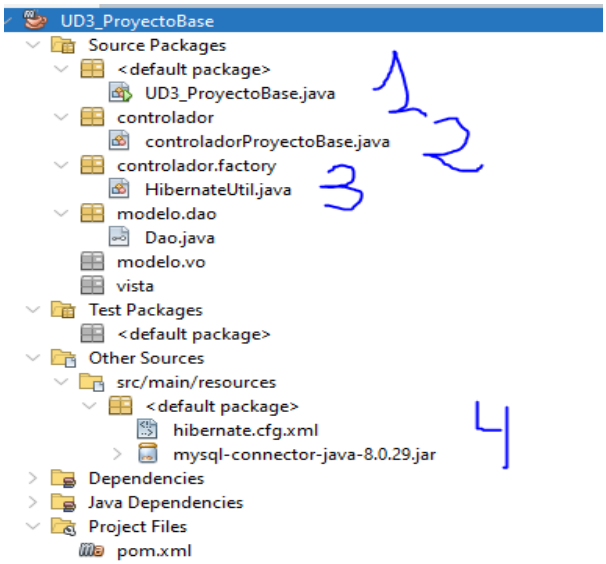
ACCESO A DATOS
UD3: MAPEO OBJETO-RELACIONAL

CRITERIOS QUE UTILIZAREMOS EN LOS EJERCICIOS QUE VAMOS A PLANTEAR A LO LARGO DE LA UNIDAD.

1.-Todos los proyectos tendrán la estructura proporcionada en el proyecto base.



A continuación explicamos como vamos a trabajar con él.

1.- El proyecto se iniciará en la clase del mismo nombre del proyecto. En este caso UD3_ProyectoBase.

2.- Observa como vamos a tener un controlador por cada formulario en la vista.

3.- En el factory es donde vamos a tener el fichero Hibernateutil.java que permite **iniciar la configuración** para conectarnos a la base de datos. Además en el fichero HibernateUtil es **donde iniciaremos los DAO**.

4.- El fichero de configuración para conectarnos a la base de datos lo tendremos en el paquete /src/main/resources. Será el fichero hibernate.cfg.xml. **Este fichero deberás modificarlos en tres puntos importantes cuando empieces a trabajar**:

- Deberás indicar la ip a la que te conectas. (1)
- Deberás indicar el nombre de la BD. (2)
- Deberás incluir las clases que mapeas (Pojos). (3)

En recursos incorporamos a mayores el driver de mysql-connector-java para crear los pojos. Los pojos se pueden crear a mano , pero os indicaré como hacerlo de un modo automático más adelante.

```
<session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
    <property name="hibernate.connection.password">root</property>
    <!--Debes modificar 2 lineas para la base de datos e ip-->
    <!-- 1.-Debes indicar la ip y la Bd a la que conectas-->
    <property name="hibernate.connection.url">jdbc:mysql://IP:3306/BD?zeroDateTimeBehavior=CONVERT_TO_NULL</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>
    <property name="hibernate.default_schema">BD</property> <!--2.- Pon aqui la BD -->
    <property name="hibernate.connection.autocommit">false</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <!--<property name="hibernate.format_sql">true</property>-->
    <property name="hibernate.query.factory_class">org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <<!-- debes incluir todas las clases que mapeas-->
    <!--         <mapping class="modelo.vo.clase"/>        -->

</session-factory>
ibernate-configuration>
```

Dependencias en el proyecto.

Si inicias los proyectos desde el proyecto base ya vienen incorporadas, pero fíjate que hemos incorporado estas dependencias:

Para hibernate,mysql y lombok (esta no obligatoria):

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.0.Final</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.30</version>
    <scope>provided</scope>
</dependency>
```

Otras que veas de org.eclipse.persistence se incorporan al crear los pojos (ya están en el proyecto base).

Antes de empezar a trabajar tenemos que crear los pojos ¿Como crear los pojos?

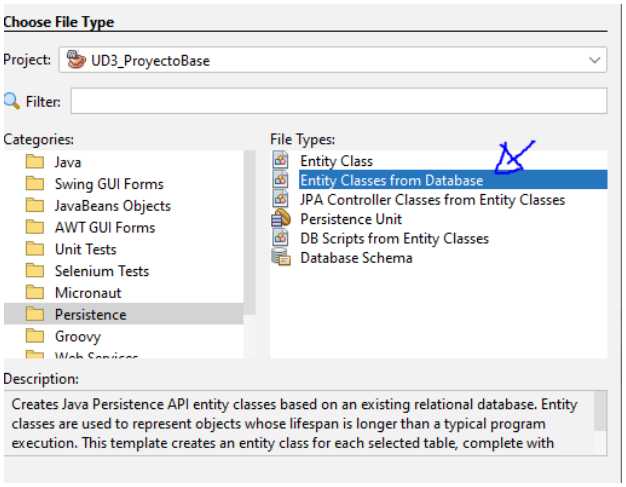
Los pojos que son las clases que se asocian con las tablas de la base de datos, pueden ser creadas directamente por nosotros , pero por comodidad os diré como hacerlo de un modo automático.

En este caso vamos a crear 2 pojos : Departamentos.java y Empleados.java

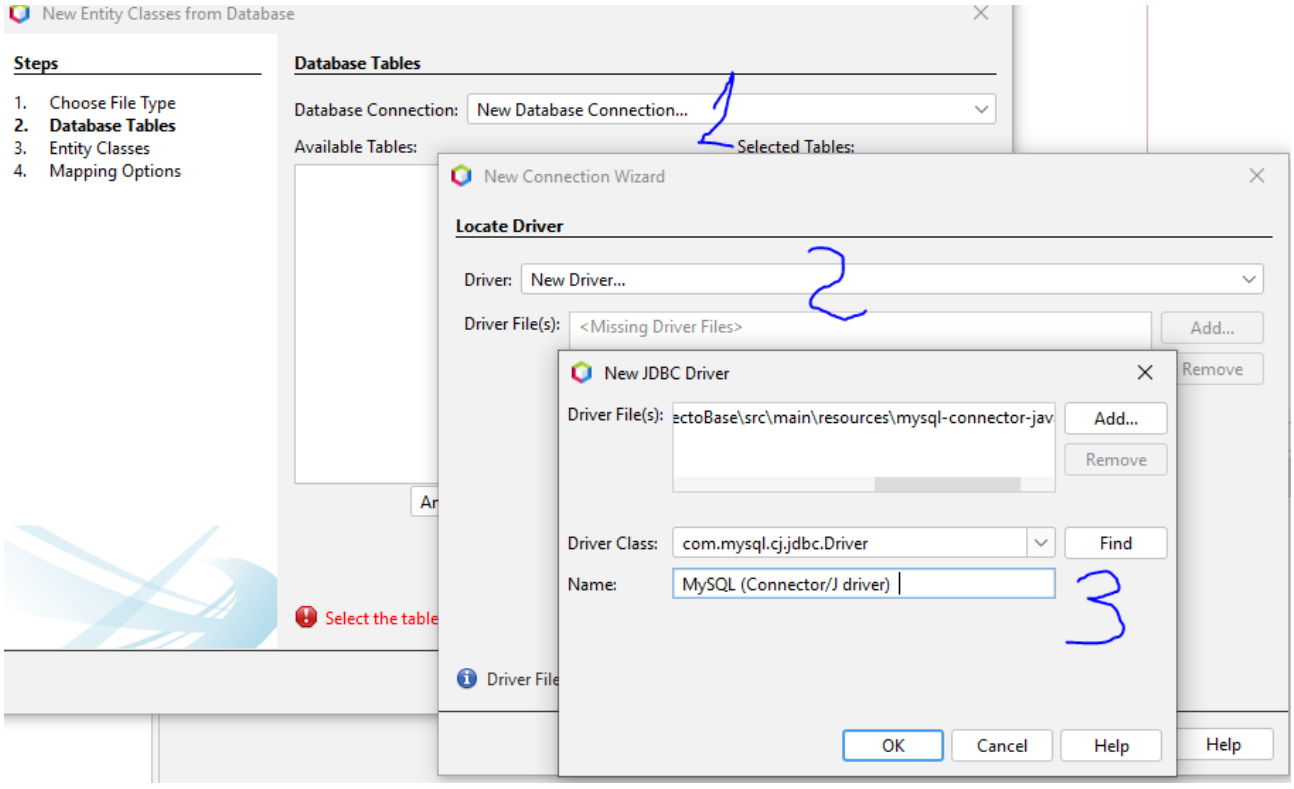
Al final veremos también como incorporamos (opcionalmente) lombok si queremos a la clase Departamento. Incorporando solo una de las etiquetas de lombok permitirá que la clase sea más sencilla y con la misma funcionalidad.

Pasos:

1.- Situaros en modelo.vo y alli con el botón derecho cliquear other. Nos vamos a la categoria de Persistence y elegimos Entity Classes from Database

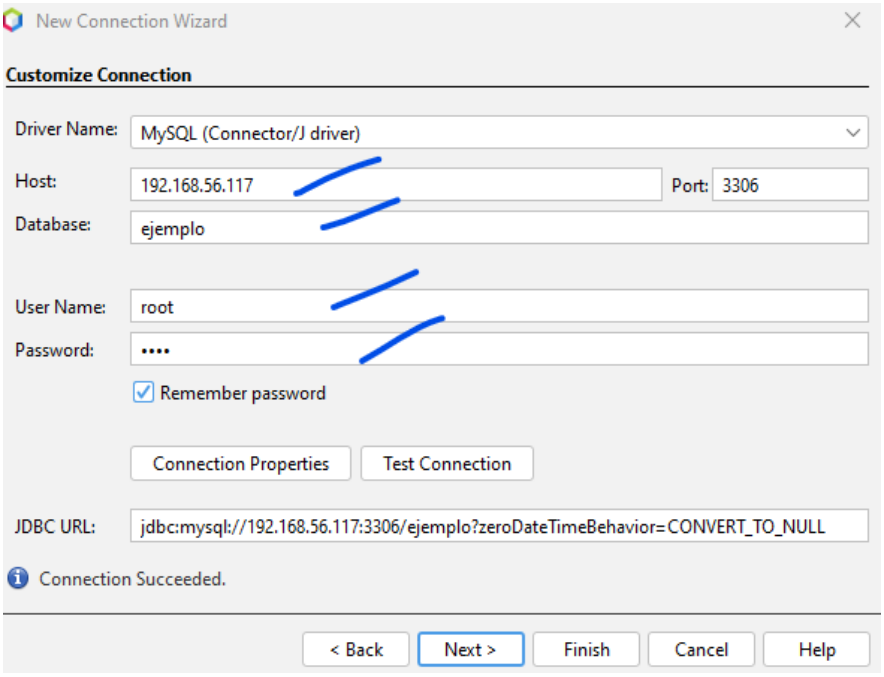


2.- A continuación creamos una New Database Conection (1). Allí le damos a New Driver (2). En el botón add buscamos la ruta donde está el fichero mysql-connector-java.

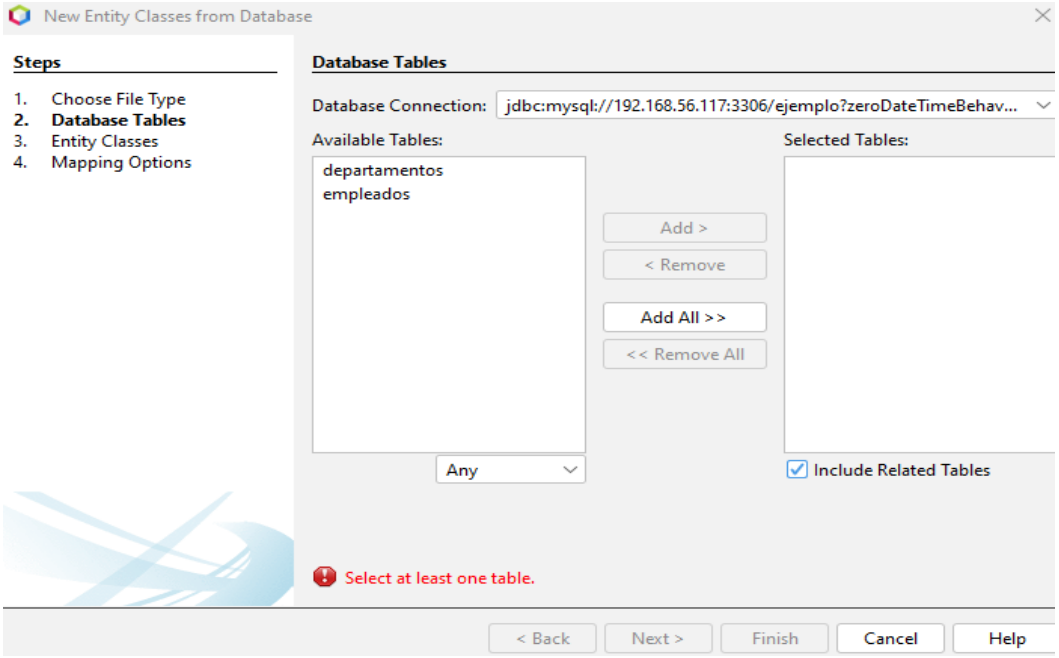


3.-Una vez le damos a siguiente tendremos que poner los parámetros básicos para conectarnos a la base de datos

(ajustalos a tu situación concreta):

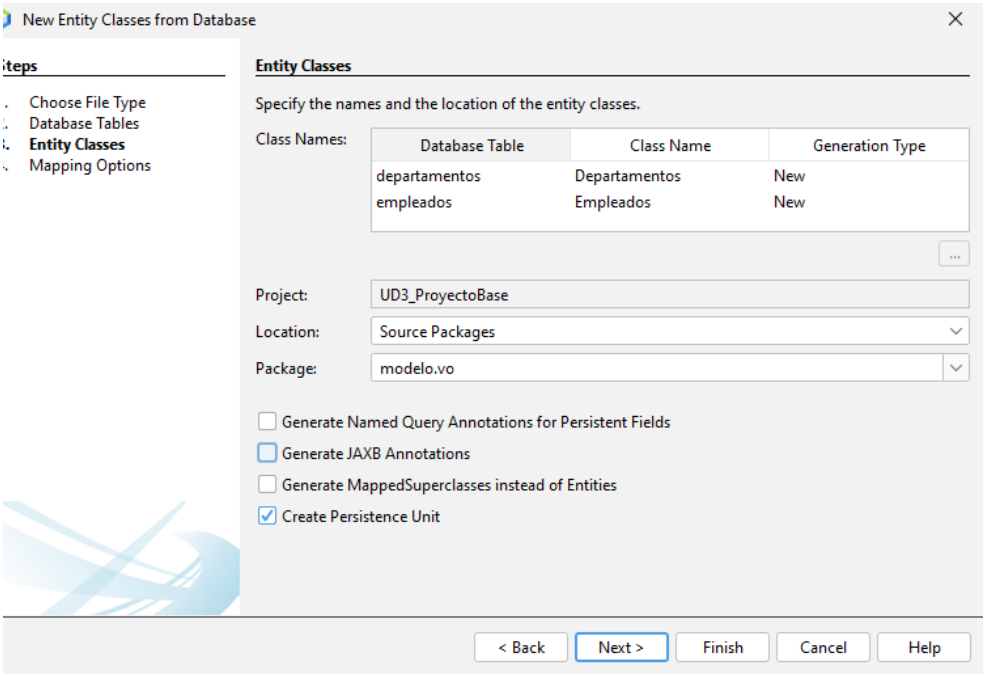


4.-Le damos a siguiente hasta que aparezca



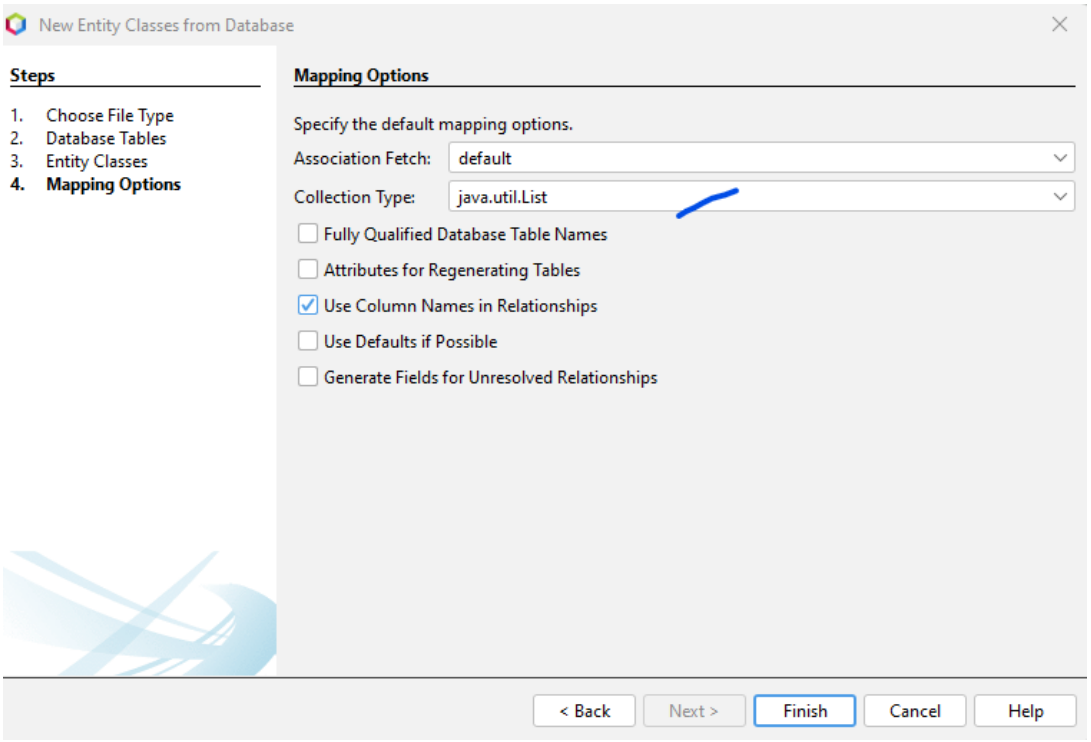
Seleccionamos todas las tablas con Add All y le damos a siguiente.

5.- Llegamos a esta pantalla

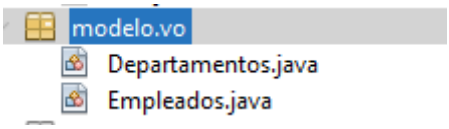


Solo nos pide obligatorio “Create Persistence Unit”. Le damos a next

6.-Nos aparece otra pantalla donde cambiamos el tipo de colección a listas.



Con este paso ya hemos conseguido los pojos



7.-Eliminar la linea de @NamedQueries

8.- Opcionalmente si quieres incorporar lombok a las clases elimina todos los métodos menos los constructores. Añades el import lombok.*; y la etiqueta @Data al inicio

```
import javax.persistence.Table;

import lombok.*;
/**
 *
 * @author acceso a datos
 */
@Data
@Entity
@Table(name = "departamentos")

public class Departamentos implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "dept_no")
    private Short deptNo;
    @Column(name = "dnombre")
    private String dnombre;
    @Column(name = "loc")
    private String loc;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "deptNo")
    private List<Empleados> empleadosList;

    public Departamentos() {
    }

    public Departamentos(Short deptNo) {
        this.deptNo = deptNo;
    }


}
```

Compresión del modelo generado.

```
@Data
@Entity
@Table(name = "departamentos")
public class Departamentos implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "dept_no")
    private Short deptNo;
    @Column(name = "dnombre")
    private String dnombre;
    @Column(name = "loc")
    private String loc;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "deptNo")
    private List<Empleados> empleadosList;

    public Departamentos() {
    }
```

1.- Si le indicamos la etiqueta @Data lombok nos permite quitar los getter/setter y toString para simplificar las clases. Etiqueta no obligatoria.

2.- Indica que la clase es una entidad referida a la tabla departamentos.

3.- La etiqueta @id indica que el atributo id que se identifica en la tabla como dept_no es clave.

4.- El atributo nombre se identifica en la tabla como dnombre.

5.- En cada objeto de departamento, guardamos la lista de empleados. Cuando pone @OnetoMany leeriamos one (nombreclase) tiene many (clase lista). Es decir 1 departamento tiene N empleados.

6.- **Observa en las clases que te he enviado hemos realizado 2 modificaciones en esta linea!!!.**

La primera es quitar la opcion de cascade , así evitamos que al borrar un departamento borre automaticamente los empleados. Queremos hacerlo nosotros programando.

Segundo cambio, para no confundir el atributo deptNo con el objeto Departamento del empleado que también le llama deptNo. En el modelo generado he cambiado el nombre a departamentos. Debo modificar también el nombre en la clase empleado ya que deben ser iguales.

En la clase departamentos

```
//@OneToMany(cascade = CascadeType.ALL, mappedBy = "deptNo")
@OneToMany(mappedBy = "departamentos")
private List<Empleados> empleadosList;
```

En la clase Empleados

```
@ManyToOne(optional = false)
//private Departamentos deptNo;
private Departamentos departamentos; //Este nombre debe ser el mismo en ambas clases.
```

COMO INICIAR UN PROYECTO

Ten en cuenta una vez que ya tenemos los pojos.

1.- Modificar el fichero hibernate.cfg.xml con los parámetros que procedan de ip, base de datos y localización de pojos.

```
<session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
    <property name="hibernate.connection.password">root</property>
    <!--Debes modificar 2 lineas para la base de datos e ip-->
    <!--1.-Debes indicar la ip y la Bd a la que conectas-->
    <property name="hibernate.connection.url">jdbc:mysql://192.168.56.117:3306/ejemplo?zeroDateTimeBe
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>
    <property name="hibernate.default_schema">ejemplo</property> <!--2.- Pon aqui la BD -->
    <property name="hibernate.connection.autocommit">false</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <!--<property name="hibernate.format_sql">true</property>-->
    <property name="hibernate.query.factory_class">org.hibernate.hql.internal.classic.ClassicQueryTra
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <!-- debes incluir todas las clases que mapeas-->
    <!--         <mapping class="modelo.vo.clase"/>        -->
    <mapping class="modelo.vo.Departamentos"/>
    <mapping class="modelo.vo.Empleados"/>

</session-factory>
```

2.- Debes incorporar al fichero HibernateUtil.java los métodos para poder crear objetos DAO.

```
/************************* INCORPORA LOS MÉTODOS PARA CREAR LOS OBJETOS DAO ********/

public static DepartamentoDAO getDepartamentoDAO() {
    return new DepartamentoDAO();
}
```

3.- El controlador por cada vista tendrá estos métodos básicos:

public static void iniciar() : Es el método para iniciar el formulario.
public static void iniciaSession() : Es el método para iniciar los parámetros básicos para conectar a la base de datos. Aqui iniciaremos el objeto factory, la session y lo DAO implicados.
public static void cerrarSession(): Es el método para cerrar la session.

Cada proceso lo vamos a realizar en una **transacción,** por eso en el controlador iniciamos siempre con un begin transacction y cerramos con commit o rollback. La variable session tiene ya incorporados los métodos de beginTransaction, commit y rollback, pero en los ejemplos utilizo los métodos de hibernateUtil para asegurarnos que si una transacción ya está abierta no la vuelva a abrir.

```
public static void iniciar() {
    ventana.setVisible(b: true);
    ventana.setLocationRelativeTo(c: null);
}

public static void iniciaSession() {
    session=HibernateUtil.getSessionFactory().openSession();
    depDAO=HibernateUtil.getDepartamentoDAO();
}

public static void cerrarSession() {
    session.close();
}
```

4.-En cada formulario que trabajemos se iniciará con iniciaSession(); en el contructor y se cerrará la session en el evento para cerrarlo.

```
public Enunciado1() {
    initComponents();
    controladorEnunciado1.iniciaSession();

}
```

```
private void formWindowClosed(java.awt.event.WindowEvent evt) {
    // TODO add your handling code here:
    controladorEnunciado1.cerrarSession();
}
```

Por ultimo os comento:

Me imagino que como ya en la unidad anterior conectateis a la BD, no deberiais tener problemas, pero si os diera problemas con la conexión remota podriais lanzar estas 2 instrucciones en el mysql.

```
create user 'root'@'%' identified by 'root';
grant all privileges on *.* to 'root'@'%' with grant option;
```

ACCESO A DATOS
UD3: MAPEO OBJETO-RELACIONAL

ENUNCIADO 1: Primera Conexión a BD. Listar los datos del departamento en un textArea.

Crea un formulario que establezca una conexión a la base de datos para listar los departamentos en un textArea.

Debes tener en cuenta:

La consulta clásica es

"Select * from departamentos".

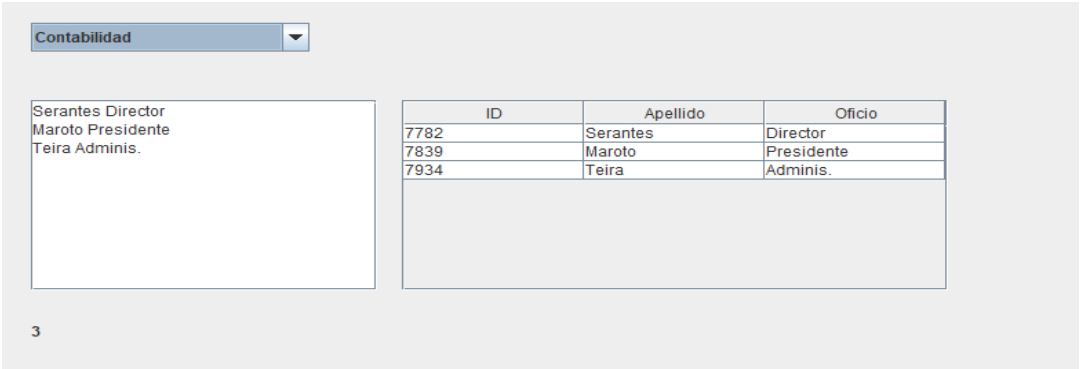
Con hibernate NO consultamos a las tablas sino a las clases que hemos mapeado, con lo que la consulta sería cualquiera de las 2 opciones:

"select d from Departamentos d"; Observa (**Departamentos y no departamentos**)
"from Departamentos d"

Donde d son los objetos de la clase (no es obligatorio poner el d, pero os va a facilitar las consultas poner el alias).

ENUNCIADO 2: Mostrar empleados por departamento.

Crea un combo de objetos departamento donde solo se visualizaran los departamentos. Una vez eliga en el combo el departamento mostrará los empleados del departamento correspondiente además del número de empleados que tiene.



Como lo que vemos en el combo es el método toString de la clase Departamentos. Añadimos el método toString a la clase.

```
@Override
public String toString() {
    return dnombre;
}
```

Nota: Fijate como si no hubieramos cambiado en las clases departamento y empleados el mapeamiento de

@OneToMany(cascade = CascadeType.ALL, mappedBy = "deptNo")

a

@OneToMany(mappedBy = "departamentos")

La consulta que devuelve los empleados de un departamento sería:

Query q = session.createQuery("from Empleados e where e.deptNo.deptNo=:deptno order by e.empNo");

en vez de la que te dejo:

from Empleados e where e.departamentos.deptNo=:deptno order by e.empNo

ENUNCIADO 3: Manejo de Departamentos

Crea un formulario que permita introducir datos de los departamentos.

Aunque la idea es que en ejercicios posteriores podamos insertar, borrar y modificar departamentos. En este primer ejercicio lo que harás es buscar los datos del departamento a partir del número de departamento en su lostfocus. De tal manera que:

1.- Si se introduce un departamento existente carga los datos cuando el cuadro de texto pierde el foco.
2.- Si se introduce un departamento no existente deja en blanco los valores.
3.- Si no se introduce nada lo indica y deja en blanco los valores.
4.- Si se introduce un valor incorrecto lo indica. (Ejemplo abc cuando debe introducir un valor númerico) y deja en blanco los valores

Prueba su funcionamiento con estos valores

numero:10 (existe el departamento y carga los datos)
numero =60 (departamento no existente y deja en blanco los cambpos
numero vacio: (indica que faltan datos)
numero = abc (valor incorrecto)