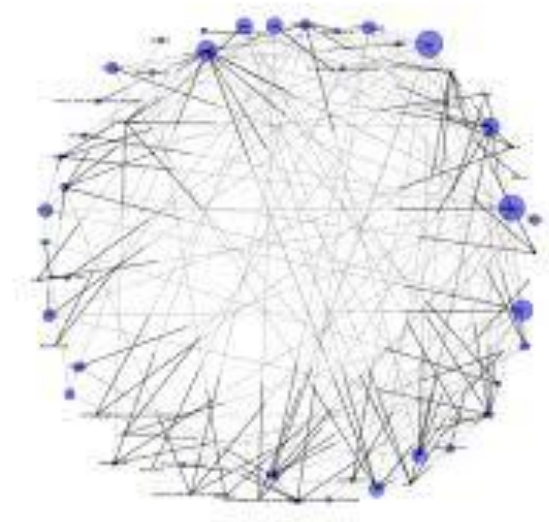


DAM

ACCESO A DATOS

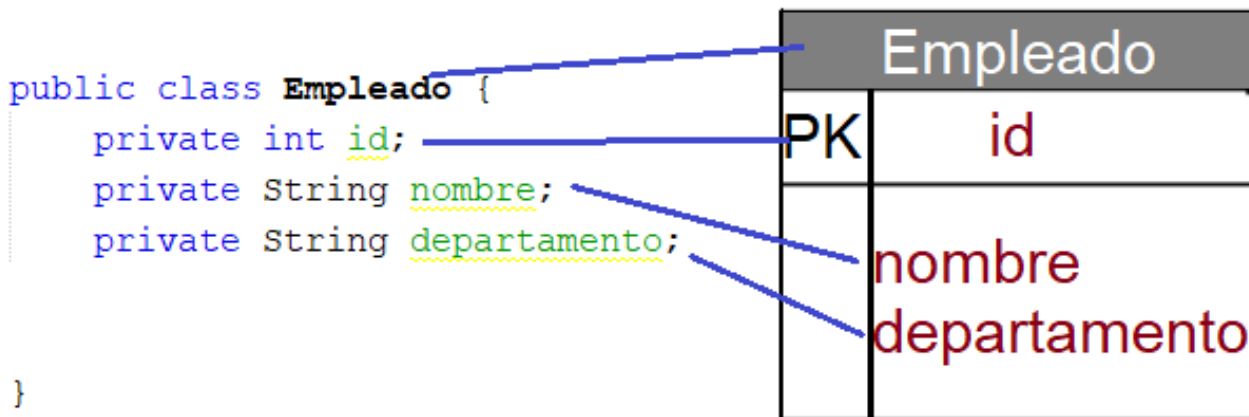


UD3

HERRAMIENTAS DE MAPEADO OBJETO- RELACIONAL (ORM)

Introducción

- En este tema mejoraremos el **acceso a las BD desde Java**, aumentando la conexión entre **la estructura de las BD relacionales** y la **estructura de clases** de la programación OO.
- Anteriormente, utilizamos **JDBC** para hacer **persistentes** los objetos en una BD, pero esta tecnología requiere de la creación de muchas líneas de código y presenta grandes **dificultades** a la hora de afrontar la persistencia de **objetos complejos** y de tratar bases de datos con **relaciones múltiples entre sus tablas**.
- Para crear esa relación entre BD relacional y programación OO es necesaria una **interfaz que traduzca la lógica de los objetos a la lógica relacional**. Esa interfaz se llama **ORM** (*Object Relational Mapping*) o **Mapeo Objeto-Relacional**.
- Una interfaz **ORM** facilita la **conversión tabla-objeto** que permitirá **transformar las tablas de una BD a clases** y las **filas de las tablas (registros)** podrán ser manejadas como instancias de una clase.



Ventajas/Inconvenientes del uso de herramientas ORM

- **Ventajas:**

1. **Mejor arquitectura del sistema:**

- El uso de herramientas *ORM* **facilita la programación por capas**, obligando a la **separación** de la **capa de acceso a datos**, de esta forma aumentará la posibilidad de **reutilización del código** ya que la mayor parte del código es independiente de la base de datos.

2. **Menor tiempo de codificación:**

- El uso de herramientas *ORM* **evita** tener que **usar** complejas y largas instrucciones **SQL** y la **conversión manual** de los **objetos** en datos almacenables en tablas, que se debe realizar con *JDBC*. *La herramienta ORM convierte datos-objetos y viceversa de forma instantánea.*

3. **Almacenamiento en caché y transacciones:**







- La mayor parte de las herramientas *ORM* incluyen funciones para el uso de caché y transacciones. El almacenamiento en caché permite mejorar el rendimiento de las operaciones de BD y el soporte de transacciones aporta un plus de seguridad a las modificaciones de la base de datos.

- **Inconvenientes:**

1. **Las aplicaciones son más lentas** debido a las transformaciones que se tienen que realizar cada vez que se produce un acceso a la base de datos.
2. **Se requiere de la introducción de metadatos** para la transformación objetos/relacional, pero esta operación es más sencilla que el mantenimiento *manual* de una solución *JDBD*.



Terminología

- 
- 
- 
- 
- 
- 
- **POJO** (*Plain Old Java Objects*) → Objeto de Java que no implementa ninguna interface especial
 - **CRUD** (*Create, Read, Update and Delete*) → Funciones básicas de almacenamiento persistente en una base de datos.
 - **Capa de persistencia** → Clases de POO que encapsulan las operaciones CRUD. Es la capa encargada de hacer que los datos se mantengan almacenados en la base de datos pero también es la encargada de recuperar esos datos.



ORM: elementos que la componen

- **Un API para realizar operaciones CRUD** (Create, Retrieve, Update, Delete) básicas sobre objetos de clases persistentes.
- **Un lenguaje o API para especificar consultas** referidas con las clases y sus propiedades (atributos).
- **Facilidades para la especificación de los metadatos.**
- **Técnicas de implementación para interactuar con los objetos transaccionales** para la realización de diversas funciones de optimización, (como dirty checking, entre otros).



Objetivos de las herramientas ORM

- Las **herramientas ORM** tienen como **objetivo**
 - **Facilitar las tareas de mapeado** entre **POJOs** (*Plain Old Java Objects*) y el almacén de datos.
 - **Aportar funciones** para realizan las operaciones **CRUD** (*Create, Read, Update and Delete*) en la base de datos.
- **Con las herramientas ORM** podremos **utilizar POJOs** de nuestra aplicación e implementar sencillas operaciones que **mediante las herramientas ORM** harán los **datos persistentes**. Por ejemplo, con una instrucción similar a `orm.save(elObjeto)`, se guardará el objetos en una tabla de la base de datos.

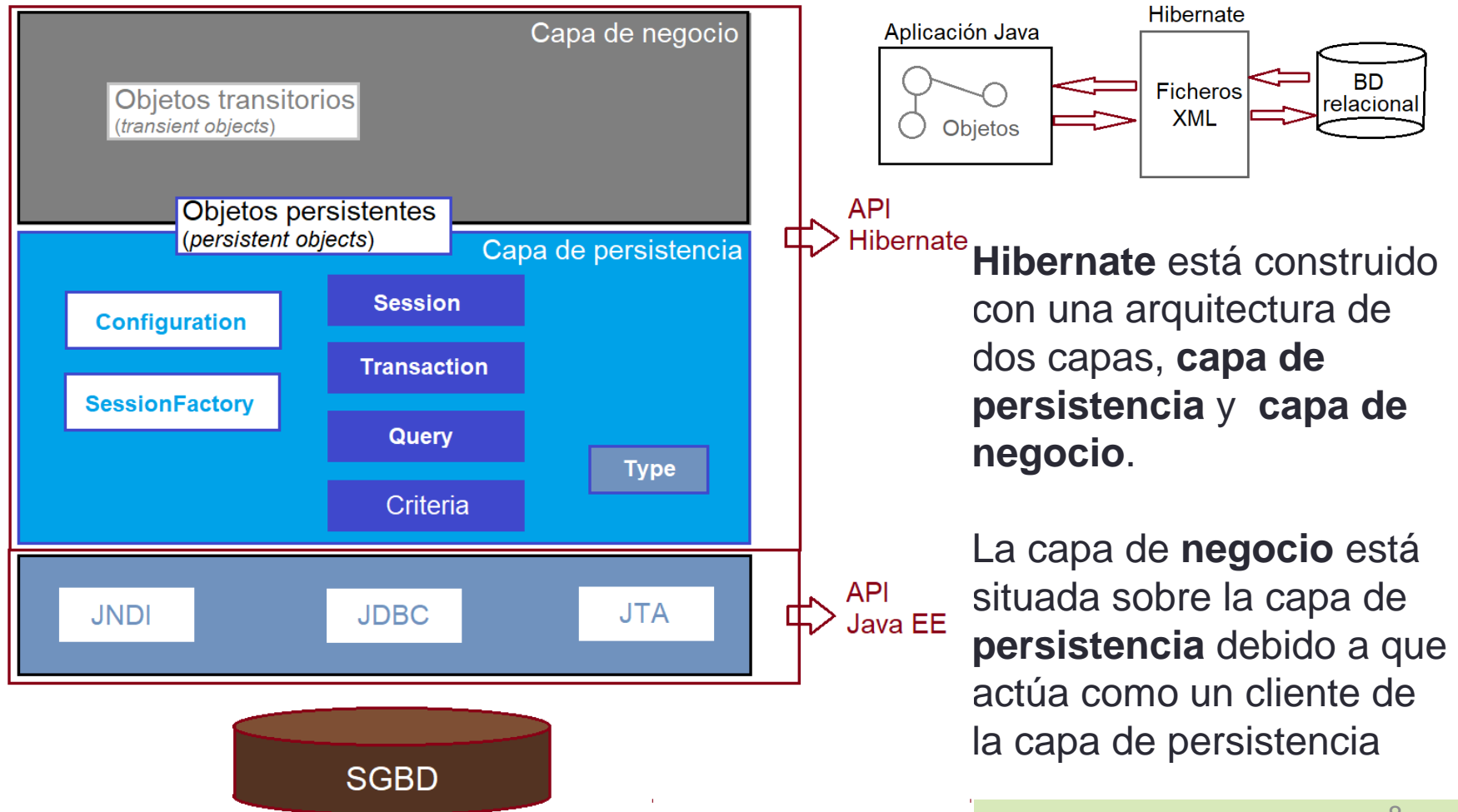


Tecnologías ORM

- Existen numerosas herramientas **ORM** tanto de pago como de uso libre.
- Utilizando los patrones *Repository* o *Active Record* es posible crear herramientas propias ORM.
- Algunos ejemplos de herramientas ORM son:
 - **Para Java:** [Hibernate](#), iBatis, Ebean, Torque
 - **Para .Net:** nHibernate, Entity Framhqlamework, DataObjects.NET
 - **Para PHP:** Doctrine, Propel , Torpor
 - **Para Python:** SQLAlchemy, Django, Tryton
- En este tema veremos como desarrollar aplicaciones Java utilizando **Hibernate**, que es una herramienta de **software libre** para la **plataforma Java** que tiene su equivalente en .Net (nHibernate).
- **Hibernate** posee su propio lenguaje de consultas, **HQL** (*Hibernate Query Language*)

Hibernate. Arquitectura

- **Hibernate** es un *framework* que **facilita el mapeo de atributos** entre una **base de datos relacional** y el **modelo de objetos** de una aplicación mediante **archivos XML** o **anotaciones en las entidades** que permiten establecer estas relaciones.



Hibernate está construido con una arquitectura de dos capas, **capa de persistencia** y **capa de negocio**.

La capa de **negocio** está situada sobre la capa de **persistencia** debido a que actúa como un cliente de la capa de persistencia

Hibernate: interfaces

- Las **interfaces** que se utilizan en **Hibernate** para realizar las operaciones básicas (inserciones, borrados, modificaciones, consultas, etc.) son:
 - **Session**: Se utiliza para obtener una conexión física con una BD. Los objetos **Session** no deben mantenerse abiertos durante mucho tiempo, ya que no suelen ser seguros para subprocesos y deben ser creados y destruidos, según sea necesario.
 - **Transaction**: Permite controlar las transacciones. Este es un objeto opcional y las aplicaciones de Hibernate puede optar por no utilizar esta interfaz y en su lugar gestionar las transacciones en su propio código de la aplicación.
 - **Query**: Objetos de consulta que utiliza SQL o *Hibernate Query Language* (HQL) para recuperar datos de la BD y crear objetos. Una instancia de consulta se utiliza para enlazar los parámetros de consulta, limitar el número de resultados devueltos por la consulta, y finalmente, para ejecutar la consulta.
 - **Criteria**: Los objetos **Criteria** se utilizan para crear y ejecutar consultas con objetos y recuperar objetos.
 - **Type**: Un objeto **Type** *Hibernate* hace corresponder un tipo Java con un tipo de una columna de una BD. Todas las propiedades persistentes de las clases persistentes tienen un tipo *Hibernate* correspondiente.
- Las interfaces llamadas por el código de la infraestructura de la aplicación para configurar **Hibernate**:
 - **Configuration**: Permite arrancar y configurar **Hibernate** al especificar la ubicación de los *documentos que contienen la información del mapeado de los objetos* y propiedades específicas de **Hibernate**. Es la encargada de crear un objeto **SessionFactory**.
 - **SessionFactory**: Objeto de configuración se utiliza para obtener un objeto **Session**. Normalmente se usa una única instancia de **SessionFactory** para toda la aplicación, solo serían necesarias distintas instancias si se accediese a varias BD.
- **Hibernate** hace uso de APIs de Java, tales como **JDBC**, **JTA** (*Java Transaction Api*) y **JNDI** (*Java Naming Directory Interface*).

Arquitectura de capas

- La arquitectura de capas requiere de la agrupación de los contenidos de la aplicación en 3 grupos, dependiendo de su futuro uso.

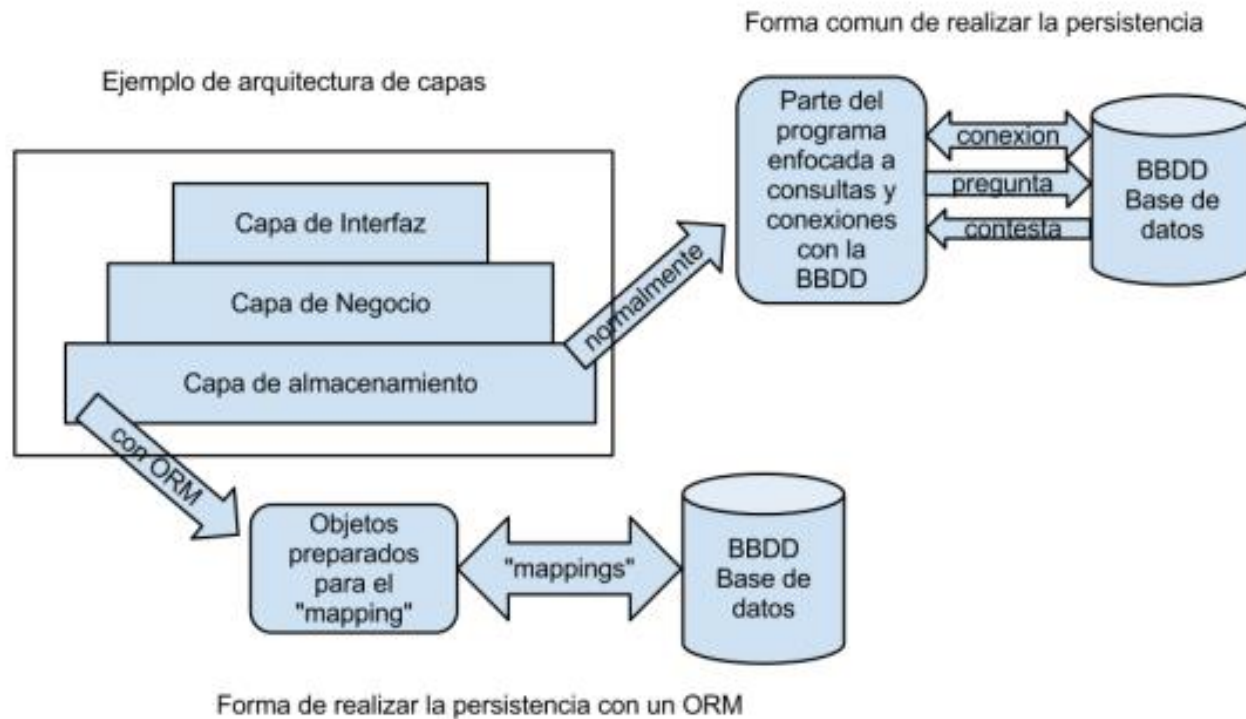


Imagen tomada de [enlace](#)

Arquitectura de capas. Descripción

- **Presentación o Interfaz:**

- Es equivalente a la capa **Vista** dentro del modelo **MVC** (*Modelo, Vista, Controlador*).
- Es la capa encargada de comunicarse con la interfaz de usuario para enviar el resultado de las operaciones y recoger las demandas mediante una interfaz gráfica o de texto.

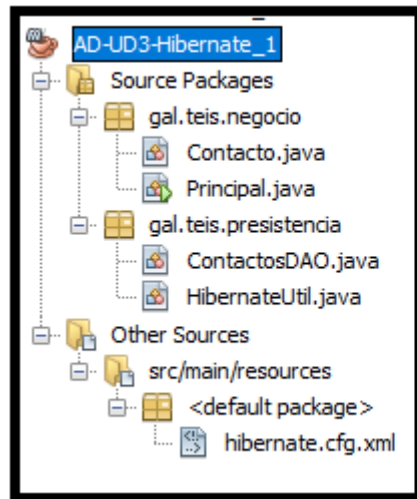
- **Negocio o aplicación:**

- Es la capa que conecta las capas de presentación y persistencia y tiene su equivalencia en la capa **Controlador** en el modelo **MVC**.
- Contiene la funcionalidad asociada a las operaciones a realizar.
- Valida los datos antes de ser tratados si es necesario, realiza operaciones con los mismos y los envía a la capa de presentación o persistencia según sea necesario.

- **Persistencia o almacenamiento:**

- Es la encargada de almacenar y recuperar los datos de la BD.
- En un escenario ORM el programador no realiza una conexión y una consulta SQL en una BD sino que es ORM el que se encarga de estas funciones. ORM transforma los datos de la BD en objetos, siendo éstos usados generalmente en las tareas propias de negocio.

Crear una aplicación con Hibernate en NetBeans con proyectos Maven, utilizando anotaciones de las entidades



Estructura del
proyecto
basado en la
arquitectura de
capas

https://github.com/estherff/AD-UD3-Hibernate_1

Enlaces de consulta:

- <http://www.jtech.ua.es/j2ee/2006-2007/restringido/hib/sesion01-apuntes.html>
- <https://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/tutorial.html#tutorial-firstapp-mapping>

Crear una base de datos para utilizar con Hibernate

- En **phpMyAdmin** creamos una base de datos MySQL llamada **agenda** que debe contener una tabla llamada **Contactos** creada con la siguiente instrucción SQL:

```
CREATE TABLE `Contactos` (  
  `id` int NOT NULL AUTO_INCREMENT,  
  `nombre` varchar(50) NOT NULL,  
  `email` varchar(25) NOT NULL,  
  `telefono` varchar(15) NOT NULL,  
  PRIMARY KEY (`id`)  
) DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;
```

1°. Agregar las dependencias necesarias a NetBeans

- Para poder utilizar **Hibernate** desde NetBeans debemos agregar la dependencia a **hibernate-core**.
- Si vamos a utilizar una base de datos **MySQL** tendremos que agregar la dependencia al **conector** correspondiente.

```
<dependencies>
  <!-- Núcleo de Hibernate-->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.21.Final</version>
  </dependency>

  <!-- Connector MySQL-->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
  </dependency>
</dependencies>
```

2º. Crear una instancia de la BD

- Debemos crear una clase que permita **arrancar Hibernate y crear una instancia de SessionFactory que utiliza el fichero de configuración (hibernate.cfg.xml)** donde está la información de la conexión con **nuestra base de datos**.

```
public class HibernateUtil {

    private final SessionFactory laSessionFactory ; // Atributos privados y finales

    private static HibernateUtil elHibernateUtil; // atributo privado y estático de la misma clase

    //Obtiene una instancia de la clase ya creada o la crea de nuevo y devuelve el atributo de tipo SessionFactory
    public static SessionFactory getSessionFactory() {
        if (Objects.isNull(elHibernateUtil)){
            elHibernateUtil = new HibernateUtil();
        }
        return elHibernateUtil.laSessionFactory;
    }
    private HibernateUtil(){// constructor privado que da valor a los atributos
        try{// carga el fichero de configuración hibernate.cfg.xml y crea un objeto SessionFactory
            laSessionFactory = new Configuration().configure().buildSessionFactory();
        }catch (HibernateException e){
            throw new ExceptionInInitializerError(e);
        }
    }
    public static void shutdown() { //Cierra la sesión
        if (getSessionFactory().isOpen()) getSessionFactory().close();
    }
}
```

3°. Mapeo de una entidad

- Una entidad es una clase de Java o *POJO* que queremos persistir en una base de datos.
- La clase Java deberá tener una implementación equivalente a la tabla en la que recuperará y almacenará los datos desde la aplicación Java. En nuestro caso, la tabla será Contactos.
- Crearemos la clase **Contacto** que deberá tener las siguientes características:
 - Deben tener un **constructor público sin** ningún tipo de **argumentos**.
 - Para cada propiedad que queramos persistir debe haber un método **get/set** asociado menos el atributo que refleja al campo clave de la tabla, que solo deberá tener un método **get**.
 - Implementar el interfaz **Serializable**. Esta condición no es imprescindible, pero sí se considera una buena práctica.


```
public class Contacto implements Serializable {  
    private int id;  
    private String nombre;  
    private String email;  
    private String telefono;
```

```
    Contacto() {}
```

```
    public Contacto(String nombre, String email, String telefono) {  
        this.nombre = nombre;  
        this.email = email;  
        this.telefono = telefono;  
    }
```

```
    public String getEmail() {  
        return email;  
    }
```

```
    public void setEmail(String email) {  
        this.email = email;  
    }
```

```
    public long getId() {  
        return id;  
    }
```

Constructor sin parámetros, ya que *Hibernate* creará instancias de esta clase usando *reflexion* cuando recupere las entidades de la BD.

```
    public String getNombre() {  
        return nombre;  
    }
```

```
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }
```

```
    public String getTelefono() {  
        return telefono;  
    }
```

```
    public void setTelefono(String telefono) {  
        this.telefono = telefono;  
    }
```

Debemos utilizar en la tabla un identificador único de cada registro para poder explotar todas las funcionalidades que nos ofrece Hibernate. Este identificador no deberá ser usado directamente desde el código, sino que debe ser la BD quien lo genere al guardar la entidad e Hibernate quien lo asigne al objeto, por ello, no hay que implementar el setter de *id*.

- Para **mapear la clase** que queremos persistir en la base de datos, se deben utilizar **anotaciones** que le indican a **Hibernate** con qué **tabla y campos** de la misma se corresponden las clases y sus atributos, respectivamente.
- Las anotaciones pertenecen al paquete **javax.persistence**.
- En la clase **Contacto** debemos agregar la siguiente información

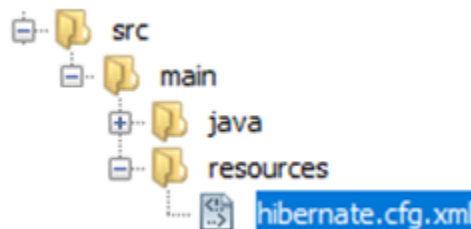
```
@Entity
@Table (name = "contactos")
public class Contacto implements Serializable {

    @Id //pk
    @GeneratedValue(strategy = GenerationType.IDENTITY) //autoincremental
    private int id;
    @Column(name = "nombre", length = 50, nullable = false)
    private String nombre;
    @Column(name = "email", length = 25, nullable = false)
    private String email;
    @Column(name = "telefono", length = 15, nullable = false)
    private String telefono;

    //El resto de la clase Contactos no necesita modificaciones
```

4º. Fichero de configuración de Hibernate (*hibernate.cfg.xml*)

- Es necesario crear un fichero de configuración **hibernate.cfg.xml** donde se encuentra la información relevante como:
 - Conector JDBC a la base de datos con su usuario y contraseña.
 - Localización de las clases de persistencia del proyecto.
 - Lenguaje de comunicación con la base de datos.
- El fichero de configuración **hibernate.cfg.xml** en **NetBeans** (Maven) debe estar almacenado en la carpeta del proyecto **\src\main\resources**.



```
<!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>

    <!-- parametros para la conexion a la base de datos -->
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost:3306/agenda
    </property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password"></property>

    <!-- lenguaje SQL con el que Hibernate se comunicará con la BD. -->
    <property name="hibernate.dialect">
      org.hibernate.dialect.MySQL5Dialect
    </property>

    <!-- Archivos de mapeo donde está la información de la clase vs tabla -->
    <mapping class="gal.teis.negocio.Contacto"/>

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

5°. Crear una clase con las operaciones sobre la BD desde Hibernate

- Crearemos una clase **ContactosDAO** basándonos en el patrón [DAO](#) (*Data Access Object*) que permitirá tener todas las operaciones de acceso a la BD en una clase.
- Las operaciones que implican acceso a la BD son:
 1. **Obtener un objeto Session.** Antes de realizar una operación en la BD se obtiene un objeto `Session` con el método `openSession()` de un objeto `SessionFactory` a partir de la clase `HibernateUtil`.
 2. **Obtejer un objeto Transaction.** A partir del objeto `Session` se obtiene un objeto de tipo `Transaction`.
 3. Ejecutar la operación de *Insertar, actualizar, eliminar y consultar* datos de la BD.
 4. Cerrar la transacción (`commit`).
 5. Cerrar la sesión (`close`)



```
public class ContactosDAO {
```

```
    private Session sesion;  
    private Transaction transa;
```

```
    //Crea una sesión y transacción en la BD. 1 por cada operación
```

```
    private void iniciaOperacion() throws HibernateException {  
        sesion = HibernateUtil.getSessionFactory().openSession();  
        transa = sesion.beginTransaction();  
    }
```

```
    private void manejaExcepcion(HibernateException he)  
        throws HibernateException {  
        transa.rollback();  
        throw new HibernateException("Error en ContactosDAO", he);  
    }
```

```
    public int guardaContacto(Contacto contacto) {  
        int id = 0;  
        try {  
            iniciaOperacion();  
            //guarda el contacto en la BD y devuelve el id generado  
            id = (int) sesion.save(contacto);  
            transa.commit();  
        } catch (HibernateException he) {  
            manejaExcepcion(he);  
            throw he;  
        } finally {  
            sesion.close();  
        }  
        return id;  
    }  
}
```

```
public void actualizaContacto(Contacto contacto) throws HibernateException {
    try {
        iniciaOperacion();
        //actualiza el contacto en la base de datos
        sesion.update(contacto);
        transa.commit();
    } catch (HibernateException he) {
        manejaExcepcion(he);
        throw he;
    } finally {
        sesion.close();
    }
}
```

```
public void eliminaContacto(Contacto contacto) throws HibernateException {
    try {
        iniciaOperacion();
        //elimina el contacto en la base de datos
        sesion.delete(contacto);
        transa.commit();
    } catch (HibernateException he) {
        manejaExcepcion(he);
        throw he;
    } finally {
        sesion.close();
    }
}
```

```
public Contacto obtenContacto(int idContacto) throws HibernateException {

    Contacto contacto = null;
    try {
        iniciaOperacion();
        //Obtiene un contacto de la BD por su id
        contacto = (Contacto) sesion.get(Contacto.class, idContacto);
    } finally {
        sesion.close();
    }
    return contacto;
}

public List<Contacto> obtenListaContactos() throws HibernateException {

    List<Contacto> listaContactos = null;

    try {
        iniciaOperacion();
        //Obtiene una lista de contactos utilizando el lenguaje HQL
        listaContactos = sesion.createQuery("from Contacto").list();
    } finally {
        sesion.close();
    }

    return listaContactos;
}

}
```


Practica Inmobiliaria 1

- Crear un proyecto de Hibernate para gestionar la base de datos MySQL llamada **Inmobiliaria**.
- Crear la tabla **Propietarios** con la siguiente instrucción SQL.

```
CREATE TABLE IF NOT EXISTS Propietarios(  
    id int(11) PRIMARY KEY AUTO_INCREMENT,  
    prDNI CHAR(9),  
    prNombre VARCHAR(15),  
    prApellidos VARCHAR(40),  
    prDireccion VARCHAR(50),  
    prTelefono CHAR(9))
```

- Implementar las operaciones CRUD sobre la tabla.
- Construir un menú para acceder a las operaciones CRUD

Exigencias de Hibernate a las tablas y clases

- **Todas las tablas deben tener clave primaria.**
 - Las tablas de las bases de datos relacionales no tienen la obligación de tener clave primaria, aunque sí que es muy aconsejable. Pero las tablas que se usan con Hibernate sí tienen esa exigencia.
- **Los valores de la clave primaria no pueden cambiar.**
 - Es aconsejable tener siempre una clave primaria autogenerada para evitar problemas en el caso de introducir un dato erróneo o un posible cambio de valor.
- **Las clases mapeadas deben implementar la interfaz *Serializable*.**
- **Las clases deben tener métodos *getX* y *setX* para obtener y asignar valores a los atributos.**

Operaciones persistentes I

- Una clase para la que se establecen correspondencias mediante **Hibernate** se denomina **clase persistente** y una instancia de una clase persistente se llama **objeto persistente**.
- Una **sesión** (`interfaz org.hibernate.Session`) se construye sobre una **conexión** a una BD y determina el **contexto de persistencia**.
- Las **transacciones** (`interfaz org.hibernate.Transaction`) se crean a partir de una sesión (`objetoSession.beginTransaction()`). La sesión debe validar cuando se hay completado la transición.
 1. **Abrir la sesión** (`laSesionAbierta = laSesion.openSession()`)
 2. **Iniciar la transición** (`laSesionAbierta.beginTransaction()`)
 3. **Realizar las operaciones de persistencia** (`save, delete, update, etc.`)
 4. **Validar la transacción** (`laTransaction.commit()`)
 5. **Finalizar la sesión** (`laSesion.close()`)
- Los cambios que se realizan sobre los **objetos persistentes** (*las clases con anotaciones que generan objetos que se almacenan en las tablas*) se reflejan en la base de datos asociada.

Operaciones persistentes II

Dada una sesión abierta de Hibernate en una BD que almacena la información de la clase **Contacto**, suponemos que tenemos creado el objeto persistente siguiente:

```
Contacto contacto = new Contacto("C1", "cont1@contacto.com", "7564356");
```

Para realizar las operaciones CRUD en la BD utilizaremos las siguientes operaciones:

```
sesion.save(contacto);
```

Guardar el objeto persistente en la BD
Devuelve el identificador generado al grabar el objeto
`int id = (int) sesión.save()`

```
sesion.get(contacto, idContacto
```

Recupera un objeto con
identificador **idContacto** de la BD

```
sesion.delete(contacto);
```

Eliminar el objeto persistente de la BD
Previamente se debe recuperar el **contacto** para
comprobar que existe y **set** el objeto con el **id** correcto

```
sesion.update(contacto);
```

Actualiza un registro de la BD
Previamente se debe recuperar el **contacto** para
comprobar que existe y **set** el objeto con el **id**
correcto y los atributos que se deseen modificar

https://github.com/estherff/AD-UD3-Hibernate_1

Ciclo de vida de los objetos persistentes

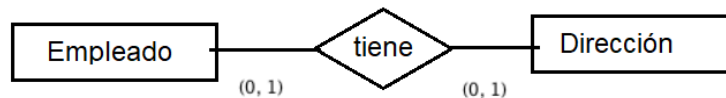
- Una clase para la que se establecen correspondencias mediante **Hibernate** se denomina **clase persistente** y una instancia de una clase persistente se llama **objeto persistente**.
- Una **sesión** (`interfaz org.hibernate.Session`) se construye sobre una **conexión** a una BD y determina el **contexto de persistencia**.
- Las **transacciones** (`interfaz org.hibernate.Transaction`) se crean a partir de una sesión (`objetoSession.beginTransaction()`). La sesión debe cerrar cuando se hay completado la transición (`commit()`)
- Los cambios que se realizan sobre los **objetos persistentes** (*las clases con anotaciones que generan objetos que se almacenan en las tablas*) se reflejan en la base de datos asociada.
- Los objetos persistentes pueden estar en los siguientes estados:
 - **Transitorio** (*transient*). El objeto se acaba de crear con **new** y no está asociado con ningún contexto de persistencia. No está grabado en la base de datos.
 - **Gestionado** (*managed*). El objeto tiene un identificador y está asociado con un contexto de persistencia. Está grabado en la base de datos. Cualquier cambio en el objeto se reflejará en la base de datos en el momento en que se cierre la sesión.
 - **Separado** (*detached*). El objeto tiene un identificador asociado, pero ya no está asociado con un objeto de persistencia, esto puede ser debido a que el contexto se ha cerrado o porque se ha desvinculado el objeto del contexto.
 - **Eliminado**: la entidad tiene un identificador asociado y está asociada con un contexto de persistencia, sin embargo, está programada para su eliminación de la base de datos.

Uso de anotaciones en Hibernate I

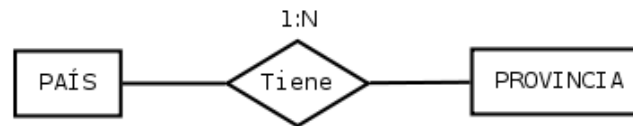
- Las anotaciones convierten nuestras clases (POJO) en objetos persistentes.
- Para usar entidades debemos importar la clase `import javax.persistence.*` y así podremos utilizar las siguientes anotaciones.
 - **@Entity** → Indica que la clase es un tabla en la base de datos.
 - **@Table(name = "nombre_tabla", catalog = "nombre_base_datos")**
Indica el nombre de la tabla y la BD a la que pertenece (este último parámetro no es necesario ya que esa información viene en el fichero de configuración)
 - **@Id** → antes del atributo donde se almacenará el identificador del objetos persistente.
 - **@GeneratedValue(strategy = GenerationType.IDENTITY)** → Indica que es un valor autonumérico (PRIMARY KEY en MySQL, por ejemplo).
 - **@Column(name = "nombre_columna")** → Se utiliza para indicar el nombre de la columna en la tabla donde debe ser mapeado el atributo

Uso de anotaciones en Hibernate II

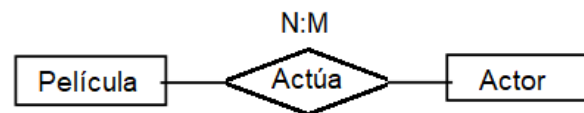
- Las anotaciones relacionadas con las relaciones son:
 - @OneToOne**. Indica una relación de uno a uno.



- @OneToMany**. Indica una relación unidireccional de uno a muchos (1-N)



- @ManyToMany**. Indica una relación unidireccional de muchos a uno (N-N). *Este tipo de relaciones no se recomiendan, se deben transformar en 1-N.*





HQL

- **HQL** (*Hibernate Query Language*) es el lenguaje de consultas orientado a objetos para las operaciones de consulta de una base de datos utilizando **Hibernate**.
- Es una versión del lenguaje SQL para manejar objetos.
- Se caracteriza por:
 - Utiliza clases y propiedades en lugar de tablas y columnas.
 - Soporta polimorfismo.
 - Permite establecer relaciones (*one-to-one*, *one-to-many*, *many-to-many*).
- Da soporte a las operaciones relacionales como:
 - Inner/outer/full joins, producto cartesiano.
 - Seleccionar campos.
 - Operaciones de agregación (MAX, AVG) y agrupamiento (GROUP BY)
 - Ordenación (ORDER BY)
 - Subconsultas.
 - Llamadas a funciones.

HQL. Query I

- La consulta más sencilla es

from Empleado

- Siendo **Persona** una clase mapeada
- Por ejemplo:

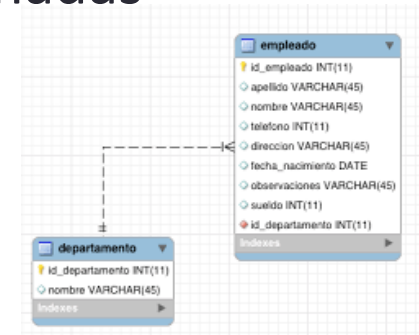
```
List lasPersonas = sesion.createQuery("from Empleados").list()
```

- Un ejemplo de una consulta en dos tablas relacionadas

```
from Empleado emp
join emp.departamento dep
where emp.formación like '%java%'
and dep.nombre = "desarrollo"
```

Devuelve una lista (List) con instancias de la clase **Empleado** que se encuentra en la tabla **empleados** indicada en las anotaciones

```
@Entity
@Table (name = "empleados")
public class Empleado implements Serializable {
```



Devuelve todos los empleados que tengan formación en *java* y formen parte del departamento de *desarrollo*.

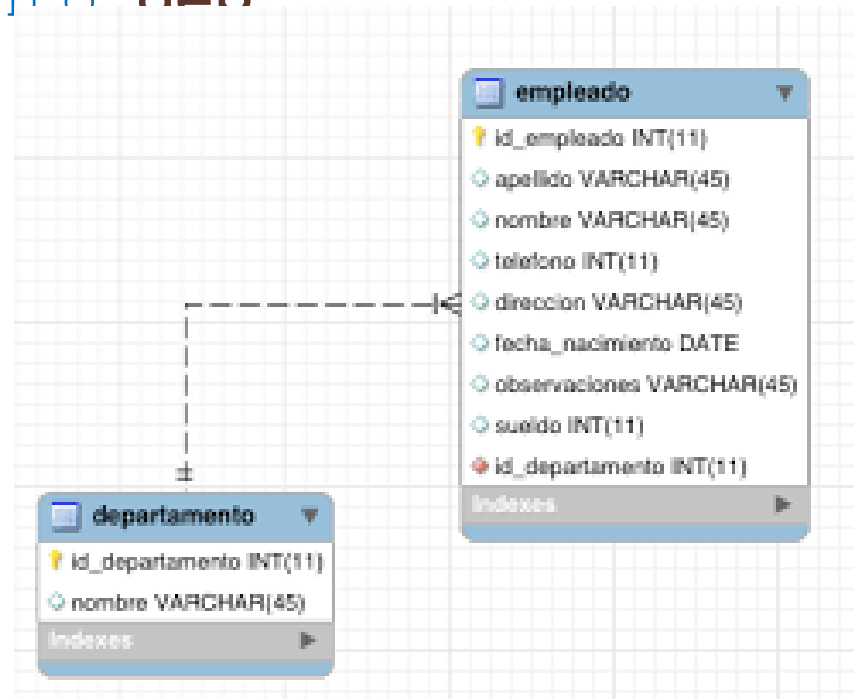
Empleado y **Departamento** tienen establecida una relación de **1 a varios**, por ello la clase mapeada **Empleado** tienen un atributo de tipo **Departamento** que hace referencia a una clave externa de **id_departamento** en la tabla **empleados**.

HQL. Query II

- Un ejemplo de una consulta en dos tablas relacionadas donde se selecciona los campos que se quieren recuperar.

```
select emp.nombre, dep.nombre  
from Empleado emp  
join emp.departamento on dep  
where emp.sueldo >
```

Devuelve el **nombre** de los **empleados** y de su **departamento** correspondiente de aquellos **empleados** que su **sueldo** sea **mayor** de 2.000



HQL. Query III

- Un ejemplo de una consulta de agregación.

```
select dep.nombre, max(emp.sueldo) , count(emp)
from Empleado emp
join emp.departamento dep
group by emp.departamento
order by dep.nombre
```

Devuelve el **nombre del departamento**, el **sueldo máximo** y el **número de empleados** para cada departamento y ordenado por el **nombre de departamento**

HQL, parámetros

- En el caso de necesitar parámetros en una consulta HML podemos utilizar identificadores de la forma **:name** en la cadena de la consulta.
- El valor del parámetro se asigna con el método **setParameter("name", vble)** donde **name** es el identificador del parámetro y **vble** su valor
- **Por ejemplo**, la siguiente consulta utiliza el parámetro **:idP** consulta

```
(from Direccion dire inner join dire.empleado as emp
where emp.id = :idP) .
setParameter("idP", idBuscar) .list() ;
```

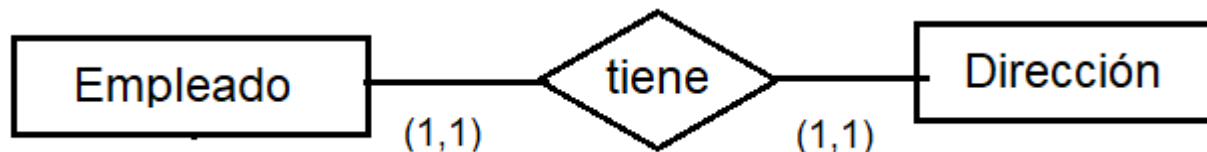
Para saber más de HQL → [enlace1](#) [enlace2](#)

Crear una BD con una relación 1 a 1

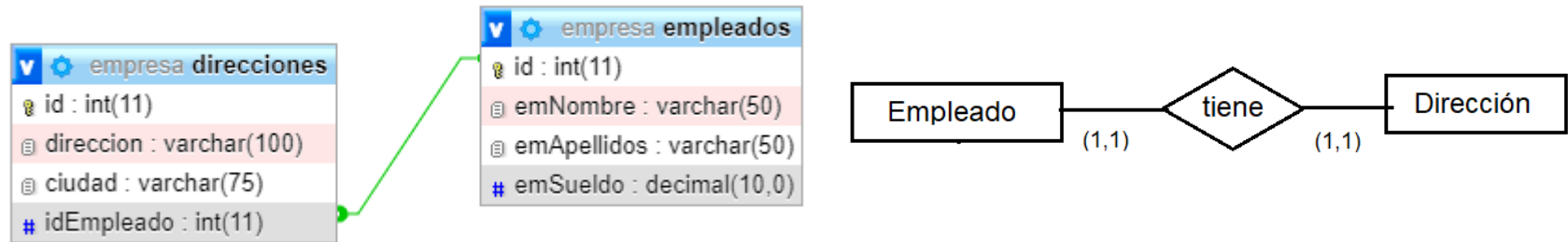
- Crear una BD llamada **empresa** con las tablas **empleados** y **direcciones**

```
CREATE TABLE empleados (  
  id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
  emNombre varchar(50) NOT NULL,  
  emApellidos varchar(50) NOT NULL,  
  emSueldo decimal(10,0) NOT NULL  
)ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

```
CREATE TABLE direcciones (  
  id int(11) NOT NULL,  
  direccion varchar(100) NOT NULL,  
  ciudad varchar(75) NOT NULL,  
  idEmpleado int(11) NOT NULL,  
  FOREIGN KEY (idEmpleado) REFERENCES empleados (id)  
  ON DELETE CASCADE ON UPDATE CASCADE  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```



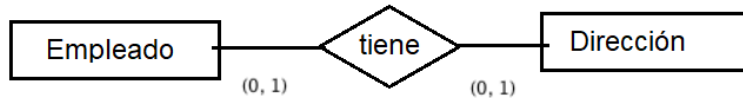
Análisis de la relación entre las tablas *empleados* y *direcciones*



- Cada empleado tiene solo una dirección y cada dirección corresponde solo a un empleado.
- Se ha establecido esta relación por medio de una clave externa (**idEmpleado**) en la tabla *direcciones* que se relaciona con la clave principal (**id**) de *empleados*.
- Se ha establecido ciertas restricciones en la relación entre *direcciones* y *empleados* que determinan que si se elimina un registro de *empleados* se eliminarán automáticamente el registro relacionado en la tabla *direcciones*.

FOREIGN KEY ('idEmpleado') REFERENCES empleados ('id')
ON DELETE CASCADE ON UPDATE CASCADE

Mapeado de una relación unidireccional 1 a 1



```
@Entity
@Table (name = "empleados")
public class Empleado implements Serializable {

    @Id //pk
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column (name = "id")
    private int id;

    @Column(name = "emNombre")
    private String nombre;

    @Column(name = "emApellidos")
    private String apellidos;

    @Column(name = "emSueldo")
    private Double sueldo;

    Empleado() {
    }
```

```
@Entity
@Table (name = "direcciones")
public class Direccion implements Serializable {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column (name = "id")
    private int id;
```

```
    @Column(name = "direccion")
    private String direccion;
```

```
    @Column(name = "ciudad")
    private String ciudad;
```

```
    @OneToOne
    @JoinColumn(name = "idEmpleado")
    private Empleado empleado;
```

```
    Direccion() {
    }
```

Se crea un atributo de tipo **Empleado** en **@JoinColumn** indicamos mediante qué campo de la tabla se relaciona con la tabla **empleados**

Es una relación unidireccional pues desde **Empleado** podemos obtener **Dirección**, pero no a la inversa

Guardar Empleado/Direccion

```
//1° Se crear instancia de Empleado y se guarda en la BD  
Empleado emp1 = new Empleado("Rosa", "Pin", 2000.0);  
Empleado emp2 = new Empleado("Alberto", "Pereira", 2500.0);
```

```
EmpresaDAO.guardaEmpleado(emp1);  
EmpresaDAO.guardaEmpleado(emp2);
```

Guardo la instancia de Empleado al crear el objeto Dirección (la clave externa está en Dirección y se mapea con un objeto)

```
//2° Se crea instancia de dirección, se hace setter a  
Empleado y se almacena en la BD  
Direccion dire1 = new Direccion("C/Sol n° 5", "Vigo");  
Direccion dire2 = new Direccion("C/Principal n° 30", "Vigo");
```

```
//3° Guardo el empleado en el atributo que hay en Dirección  
dire1.setEmpleado(emp1);  
dire2.setEmpleado(emp2);
```

```
//4° Almaceno cada dirección (cada una lleva a su empleado)  
EmpresaDAO.guardaDireccion(dire1);  
EmpresaDAO.guardaDireccion(dire2);
```


Listar Empleados+Direccion I

```
//Al acceder a dos tablas relacionadas, el resultado será un listado de
//arrays de objetos List<Object[]>
List<Object[]> listaEmpleadosDireccion = EmpresaDAO.obtenerListaEmpDirec();

if (!Objects.isNull(listaEmpleadosDireccion)) {
    System.out.println("Hay " + listaEmpleadosDireccion.size()
        + " empleados en la base de datos");
    for (int i = 0; i < listaEmpleadosDireccion.size(); i++) {
        //Para cada posición de la lista, recupera el índice 0 que son los
        //datos de la dirección y el índice 1 que son los datos del empleado
        System.out.println(listaEmpleadosDireccion.get(i)[0]
            + listaEmpleadosDireccion.get(i)[1]);
    }
}
```

En cada posición de la lista hay un array que tiene en la posición 0 la Dirección y en la posición 1 el Empleado. Este es el orden pues se almacenó el objeto Direccion con un atributo que contiene Empleado

Listar Empleados+Direccion II

```
public static List<Object[]> obtenerListaEmpDirec()  
                                throws HibernateException {  
    List<Object[]> empleadoDireccion = null;  
  
    try {  
        iniciaOperacion();  
        // Recuperamos una lista de arrays (.list()) de objetos  
        // pues la consulta incluye dos tablas.  
        empleadoDireccion = sesion.createQuery("from Direccion as dire"  
                                                + " inner join dire.empleado as emp").list();  
    } finally {  
        sesion.close();  
    }  
    return empleadoDireccion;  
}
```

Instrucción HQL para acceder a dos tablas relacionadas 1 a 1 de forma unidireccional.

“ver elementos mapeados con la clase Direccion (alias dire) enlazada con dire.empleado (alias emp)”

Localizar un empleado sin dirección a partir de su id

```
public static Empleado obtenEmpleado(int id) {  
    Empleado empleado = null;  
    boolean obtenido = false;  
  
    try {  
        //abre la sesión e inicia la transición  
        iniciaOperacion();  
        //obtiene un objeto Empleado a partir de su id  
        empleado = sesion.get(Empleado.class, id);  
        //ejecuta la transacción  
        transa.commit();  
    } catch (HibernateException he) {  
        manejaExcepcion(he);  
    } finally {  
        sesion.close();  
    }  
    return empleado;  
}
```

Localizar un empleado+dirección a partir de su id I

```
//Al acceder a dos tablas relacionadas, el resultado será un  
listado de //arrays de objetos List<Object[]>  
List<Object[]> empleadosDireccion = EmpresaDAO.obtenEmplDirec(id);  
  
if (!Objects.isNull(empleadosDireccion)) {  
    //Solo se accede a un elemento con la orden HQL, por lo tanto solo  
    //se tiene en cuenta la primera posición de la lista.  
  
    System.out.println(empleadosDireccion.get(0)[0] +  
                        empleadosDireccion.get(0)[1]);  
}
```

Localizar un empleado+dirección a partir de su id II

```
public static List<Object[]> obtenEmplDirec(int idBuscar) {  
    List<Object[]> empleadoDireccion = null;  
    boolean obtenido = false;  
  
    try {  
        //abre la sesión e  
        //inicia la transición  
        iniciaOperacion();  
  
        empleadoDireccion = sesion.createQuery("from Direccion as dire "  
            + "inner join dire.empleado as emp where emp.id = :idP").  
            setParameter("idP", idBuscar).list();  
        //query.uniqueResult();  
        transa.commit();  
    } catch (HibernateException he) {  
        manejaExcepcion(he);  
    } finally {  
        sesion.close();  
    }  
    return empleadoDireccion;  
}
```

Los parámetros se ponen en la cadena HQL como :nombre y después se da valor con `setParameter("nombre", valor)`

Instrucción HQL para acceder a dos tablas relacionadas 1 a 1 de forma unidireccional.

“ver elementos mapeados con la clase Direccion (alias dire) enlazada con dire.empleado (alias emp) donde el id de empleado sea igual a uno dado”

Eliminar empleado

- El proceso de eliminar un empleado de la base de datos es similar al realizado cuando eliminamos un elemento de una tabla no relacionada.
- En este caso, la diferencia estará en la consecuencia que tiene eliminar un empleado.
- Eliminar un empleado implica eliminar su dirección de forma automática debido a las relaciones establecidas y a las condiciones de ésta.

FOREIGN KEY ('idEmpleado') REFERENCES empleados ('id')
ON DELETE CASCADE ON UPDATE CASCADE

Actualizar Empleado

- El proceso de actualización de un empleado de la base de datos es similar al realizado cuando eliminamos un elemento de una tabla no relacionada
 1. Obtener, en un objeto, el elemento de la base de datos con **get**.
 2. Realizar la actualización de datos sobre el objeto.
 3. Realizar una operación **session.update (objetomodificado)**



[Enlace al proyecto en GitHub](#)

```
System.out.println("Introduce el id del elemento a modificar ");
int id3 = ControlData.lerInt(sc);
System.out.println("Introduce el nuevo nombre del empleado");
String nombre = ControlData.lerString(sc);
List<Object[]> empDire1 = EmpresaDAO.obtenEmplDirec(id3);
if (!Objects.isNull(empDire1)) {
    Empleado empActualizar = ((Empleado) (empDire1.get(0)[1]));
    empActualizar.setNombre(nombre);
    EmpresaDAO.actualizaEmpleado(empActualizar);
}
```

```
public static void actualizaEmpleado(Empleado empleado)
    throws HibernateException {

    try {
        iniciaOperacion();
        sesion.update(empleado);
        transa.commit();
    } catch (HibernateException he) {
        manejaExcepcion(he);
    } finally {
        sesion.close();
    }
}
```

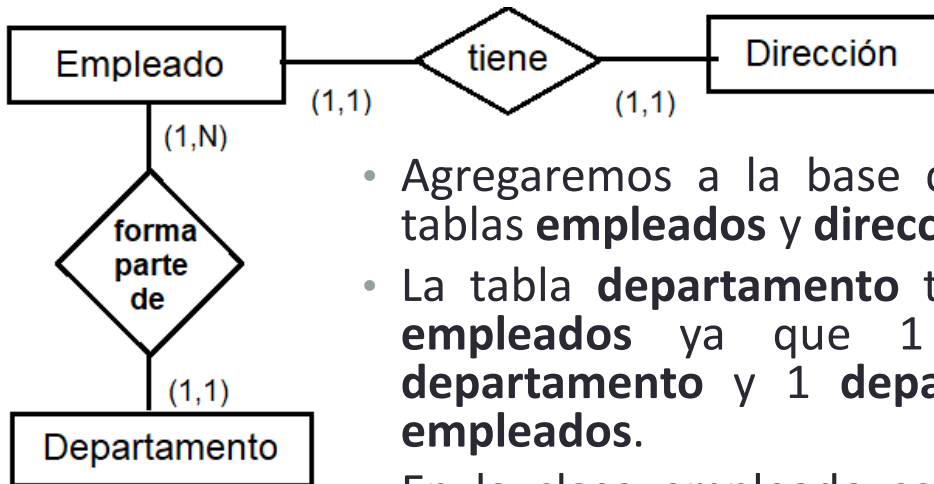

Practica Inmobiliaria 2

- Crear la tabla **DatosBancarios** que tienen establecida una relación **OneToOn** con **Propietarios**.
- Al **eliminar Propietario** se debe eliminar su datos bancarios.
- La tabla de **DatosBancarios** se creará con la siguiente estructura:

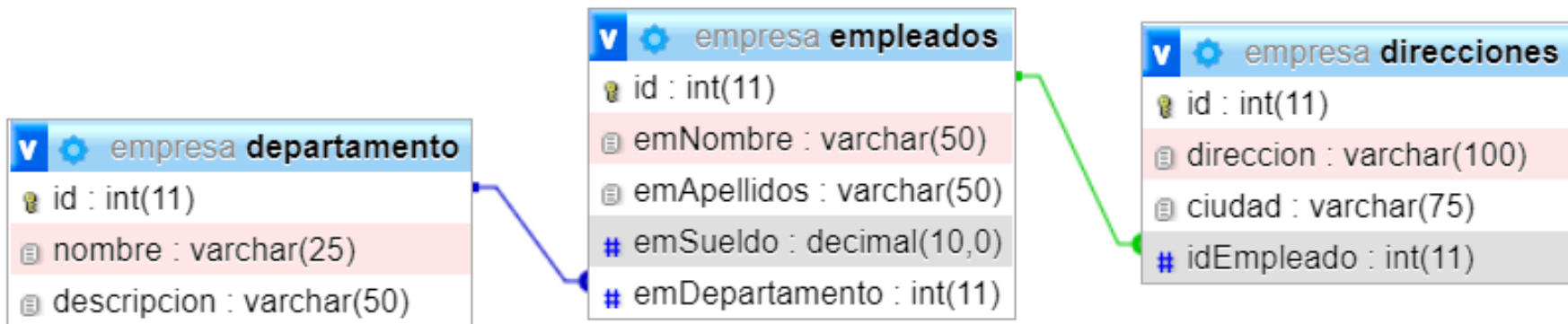
```
CREATE TABLE `datosBancarios` (  
  `id` int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
  `numCuenta` varchar(24) NOT NULL,  
  `nombreBanco` varchar(100) NOT NULL,  
  `idPropietario` int(11) NOT NULL,  
  FOREIGN KEY (`idPropietario`) REFERENCES propietarios (`id`)  
ON DELETE CASCADE ON UPDATE CASCADE  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

- Implementa las operaciones:
 - Añadir datos bancarios
 - Eliminar datos bancarios
 - Modificar datos bancarios
 - Ver propietario junto con sus datos bancarios
 - Comprobar que al eliminar el propietario se eliminan sus datos bancarios.

Tablas relacionadas 1 a N

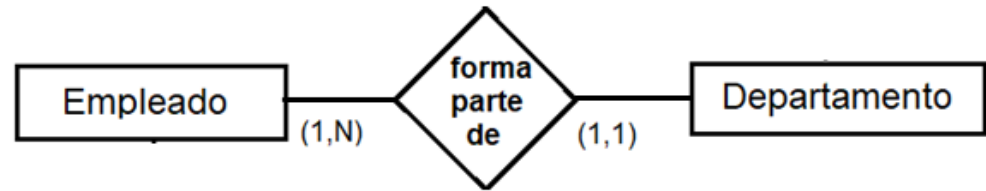


- Agregaremos a la base de datos **empresa**, que ya tenía las tablas **empleados** y **direcciones**, la tabla **departamento**.
- La tabla **departamento** tendrá una relación **1N** con la tabla **empleados** ya que 1 **empleado** solo pertenece a 1 **departamento** y 1 **departamento** está formado por varios **empleados**.
- En la clase empleado se ha definido una clave externa que establece la relación con la tabla departamento
- ...FOREIGN KEY (`idPropietario`) REFERENCES propietarios (`id`)



Mapeado de una relación bidireccional

1 a N (I)



@Entity

@Table(name = "empleados")

```
public class Empleado implements Serializable {
```

@Id //pk

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

```
private int id;
```

@Column(name = "emNombre")

```
private String nombre;
```

@Column(name = "emApellidos")

```
private String apellidos;
```

@Column(name = "emSueldo")

```
private Double sueldo;
```

```
/*Tenemos que asociar el objeto departamento con el empleado,
para ello creamos un atributo con el objeto Empleado*/
```

```
//La cardinalidad es N,1 con empleados
```

@ManyToOne

@JoinColumn(name = "emDepartamento")

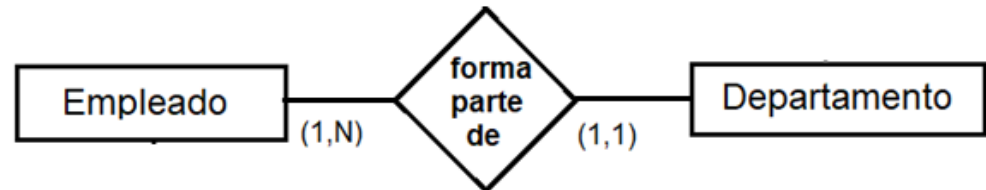
```
private Departamento departamento;
```

```
.....
```

Le indicamos que *muchos empleados* tiene un *departamento* con **@ManyToOne** La columna que tiene la información del **departamento** en la bd es **emDepartamento** que establece una relación con un registro de **departamento**.

Mapeado de una relación bidireccional

1 a N (II)



@Entity

@Table(name = "departamentos")

```
public class Departamento {
```

```
    @Id //pk
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "id")
```

```
    private int id;
```

```
    @Column(name = "nombre")
```

```
    private String nombre;
```

```
    @Column(name = "descripcion")
```

```
    private String descripcion;
```

```
    //La cardinalidad es 1,N con empleados
```

```
    //mappedBy="departamento" hace referencia al nombre de la propiedad
```

```
    //en la clase Departamento (con minúsculas)
```

```
    @OneToMany(mappedBy="departamento")
```

```
    private List<Empleado> empleados = new ArrayList<Empleado>();
```

```
    public List<Empleado> getEmpleados() {
```

```
        return empleados;
```

```
    }
```

```
    public void setEmpleados(List<Empleado> empleados) {
```

```
        this.empleados = empleados;
```

```
    }
```

```
    public Departamento() { }
```

Estableceremos una relación bidireccional, de tal forma que la clase Departamento contenga un atributo de tipo

List<Empleados> para poder listar los empleados a partir de un departamento

La relación @OneToMany (un departamento tiene muchos empleados) se debe indicar delante de un atributo **List<Empleados>**

Orden de operaciones de insercción

- En principio, es necesario **establecer un orden a la hora de introducir los datos** debido a las restricciones provocadas por las relaciones entre las tablas.
- Debemos añadir primero el **Departamento** antes de incorporar empleados que pertenezcan al mismo.

//1º Se crear instancia de Departamento

```
Departamento dep1 = new Departamento ("Comercial", "Se ocupa de las operaciones comerciales");  
DepartamentoDAO.guardaDepartamento(dep1);  
Departamento dep2 = new Departamento ("Marketing", "Maneja y coordina estrategias de venta");  
DepartamentoDAO.guardaDepartamento(dep2);
```

//2º Se crear instancia de Empleado y se guarda en la BD

```
Empleado emp1 = new Empleado("Rosa", "Pin", 2000.0);  
Empleado emp2 = new Empleado("Alberto", "Pereira", 2500.0);  
emp1.setDepartamento(dep1);  
emp2.setDepartamento(dep2);
```

```
EmpleadoDAO.guardaEmpleado(emp1);  
EmpleadoDAO.guardaEmpleado(emp2);
```

//3º Se crea instancia de dirección, se hace setter a Empleado y se almacena en la BD

```
Direccion dire1 = new Direccion("C/Sol nº 5", "Vigo");  
Direccion dire2 = new Direccion("C/Principal nº 30", "Vigo");
```

//4º Guardo el empleado en el atributo que hay en Dirección

```
dire1.setEmpleado(emp1);  
dire2.setEmpleado(emp2);
```

Listado de los departamentos con sus empleados

```
//Obtener una lista de los departamentos con sus empleados
List<Object[]> listaDepEmpleados =
DepartamentoDAO.obtenListaDepEmpleados();
if (!Objects.isNull(listaDepEmpleados)) {
    System.out.println("Hay " + listaDepEmpleados.size() + " empleados en la
base de datos");
    for (int i = 0; i < listaDepEmpleados.size(); i++) {
        System.out.println("Empleado " + i + ": " +
            listaDepEmpleados.get(i)[0] + ", Empleado: " +
            listaDepEmpleados.get(i)[1]);
    }
}
```

https://github.com/estherff/AD-UD3-Hibernate_OneToMany_Bidireccional

Inmobiliaria. Agregar una tabla y una relación 1 a N

- En la práctica debéis agregar la tabla Inmuebles que tendrá una relación 1N con propietarios.

```
CREATE TABLE IF NOT EXISTS Inmuebles(  
    id CHAR(5) PRIMARY KEY,  
    inDireccion VARCHAR(50),  
    inCodZona CHAR(5),  
    inEstado CHAR(1),  
    CONSTRAINT FKPropietario FOREIGN KEY (id)  
    REFERENCES Propietarios (id))
```

Menú 1	Menú 2	Comentarios de operaciones
Propietarios	Buscar/Mostar un propietario por su id	<i>Al localizarlo se debe dar la opción a ver sus datos bancarios</i>
	Agregar propietario	<i>Se debe dar opción a agregar sus datos bancarios</i>
	Eliminar propietario	<i>Se deben eliminar todos los inmuebles relacionados</i>
	Modificar propietario	<i>Se debe dar opción a modificar sus datos bancarios</i>
	Listar todos los propietarios	<i>Sin datos bancarios</i>
	Listar todos los propietarios/inmuebles	<i>Sin datos bancarios</i>
Inmuebles	Buscar un inmueble por su código	<i>Se debe dar opción a ver su propietario</i>
	Agregar inmueble	<i>Un inmueble se agrega a un propietario ya existente</i>
	Eliminar inmueble	
	Modificar inmueble	
	Listar todos los inmuebles/propietarios	