



Desarrollo de Aplicaciones Multiplataforma



Tema 3.

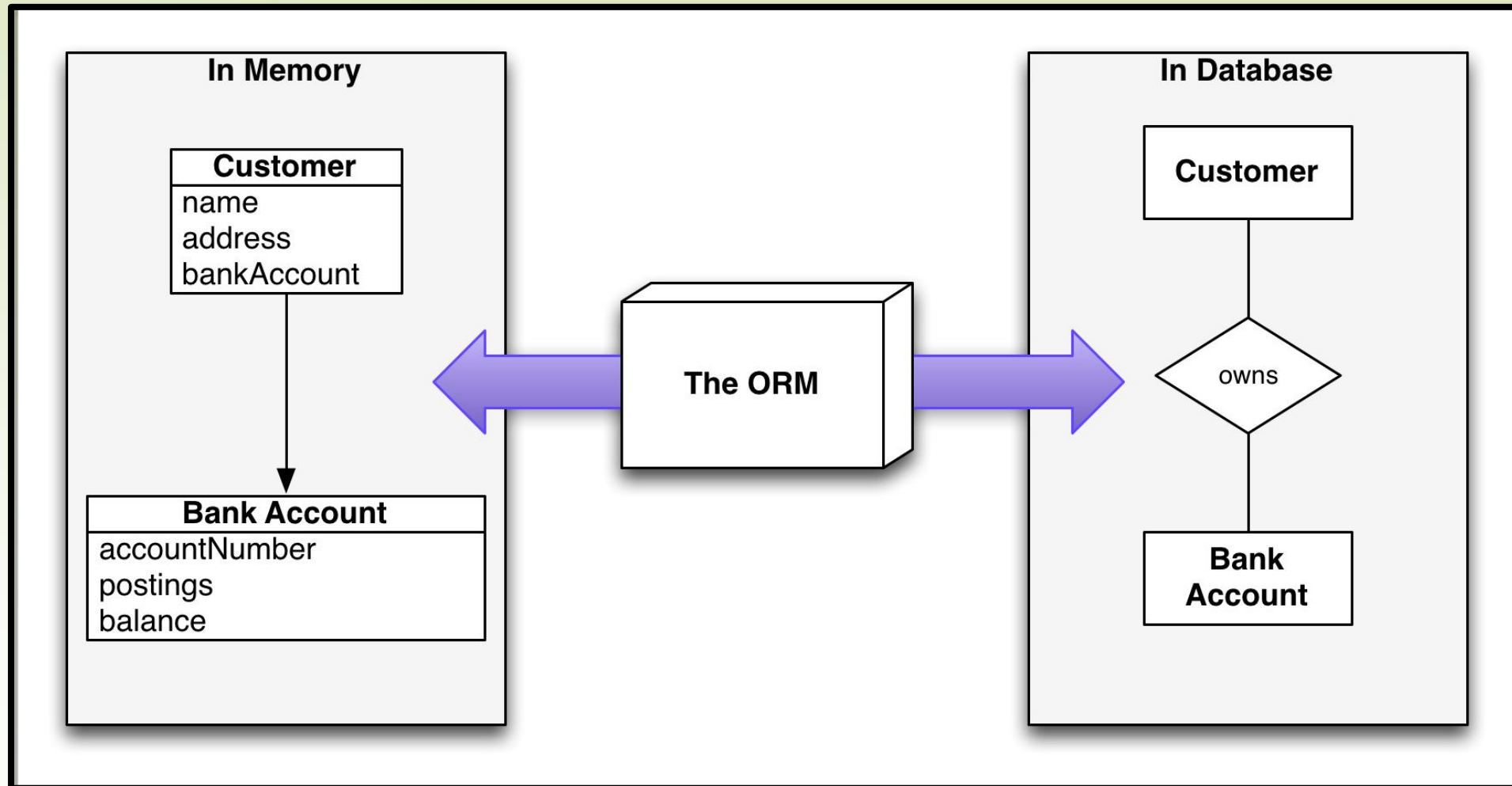
Mapeo Objeto-Relacional

Índice de contenidos

- ❑ Concepto de mapeo objeto relacional.
- ❑ Persistencia ORM con Hibernate.
- ❑ Bases de datos objeto-relacionales.
- ❑ Bases de datos orientadas a objetos.

- ❑ En este tema vamos a exponer soluciones al problema del **desfase objeto-relacional**. La primera solución es el mapeo objeto-relacional (**ORM**).
- ❑ Convierte los objetos de un POO a datos en un sistema relacional. Los datos del objeto se transforman en **datos primitivos** que si se pueden almacena en tablas.
- ❑ Después el ORM se encarga de **construir de nuevo los objetos** a partir de los datos primitivos de las tablas.

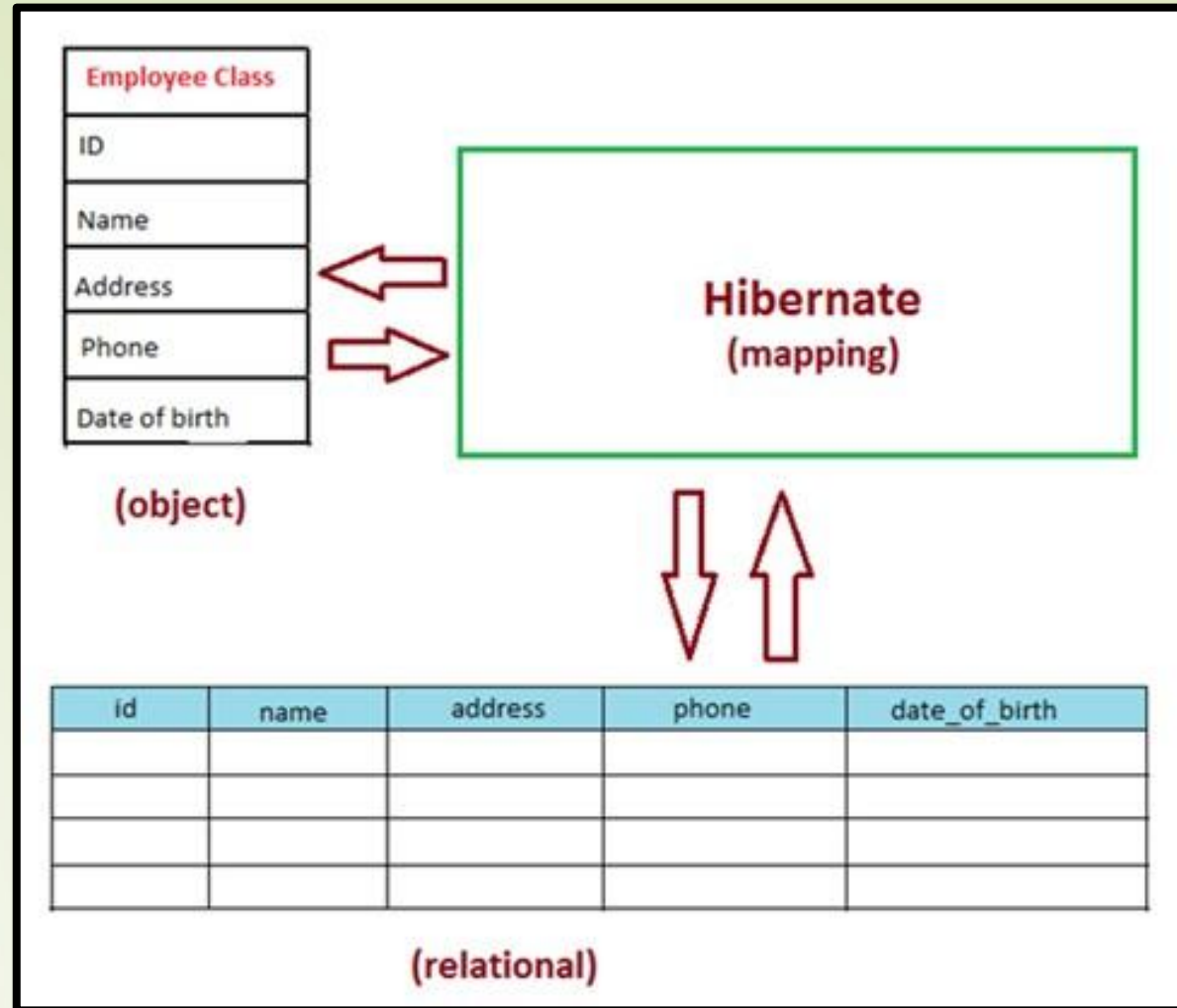
Esquema conceptual del ORM



- ❑ **Rapidez de desarrollo** porque ya no hay que mezclar orientación a objetos con consultas relacionales.
- ❑ **Independencia del sistema gestor** de bases de datos en el desarrollo de la aplicación.
- ❑ La única pega es que nuestra aplicación solo puede manipular los datos y **no su estructura**.
- ❑ Se introduce el concepto de **persistencia** donde los objetos modificados por la aplicación ORM, **automáticamente guardan los cambios en la BD**.

- ❑ **Hibernate** es un software libre bajo licencia GNU de **ORM en Java**, ampliamente utilizado en desarrollos profesionales.
- ❑ **Esta diseñado para ser flexible** en cuanto al esquema de tablas utilizado y para poder adaptarse a su uso sobre una base de datos ya existente.
- ❑ También ofrece un lenguaje de consulta de datos llamado **HQL (Hibernate Query Language)**.
- ❑ El concepto mapeo objeto-relacional también lo veremos en **sistemas NoSQL en siguiente tema**.

- ❑ El mecanismo de funcionamiento ORM de Hibernate se puede resumir en el siguiente esquema.



- ❑ Para hacer ORM en Hibernate se tienen que crear los siguientes ficheros necesarios para el proceso de mapeado:
- ❑ Clases que representan a los objeto almacenados (POJOs)
- ❑ **Fichero de Mapeo HBM** que es un fichero XML con la información para hacer el mapeo.
- ❑ Fichero de configuración **hibernate.cfg** con los datos para que hibernate pueda conectar con la base de datos.

- ❑ Estos ficheros se generan automáticamente desde IntelliJ.
- ❑ Para realizar el mapeo seguiremos la siguiente guía sobre la base de datos **usuariosdb** de ejercicios de temas anteriores.

Tabla: usuarios	
Nombre	Tipo
<u>id</u>	int(4)
login	varchar(20)
pass	varchar(20)
tipo	varchar(10)

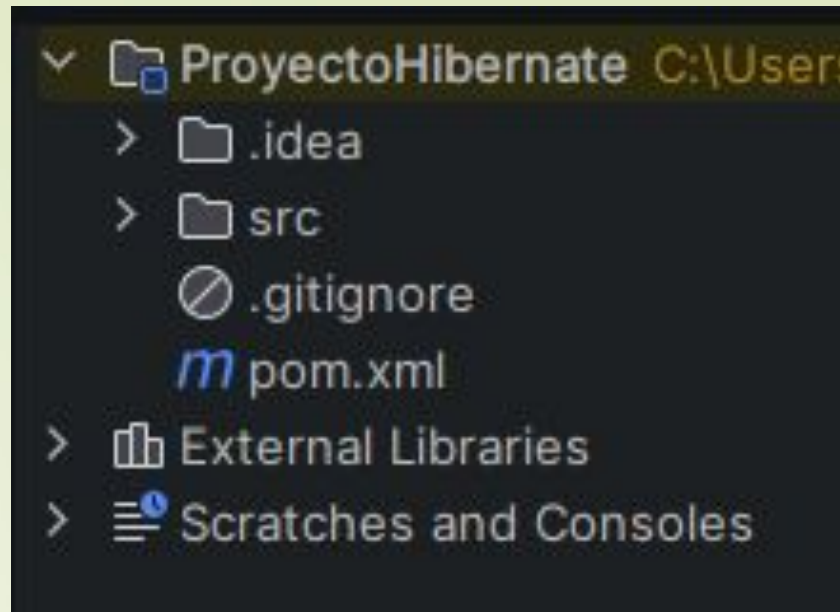
- ❑ A continuación vamos a mostrar una guía paso a paso de como se genera cada fichero.

Requisitos previos

- ❑ Partimos de la base que existe un **SGBD instalado** con la **base de datos creada** y las **tablas creadas**.
- ❑ Tenemos el **conector necesario** o nuestro IDE lo contiene.
- ❑ Disponemos de un trial de IntelliJ IDEA. Ya sea el free trial o la licencia educativa. Así podremos hacer uso del framework hibernate
- ❑ Las dependencias para el conector e hibernate

Añadir dependencias de Hibernate y MySQL

- ❑ Abrimos el proyecto de IntelliJ IDEA, y nos dirigimos al archivo de configuracion maven pom.xml



Añadir dependencias de Hibernate y MySQL

- ❑ En este fichero en el bloque “dependencies” debemos introducir las dependencias. Pero de donde sacamos las dependencias, del repositorio de maven:
- ❑ Hibernate:
<https://mvnrepository.com/artifact/org.hibernate/hibernate-core/5.6.14.Final>
- ❑ MySQL:
<https://mvnrepository.com/artifact/mysql/mysql-connector-java/8.0.33>

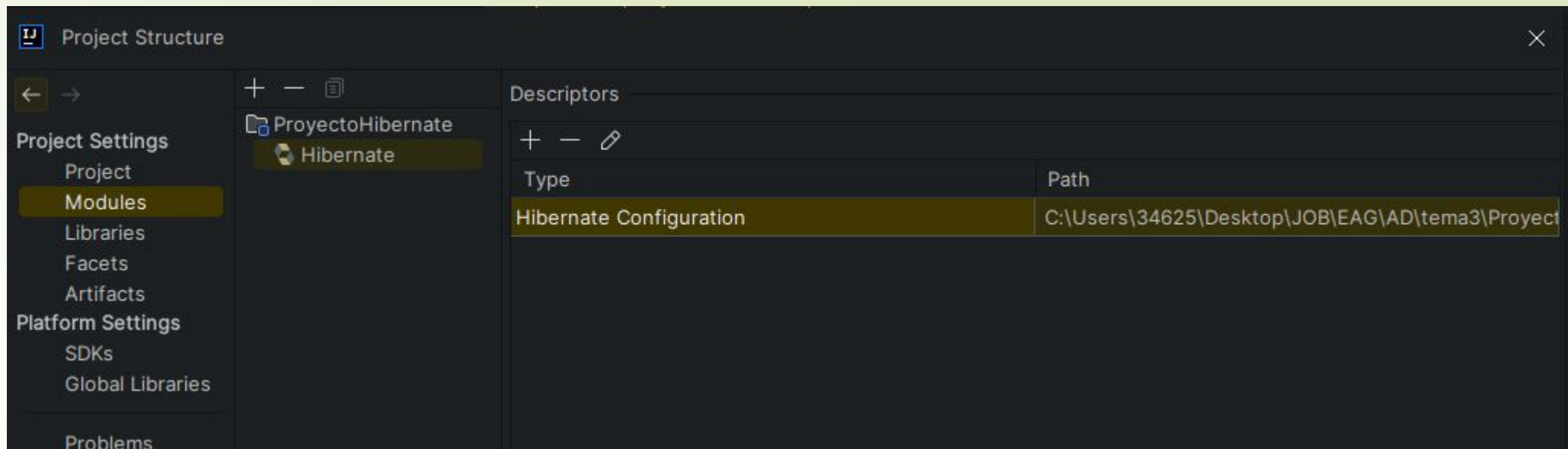
Añadir dependencias de Hibernate y MySQL

- ❑ De esta manera añadimos el código indicado, quedando algo tal que así:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.14.Final</version>
  </dependency>
</dependencies>
</project>
```

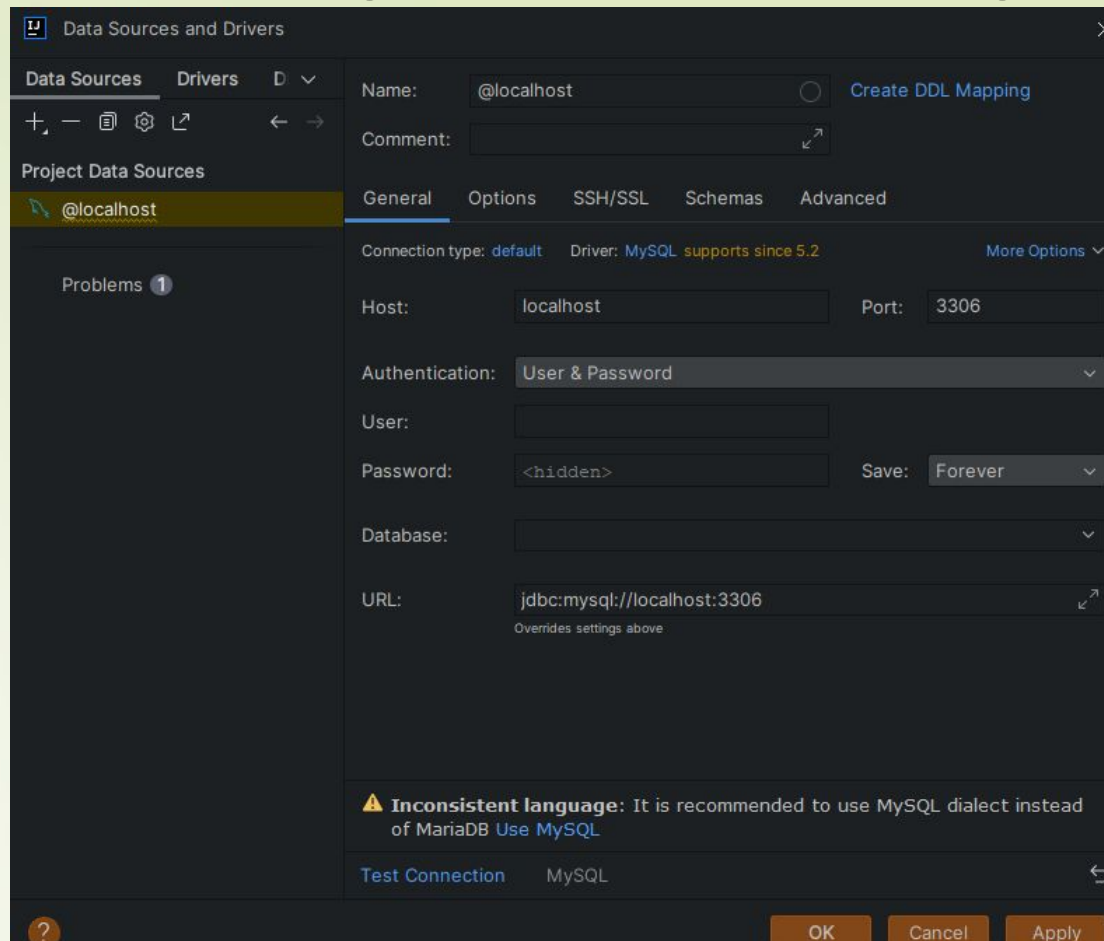
Añadir Faceta y descriptor al módulo

- ❑ Para ello nos vamos a File>Project Structure>Facets
- ❑ Presionamos en el “+”, y añadimos Hibernate
- ❑ Tras esto nos llevará a la pestaña Modules ahí añadiremos el descriptor, fichero donde se van a almacenar los mapeos de Hibernate



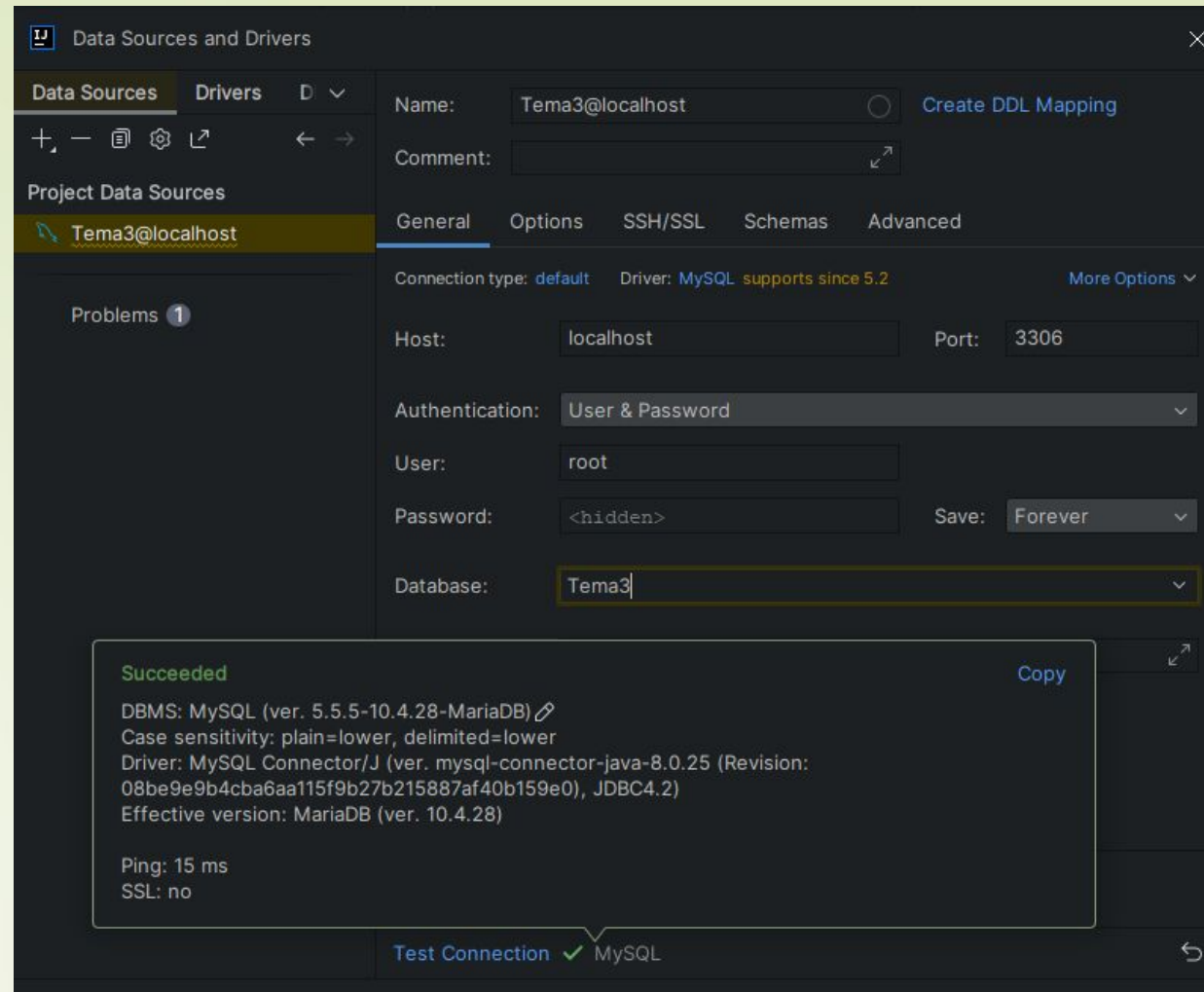
Conexión a la BD

- ❑ Ahora nos iremos al icono de bases de datos, a la derecha pulsamos botón izquierdo y pulsar en nueva conexión. Y nos aseguramos de escoger MySQL



Conexión a la BD

- ❑ Rellenamos los campos de user, password y el nombre de la base de datos. Tras esto podremos comprobar la conexión

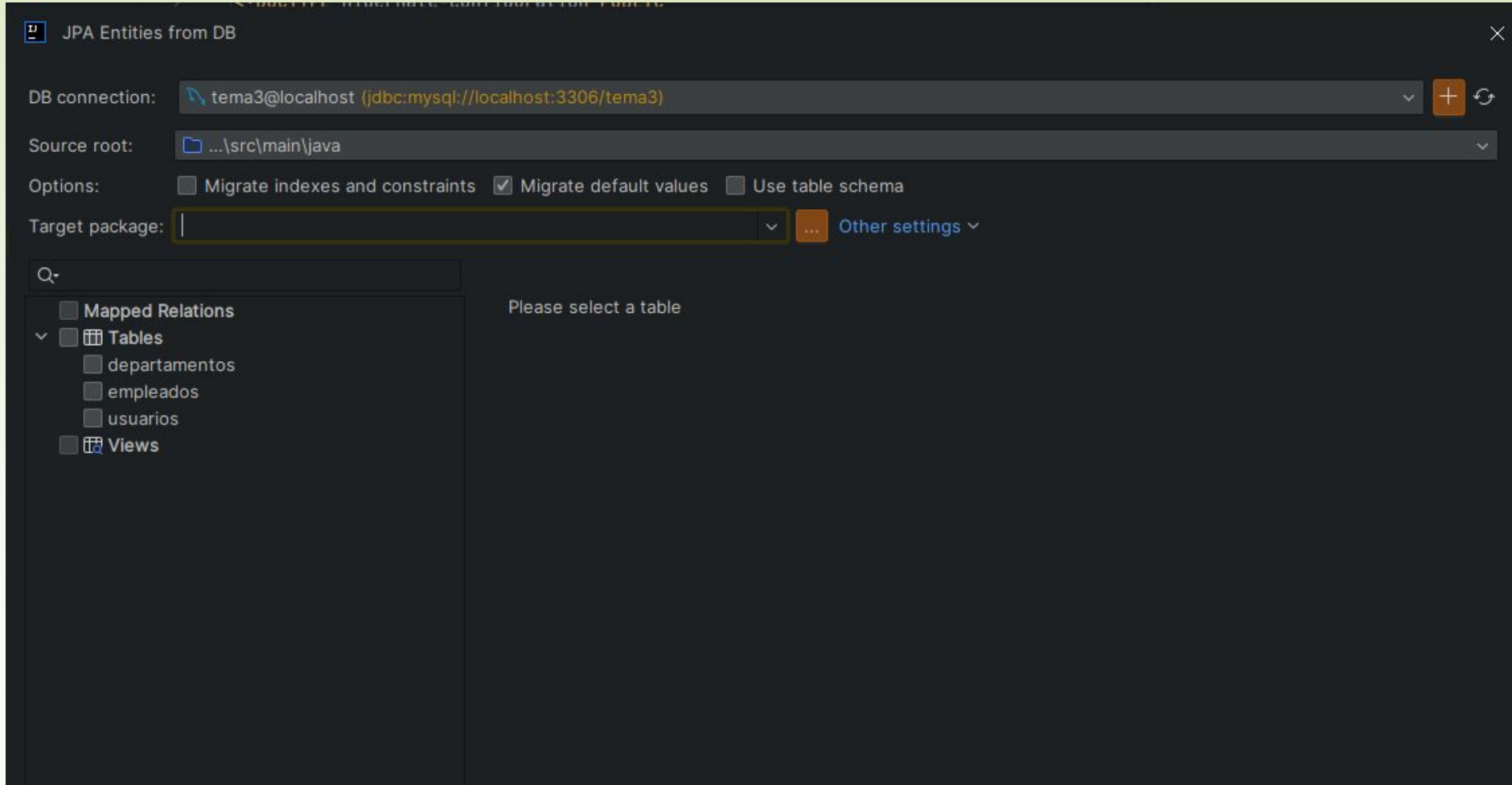


Ficheros de POJOS y de mapeo

- ❑ POJO (Plain Old Java Object) son las clases Java obtenidas de las tablas de la bases de datos.
- ❑ Estos objetos mantendrán los datos en nuestra aplicación. Son ficheros .java y .xml por cada tabla.
- ❑ Primero necesitamos instalar el plugin JPA buddy.
View>Tool Windows>JPA buddy o directamente click derecho en nuestra bbdd >> Create JPA Entities from DB

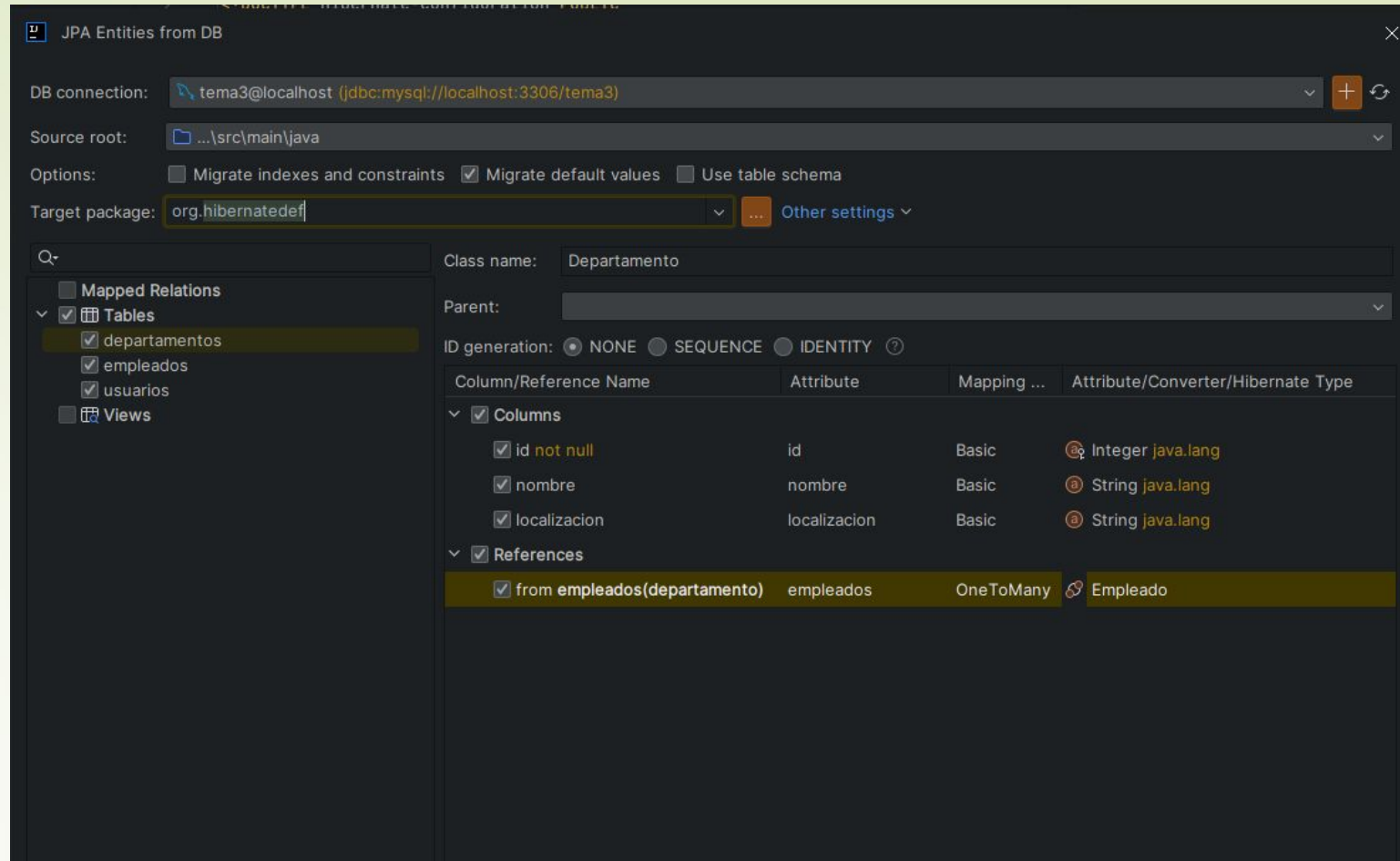
Ficheros de POJOS y de mapeo

Estaremos en esta ventana



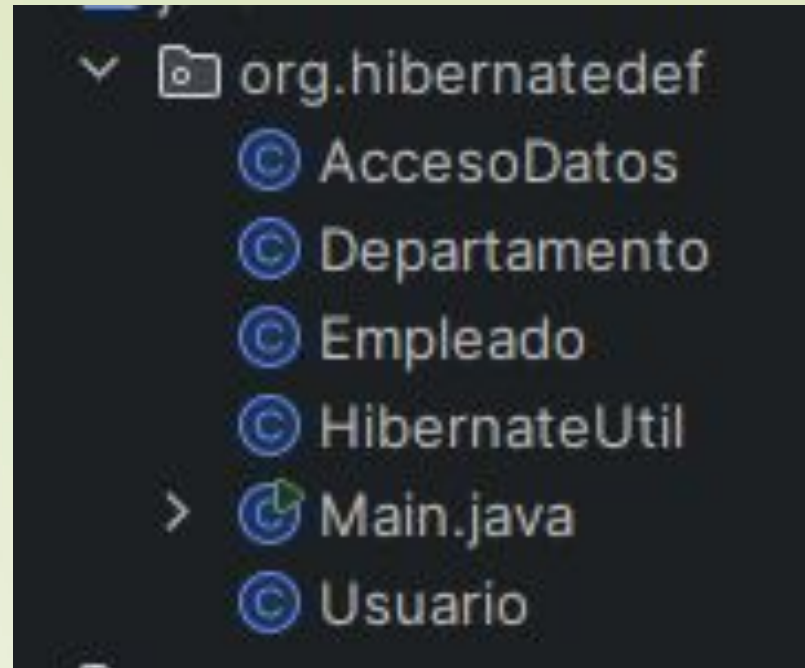
Ficheros de POJOS y de mapeo

- Indicamos el nombre del paquete de nuestro proyecto y seleccionamos las tablas que queremos y las relaciones



Ficheros de POJOS y de mapeo

- ❑ Al darle okay, se nos generarán los POJOS



Ficheros de POJOS y de mapeo

- ❑ El mapeo se realiza a través de etiquetas dentro de los POJOS además de indicarlo en el archivo de configuración de hibernate

```
@Entity 3 usages
@Table(name = "departamentos")
public class Departamento {
    @Id
    @Column(name = "id", nullable = false)
    private Integer id;

    @Column(name = "nombre", length = 20) 2 usages
    private String nombre;

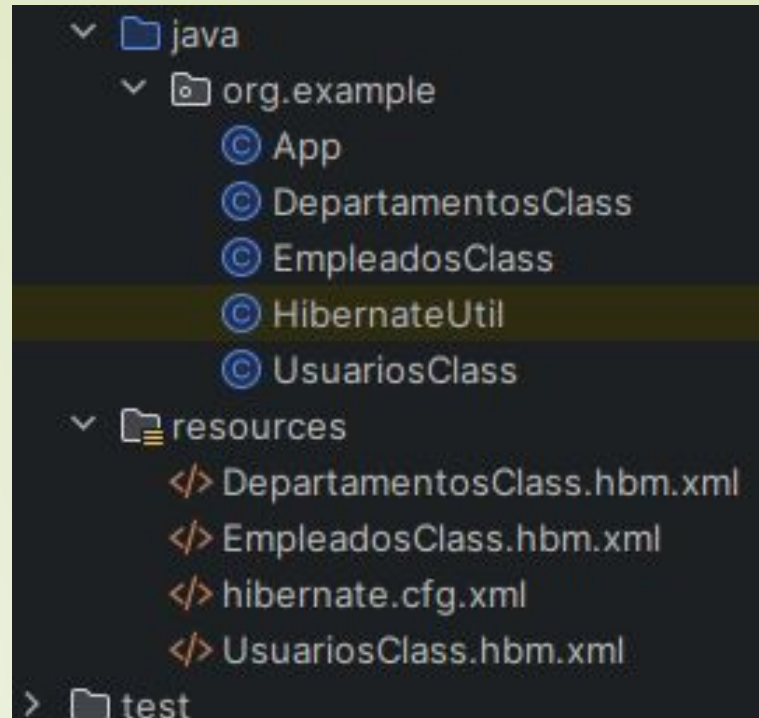
    @Column(name = "localizacion", length = 50) 2 usages
    private String localizacion;

    @OneToMany(mappedBy = "departamento") 2 usages
    private Set<Empleado> empleados = new LinkedHashSet<>();
}
```

```
<hibernate-configuration>
  <session-factory>
    <property name="connection.url">jdbc:mysql://localhost:3306/Tema3</property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.username">root</property>
    <!-- <property name="connection.password"/> -->
    <mapping class="org.hibernate.def.Departamento"/>
    <mapping class="org.hibernate.def.Empleado"/>
    <mapping class="org.hibernate.def.Usuario"/>
  </session-factory>
</hibernate-configuration>
```

Resultado final

- ❑ Por ultimo necesitaremos un fichero en el que iniciar el puente entre los ficheros de mapeo y los POJOS. Llamándolo HibernateUtil.



Crear un usuario

- ❑ Paracrear un usuario usando el objetos Usuarios que nos permite representar en Java datos de la BD

```
public void CrearUsuario(String login,String password,String tipo)
{
    Transaction tx;
    Session session=HibernateUtil.getSessionFactory().openSession();

    tx=session.beginTransaction(); //Crea una transacción
    Usuarios u = new Usuarios();
    u.setLogin(login);
    u.setPass(password);
    u.setTipo(tipo);
    session.save(u);//Guarda el objeto creado en la BBDD.
    tx.commit(); //Materializa la transacción
    session.close();
}
```


Consulta HQL

- ❑ Una consulta HQL es prácticamente idéntica al lenguaje SQL.
- ❑ Vamos a obtener una colección Java (List) que podemos recorrer y usar para nuestra aplicación.
- ❑ Los objetos conectan con sus datos relacionados y permiten añadir y modificar la información.

Consultas HQL Simples

- ❑ Podemos recuperar datos en forma de objetos.

```
public void MostrarAdmins()
{
    Session session = HibernateUtil.getSessionFactory().openSession();
    String c="from Usuarios u where u.tipo='admin'";

    Query q= session.createQuery(c);
    List<Usuarios> lista_usuarios = q.list();
    for (Usuarios u:lista_usuarios)
    {
        System.out.println("Nombre: "+u.getLogin());
    }
    session.close();
}
```

Consultas HQL Simples

- También podríamos modificar datos del usuario mediante los setters del objeto y quedaría reflejado en la BD.

```
public void ModificarAdmins()
{
    Session session = HibernateUtil.getSessionFactory().openSession();
    Transaction tx=session.beginTransaction();
    String c="from Usuarios u";

    Query q= session.createQuery(c);
    List<Usuarios> lista_usuarios = q.list();
    for (Usuarios u:lista_usuarios)
    {
        if(u.getTipo().equals("admin")){
            System.out.println("Nombre: "+u.getLogin());
            u.setPass("asasasasasP");
        }
    }
    tx.commit();
    session.close();
}
```

Consultas HQL Simples

- ❑ En el caso de que sepamos que solo va a devolver una única instancia usamos uniqueResult


```
public void recuperarUsuario(String user)
{
    Session session = HibernateUtil.getSessionFactory().openSession();
    String c="from Usuarios u where u.login='"+user+"'";

    Usuarios u=(Usuarios) session.createQuery(c).uniqueResult();

    System.out.println("Tipo: "+u.getTipo());
    session.close();
}
```

- ❑ Vamos a analizar como utilizar Hibernate para bases de datos con tablas relacionadas.
- ❑ Para ello nos basaremos en un base de datos llamada empresa con las siguientes tablas.

Tabla: departamentos	
Nombre	Tipo
id 	int(2)
nombre	varchar(20)
localizacion	varchar(50)

Tabla: empleados	
Nombre	Tipo
id 	int(4)
apellido	varchar(20)
cargo	varchar(9)
jefe	int(4)
fecha_alta	date
salario	double
comision	double
departamento	int(4)

- ❑ Debemos generar los ficheros necesarios como en el caso anterior

- ❑ Hibernate hace el mapeo objeto-relacional interpretando las **cardinalidades** de cada lado de la relación.
- ❑ Permite **comunicación bidireccional** entre relaciones de forma transparente, **sin necesidad de conocer el diseño** de la base de datos para poder manipularla.
- ❑ Hibernate nos reduce la **programación a colecciones** de objetos de los datos que necesitemos.
- ❑ Estudiaremos casos en función de la cardinalidad de la relación.

- ❑ Un departamento tiene muchos empleados.
Cardinalidad 1 a muchos.
- ❑ Hibernate proporciona un método **getEmpleados** que nos devuelve una colección con solo los empleados que pertenecen al departamento.
- ❑ Dichas colección es un conjunto de Java (**tipo Collection**) que se puede recorrer mediante un bucle for.

Empleados de un departamento

```
public void EmpleadosDepartamento(String dept)
{
    Session session = HibernateUtil.getSessionFactory().openSession();
    String c="from Departamentos d where d.nombre='"+dept+"'";

    Query q= session.createQuery(c);
    Departamentos d=(Departamentos) q.uniqueResult();
    Set<Empleados> lista_emp=d.getEmpleadoses();

    for (Empleados emp:lista_emp){
        System.out.println(emp.getApellido()+" "+emp.getCargo());
    }
    session.close();
}
```

- ❑ Podemos modificar datos y se actualizaran de manera automática. Como por ejemplo aumentar 3% el sueldo.

- ❑ Un empleado pertenece a un departamento y tiene un jefe (**Relación muchos a 1**).
- ❑ Hibernate proporciona métodos **getDepartamento** y **getEmpleados** que nos devuelven directamente los objetos relacionados.
- ❑ Igual que antes podemos obtener y modificar cualquier datos del objeto relacionado, actualizando la **base de datos en su tabla correspondiente** de manera automática.

Información de un Empleado

- ❑ Queremos saber la localización del departamento y el nombre del jefe del empleado llamado JONES

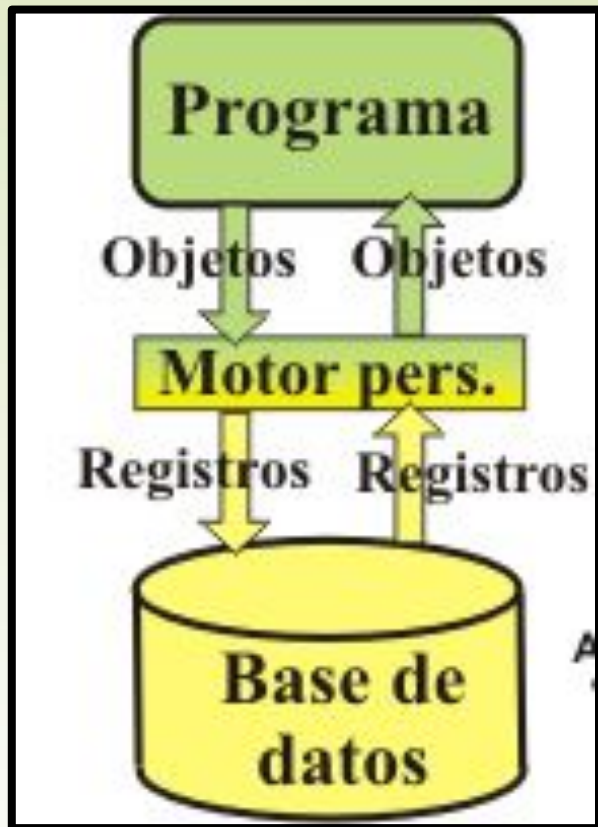
```
public void InfoEmpleado(String nombre_emp)
{
    Session session = HibernateUtil.getSessionFactory().openSession();
    String c="from Empleados e where e.apellido='"+nombre_emp+"'";

    Query q= session.createQuery(c);
    Empleados e=(Empleados) q.uniqueResult();
    Empleados jefe=e.getEmpleados();
    Departamentos dep=e.getDepartamentos();

    System.out.println("Localizacion del dept: "+dep.getLocalizacion()+
        "\nNombre jefe: "+jefe.getApellido());
    session.close();
}
```

- ❑ Usar un SGBD relacional **incrementa significativamente la complejidad** de un desarrollo software ya que obliga a la conversión de objetos a tabla relacionales.
- ❑ En un **SGBD-OO** los datos se almacenan directamente en objetos en la base de datos usando las mismas **estructuras y relaciones** que los lenguajes OO.
- ❑ El esfuerzo del programador y la complejidad del desarrollo software **se reducen considerablemente** y se mejora el flujo de comunicación entre usuarios, ingenieros y desarrolladores.

Justificación del uso de SGBD totalmente OO



- ❑ Un **SGBD-OO** contempla características como la encapsulación, identificadores de objetos(OID), herencia, polimorfismo y control de tipos de datos.
- ❑ Son **características** de un lenguaje orientados a objetos como Java, que añaden la persistencia en el uso de la información de una BD.
- ❑ La **persistencia** consiste en que cualquier acción que se haga en un tu aplicación automáticamente **se ve reflejada en la BD.**

- ❑ En 1993 se publico el estándar para el modelo de datos ODMG (Object Data Management Group) define todo lo necesario para que un lenguaje orientado a objetos y un SGBD puedan cooperar.
- ❑ Se define el ODL (object definition language) equivalente al DDL del modelo relacional que permite definir la especificación de tipos de objetos en sistemas compatibles con ODMG.
- ❑ Define atributo, relaciones ,clases , herencia, etc.
- ❑ Existe un OQL con el mismo objetivo que el SQL.

- ❑ Los SGBD-OO no han explotado todavía sobre todo por el arraigo que tiene al modelo relacional desde la industria y a la aparición de JSON, MongoDB que se adaptan rápidamente a las necesidades actuales de las aplicaciones.
- ❑ Disponéis de un manual sobre Matisse que es un ejemplo sencillo de SGBD-OO.
- ❑ Todavía queda un largo camino para que esto sea una alternativa a los sistemas tradicionales.

- ❑ El termino base de datos **objeto-relacional** se usa para describir una base de datos que ha evolucionado desde el modelo relacional hasta una **base de datos híbrida**.
- ❑ Las bases de datos objeto-relacionales tales como **Oracle son compatibles** en sentido ascendente con las bases de datos relacionales actuales y que además son familiares a los usuarios.
- ❑ Los usuarios pueden pasar sus aplicaciones actuales sobre bases de datos relaciones al nuevo modelo **sin tener que reescribirlas**.

- ❑ Posteriormente se pueden ir adaptando las aplicaciones y bases de datos para que utilicen las funciones orientadas a objetos. Algunos ejemplos interesantes:

Constructores de tipo:

```
create type tipo_dir as (  
  calle      varchar(20),  
  ciudad     varchar(20),  
  cod_postal char(5) );
```

Definición de un
tipo de usuario

```
create type tipo_empl as (  
  nombre     varchar(20),  
  apellido   varchar(20),  
  direccion  tipo_dir );
```

tipo de usuario para
definir otro tipo

```
create type tipo_empr as (  
  nombre     varchar(20),  
  lugares    varchar(20) array[10] );
```

Vector de 10
posiciones de
tipo varchar(20)

```
create table empresa of tipo_empr;  
create table empleado of tipo_empl;
```

tipos de usuario
para definir tablas

```
create type tipo_dir as (  
  calle      varchar(20),  
  ciudad     varchar(20),  
  cod_postal char(5)  
)  
method piso returns integer;
```

Función definida
Por el usuario

```
create table estado_act(  
  dueño char(25),  
  precio money );
```

```
create table est_europa under estado_act;  
create table est_españa under est_europa;  
create table est_rioja under est_españa;
```

Bibliografía

- ❑ **Ramos Martín, Alicia y Ramos Martín, M^aJesús:**
“Acceso a Datos”. Editorial Garceta. 2012
- ❑ **Córcoles Tendero, J.Ed. y Montero Simarro, Francisco:**
“Acceso a Datos. CFGS”. Editorial Ra-Ma. 2012
- ❑ **“Sistemas abiertos”. Contenidos de la asignatura.**
<http://deim.urv.cat/~pedro.garcia/SOB/>
Última visita: Septiembre 2017.
- ❑ **Ejercicios Acceso a Datos 2^a CFGS DAM**
<https://github.com/andresmr/AccessoDatos>
Última visita: Septiembre 2017
- ❑ **Master en desarrollo de aplicaciones Android. Universidad Politécnica de Valencia**
<http://www.androidcurso.com/index.php/recursos/42-unidad-9-almacenamiento-de-datos/299-preferencias>
Última visita: Septiembre 2017