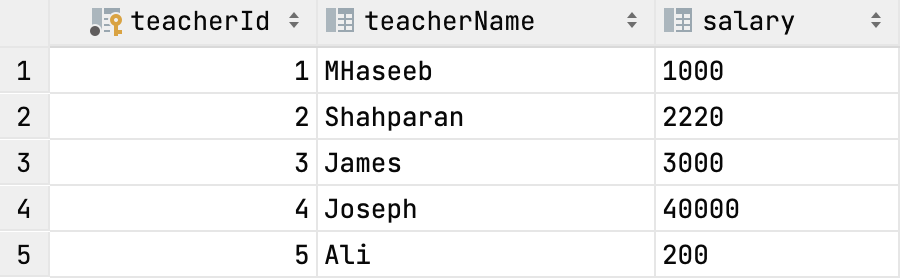
Now let us execute the code we created above. Run the App class.

**Explanation**: In the above code, we have created one department object and several teacher objects. We added all teacher objects to one department. As a result, all teachers will belong to one department.

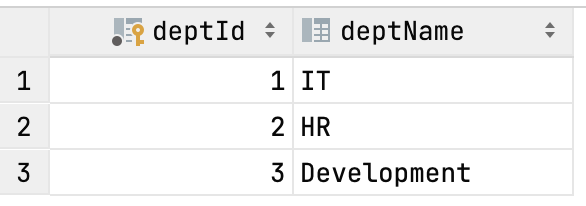
The **“tit**” value is auto generated; therefore, the value need not be set here. The **session.persist()** method is used to persist the value in the database, and once the value is saved, the id value (Primary Key) is returned. Once the object is saved, the transaction is committed. If an exception occurs, the transaction is rolled back. The transaction ends either through a commit or rollback action. Once the transaction ends, the session is closed.

**Result**: At the start of each thread, a database schema will be created and you will see the following result in ***userDb*** Database

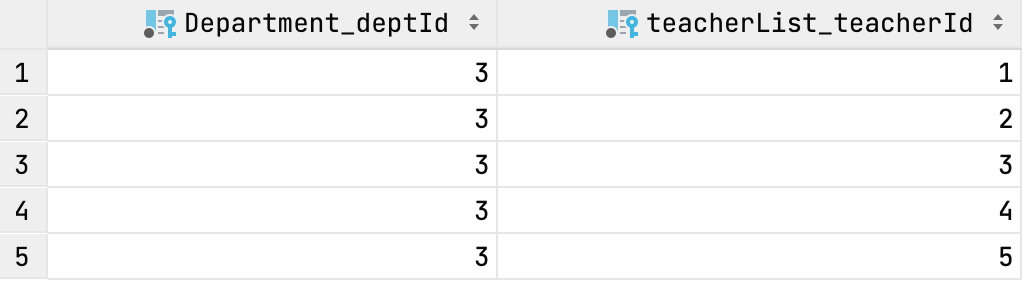
Teacher screenshot



Department screenshot



Department\_teacher table:



As you can see in the above screenshot, the joined table has been created. The join table (Department\_teacher) will have two foreign key columns (i.e., it will have both primary key columns from the two entity tables from the Department and Teacher tables [**Department\_depId** and **teacherlist\_teacherId]**, respectively). One of the foreign keys will serve as the primary key for the join table. A unique constraint will be applied to the remaining foreign key column; hence, both foreign key columns will not have duplicate values.

**Submission Instructions:**

Include the following deliverables in your submission:

* Submit your source code or screenshot using the Start Assignment button in the top-right corner of the assignment page in Canvas.

## 