

U3 — HERRAMIENTAS DE MAPEO OBJETO-RELACIONAL

1. INTRODUCCIÓN

Acceder a una BD relacional utilizando el lenguaje OO Java.

ORM: Object Relational Mapping. Interfaz que traduzca la lógica de los objetos a la lógica relacional.

- Tablas de la BD ↔ Clases
- Filas de una tabla de BD ↔ Objetos

2.MAPEO OBJETO-RELACIONAL



En la práctica, crea una BD OO virtual sobre la BD relacional.

Esto posibilita el uso de herencia y polimorfismo.

Hay paquetes comerciales y de uso libre que desarrollan el mapeo relacional de objetos, aunque algunos programadores prefieren crear sus propias herramientas ORM.

3. HERRAMIENTAS ORM. CARACTERÍSTICAS

Nos permiten crear una capa de acceso a datos.

Una forma sencilla: crear una clase por cada tabla de la BD y mapearlas una a una.

Estas herramientas aportan un lenguaje de consultas OO propio e independiente de la BD.

Algunas ventajas:

- Reduce tiempo de desarrollo de software
- Abstracción de la BD
- Reutilización
- Independencia de la BD
- Lenguaje propio para consultas

Inconveniente:

- Las aplicaciones son algo más lentas, ya que el sistema tiene que transformar las consultas al lenguaje propio de la herramienta, luego leer los registros, y por último, crear los objetos.

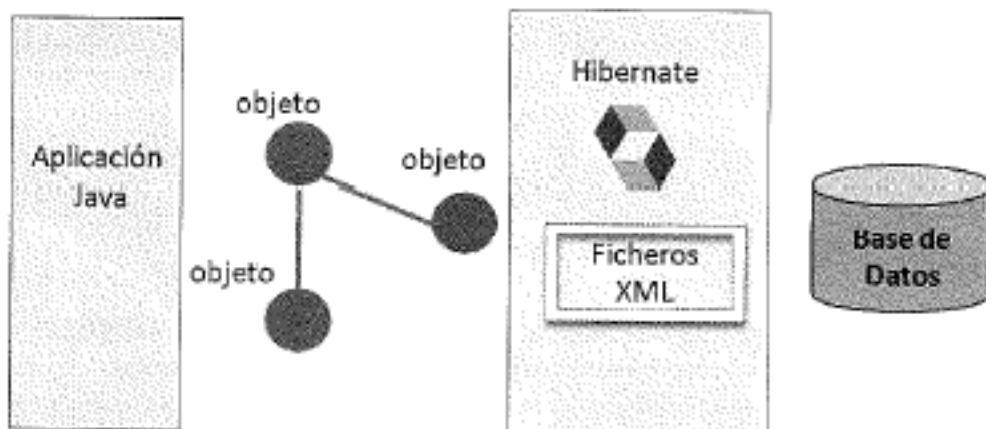
3.HERRAMIENTAS ORM. CARACTERÍSTICAS

Algunas herramientas ORM:

- Para incluir en proyectos PHP: Doctrine, Propel, ADOdb Active Record
- Para Visual Basic, .Net y C#: LINQ
- Para Java y .NET: Hibernate y Nhibernate respectivamente (software libre)
- Otros: QuickDB, iPersist, Java Data Objects, Oracle Toplink, etc.

En este tema estudiaremos **Hibernate**, que es uno de los ORM más populares.

Hibernate facilita el mapeo mediante **ficheros declarativos (XML)** que permiten establecer las relaciones entre la BD relacional y el modelo de objetos de una aplicación.



Con Hibernate no utilizaremos habitualmente SQL, sino que el propio motor de Hibernate, mediante el uso de factorías (patrón de diseño **Factory**), construirá esas consultas.

Lenguaje **HQL** (Hibernate Query Language) para acceder a datos mediante POO.

4.ARQUITECTURA DE HIBERNATE

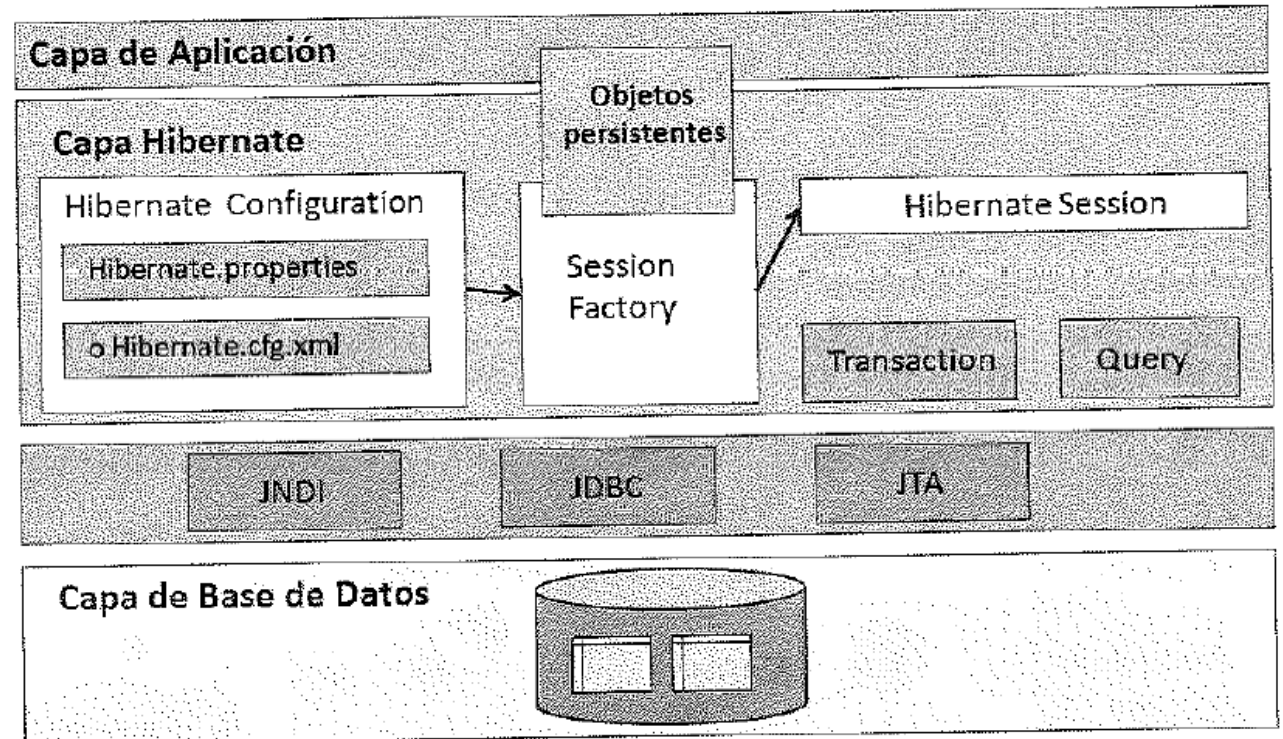
Filosofía Hibernate: mapear objetos POJOs (Plain Old Java Objects).

Para almacenar y recuperar estos objetos de la BD, el desarrollador debe mantener una conversación con el motor de Hibernate mediante un objeto especial que es la sesión (clase Session).

Igual que con las conexiones JDBC, hemos de crear y cerrar sesiones.

Arquitectura de Hibernate: varias capas.

Entre la capa de Hibernate y la de BD, se muestran diferentes APIs Java que usa Hibernate para interactuar con la BD.



4.ARQUITECTURA DE HIBERNATE

La clase Session (**org.hibernate.Session**) ofrece métodos como:

- *save(Object objeto)*
- *createQuery(String consulta)*
- *beginTransaction()*
- *close()*

para interactuar con la BD tal como se hace con una conexión JDBC, pero más simple.

Por ejemplo: para guardar un objeto, no hay que ejecutar SQL, sino:

session.save(miObjeto)

Una instancia Session no consume mucha memoria, y su creación y destrucción es muy barata.

4.ARQUITECTURA DE HIBERNATE

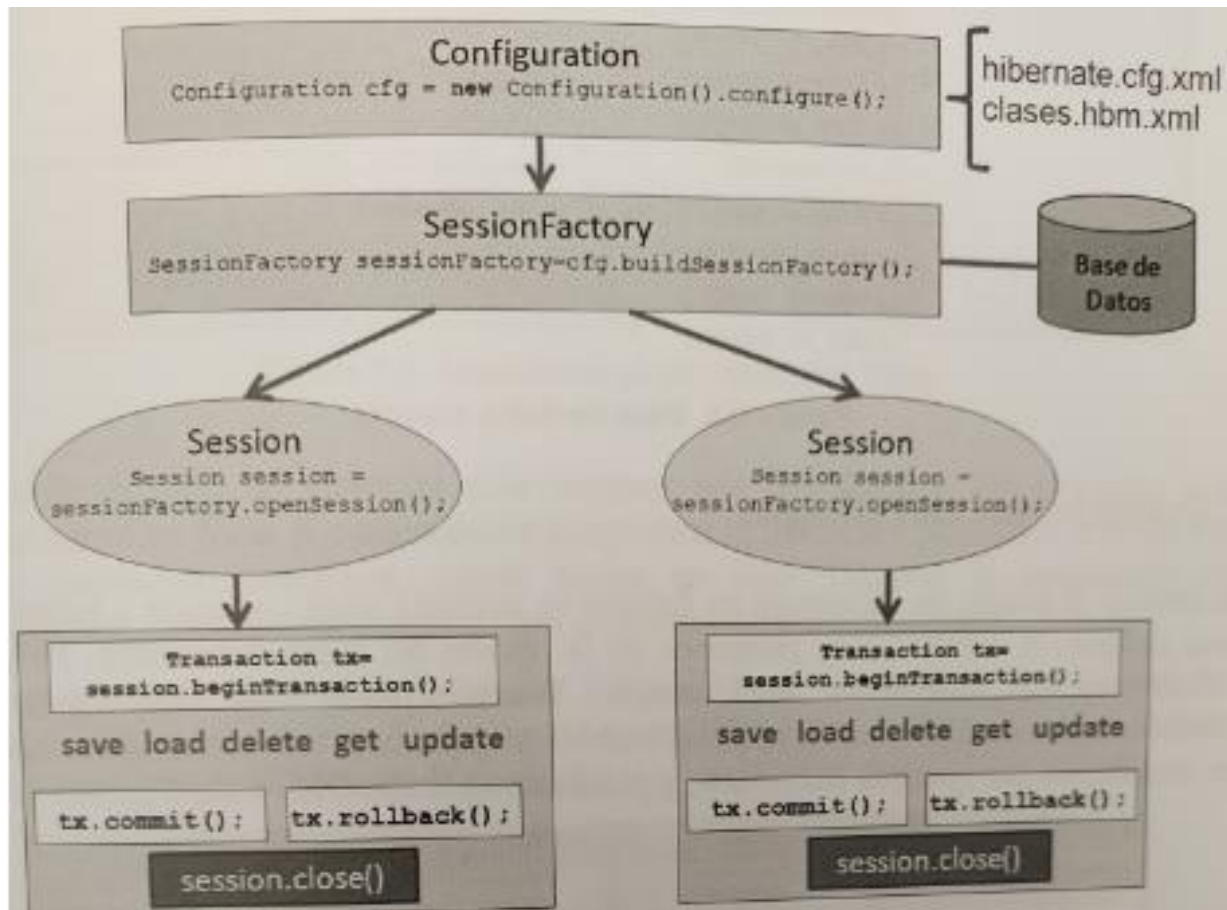
Interfaces de Hibernate:

- **org.hibernate.SessionFactory** permite obtener instancias Session.
 - Esta interfaz debe compartirse entre varios hilos de ejecución.
 - Normalmente hay una única SessionFactory para toda la aplicación, creada durante la inicialización de la misma.
 - Una SessionFactory por cada BD si la aplicación accede a más de una.
- **org.hibernate.cfg.Configuration** se utiliza para configurar Hibernate.
 - La aplicación utiliza una instancia de Configuration para especificar la ubicación de los documentos que indican el mapeado de los objetos y propiedades específicas de Hibernate, y a continuación crea la SessionFactory.
- **org.hibernate.Query** permite realizar consultas a la BD y controla cómo se ejecutan dichas consultas.
 - Las consultas se escriben en HQL o en el dialecto SQL nativo de la BD que estemos usando.
- **org.hibernate.Transaction** nos permite asegurar que cualquier error que ocurra entre el inicio y final de la transacción produzca el fallo de la misma.

4.ARQUITECTURA DE HIBERNATE

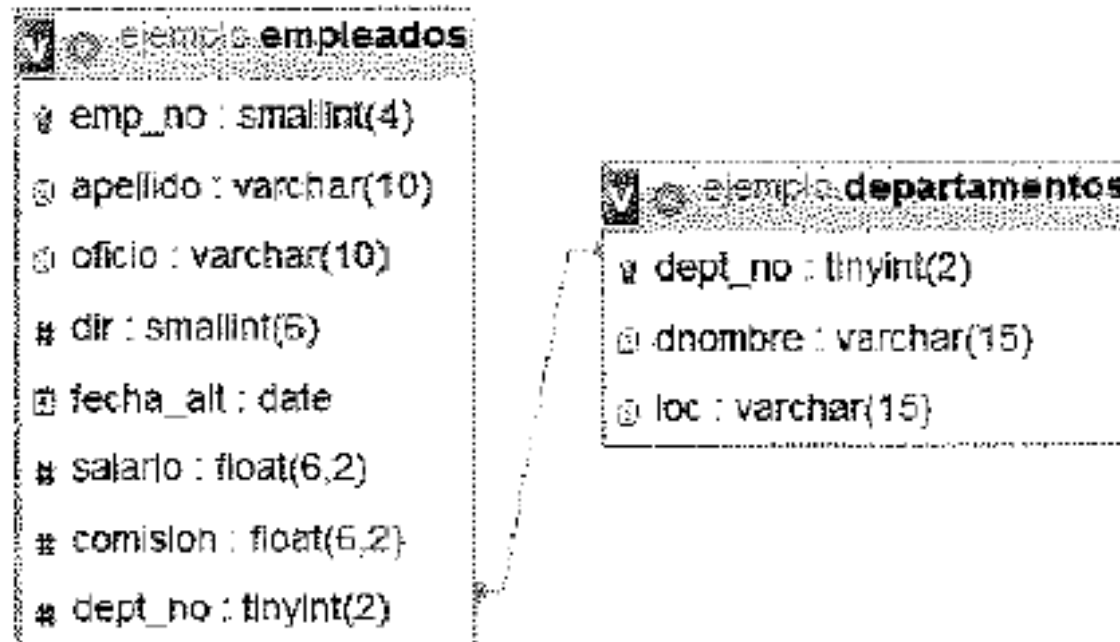
Hibernate hace uso de APIs de Java como JDBC, JTA (*Java Transaction Api*) y JNDI (*Java Naming Directory Interface*).

Aplicación con Hibernate:



5.INSTALACIÓN Y CONFIGURACIÓN DE HIBERNATE

BD ejemplo Mysql: RENOMBRAR ANTES LAS TABLAS: empleados->empleado, y departamentos->departamento. Clarificará mucho el resto del trabajo con este cambio.



5. CREAR PROYECTO HIBERNATE.

PASO 1: AÑADIR HIBERNATE TOOLS

Eclipse -> Help -> Eclipse Marketplace -> JBoss Tools 4.11.0.Final -> Install

(solo marcar la de Hibernate Tools -> Confirm)

Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation.
Press the "more info" link to learn more about a solution.



Search Recent Popular Favorites Installed Giving IoT an Edge

Find: All Markets All Categories Go


FreeMarker IDE from JBoss Tools 1.5

This is quick way to install FreeMarker IDE plugin from JBoss Tools <http://tools.jboss.org/>. Sources are at <https://github.com/jbosstools/jbosstools-freemarker...> [more info](#)

by JBoss/Red Hat, EPL
[freemarker](#) [html](#) [template](#) [fileExtension](#) [ftl](#)

★ 144 Installs: 112K (508 last month) **Install**

JBoss Tools 4.11.0.Final

 JBoss Tools is an umbrella project for a set of Eclipse plugins that includes support for JBoss and related technologies, such as Hibernate, JBoss AS / WildFly,... [more info](#)

by Red Hat, Inc., EPL
[openshift](#) [WildFly](#) [jbosstools](#) [maven](#) [hibernate](#)

★ 1094 Installs: 1,14M (7.909 last month) **Installed**

Hibernate Search Plugin v2.0.0.Final

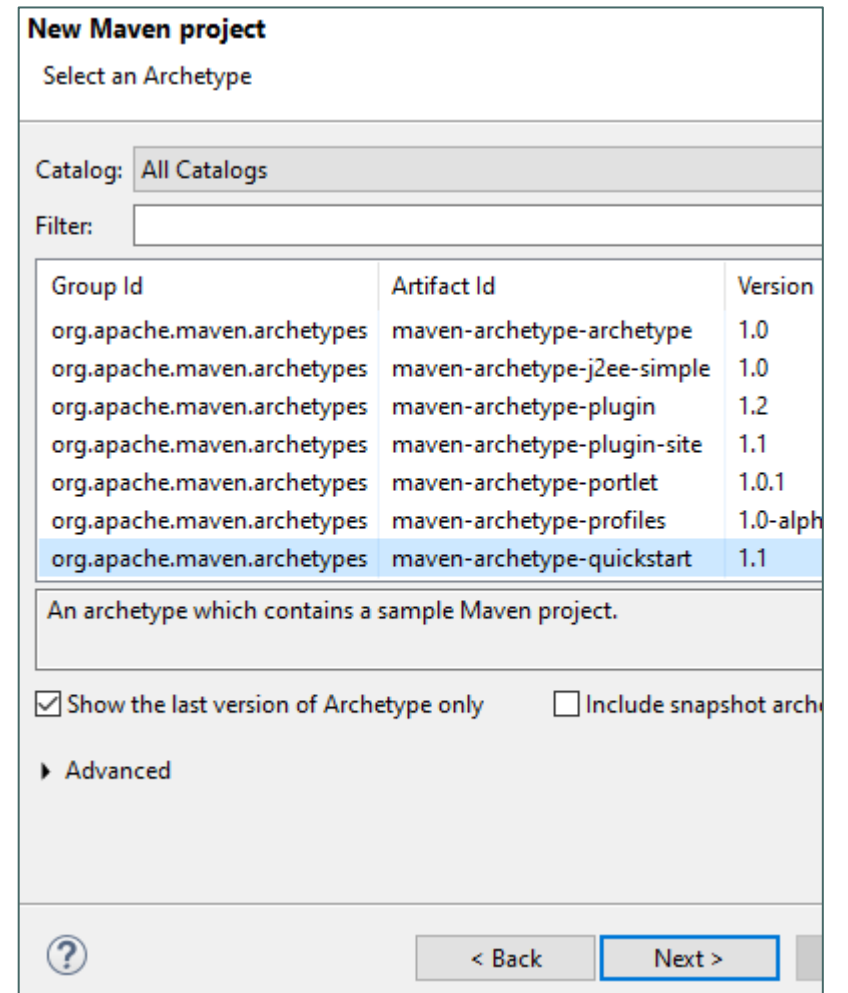
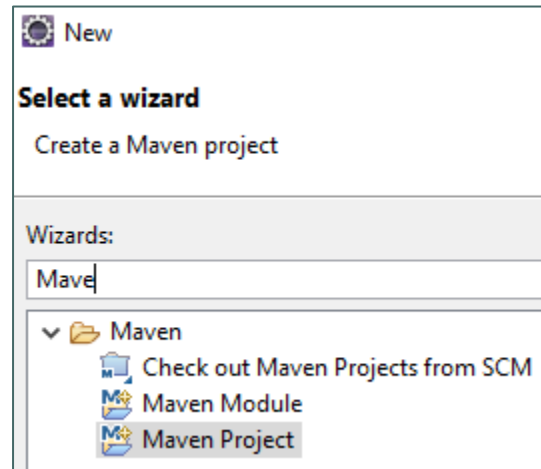
PASO 2: CREAR PROYECTO MAVEN

Para incluir Hibernate, hay que incluir más de 10 jars, lo mejor es usar el sistema de gestión de dependencias Maven. (dependencias en pom.xml)

2.1. New -> Maven Project -> Next -> escogemos el arquetipo QuickStart (proyecto de plantilla)

2.2. Indicar **Group Id:**
`com.accesodatos.hibernate` y
Artifact Id:
`ProyectoHibernateAnotaciones`

2.3. Una vez creado el proyecto, modificamos en Build Path del proyecto (pestaña Libraries) la versión de java a la 1.8



PASO 3: INDICAR DEPENDENCIAS (pom.xml)

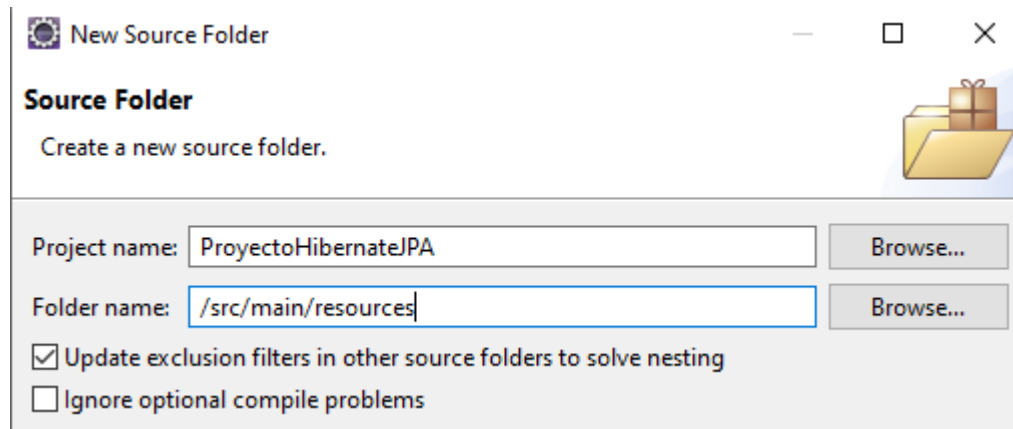
Añadir dependencia Maven de Hibernate y el conector MySQL al archivo pom.xml:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.4.25.Final</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.49</version>
</dependency>
```

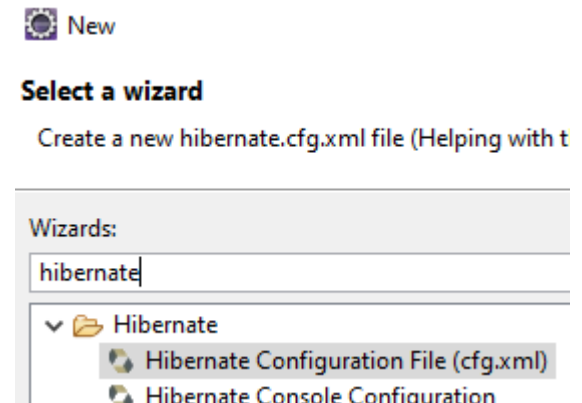
```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.25.Final</version>
    <type>pom</type>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.49</version>
  </dependency>
</dependencies>
```

PASO 4: CREAR EL FICHERO DE CONFIGURACIÓN DE HIBERNATE (hibernate.cfg.xml)

4.1. En el proyecto, creamos una ubicación donde situaremos el fichero de configuración de hibernate. New -> Source Folder



4.2. En el proyecto -> New -> Hibernate Configuration File



PASO 4: CREAR EL FICHERO DE CONFIGURACIÓN DE HIBERNATE (hibernate.cfg.xml)

4.3. Indicar datos de conexión a nuestra BD. Cuidado con escoger el driver correcto. (Si no os deja, poned Database dialect Mysql, seleccionar driver, y luego cambiar a MySQL 5 (InnoDB))

Hibernate Configuration File (cfg.xml)

This wizard creates a new configuration file to use with Hibernate.

Container:	/ProyectoHibernateJPA/src/main/resources
File name:	hibernate.cfg.xml
Hibernate version:	5.4 ▾
Session factory name:	
Get values from Connection	
Database dialect:	MySQL 5 (InnoDB)
Driver class:	com.mysql.jdbc.Driver
Connection URL:	jdbc:mysql://localhost/ejemplo
Default Schema:	ejemplo
Default Catalog:	
Username:	ejemplo
Password:	ejemplo
<input type="checkbox"/> Create a console configuration	

PASO 5: CREAR LAS CLASES PERSISTENTES (con anotaciones compatibles con JPA y anotaciones propias)

- Vamos a crear la clase persistente ***Departamento.java***.
- Son las clases que implementan las entidades del problema, y deben implementar ***Serializable***.
- Un registro o fila de su tabla en BD es un **objeto persistente** de esa clase.
- Cumplen normas del modelo de programación **POJO** (Plain Old Java Object):
 - utilizan convenciones de nombrado estándares para getters y setters:
atributo deptNo -> getDeptNo(), setDeptNo(...)
 - tienen atributos privados + getters y setters públicos

PASO 5: CREAR LAS CLASES PERSISTENTES

5.1. Creamos la clase **Departamento** (mismo nombre que la tabla en BD), con atributos privados, y anotando de esta forma:

- Anotamos la clase con **@Entity**
- (opcional) **@Table(name="")**
- Anotamos el atributo correspondiente a la PK con **@Id**
- (opcional) Anotamos el resto de columnas con **@Column(name="")**. Otras propiedades además de name: nullable, lenght (tamaño que tendrá el campo en la bd),...

Generar también un constructor sin parámetros, y los getters y setters de los atributos creados. (Desde el menú Source de Eclipse)

5.2. Añadir la entidad dentro de nuestro fichero de configuración de Hibernate.

PASO 5: CREAR LAS CLASES PERSISTENTES

5.1. Clase Departamento que implementa Serializable. Sus atributos y anotaciones:

```
@Id
@Column(name="dept_no", unique=true, nullable=false)
private byte deptNo;

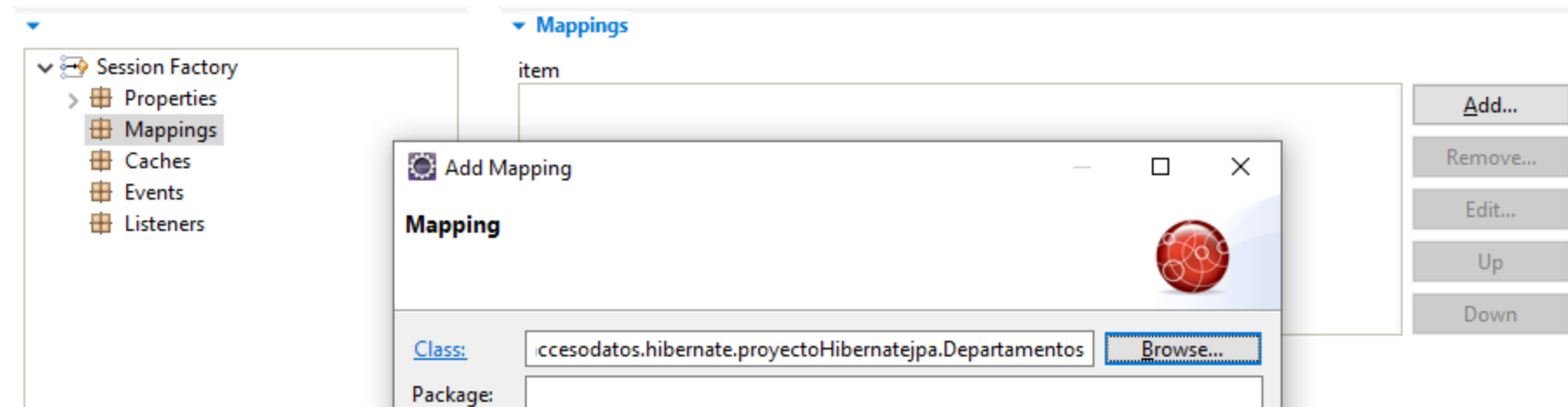
@Column(name="dnombre")
private String dnombre;

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

PASO 5: CREAR LAS CLASES PERSISTENTES

5.2. Añadir la entidad dentro de nuestro fichero de configuración de Hibernate.

Hibernate Configuration 3.0 XML Editor



PASO 6: CREAR LA CLASE DE APLICACIÓN

Editamos App.java:

- 1°. Inicializar un SessionFactory
- 2°. Abrir una sesión
- 3°. Crear instancias de Departamento (en este ejemplo)
- 4°. Iniciar transacción
- 5°. Persistir datos
- 6°. Commit de la transacción
- 7°. Cerrar Session y SessionFactory

PASO 6: CREAR LA CLASE DE APLICACIÓN

```
StandardServiceRegistry sr=new StandardServiceRegistryBuilder().configure().build();
SessionFactory sf= new MetadataSources(sr).buildMetadata().buildSessionFactory();

Session session=sf.openSession();

Departamentos departamento= new Departamentos();
departamento.setDeptNo((byte)11);
departamento.setDnombre("MIDEP");
departamento.setLoc("SEVILLA");

session.getTransaction().begin();
session.save(departamento);
session.getTransaction().commit();

session.close();
sf.close();
```

PASO 6: CREAR LA CLASE DE APLICACIÓN

En el ejemplo anterior hemos usado:

- ***save(Object objeto)***: método de la sesión (interface Session). Para guardar el objeto que le pasamos por parámetro
- ***commit()***: Hace commit de la transacción actual. La transacción empieza con ***beginTransaction()***
- ***close()***: Para cerrar la sesión.

6. TRANSACCIONES

Un objeto Session de Hibernate representa una única unidad de trabajo para un almacén de datos dado y lo abre un ejemplar de **SessionFactory**.

Tras crear la sesión se crea la transacción para dicha sesión.

Se deben cerrar las sesiones cuando se haya completado todo el trabajo de una transacción.

El siguiente código ilustra una sesión de persistencia de Hibernate:

```
Session session = session.openSession();
Transaction tx = session.beginTransaction();
//Código de persistencia
//...
tx.commit(); //valida la transacción
session.close(); //finaliza la sesión
```

- ***beginTransaction()*** marca el comienzo de una transacción.
- ***commit()*** valida una transacción.
- ***rollback()*** deshace la transacción.

7. ESTADOS DE UN OBJETO HIBERNATE

Hibernate define y soporta los siguientes estados de objeto:

- **Transitorio (Transient):** Objeto creado mediante operador new, pero aún no asociado a una Session de Hibernate.

```
Departamentos dep = new Departamentos();  
dep.setDeptNo((byte) 80);  
dep.setDnombre("MARKETING");  
dep.setLoc("GUADALAJARA"); //por ahora, dep es un objeto transitorio  
session.save(dep); //Hace que la instancia sea persistente
```

- **Persistente (Persistent):** Objeto que se encuentra en el ámbito de una Session.
 - El objeto ya está almacenado en la BD (puede haber sido guardado en la BD, o cargado de la BD).
 - Hibernate detectará cualquier cambio sobre el objeto persistente, y sincronizará el estado con la BD cuando se complete la unidad de trabajo.
- **Separado (Detached):** Una instancia separada es un objeto que se ha hecho persistente, pero su sesión ha sido cerrada.
 - Una instancia separada puede ser asociada a una nueva Session, haciéndola persistente de nuevo.

8. CARGA DE OBJETOS

Para la carga de objetos usaremos los siguientes métodos de Session:

MÉTODO	DESCRIPCIÓN
<code><T> T load(Class<T> Clase, Serializable id)</code>	Devuelve la instancia persistente de la clase indicada con el identificador dado. La instancia tiene que existir, si no existe el método lanza una excepción
<code>Object load(String nombreClase, Serializable id)</code>	Similar al método anterior, pero en este caso indicamos en el primer parámetro el nombre de la clase en formato de <i>String</i>
<code><T> T get(Class<T> Clase, Serializable id)</code>	Devuelve la instancia persistente de la clase indicada con el identificador dado. Si la instancia no existe, devuelve <i>null</i>
<code>Object get(String nombreClase, Serializable id)</code>	Similar al método anterior, pero en este caso indicamos en el primer parámetro el nombre de la clase

8. CARGA DE OBJETOS

El siguiente ejemplo utiliza el método *load()* para obtener los datos del departamento 20.

```
//Visualizamos los datos del departamento 20
Departamentos departamento=new Departamentos();
try {
    //Parámetros: clase del departamento, número de departamento que queremos recuperar
    //Lo casteamos a byte para convertirlo al tipo de dato definido en el atributo
    //identificador de la clase (deptNo)
    departamento=session.load(Departamentos.class, (byte) 20);
    System.out.println("Nombre del departamento: "+departamento.getDnombre());
    System.out.println("Localidad: "+departamento.getLoc());
}catch (ObjectNotFoundException o) {
    System.out.println("No existe el departamento.");
}
```

El método *load()* lanza una excepción *ObjectNotFoundException* si la fila no existe. Si no estamos seguros de que exista, es mejor utilizar *get()*.

8. CARGA DE OBJETOS

get() devuelve null si no existe la fila correspondiente.

El siguiente ejemplo comprueba si el departamento 11 existe.

```
//Comprobamos si existe el dpto 11 con get
Departamentos departament= session.get(Departamentos.class, 11);
if(departament==null) {
    System.out.println("El dpto no existe");
}else {
    System.out.println("Nombre del departamento: "+departament.getDnombre());
    System.out.println("Localidad: "+departament.getLoc());
}
```

9. ALMACENAMIENTO, MODIFICACIÓN Y BORRADO DE OBJETOS

Los métodos de Session que utilizaremos son:

MÉTODO	DESCRIPCIÓN
Serializable save(Object obj)	Guarda el objeto que se pasa como argumento en la base de datos. Hace que la instancia transitoria del objeto sea persistente.
void update(Object objeto)	Actualiza en la base de datos el objeto que se pasa como argumento. El objeto a modificar debe ser cargado con el método <i>load()</i> o <i>get()</i>
void delete(Object objeto)	Elimina de la base de datos el objeto que se pasa como argumento. El objeto a eliminar debe ser cargado con el método <i>load()</i> o <i>get()</i>

Ejemplos:

```
Departamentos dep = new Departamentos();  
dep.setDeptNo((byte) 80);  
dep.setDnombre("MARKETING");  
dep.setLoc("GUADALAJARA");  
  
session.save(dep); //almacena el objeto
```

```
//Para borrar un objeto, es necesario cargarlo antes con load o get.  
Empleados em=new Empleados();  
em=(Empleados) session.load(Empleados.class, (short)7369);  
session.delete(em); //elimina el objeto
```

9. ALMACENAMIENTO, MODIFICACIÓN Y BORRADO DE OBJETOS

Ejemplo que **elimina** un departamento controlando las excepciones: que el departamento no exista o que tenga empleados:

```
Departamentos de = (Departamentos) session.load(Departamentos.class, (byte) 10);

try {
    session.delete(de); // elimina el objeto
    tx.commit();
    System.out.println("Departamento eliminado");
} catch (ObjectNotFoundException o) {
    System.out.println("NO EXISTE EL DEPARTAMENTO...");
} catch (ConstraintViolationException c) {
    System.out.println("NO SE PUEDE ELIMINAR, TIENE EMPLEADOS...");
}
```

10. ASOCIACIONES ENTRE ENTIDADES

1. ManyToOne (unidireccional)

Creamos ahora la entidad Empleado (sigue los mismos pasos que para crear Departamento).
Particularidad: Muchos empleados pertenecen a un Departamento -> ManyToOne:

```
@ManyToOne
@JoinColumn(name="dept_no", foreignKey = @ForeignKey(name="FK_DEP"))
public Departamentos getDepartamentos() {
    return this.departamento;
}
```

(ojo en todos los ejemplos: para vosotros la clase se llama Departamento, no Departamentos.
Lo mismo con Empleado/s.)*

Crea ahora una aplicación que inserte un nuevo empleado asociado a un departamento que ya exista en la BD.

10. ASOCIACIONES ENTRE ENTIDADES

2. OneToMany (unidireccional)

La asociación la tendríamos solo del lado Departamento, que contendría un atributo Lista de Empleados. (Tendríamos que borrar el atributo Departamento de Empleado)

```
@OneToMany(cascade=CascadeType.ALL)
private List<Empleados> empleadotes= new ArrayList<>();
```

No vamos a hacerlo por ahora, ya que en este caso tendríamos que permitir que la aplicación pudiera crear tablas, y crearía la asociación entre empleados y departamentos en una tabla nueva en lugar de como lo tenemos actualmente en BD.

10. ASOCIACIONES ENTRE ENTIDADES

3. OneToMany (bidireccional)

Dejaríamos el atributo Departamento de Empleado tal y como lo teníamos anotado, y añadiríamos la lista del lado de Departamento, así:

```
@OneToMany(mappedBy="departamento", cascade=CascadeType.ALL)
private List<Empleados> empleadotes= new ArrayList<>();
```

Donde el valor de *mappedBy* es el nombre del atributo de Empleado mediante el que se hace el vínculo.

Añadiremos además métodos que faciliten la gestión de la asociación:
(Añadir también hashCode() y equals() a Empleado)

```
public void addEmpleado(Empleados empleado) {
    empleadotes.add(empleado);
    empleado.setDepartamentos(this);
}

public void removeEmpleado(Empleados empleado) {
    empleadotes.remove(empleado);
    empleado.setDepartamentos(null);
}
```


10. ASOCIACIONES ENTRE ENTIDADES

3. OneToMany (bidireccional)

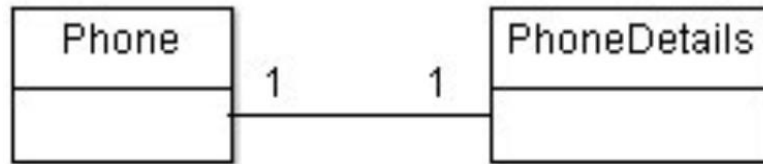
a) Carga un departamento de la BD, y muestra los empleados que pertenecen a ese departamento.

b) Crea un empleado con el mismo número de empleado que uno que ya pertenezca al departamento anterior, y comprueba mediante el método `contains` si ese empleado ya está contenido en la lista de empleados de ese departamento. (Fíjate, poniendo puntos de interrupción, en que ese método llama al método `equals` de `Empleado`. ¿Por qué?)

c) Crea una aplicación que cree un departamento con nuevos empleados asociados a él, y lo guarde. ¿Guarda también los empleados? ¿Por qué?

10. ASOCIACIONES ENTRE ENTIDADES

4. OneToOne (unidireccional)



```
@Entity
public class Phone {

    @Id
    @GeneratedValue
    private long id;

    private String number;

    @OneToOne
    @JoinColumn(name = "details_id")
    private PhoneDetails details;
```

```
@Entity
public class PhoneDetails {

    @Id
    @GeneratedValue
    private Long id;

    private String provider;

    private String technology;
```

10. ASOCIACIONES ENTRE ENTIDADES

5. OneToOne (bidireccional)

```
@Entity
public class Phone {

    @Id
    @GeneratedValue
    private Long id;

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

    @OneToOne(mappedBy = "phone", cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
    private PhoneDetails details;
```

```
@Entity
public class PhoneDetails {

    @Id
    @GeneratedValue
    private Long id;

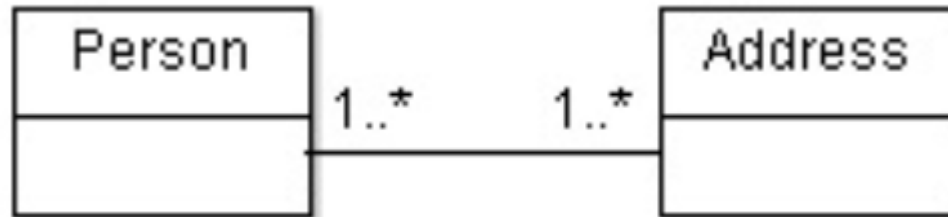
    private String provider;

    private String technology;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "phone_id")
    private Phone phone;
```

10. ASOCIACIONES ENTRE ENTIDADES

6. ManyToMany (unidireccional) – Si no tuvieramos la clase intermedia creada y queremos que la cree Hibernate (no es nuestro caso)



```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private List<Address> addresses = new ArrayList<>();
}
```

```
@Entity
public class Address {

    @Id
    @GeneratedValue
    private Long id;

    private String street;

    private String number;
}
```

10. ASOCIACIONES ENTRE ENTIDADES

7. ManyToMany (bidireccional) – Si no tuvieramos la clase intermedia creada y queremos que la cree Hibernate (no es nuestro caso)

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private List<Address> addresses = new ArrayList<>();
```

```
    public void addAddress(Address address) {
        addresses.add( address );
        address.getOwners().add( this );
    }

    public void removeAddress(Address address) {
        addresses.remove( address );
        address.getOwners().remove( this );
    }
}
```

```
@Entity
public class Address {

    @Id
    @GeneratedValue
    private Long id;

    private String street;

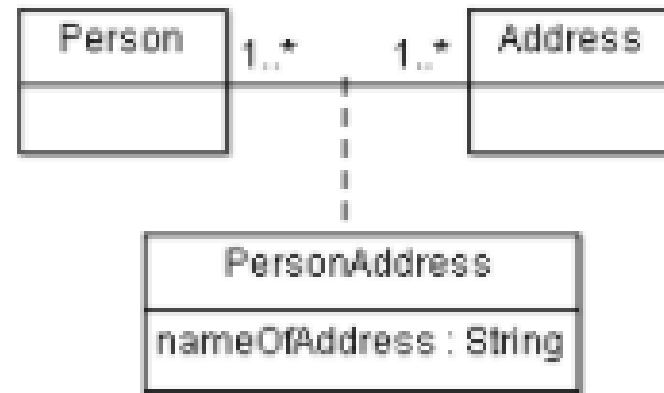
    private String number;

    private String postalCode;

    @ManyToMany(mappedBy = "addresses")
    private List<Person> owners = new ArrayList<>();
```

10. ASOCIACIONES ENTRE ENTIDADES

7. **ManyToMany (bidireccional)** – Si ya tenemos la clase intermedia (es nuestro caso)



1. Generar una nueva entidad **PersonAddress**
2. Romper la asociación **@ManyToMany** en ambos extremos en dos asociaciones que den el mismo resultado: **@ManyToOne** + **@OneToMany**.
3. Manejar de forma conveniente la clave primaria de esta nueva entidad. Al ser una clave primaria compuesta, necesitaremos de una clase extra, **PersonAddressId**, y de la anotación **@IdClass**, para poder manejarla.

10. ASOCIACIONES ENTRE ENTIDADES

7. ManyToMany (bidireccional)

```
@Entity
public class Address {

    @Id
    @GeneratedValue
    private Long id;

    private String street;

    private String number;

    private String postalCode;

    @OneToMany(mappedBy = "address", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<PersonAddress> owners = new ArrayList<>();
```

```
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    @NaturalId
    private String registrationNumber;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<PersonAddress> addresses = new ArrayList<>();
```

10. ASOCIACIONES ENTRE ENTIDADES

7. ManyToMany (bidireccional)

```
public class PersonAddressId implements Serializable {  
  
    private Long person;  
    private Long address;  
  
    public PersonAddressId() {  
  
    }  
  
    //Getters, setters, equals y hashCode  
  
}
```

```
@Entity  
@IdClass(PersonAddressId.class)  
public class PersonAddress {  
  
    @Id  
    @ManyToOne  
    @JoinColumn(  
        name="person_id",  
        insertable = false, updatable = false  
    )  
    private Person person;  
  
    @Id  
    @ManyToOne  
    @JoinColumn(  
        name="address_id",  
        insertable = false, updatable = false  
    )  
    private Address address;
```


10. ASOCIACIONES ENTRE ENTIDADES

7. ManyToMany (bidireccional)

Realiza el mapeo de las tablas Proyecto y Trabaja, y crea la asociación muchos a muchos correspondiente. Crea una aplicación que inserte un empleado que trabaje en un proyecto.

11. CONSULTAS

Las consultas HQL y SQL nativas son representadas con una instancia de `org.hibernate.query.Query`. Algunos métodos importantes de esta interfaz son:

MÉTODO	
<code>Iterator iterate()</code>	Devuelve en un objeto <code>Iterator</code> el resultado de la consulta
<code>List list()</code>	Devuelve el resultado de la consulta en un <code>List</code>
<code>Query setFetchSize (int size)</code>	Fija el número de resultados a recuperar en cada acceso a la base de datos al valor indicado en <code>size</code>
<code>int executeUpdate()</code>	Ejecuta la sentencia de modificación o borrado. Devuelve el número de entidades afectadas
<code>String getQueryString()</code>	Devuelve la consulta en un <code>String</code>
<code>Object uniqueResult()</code>	Devuelve un objeto (cuando sabemos que la consulta devuelve un objeto) o nulo si la consulta no devuelve resultados
<code>Query setCharacter(int posición, char valor)</code>	Asigna el valor indicado en el método a un parámetro de tipo <code>CHAR</code> <i>posición</i> , indica la posición del parámetro dentro de la consulta, empieza en 0
<code>Query setCharacter(String nombre, char valor)</code>	<i>nombre</i> es el nombre (se indica como <code>:nombre</code>) del parámetro dentro de la consulta
<code>Query setDate(int posición, Date fecha)</code> <code>Query setDate(String nombre, Date fecha)</code>	Asigna la <i>fecha</i> a un parámetro de tipo <code>DATE</code>

11. CONSULTAS

Las consultas HQL y SQL nativas son representadas con una instancia de `org.hibernate.query.Query`. Algunos métodos importantes de esta interfaz son:

<code>Query setDouble(int posición, double valor)</code> <code>Query setDouble(String nombre, double valor)</code>	Asigna valor a un parámetro de tipo decimal (en MySQL tipo FLOAT)
<code>Query setInteger(int posición, int valor)</code> <code>Query setInteger(String nombre, int valor)</code>	Asigna valor a un parámetro de tipo entero
<code>Query setString(int posición, String valor)</code> <code>Query setString(String nombre, String valor)</code>	Asigna valor a un parámetro de tipo VARCHAR
<code>Query setParameterList(String nombre, Collection valores)</code>	Asigna una colección de valores al parámetro cuyo nombre se indica en <i>nombre</i>
<code>Query setParameter(int posición, Object valor)</code>	Asigna el valor al parámetro indicado en <i>posición</i>
<code>Query setParameter(String nombre, Object valor)</code>	Asigna el valor al parámetro indicado en <i>nombre</i>
<code>int executeUpdate()</code>	Ejecuta una sentencia UPDATE o DELETE, devuelve el número de entidades afectadas por la operación
Consulta la API de Hibernate: https://docs.jboss.org/hibernate/orm/current/javadocs/	

11. CONSULTAS

Ejemplo 1: Consulta de un departamento:

```
StandardServiceRegistry sr=new StandardServiceRegistryBuilder()  
SessionFactory sf= new MetadataSources(sr).buildMetadata().buildSessionFactory()  
  
Session session=sf.openSession();  
  
Departamentos dep= session.get(Departamentos.class, (byte)11);  
System.out.println(dep);  
  
session.close();  
sf.close();
```

11. CONSULTAS

Ejemplo 2: Consulta de todos los departamentos:

```
StandardServiceRegistry sr=new StandardServiceRegistryBuilder().configure().build();
SessionFactory sf= new MetadataSources(sr).buildMetadata().buildSessionFactory();

Session session=sf.openSession();

Query<Departamentos> q = session.createQuery("from Departamentos");
List <Departamentos> lista = q.list();
// Obtenemos un Iterador y recorremos la lista.
Iterator <Departamentos> iter = lista.iterator();
System.out.printf("Número de registros: %d%n",lista.size());
while (iter.hasNext())
{
    //extraer el objeto
    Departamentos depar = iter.next();
    System.out.printf("%d, %s%n", depar.getDeptNo(), depar.getDnombre());
}
session.close();
sf.close();
```

11. CONSULTAS

Ejemplo 3: Consulta de todos los empleados que pertenecen a un departamento:

```
Query<Empleados> q = session.createQuery("from Empleados as e where e.departamento.deptNo=20");
List<Empleados> lista = q.list();
// Obtenemos un Iterador y recorremos la lista.
Iterator<Empleados> iter = lista.iterator();
System.out.printf("Número de registros: %d\n", lista.size());
while (iter.hasNext())
{
    //extraer el objeto
    Empleados emple = iter.next();
    System.out.printf("%s\n", emple.getApellido());
}
```

11. CONSULTAS

Ejemplo 4: Consulta del departamento 20 con ***UniqueResult()***:

```
Query q = session.createQuery("from Departamentos as e where e.deptNo=20");  
Departamentos dep= (Departamentos) q.uniqueResult();  
System.out.println(dep);
```

11. CONSULTAS

Ejemplo 5: Consulta con parámetros utilizando **setParameter()**

```
String hql = "from Empleados where empNo = :numemple";  
Query q = session.createQuery(hql);  
q.setParameter("numemple", (short) 1003);  
Empleados emple = (Empleados) q.uniqueResult();  
System.out.printf("%s, %s %n", emple.getApellido(), emple.getOficio());
```

Ejemplo 6: Consulta los departamentos cuyo número de departamento esté entre 10 y 20. Uso de **setParameterList()**:

```
// from Empleados emp where emp.departamentos.deptNo in (10,20)  
System.out.println("Empleados con departamento 10, 20 ");  
List<Byte> numeros = new ArrayList<Byte>();  
numeros.add((byte) 10);  
numeros.add((byte) 20);  
String hql5 = "from Empleados emp where emp.departamentos.deptNo in (:listadep) "  
            + "order by emp.departamentos.deptNo ";  
Query q5 = session.createQuery(hql5);  
q5.setParameterList("listadep", numeros);
```


11. CONSULTAS

Ejemplo 7: Consultas sobre varias tablas (no tenemos ninguna clase asociada a los atributos que devuelve la consulta).

```
String hql = "from Empleados e, Departamentos d where e.departamento.deptNo = d.deptNo order by apellido";
Query q = session.createQuery(hql);
List lista = q.list();
// Obtenemos un Iterador y recorremos la lista.
Iterator iter = lista.iterator();
while(iter.hasNext()) {
    Object[] par=(Object[]) iter.next();
    Empleados em= (Empleados) par[0];
    Departamentos dep=(Departamentos) par[1];
    System.out.println("Apellido empleado: "+em.getApellido()+" , Departamento: "+dep.getDnombre());
}
```

11. CONSULTAS

<https://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/queryhql.html>