1. CONCEPTO DE MAPEO OBJETO-RELACIONAL

En el **modelo relacional** una base de datos es un conjunto de tablas estructuradas en registros (filas) y campos (columnas), que se relacionan entre sí por uno o varios campos en común. Sin embargo, en el **modelo orientado a objetos** se organizan en objetos, combinando las estructuras de datos con sus comportamientos. En este modelo destacan conceptos básicos tales como clases, objetos, herencia y polimorfismo.

Entre estos dos modelos existe un **desfase objeto-relacional** (también denominado desajuste por impedancia), debido a sus diferencias:

- En los sistemas de bases de datos relacionales, los datos siempre se manejan en forma de tablas, formadas por un conjunto de tuplas. Además, en el modelo relacional no se puede modelar la herencia.
- En los entornos orientados a objetos, los datos se manipulan como objetos, que suelen estar formados por tipos elementales y a su vez otros objetos.
- Existen desajustes en los tipos de datos, porque los tipos y denotaciones de tipos asumidos por los lenguajes de programación y de consultas difieren.

El **mapeo objeto-relacional** (ORM, Object-Relational Mapping) es una técnica de programación que permite convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos (como Java) y el utilizado en una base de datos relacional utilizando un motor de persistencia (como Oracle o MySQL).

En la práctica, este mapeo crea una base de datos orientada a objetos virtual sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (herencia y polimorfismo).

Actualmente, las bases de datos relacionales solo pueden guardar datos primitivos, por lo que impiden almacenar directamente objetos en la base de datos. El mapeo ORM se basa en la conversión de los objetos usados en un programa orientado a objetos en datos primitivos que sí se pueden almacenar en las tablas correspondientes en una base de datos relacional. Cuando se desea recuperar los datos del sistema relacional, el mapeo ORM obtiene los datos primitivos de la base de datos y reconstruye los objetos.

2. HERRAMIENTAS ORM.

2.1. CARACTERÍSTICAS

Las herramientas ORM permiten crear una capa intermedia de acceso a datos; es decir, ofrecen un proceso transparente de conversión de datos relacionales a objetos y viceversa.

Una forma sencilla y válida de hacerlo es crear una clase por cada tabla de la base de datos y mapearlas una a una. Estas herramientas aportan un lenguaje de consultas orientado a objetos propio y totalmente independiente de la base de datos que usemos, lo que nos permitirá migrar de una base de datos a otra sin tocar nuestro código, solo será necesario cambiar alguna línea en el fichero de configuración.

El mapeo objeto-relacional tiene las siguientes **ventajas**:

- **Rapidez en el desarrollo.** La mayoría de las herramientas actuales permiten la creación del modelo adecuado a partir del esquema de la base de datos, reduciendo el tiempo de desarrollo de software.
- **Abstracción de la base de datos.** Las herramientas ORM permiten abstraer la aplicación del sistema gestor de base de datos utilizado. Por tanto, si se cambia de sistema gestor en el futuro, este cambio no debería afectar al código del programa desarrollado.
- **Reutilización.** Las herramientas ORM permiten utilizar los métodos de un objeto de datos desde distintas zonas de la aplicación, incluso desde aplicaciones distintas.
- **Mantenimiento del código.** El mapeo facilita y simplifica el mantenimiento del código debido a la correcta ordenación de la capa de datos.
- Incentivan la **portabilidad y la escalabilidad** de los programas de software.
- Permiten la producción de **mejor código** y son **independientes** de la Base de Datos, funcionan en cualquier BD.
- **Lenguaje propio para realizar consultas.** Los mecanismos de mapeo ofrecen su propio lenguaje para hacer las consultas, de forma que los usuarios dejan de usar las sentencias SQL para pasar a utilizar el lenguaje propio de la herramienta. Esto también puede verse como un inconveniente, pues obliga al programador a conocer más acerca de este lenguaje propio.

El mapeo objeto-relacional tiene las siguientes **desventajas**:

- **Tiempo utilizado en el aprendizaje.** Las herramientas ORM suelen ser un poco más complejas, por lo que su correcta utilización requiere un tiempo de aprendizaje no despreciable.
- **Aplicaciones algo más lentas.** Esto es debido a que, para cada consulta realizada sobre la base de datos, primero el sistema tendrá que transformarla al lenguaje propio de la herramienta, después deberá leer los registros necesarios, y por último, deberá crear los objetos correspondientes.

2.2. HERRAMIENTAS

Cuando se trabaja con programación orientada a objetos y bases de datos relacionales, se utilizan paradigmas y formas de pensar distintas: el modelo relacional trata con relaciones y conjuntos de datos y el paradigma orientado a objetos trata con clases, objetos, atributos, métodos y asociaciones entre objetos. El mapeo objeto-relacional tiene como misión evitar estas diferencias.

Las herramientas ORM sirven para crear una **capa intermedia de acceso a datos**, como por ejemplo, crear una clase por cada tabla de la base de datos y mapearlas una a una. Estas herramientas aportan un lenguaje de consultas orientado a objetos propio y totalmente independiente de la base de datos usada. Esto permitirá en el futuro migrar de una base de datos a otra sin necesidad de tocar el código fuente, sólo se precisarán algunos cambios en el fichero de configuración.

En la actualidad, existen muchas herramientas ORM que permiten el mapeo objeto-relacional, según el lenguaje de programación utilizado algunas son: *Doctrine, propel o ADOdb Active Record* para incluir en proyectos PHP, Language INtegrated query (LINQ) desarrollado por Microsoft para el mapeo objeto-relacional para Visual Basic .NET y C#, Hibernate para la tecnología Java y disponible también para la tecnología .NET con el nombre de *NHibernate,* que es software libre bajo la licencia GNU LGPL.

Además de estos nombreado hay muchos otros como pueden ser *QuickBD, iPersist, Java Data Objects Oracle Toplink, etc*.

2.2.1. LANGUAGE INTEGRATED QUERY (LINQ)

Language INtegrated Query (LINQ) es un componente de la plataforma .NET de Microsoft que se utiliza para el mapeo objeto-relacional en los lenguajes Visual Basic .NET y C#, y que agrega capacidades de consulta a datos de manera nativa a los lenguajes .NET.

LINQ extiende el lenguaje a través de expresiones de consulta, que son parecidas a las sentencias SQL y que pueden ser usadas para extraer y procesar convenientemente datos de vectores, clases enumerables, documentos XML y bases de datos relacionales. Otros usos incluyen la construcción de manejadores de eventos.

Además, LINQ define un conjunto de nombres de métodos (llamados operadores de consulta estándar) y un conjunto de reglas de traducción, que son usadas por el compilador para traducir las expresiones de consulta en expresiones normales del lenguaje, utilizando estos nombres de métodos, expresiones lambda y tipos de datos anónimos.

2.2.2. DOCTRINE

Doctrine es un mapeador objeto-relacional escrito en PHP que proporciona una capa de persistencia para objetos PHP y que actúa como una capa de abstracción ubicada justo encima de un sistema gestor de bases de datos.

Una característica de Doctrine es el bajo nivel de configuración que se requiere para comenzar un proyecto. Doctrine puede generar clases a partir de una base de datos existente, y después el programador puede especificar relaciones y añadir funcionalidades personalizadas a las clases autogeneradas. No es necesario generar o mantener complejos esquemas XML de bases de datos, como en otros *frameworks*.

Doctrine incluye un dialecto de SQL orientado a objetos denominado DQL (Doctrine Query Language), que está inspirado en Hibernate (Java), para escribir consultas de bases de datos. No obstante, no siempre se necesitan escribir consultas explícitamente, pues Doctrine realiza consultas JOIN y recoge objetos asociados automáticamente.

Asimismo, Doctrine ofrece soporte para datos jerárquicos mediante una columna agregada (que especifica el subtipo de un objeto particular), soporte para *hooks* (métodos que pueden validar o modificar las escrituras y lecturas de la base de datos) y eventos para manejar la lógica de negocio relacionada, transacciones ACID, migraciones de bases de datos, y un *framework* de caché que utiliza diversos motores como Memcached, APC o SQLite.

2.2.3. HIBERNATE

Hibernate es una herramienta de mapeo objeto-relacional (ORM) para la plataforma Java (también está disponible para .NET con el nombre de NHibernate) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante ficheros declarativos (XML) o anotaciones que permiten establecer estas relaciones. Es software libre, distribuidos bajo los términos de la licencia GNU LGPL.

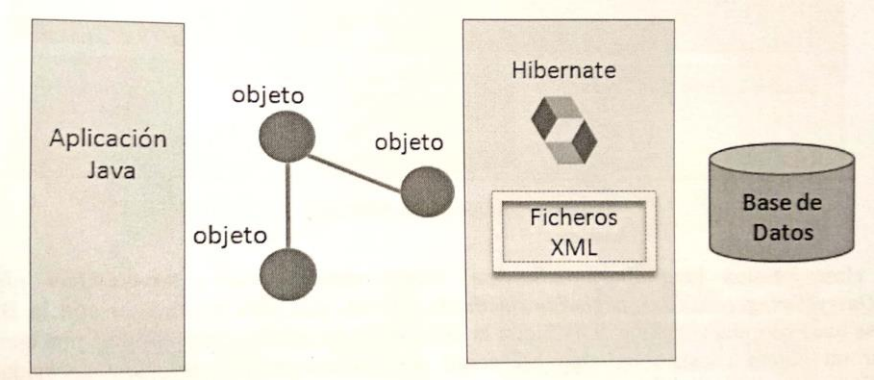


Ilustración 1.- Hibernate, mapeo objeto-relacional

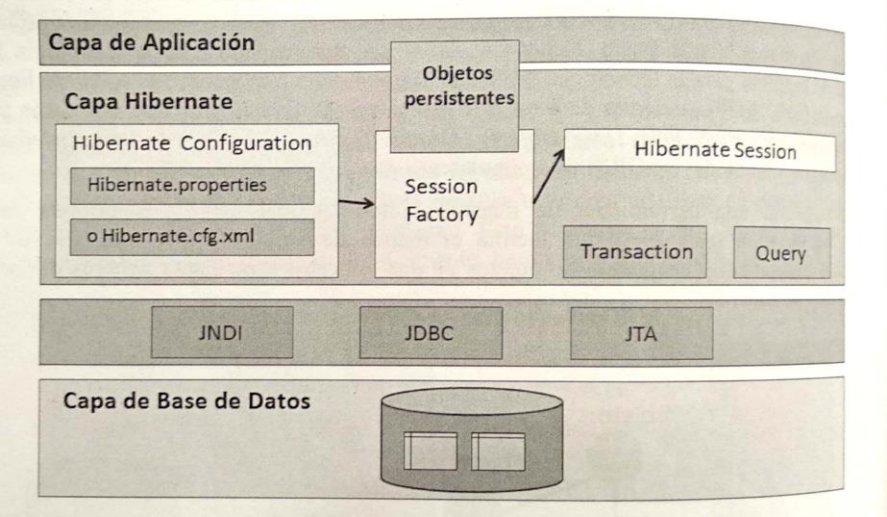
Se está convirtiendo en el estándar de facto para el almacenamiento persistente cuando se desea independizar la capa de negocio del almacenamiento de la información. Esta capa de persistencia permite abstraer al programador Java de las particularidades de una determinada base de datos, proporcionando clases que envolverán los datos recuperados de las tablas. Hibernate busca solucionar la diferencia entre los modelos de datos usados para organizar y manipular datos: el modelo de objetos proporcionado por el lenguaje de programación y el modelo relacional usado en las bases de datos.

Hibernate pone a disposición del diseñador un lenguaje de consulta de datos llamado **HQL (Hibernate Query Language)** que permite acceder a datos mediante programación orientada a objetos y una API para construir las consultas programáticamente. De esta forma, no es necesario emplear consultas SQL para ello, ya que el motor de Hibernate generará las consultas mediante el uso de factorías (patrón de diseño *Factory*) y otros elementos de programación para nosotros.

3. ARQUITECTURA DE HIBERNATE

Hibernate parte de una filosofía de mapear objetos Java normales, o también denominados POJO (Plain Old Java Objects). Para almacenar y recuperar estos objetos de la base de datos, el desarrollador debe mantener una conversación con el motor de Hibernate mediante un objeto especial de la clase **sesión** (clase **Session**) (similar al concepto de conexión de JDBC). Al igual que ocurre con las conexiones JDBC, las sesiones de Hibernate se deben crear y cerrar. Entre la capa de Hibernate y la de base de datos se muestran diferentes APIs Java que usan Hibernate para interactuar con la base de datos.

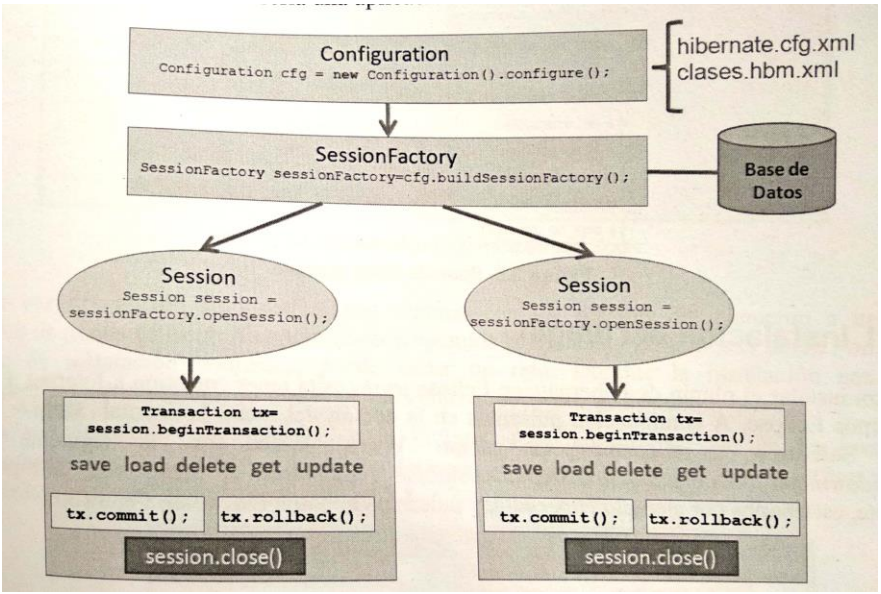
La siguiente figura muestra la arquitectura de Hibernate:



La clase **Session** dispone de varios métodos: save(Object objeto), createQuery(String consulta), beginTransaction(), close(), etc. para interactuar con la base de datos tal y como se hace con una conexión JDBC, con la diferencia que resulta más simple; por ejemplo, guardar un objeto consiste en algo asi como ***session.save(miObjeto)***, sin necesidad de especificar una sentencia SQL.

Una instancia de Session no consume mucha memoria y su creación y destrucción es muy barata. Esto es importante, ya que una aplicación necesitará crear y destruir sesiones todo el tiempo, quizá en cada petición.

Una aplicación Java con Hibernate tiene la siguiente estructura:



Hibernate tiene las siguientes interfaces:

- **Configuration** (*org.hibernate.cfg.Configuration*). Se utiliza para configurar Hibernate. Se 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 se crea la SessionFactory.
- **SessionFactory** (*org.hibernate.SessionFactory*). Permite obtener instancia Session. Esta interfaz se debe compartir entre muchos hilos de ejecución. Normalmente hay una única SessionFactory para toda la aplicación, creada al inicio durante la inicialización de la misma, y utilizada para crear todas las sesiones relacionadas con un contexto dado. Si la aplicación accede a varias bases de datos, se necesitará una SessionFactory por cada base de datos.
- **Query** (*org.hibernate.Query*). Permite realizar consultas a la base de datos, escritas en HQL, y controla cómo se ejecutan dichas consultas. Las consultas se encriben en HQL o en el dialecto de SQL nativo de la base de datos que estemos utilizando. Una instancia Query se utiliza para enlazar los parámetros de la consulta, limitar el número de resultados devueltos y para ejecutar la consulta.
- **Transaction** (*org.hibernate.Transaction*). Permite asegurar que cualquier error que ocurra entre el inicio y el final de la transacción produzca el fallo de la misma.

Hibernate hace uso de varias API de Java, tales como JDBC, JTA (Java Transaction API) y JNDI (Java Naming Directory Interface).

4. INSTALACIÓN Y CONFIGURACIÓN DE HIBERNATE

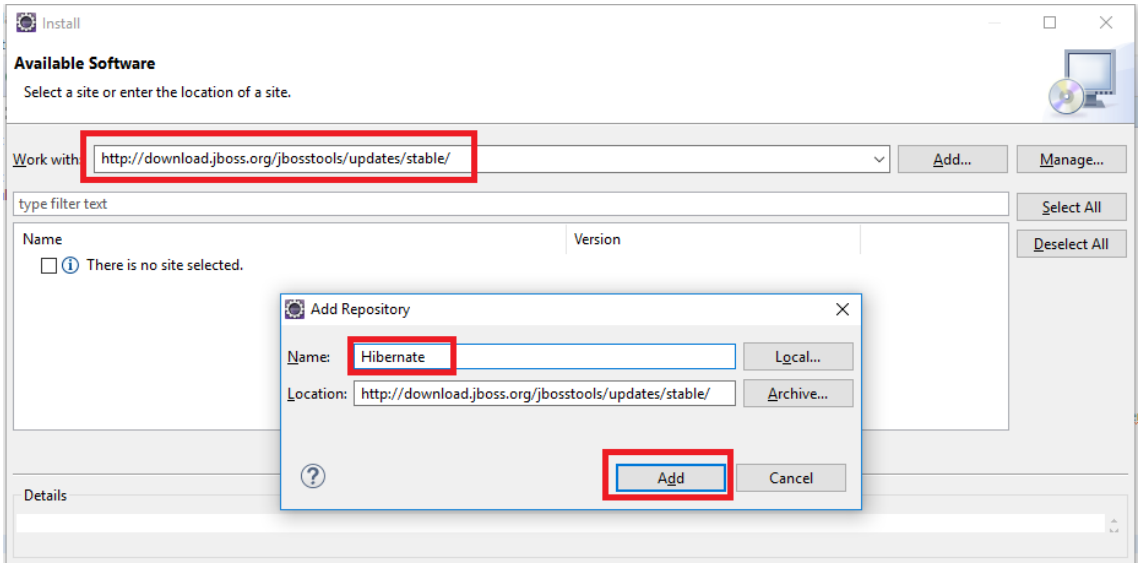
A continuación, se va a detallar el proceso de instalación y configuración de Hibernate en el entorno de desarrollo Eclipse. Previamente se requiere tener un JDK (Java Development Kit) instalado.

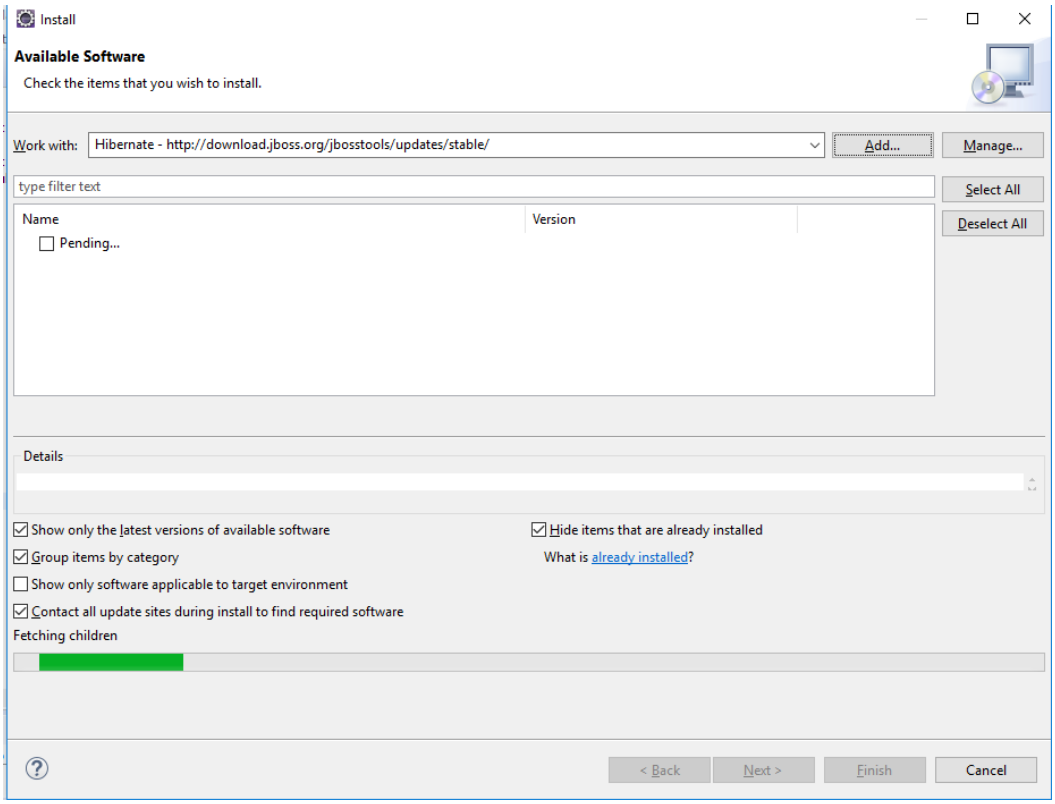
4.1. INSTALACIÓN DEL PLUGIN DE HIBERNATE

JBoss Tools es un conjunto de plugins de Eclipse diseñados para ayudar a desarrollar aplicaciones Java. Contiene herramientas de Hibernate (**Hibernate Tools**), que soportan ficheros de mapeo, anotaciones y JPA con ingeniería inversa, completado de código, asistentes de proyectos, refactorización y ejecución interactiva de sentencias HQL.

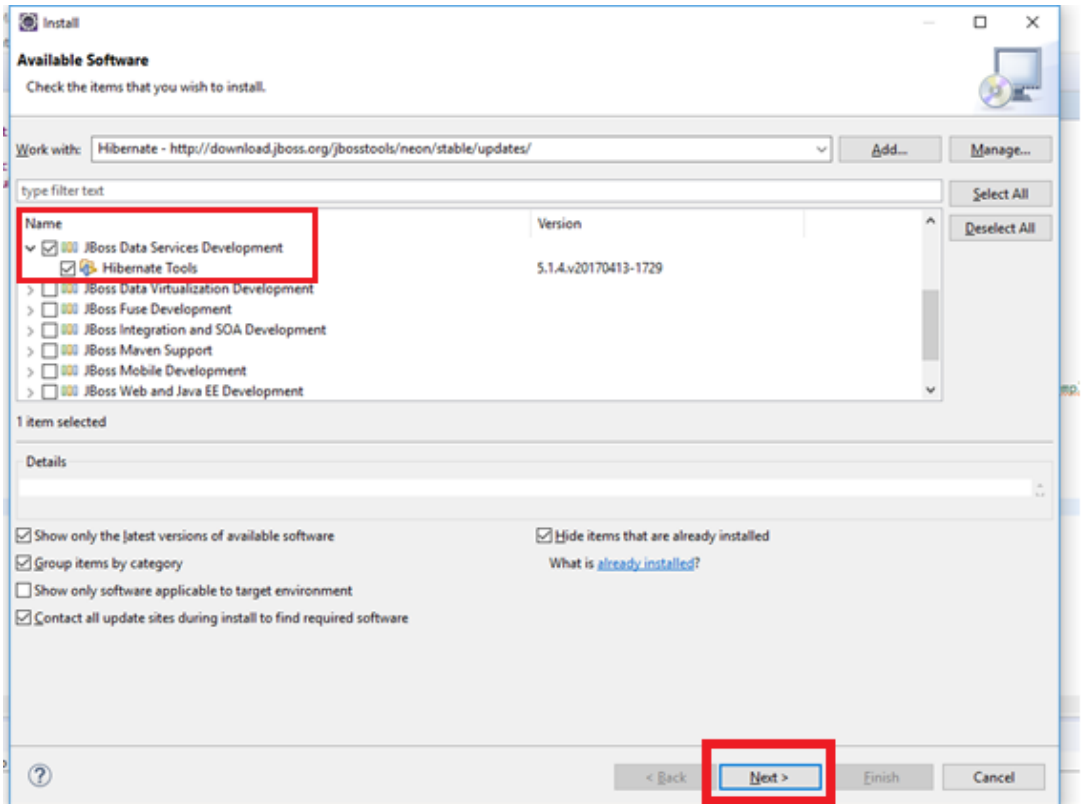
JBoss Tools se encuentra disponible para distintas versiones de Eclipse en la siguiente URL oficial: http://download.jboss.org/jbosstools/updates/stable/

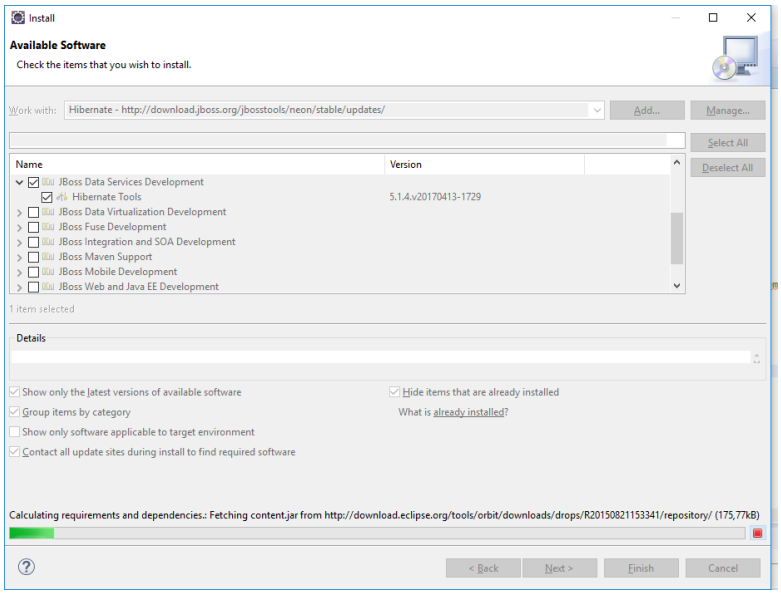
Para instalar el plugin *Hibernate Tools* de *JBoss Tools* se necesita una conexión a internet. Primero inicializamos Eclipse. A continuación pulsamos en la opción del menú Horizontal **Help->Install New Software,** rellenamos el campo **Work With** con la siguiente URL: Accedemos a repositorio general: http://download.jboss.org/jbosstools/updates/stable/, podemos acceder a diferentes versiones que se van actualizando:
http://download.jboss.org/jbosstools/neon/stable/updates/ y
http://download.jboss.org/jbosstools/photon/stable/updates/ y pulsamos el botón *Add,* no pide un nombre y estableceremos por ejemplo Hibernate y pulsamos el botón *Add*.



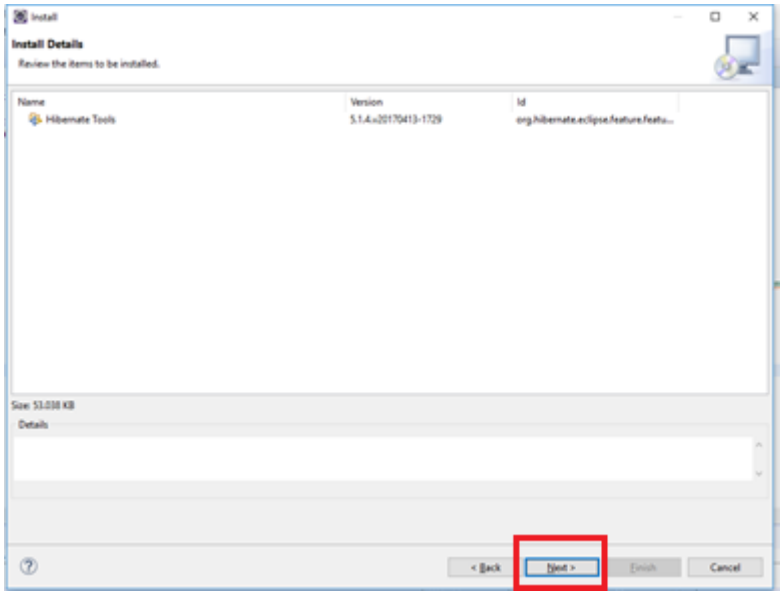


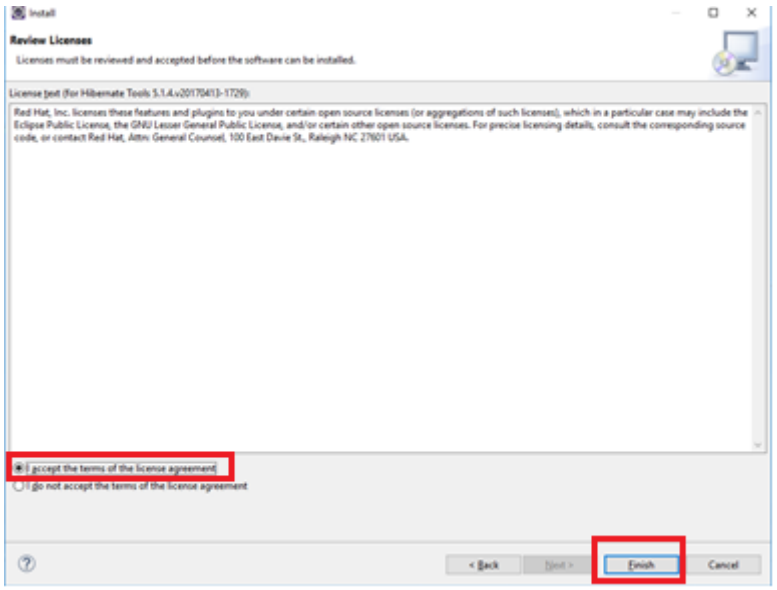
Pasado un rato, aparece la lista de plugins. Pulsamos en la flechita que aparece a la izquierda de *JBoss Data Services Development* y seleccionamos *Hibernate Tools*. A continuación pulsamos el botón *Next,* y comienza el proceso de descarga.



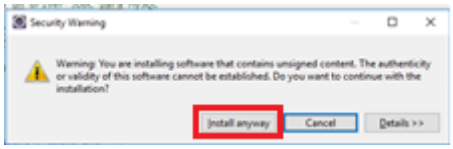


Una vez se haya descargado se visualiza una ventana con los detalles del elemento a instalar, pulsamos de nuevo *Next*. A continuación, aceptamos la licencia y pulsamos el botón *Finish*.



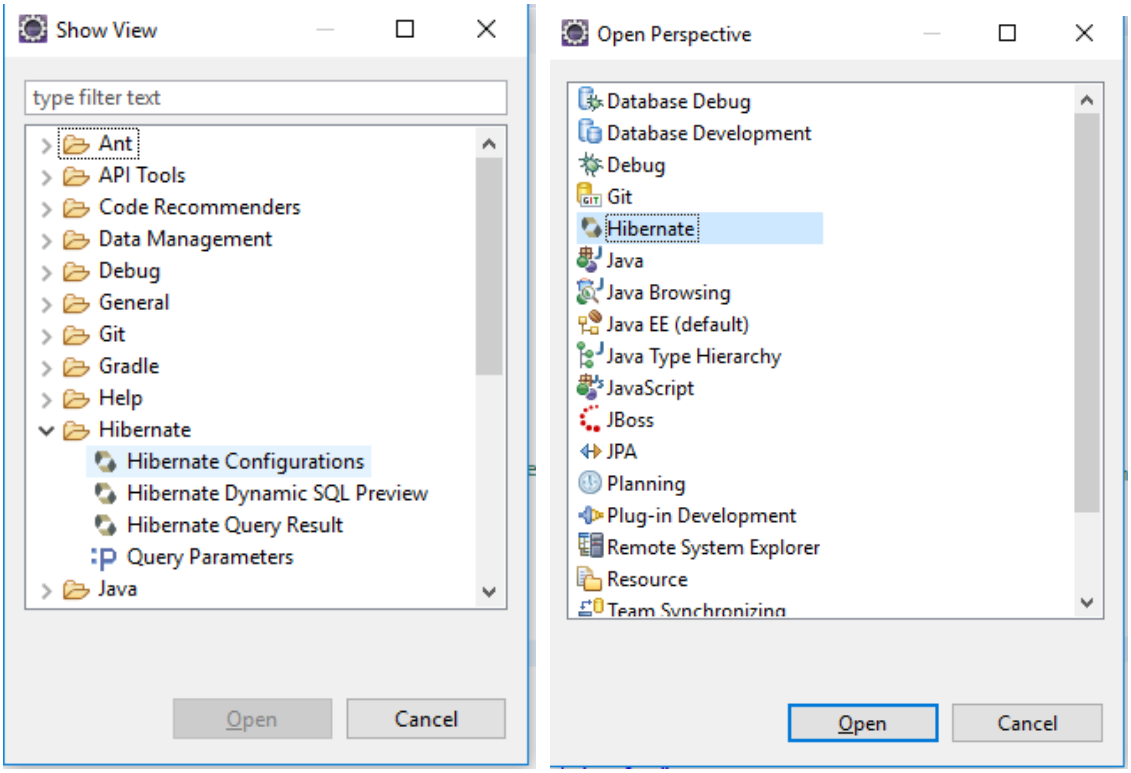


El proceso de instalación comienza, puede tardar un rato. Durante la instalación nos pide confinación para continuar, ya que el plugin contiene software sin firmar.



A continuación, y tras aceptar los términos de licencia, se inicia el proceso de descarga e instalación. Por último, será necesario reiniciar Eclipse para hacer efectivos los cambios realizados.

Unavez instalado, para comprobar que Hibernate se ha instalado correctamente, podemos ir al menú **Window** -> **Show View** -> **Other** o también desde el menú **Window** -> **Open Perspective** -> **Other** -> **Hibernate**, deben aparecer las opciones de Hibernate.



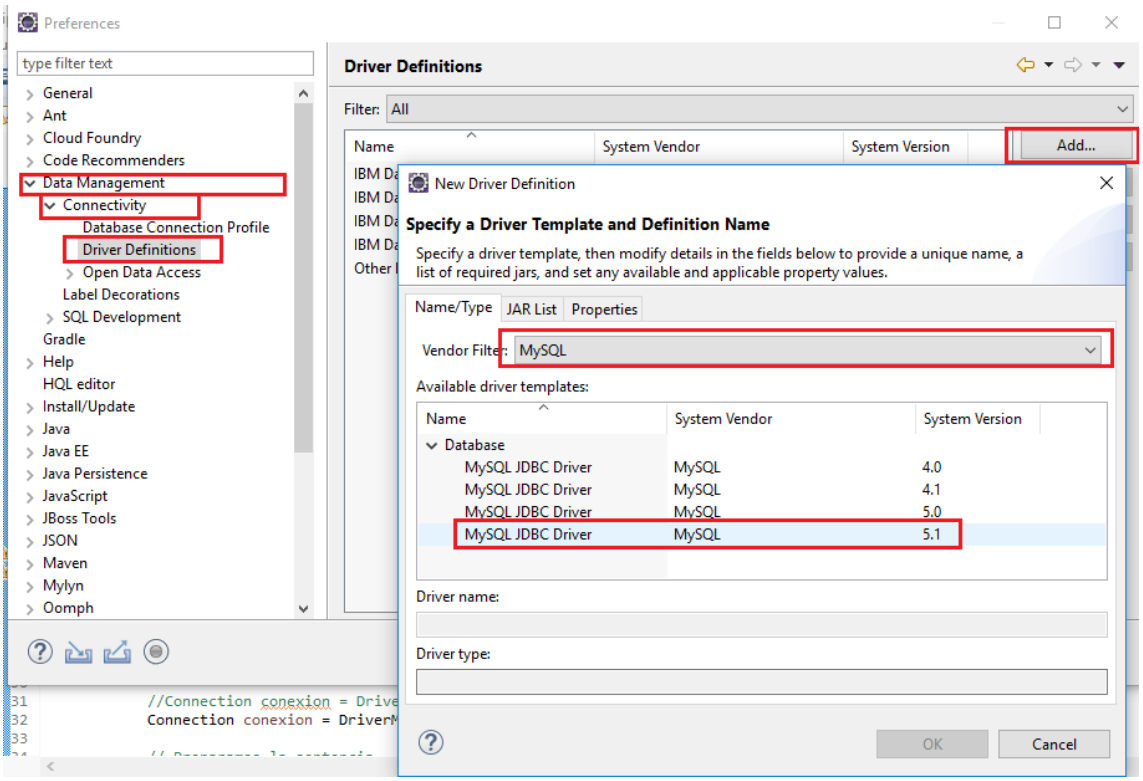
4.2. CONFIGURACIÓN DEL DRIVER DE MYSQL

Una vez instalado Hibernate, el siguiente paso es configurarlo para que se comunique con la base de datos MySQL, vamos a usar la base de datos *ejemplo.*

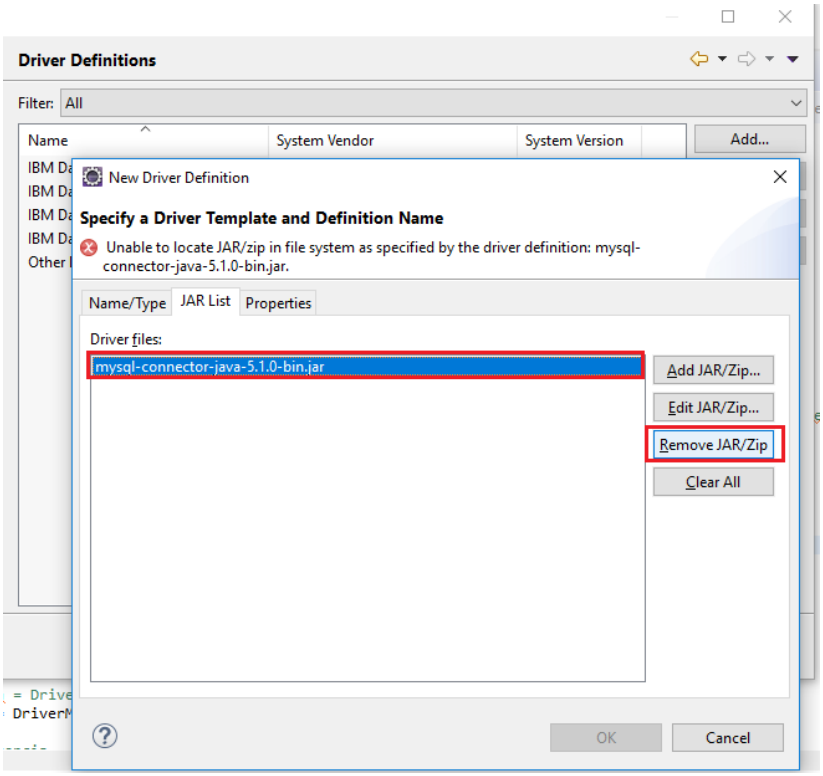
En primer lugar, se debe descargar el driver o conector de MySQL desde la URL http://dev.mysql.com/downloads/connector/j/ e instalarlo (lo tendremos localizado en alguna carpeta de la descarga y uso del tema anterior).

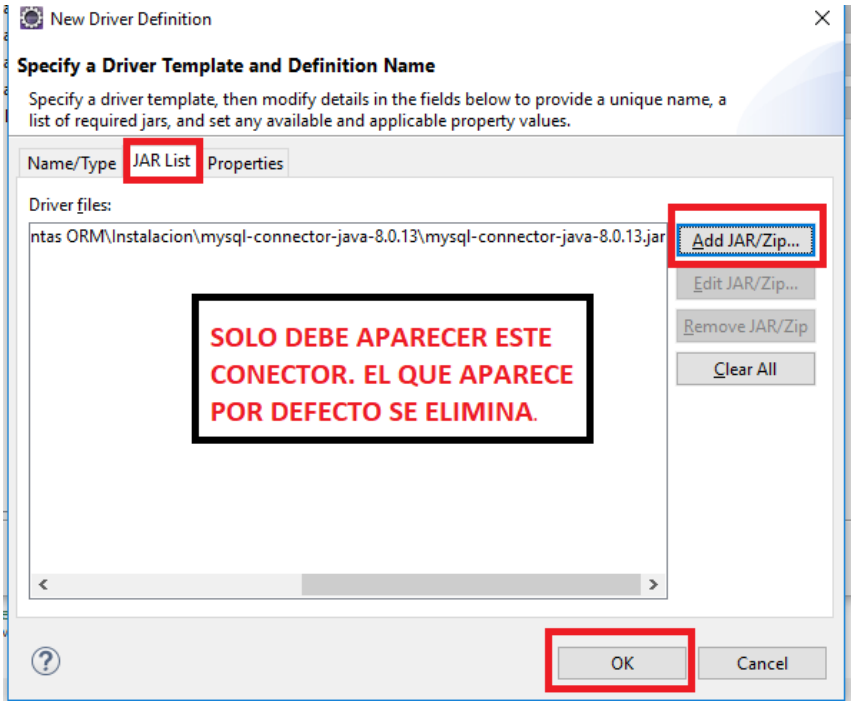
A continuación, desde el entorno Eclipse, vamos al menú **Window** -> **Preferences** -> **Data Management** -> **Driver Definitions** y pulsamos el botón *Add*:

- Desde la pestaña ***Name/Type***: seleccionamos MySQL JDBC Driver en la versión 5.1.



- En la pestaña *JAR List* pulsamos el botón *Add JAR/Zip* para localizar el conector descargado. Si se visualiza el driver por defecto *mysql-connector-java-5.1.0-bin.jar*, lo seleccionamos y lo eliminamos mediante el botón *Remove JAR/Zip*.

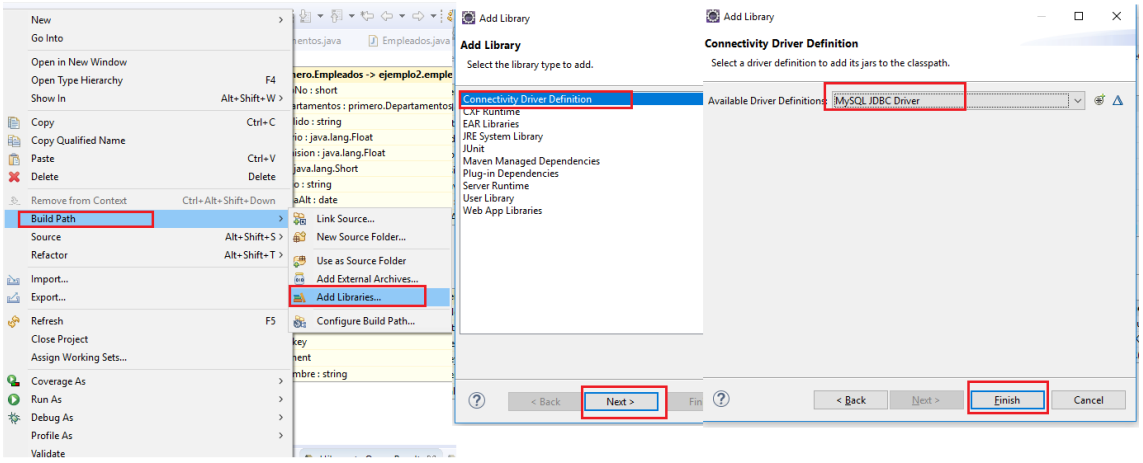




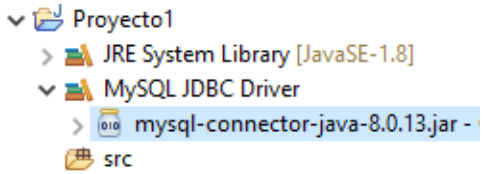
4.3. CREACIÓN DE UN NUEVO PROYECTO

Ahora vamos a crear un nuevo proyecto y configurar Hibernate para que se comunique con MySQL y cree las clases correspondientes de cada tabla una base de datos que se desea mapear. Para ello, pulsamos en el menú **File** -> **New** -> **Project** -> **Java Project**, pulsamos el botón *Next,* nos pide en nombre del proyecto, por ejemplo, *Proyecto1* y pulsamos el botón *Finish*.

Para agregar el driver de MySQL al proyecto creado, en el explorador de paquetes pulsamos sobre el proyecto con el botón derecho del ratón y vamos a **Build Paths** -> **Add Libraries**.



En la ventana que se abre seleccionamos *Connectivity Driver Definition* y pulsamos el botón *Next*. La nueva ventana dispone de varias opciones para conectarse a una fuente de datos. Aquí elegimos *MySQL JDBC Driver* y pulsamos el botón *Finish*. Así vemos el proyecto:

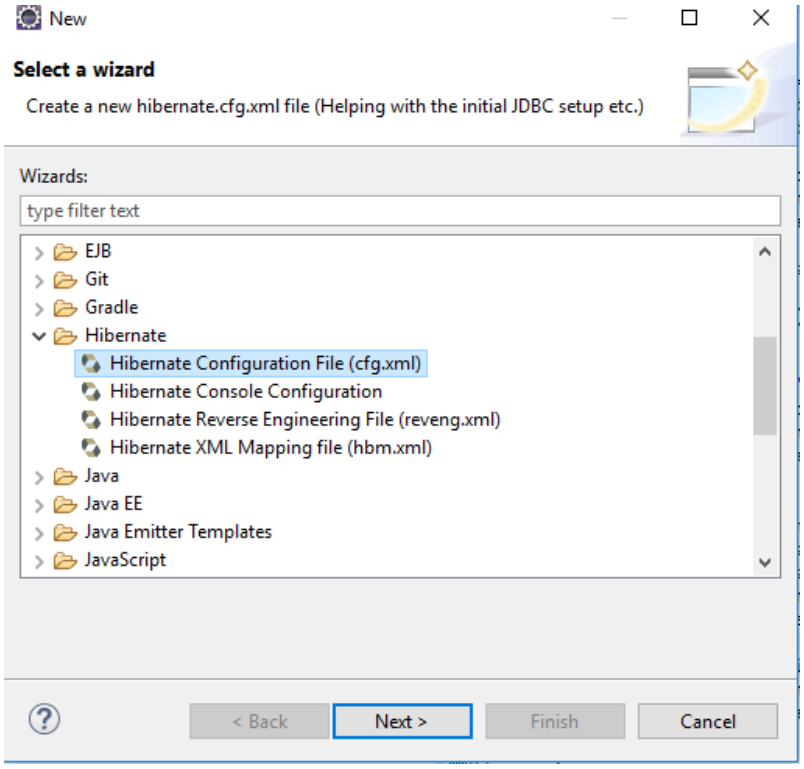


4.4. CONFIGURACIÓN DE HIBERNATE

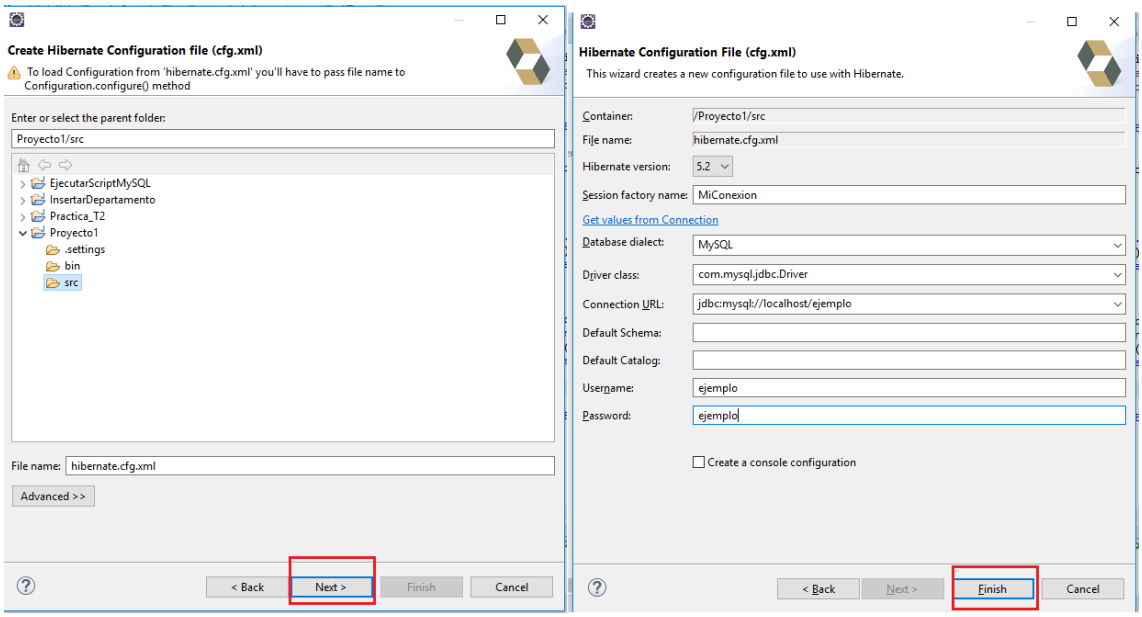
Tras haber añadido el conector de MySQL y las librerías de Hibernate al proyecto, se deben generar los ficheros de configuración de Hibernate:

- Fichero de Configuración *hibernate.cfg.xml*.
- Fichero de Configuración de Consola.
- Fichero de Ingeniería Inversa *reveng.xml*.

En primer lugar, se debe crear el fichero de configuración de Hibernate **hibernate.cfg.xml**. Para ello, seleccionamos el proyecto, pulsamos el botón derecho del ratón y vamos a **New** -> **Other** -> **Hibernate** -> **Hibernate Configuration File (cfg.xml)**. Este fichero es un XML que contiene todo lo necesario para realizar la conexión a la base de datos.



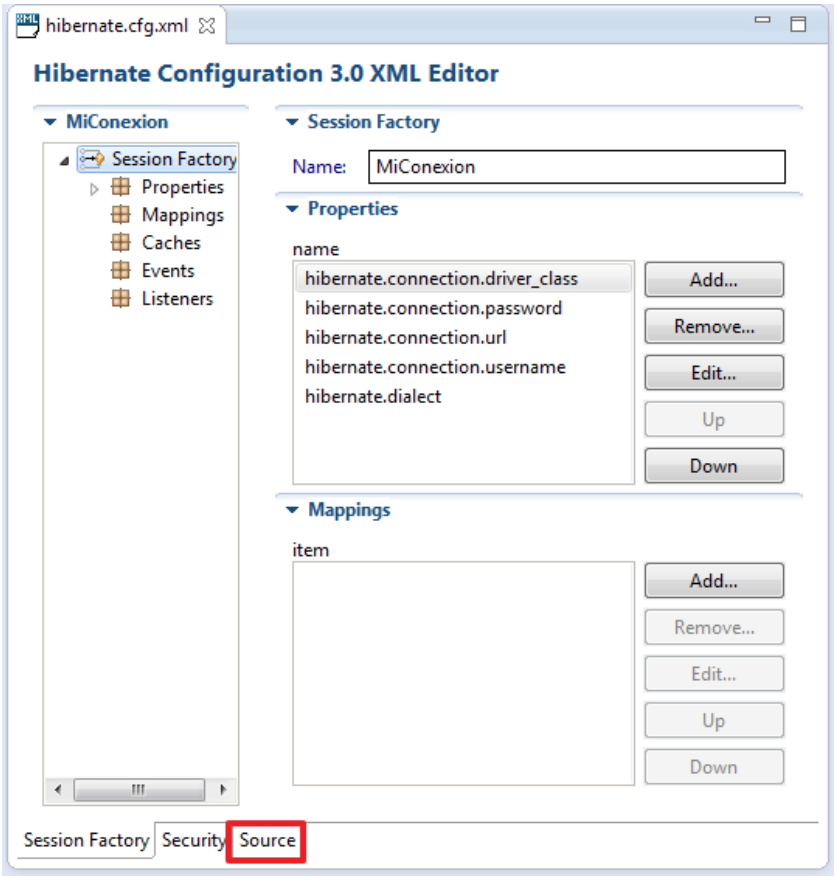
Después de pulsar el botón *Next*, se pide dónde crear el fichero, en este caso, en la carpeta por defecto **src** del proyecto. Seguidamente, se deben introducir los datos necesarios para realizar la conexión a la base de datos:



Los campos a rellenar son los siguientes:

- **Session Factory Name.** Es el nombre de la conexión a MySQL.
- **Database Dialect.** En esta lista desplegable se puede elegir como se comunica JDBC con la base de datos.
- **Driver Class.** Es la clase de JDBC que se va a usar para la conexión.
- **Connection URL.** Es la ruta de conexión a la base de datos: *jdbc:mysql://<hostname>/<databse>*
- **Username.** Es el usuario que se va a conectar a la base de datos MySQL.
- **Password.** Es la clave del usuario que se va a conectar a la base de datos MySQL.

Para terminar pulsamos el botón Finish. Aparece el editor de configuración de Hibernate:



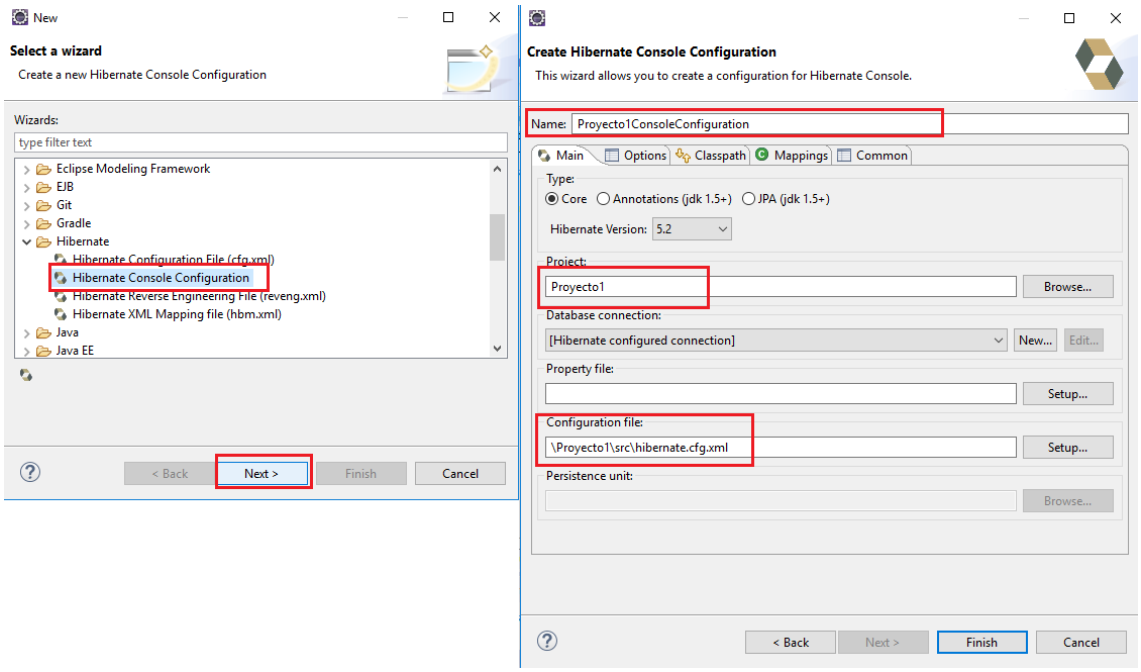
Desde la pestaña *Source* se puede editar el fichero XML **cfg.xml** generado:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="MiConexion">
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.password">ejemplo</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost/ejemplo</property>
        <property name="hibernate.connection.username">ejemplo</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    </session-factory>
</hibernate-configuration>

```

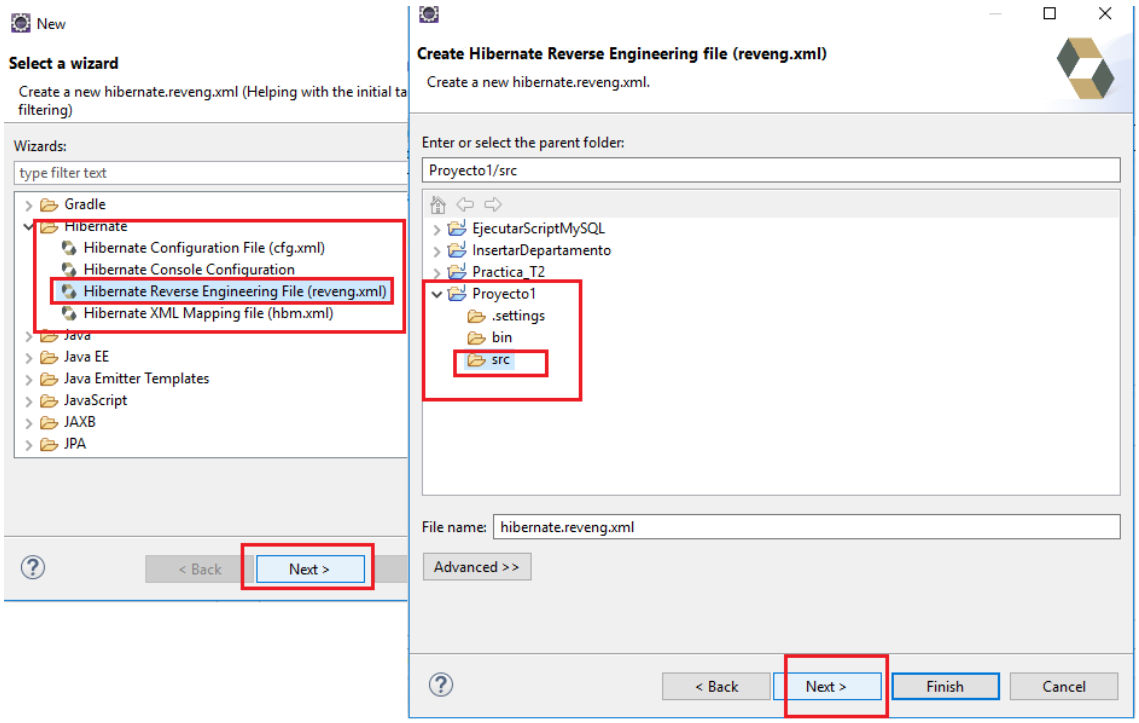
Después de la creación de este fichero *hibernate.cfg.xml*, se debe generar el fichero XML **Hibernate Console Configuration**. Para ello, seleccionamos el proyecto, pulsamos el botón derecho del ratón y vamos a **New** -> **Other** -> **Hibernate** -> **Hibernate Console Configuration** y pulsamos *Next.*.

En el campo *Name* escribimos un nombre para la configuración, nos aseguramos que en el campo Project aparezca nuestro proyecto y en el campo *Configuration File* debe aparecer el fichero de configuración creado anteriormente (*hibernate.cfg.xml*). Pulsamos el botón *Finish* para terminar el proceso de creación.



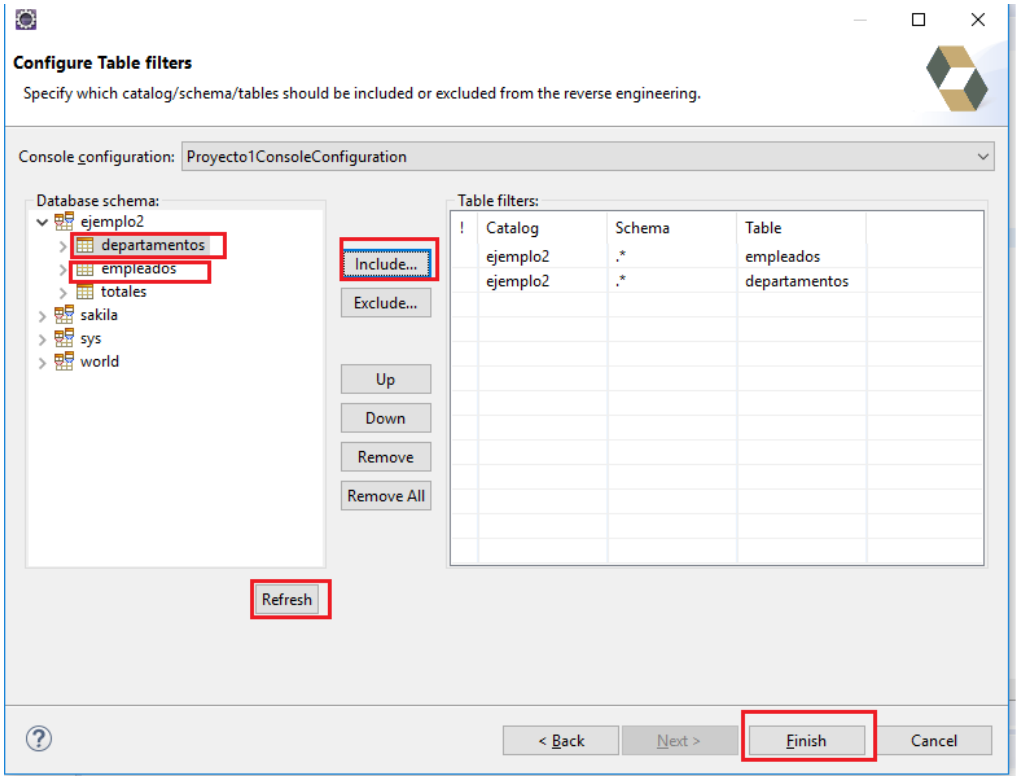
Por último, se debe crear el fichero XML **Hibernate Reverse Engineering (reveng.xml)**, que se encarga de crear las clases correspondientes a las tablas de la base de datos MySQL. Para ello, pulsamos el botón derecho del ratón y seleccionamos **New** -> **Other** -> **Hibernate** -> **Hibernate Reverse Engineering File (reveng.xml)**.

Cuando pulsamos el botón *Next*, se nos pide la ruta donde se va a guardar el fichero, que debe ser la misma carpeta que el fichero *hibernate.cfg.xml*, en este caso, en la carpeta **src**. Pulsamos el botón *Next* para realizar el siguiente paso.

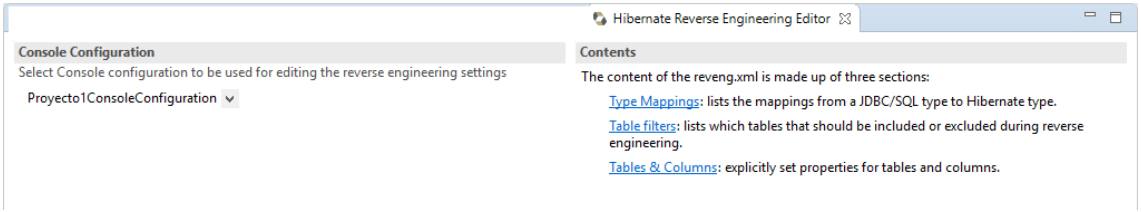


Aparece una nueva ventana en la que se deben indicar las tablas que se desean mapear. En la lista *Console Configuration* seleccionamos el nombre del fichero de *Hibernate Console Configuration* y pulsamos el botón *Refresh* para que se muestren todas las tablas de la base de datos.

Seleccionamos una a una o todas las tablas y pulsamos el botón *Include*. Para finalizar pulsamos el botón *Finish*.



Seguidamente, aparece el editor de *Hibernate Reverse Engineering*. En la pestaña *Source* se puede visualizar el fichero XML **reveng.xml** generado:

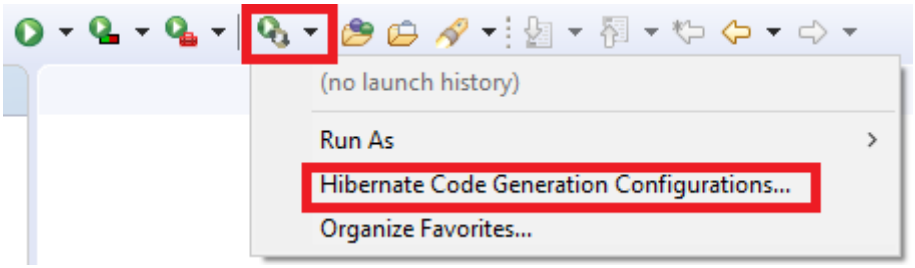



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate Reverse Engineering DTD 3.0//EN" "http://hibernate.sourceforge.n

<hibernate-reverse-engineering>
  <table-filter match-catalog="ejemplo2" match-name="empleados"/>
  <table-filter match-catalog="ejemplo2" match-name="departamentos"/>
</hibernate-reverse-engineering>
```

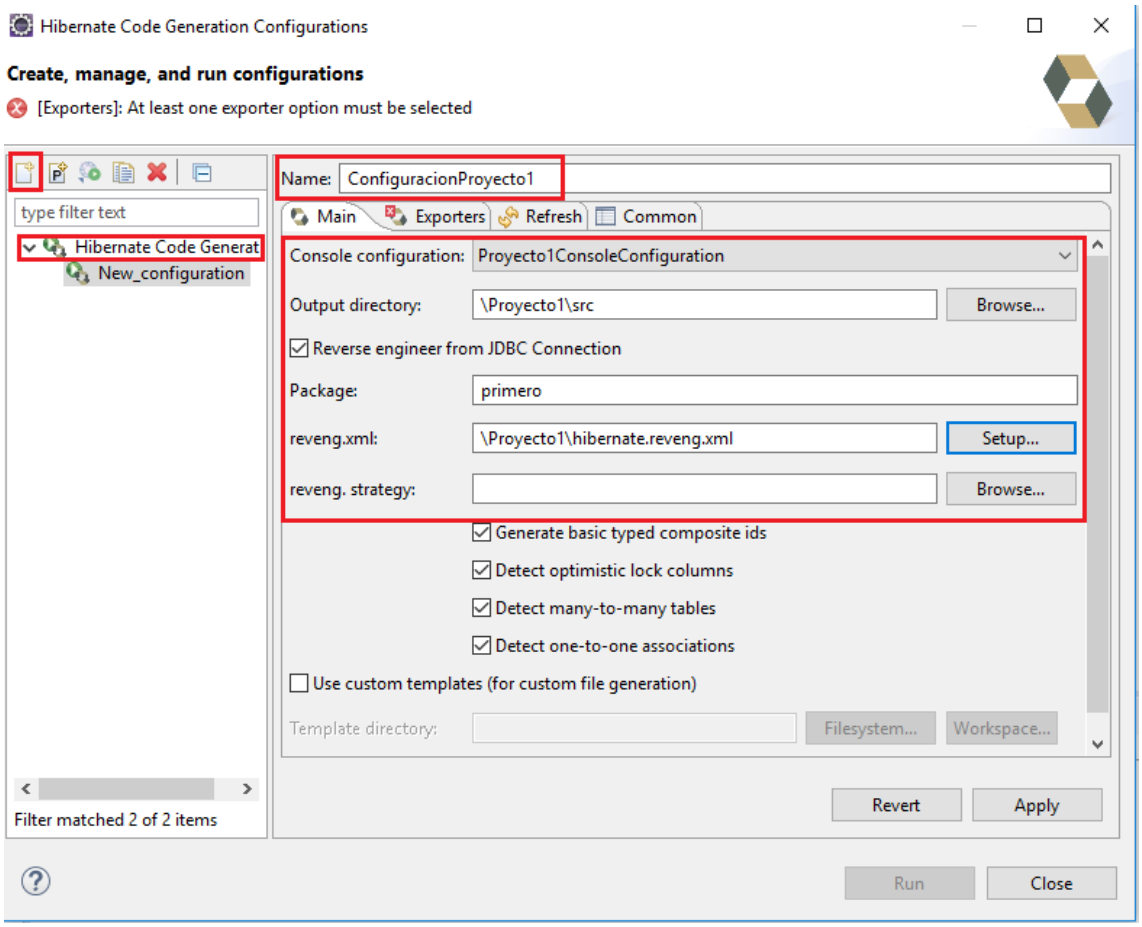
4.5. GENERACIÓN DE CLASES ASOCIADAS

Para generar las clases de la base de datos seleccionada, pulsamos el botón *Run As* en la barra superior de iconos y seleccionamos *Hibernate Code Generation Configurations*:

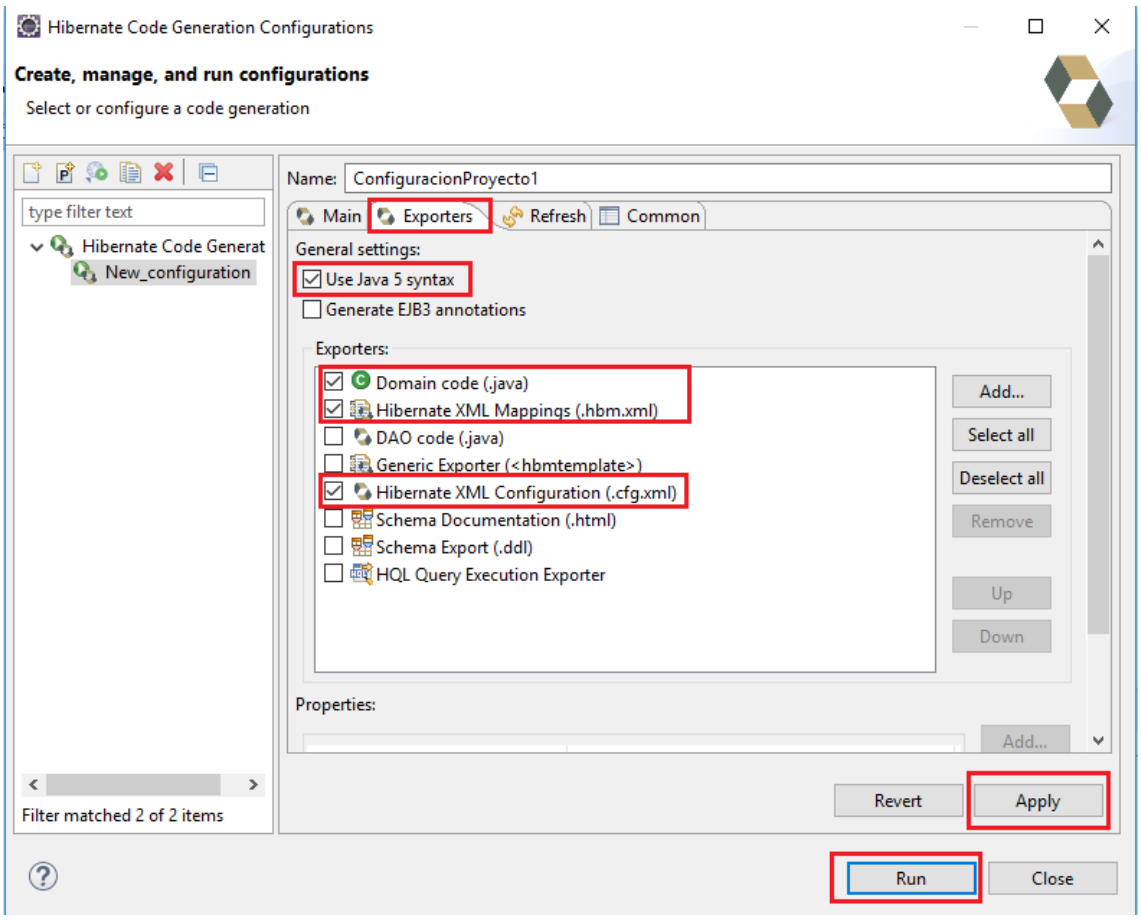


Aparece una ventana con varias pestañas. En la pestaña *Main* se deben rellenar los siguientes campos:

- **Name:** escribimos el nombre de la configuración.
- **Console Configuration.** Es una lista desplegable que muestra todas las configuraciones de consola creadas anteriormente.
- **Output Directory.** Debe ser la carpeta **src**.
- **Reverse Engineer from JDBC Connection.** Indica si se utiliza JDBC para la ingeniería inversa.
- **Package.** Indica el paquete en el que se guardarán las clases Java generadas, le ponemos un nombre.
- **reveng.xml.** Es la ruta del fichero XML *reveng.xml* creado anteriormente. Pulsamos sobre el botón *Setup* para localizarlo.

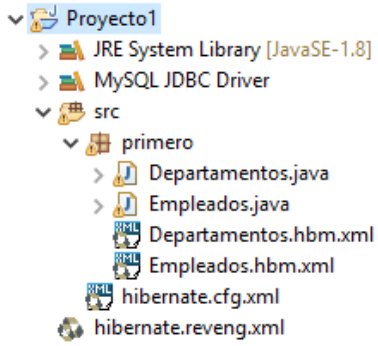


En la pestaña *Exporters* se indican los ficheros que se desea generar. En este caso, marcamos las casillas *Domain Code (.java)*, *Hibernate XML Mapping (.hbm.xml)* e *Hibernate XML Configuration (.cfg.xml).*



Una vez seleccionados los ficheros a generar, pulsamos el botón *Apply* y luego el botón *Run*.

El proyecto con las clases Java y los ficheros hbm.xml generados quedaría así:



En este ejemplo, al ejecutarse se genera un paquete que tiene:

- Las clases Java correspondientes a las tablas EMPLEADOS (*Empleados.java*) y DEPARTAMENTOS (*Departamentos.java*), que incluyen métodos de acceso (*getters* y *setters*) para cada campo de la tabla.
- Los ficheros XML *Departamentos.hbm.xml* y *Empleados.hbm.xml*, que contienen la información del mapeo de la respectiva tabla.

Para cada clase se generan una serie de atributos que tienen que ver con las columnas de la tabla que mapean y las relaciones de claves ajenas, varios constructores; y los métodos getters y setters. En la clase *Departamentos* se observan los siguientes atributos:

```
private byte deptNo;
	private String dnombre;
	private String loc;
	private Set<Empleados> empleadoses = new HashSet<Empleados>(0);
```

Los atributos deptNo, dnombre y loc se corresponden con las columnas de la tabla. El atributo empleadoses define la relación de clave ajena entre las tablas empleados y departamentos. Este atributo sirve para almacenar los empleados del departamento.

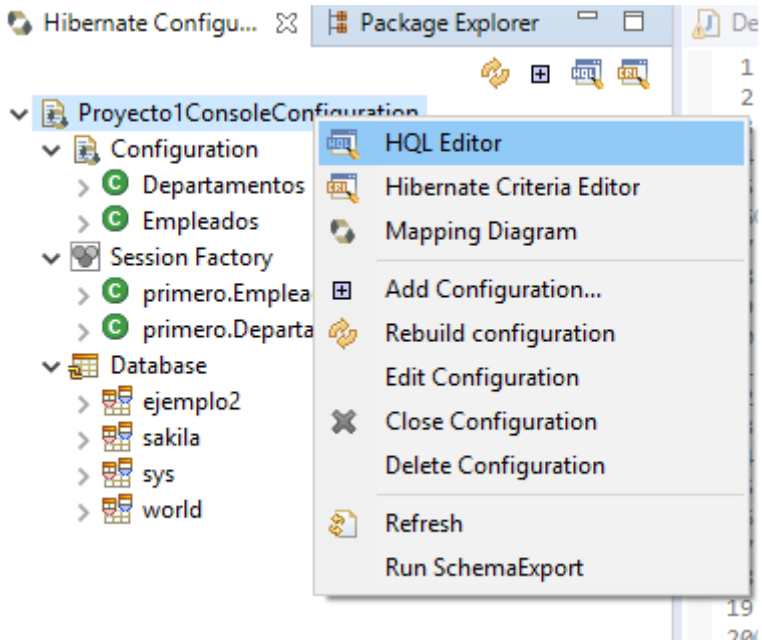
En la clase empleados se observan los siguientes atributos:

```
	private short empNo;
	private Departamentos departamentos;
	private String apellido;
	private String oficio;
	private Short dir;
	private Date fechaAlt;
	private Float salario;
	private Float comision;
```

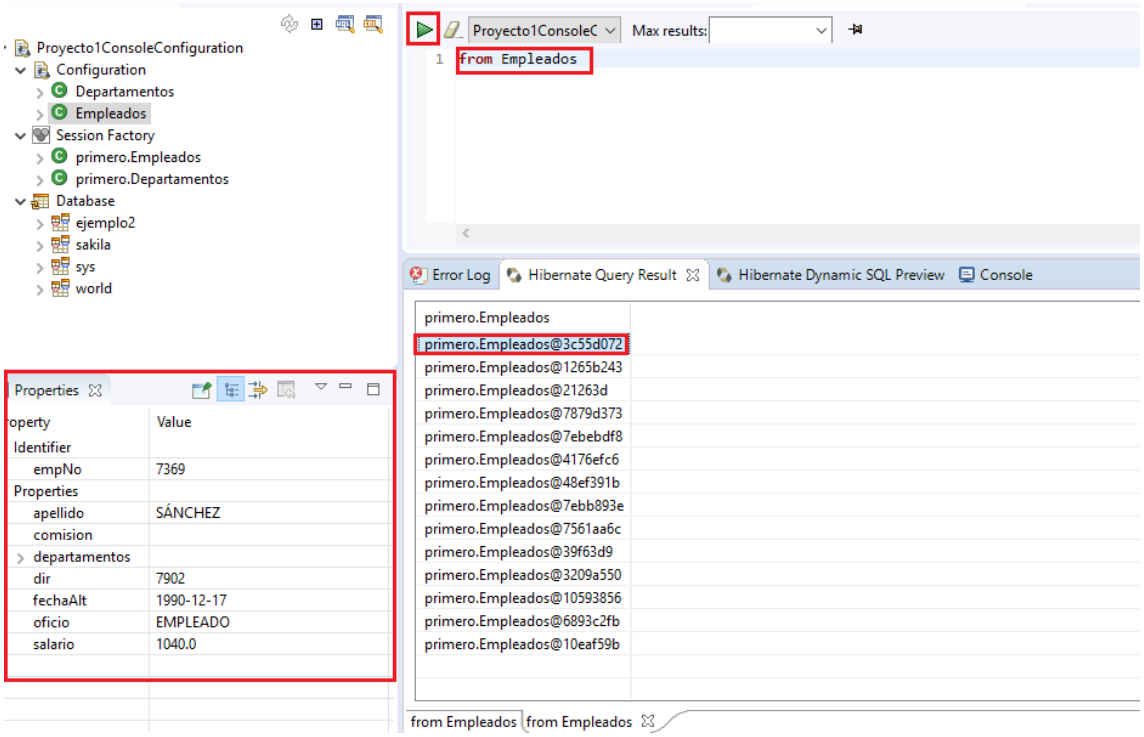
4.6. CONSULTAS EN HQL

Hibernate utiliza un lenguaje de consulta potente (HQL) que se parece a SQL. Sin embargo, comparado con SQL, HQL es completamente orientado a objetos y comprende nociones como herencia, polimorfismo y asociación.

A continuación vamos a realizar consultas en HQL para comprobar si la conexión a la base de datos funciona correctamente. Abrimos la perspectiva de Hibernate desde el menú **Window** -> **Open Perspective** -> **Other** -> **Hibernate**. En la pestaña Hibernate, pulsamos en la configuración y con el botón derecho vamos a **HQL Editor**.



Se abre una nueva pestaña con el mismo nombre que la configuración de Hibernate. Desde aquí podemos escribir sentencias HQL para realizar consultas a la base de datos. A continuación escribimos el siguiente código HQL para consultar los empleados: *from Empleados,* y pulsamos la flechita verde (“*Play”)* para ejecutarla. Desde la ventana **Hibernate Query Result** se puede ver el resultado de la consulta. Si seleccionamos una fila, en el panel Property vemos el contenido de la misma.

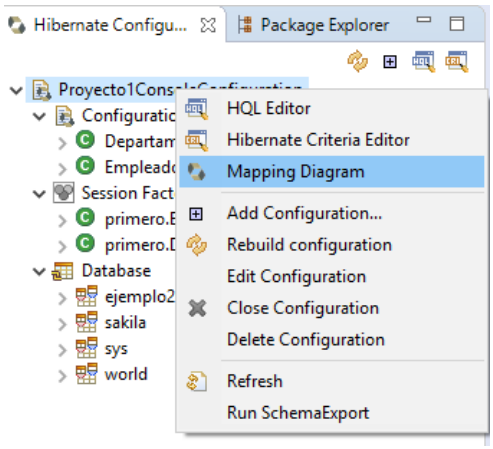


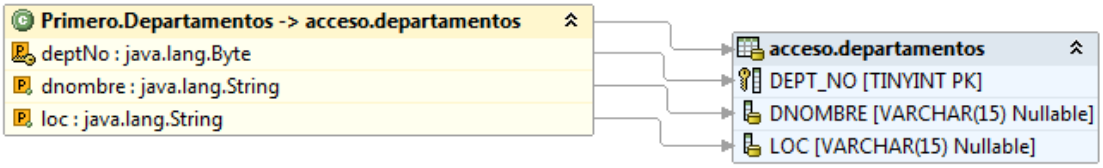
En las consultas HQL se utiliza el nombre de los atributos de las clases que se han generado, no el de los campos de las tablas originales.

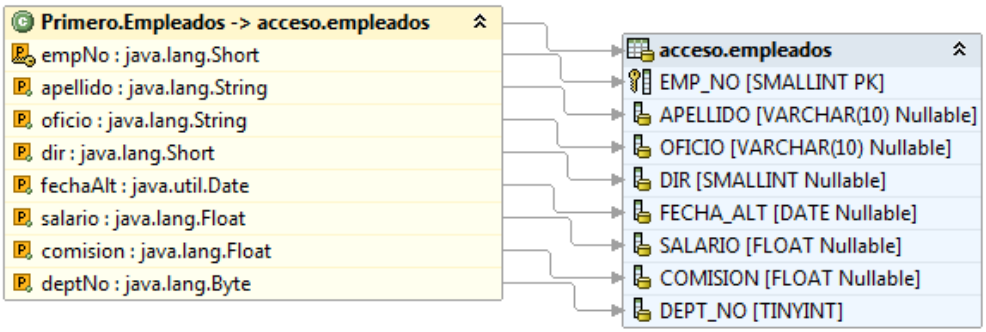
Ejemplos de consultas HQL son los siguientes:

- **from** Empleados
- **select** deptNo, dnombre, loc **from** Departamentos
- **select** empNo, apellido,salario **from** Empleados **where** departamentos.deptNo=1
- **from** Empleados **where** departamentos.deptNo=1 **order** by apellido
- **from** Empleados as em **where** em.empNo=10 **order** by 1 **desc**
- **select** avg(salario) as med, count(empNo) as c **from** Empleados
- **select** avg(salario), count(empNo) **from** Empleados
- **select** avg(salario)+sum(salario), count(empNo) **from** Empleados
- **select** apellido ||'*'||oficio as campo **from** Empleados
- **from** Empleados **where** departamentos.deptNo **in** (1,2)
- **from** Empleados **where** departamentos.deptNo **not in** (1,2)
- **from** Empleados **where** salario **not between** 2000 **and** 3000
- **from** Empleados **where** comision **is not null**
- **from** Empleados **where** comision **is null**
- **select** lower(apellido),coalesce(comision,0) **from** Empleados
- **select** apellido **from** Empleados **where** apellido **like** 'L%'
- **select** de.dnombre, avg(em.salario) **from** Empleados em, Departamentos de **where** em.departamentos.deptNo = de.deptNo

Desde la perspectiva Hibernate y pulsando con el botón derecho del ratón en la configuración, seleccionamos **Mapping Diagram** para visualizar el diagrama de mapeo entre las clases Java generadas y las tablas de la base de datos relacional:


Hibernate Configu...
Package Explorer
HQL Editor
Hibernate Criteria Editor
Mapping Diagram
Add Configuration...
Rebuild configuration
Edit Configuration
Close Configuration
Delete Configuration
Refresh
Run SchemaExport
Session Fact
Database
ejemplo2
sakila
sys
world

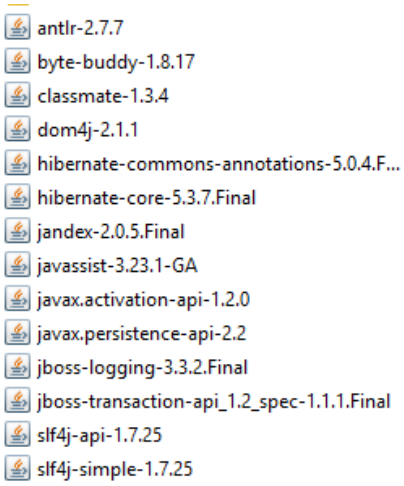

Primero.Departamentos -> acceso.departamentos
deptNo : java.lang.Byte
dnombre : java.lang.String
loc : java.lang.String
acceso.departamentos
DEPT_NO [TINYINT PK]
DNOMBRE [VARCHAR(15) Nullable]
LOC [VARCHAR(15) Nullable]


Primero.Empleados -> acceso.empleados
empNo : java.lang.Short
apellido : java.lang.String
oficio : java.lang.String
dir : java.lang.Short
fechaAlt : java.util.Date
salario : java.lang.Float
comision : java.lang.Float
deptNo : java.lang.Byte
acceso.empleados
EMP_NO [SMALLINT PK]
APELLIDO [VARCHAR(10) Nullable]
OFICIO [VARCHAR(10) Nullable]
DIR [SMALLINT Nullable]
FECHA_ALT [DATE Nullable]
SALARIO [FLOAT Nullable]
COMISION [FLOAT Nullable]
DEPT_NO [TINYINT]

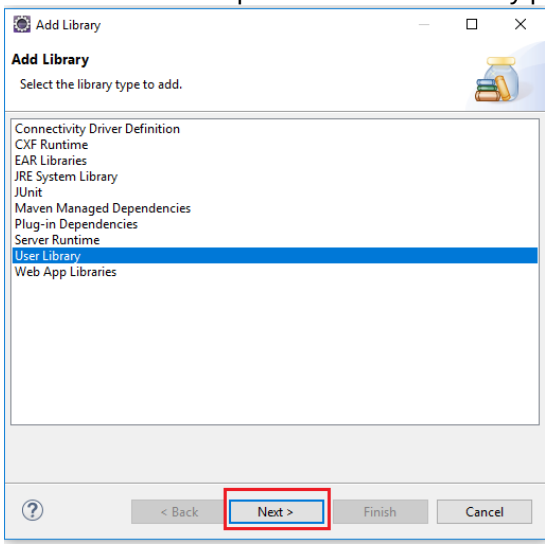
4.7. EMPEZANDO A PROGRAMAR CON HIBERNATE EN ECLIPSE

Con la configuración realizada hasta ahora no se puede programar en Java, es neceario realizar los siguientes pasos:

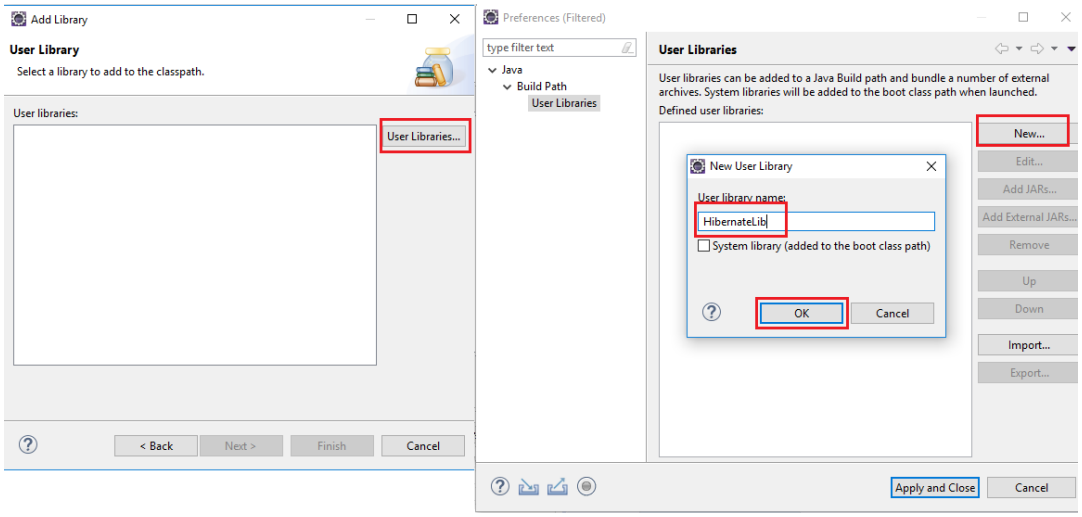
1. Desde la URL http://hibernate.org/orm/downloads/, se debe descargar la última distribución estable de Hibernate., En este caso se ha bajado el fichero **hibernate-release-5.3.7.Final.zip**.
2. A continuación hemos de preparar la distribución para agregarla a Eclipse. Creamos una carpeta dentro de Eclipse llamada por ejemplo Hibernate y aquí es donde copiaremos las librerías que vamos a necesitar, nos debe de quedar: *D:\eclipse\Hibernate*) las librerías que vamos a necesitar. Descomprimimos el ZIP en esta capeta.
3. Desde la capeta *D:\eclipse\Hibernate\hibernate-release-5.3.7.Final\lib\required* seleccionamos todos los ficheros JAR contenidos y los copiamos a nuestra carpeta *D:\eclipse\Hibernate.*
4. Descargamos la última versión de la librería **slf** desde la URL: https://www.slf4j.org/download.html, se ha descargado el fichero **slf4j-1.7.25.zip**, lo descomprimimos y localizamos los ficheros slf4j-api-1.7.25.jar y slf4j-simple-1.7.25.jar para copiarlos en la carpeta *D:\eclipse\Hibernate.*
5. Los fichero JAR que deben contener la carpeta son los que se muestran a continuación:



6. Para agregar las librerías de Hibernate en nuestro proyecto creado, en el explorador de paquetes pulsamos sobre el proyecto con el botón derecho del ratón y vamos a **Build Paths** -> **Add Libraries**. En la ventana que se abre seleccionamos *User Library*. En este paso crearemos una librería con todos los JAR que necesita Hibernate y pulsamos el botón *Next*

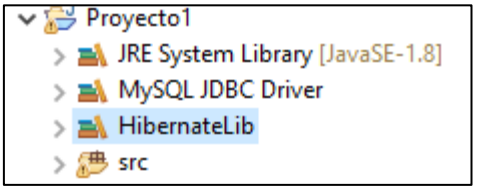


7. En la siguiente ventana pulsamos el botón *User Libraries* y luego pulsamos el botón *New* para crear una nueva librería de usuario. Nos pedirá el nombre de la librería que queremos agregar, escribimos por ejemplo: *HibernateLib* y pulsamos *OK*



8. *Una* vez creada la librería de usuario *HibernateLib*, pulsamos el botón *Add External JARs* para seleccionar todos los ficheros JAR de la carpeta que contiene las librerías de Hibernate (*D:/Eclipse/HibernateLib*). Los seleccionamos todos y pulsamos el botón *Abrir.* Se mostrarña un apantalla con los JAR. Pulsamos *Ok* y a acontinuación el botón *Finish* para finalizar.

En el proyecto aparece la nueva librería creada *HibernateLib*:



Establecimiento de una sesión

En primer lugar vamos a crear una instancia de la base de datos para poder trabajar con ella y que se utilizará a lo largo de la aplicación. Para ello, se requerirá un patrón *singleton*. Necesitamos crear un **Singleton.**

El **Singleton** es un patrón de diseño diseñado para restringir la creación de objetos pertenecientes a una clase. Su intención consiste en garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella (así tenemos un único objeto creado de una clase).

El patrón **Singleton** se implementa creando en nuestra clase un método que crea una instancia del objeto, solo si todavía no existe alguna. Para asegurar que la clase no pueda se instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado).

Nuestro Singleton será una clase de ayuda que accede a **SessionFactory** para obtener un objeto sesión, hay una única **SessionFactory** para toda la aplicación. En la clase se define una variable estática definida, o lo que es lo mismo, devuelve un objeto SessionFactory creado. El nombre de la clase es *HibernateUtil.java* y se incluirá en el paquete *primero* de nuestro proyecto, el código es el siguiente:

```java
package primero;

import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
  private static final SessionFactory sessionFactory = buildSessionFactory();

  private static SessionFactory buildSessionFactory() {
     try {
        //Create the SessionFactory from hibernate.cfg.xml
        return new Configuration().configure().buildSessionFactory(
                new StandardServiceRegistryBuilder().configure().build() );
     }
     catch (Throwable ex) {
        // Make sure you log the exception, as it might be swallowed
        System.err.println("Initial SessionFactory creation failed." + ex);
        throw new ExceptionInInitializerError(ex);
     }
  }

  public static SessionFactory getSessionFactory() {
      return sessionFactory;
  }
}//
```

Con esta clase podemos obtener ña sesión actual desde cualquier parte de nuestra aplicación. Ahora pulsando con el botón derecho del ratón sobre nuestro proyecto creamos una clase de nombre *Main.java*. Esta se creará en el *default package* del proyecto, o en otro paquete que definamos. El método *main()* inserta una fila en la tabla departamentos. El código quedaría:

```java
import primero.*;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.exception.ConstraintViolationException;

public class Main {
    public static void main(String[] args) {

        //obtener la sesion actual
        SessionFactory sesion = HibernateUtil.getSessionFactory();
        //crear la sesion
        Session session = sesion.openSession();
        //crear una transaccion de la sesion
        Transaction tx = session.beginTransaction();

        System.out.println("Inserto una fila en la tabla DEPARTAMENTOS.");

        Departamentos dep = new Departamentos();
        dep.setDeptNo((byte) 60);
        dep.setDnombre("MARKETING");
        dep.setLoc("GUADALAJARA");

        session.save(dep);
        try {
            tx.commit();
        } catch (ConstraintViolationException e) {
            System.out.println("DEPARTAMENTO DUPLICADO");
            System.out.printf("MENSAJE: %s%n",e.getMessage());
            System.out.printf("COD ERROR: %d%n",e.getErrorCode());
            System.out.printf("ERROR SQL: %s%n" ,
                    e.getSQLException().getMessage());
        }
        session.close();
        System.exit(0);
    }
}
```

Con el siguiente código SessionFactory sesion = HibernateUtil.getSessionFactory();se puede obtener la sesión actual, creada por el patrón *singleton*, desde cualquier parte de la aplicación. Este método se deberá usar en todas aquellas clases que realicen operaciones con la base de datos, es decir, se creará a lo largo de todas las clases en las que deseemos realizar operaciones con nuestra base de datos.

Antes de ejecutar la aplicación (si es necesario) debemos editar el fichero *hibernate.cfg.xml* y cambiar la línea:

```
<session-factory name="Miconexion">
```

Por la siguiente eliminando el parámetro name, para evitar un error de ejecución:

```
<session-factory>
```

Veamos un ejemplo de inserción de un empleado en la tabla *empleados* en el departamento10, para el que será necesario crear un objeto de tipo *Departamento*s y asignar como número de departamento el valor 10, es lo que se hace en el método *setDeptNo()* de esta clase:

```
import primero.*;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

public class MainEmpleado {
    public static void main(String[] args) {
        //obtener la sesion actual
        SessionFactory sesion = HibernateUtil.getSessionFactory();
        //crear la sesion
        Session session = sesion.openSession();
        //crear la transanccion de la sesion
        Transaction tx = session.beginTransaction();

        System.out.println("Inserto un EMPLEADO EN EL DEPARTAMENTO 10.");

        Float salario = new Float(1500);// inicializo el salario
        Float comision = new Float(10); // inicializo la comisión

        Empleados em = new Empleados(); // creo un objeto empleados
        em.setEmpNo((short) 4456); // el numero de empleado es 4455
        em.setApellido("PEPE");
        em.setDir((short) 7499); // el director es el numero de empleado 7499
        em.setOficio("VENDEDOR");
        em.setSalario(salario);
        em.setComision(comision);
        //se crea un objeto Departamentos para asignarselo al empleado
        Departamentos d = new Departamentos(); // creo un objeto Departamentos
        d.setDeptNo((byte) 19); // el número de dep es 10
        em.setDepartamentos(d);

        // fecha de alta actual
        java.util.Date hoy = new java.util.Date();
        java.sql.Date fecha = new java.sql.Date(hoy.getTime());
        em.setFechaAlt(fecha);

        session.save(em);
        tx.commit();
        session.close();
        System.exit(0);
    }
}
```

En el ejemplo anterior se ha utilizado los siguientes métodos:

save(): Este método de la sesión (interface **Session)** lo usamos para guardar el objeto, le pasamos como argumento el objeto a guardar: ***save(Objetct object).***

commit():Este método hace un commit de la transacción actual. La transacción se crea al método ***beginTransaction()*** de la sesión actual. Es necesario para que los datos se almacenen en la BD.

close(): Este método se utiliza para cerrar la sesión.

Si el ejemplo anterior lo ejecutamos más de una vez se producirán excepciones de error, ya que el departamento a insertar o el empleado a insertar ya existe. La excepción es la siguiente: `org.hibernate.exception.ConstraintViolationException`, y se produce al hacer el commit en el método `tx.commit()`.

También se producirá un error si al insertar un empleado creamos un objeto departamento que no existía, entonces, se produce la siguiente excepción: `org.hibernate.TransientPropertyValueException;` esta vez ocurre sobre el método `session.save(em)`. El siguiente código controlará las excepciones de la existencia de empleado y de la no existencia de departamento en el programa de inserción de un empleado:

```
try {
    session.save(em);
    try {
        tx.commit();
    } catch (ConstraintViolationException e) {
        System.out.println("EMPLEADO DUPLICADO");
        System.out.printf("MENSAJE: %s%n", e.getMessage());
        System.out.printf("COD ERROR: %d%n", e.getErrorCode());
        System.out.printf("ERROR SQL: %s%n", e.getSQLException().getMessage());
    }
} catch (TransientPropertyValueException e) {
    System.out.println("EL DEPARTAMENTO NO EXISTE");
    System.out.printf("MENSAJE: %s%n", e.getMessage());
    System.out.printf("Propiedad: %s%n", e.getPropertyName());
} catch (Exception e) {
    System.out.println("ERROR NO CONTROLADO....");
    e.printStackTrace();
}
```

5. ESTRUCTURA DE LOS FICHEROS DE MAPEO

Hibernate utiliza unos ficheros de mapeo para relacionar las tablas de la base de datos relacional con las clases Java generadas. Estos ficheros están en formato XML y tienen la extensión *.hbm.xml.*

En nuestro proyecto de ejemplo se han creado los ficheros: **Empleados.hbm.xml** y **Departamentos.hbm.xml**, el primero asociado a la tabla *empleados* y el segundo a la tabla *departamentos*. Estos ficheros se guardan en el mismo directorio que las clases Java *Empleados.java* y *Departamentos.java*. Estas clases representan un objeto *empleados* y un objeto *departamentos* respectivamente y todos los ficheros estaban en el paquete *primero*.

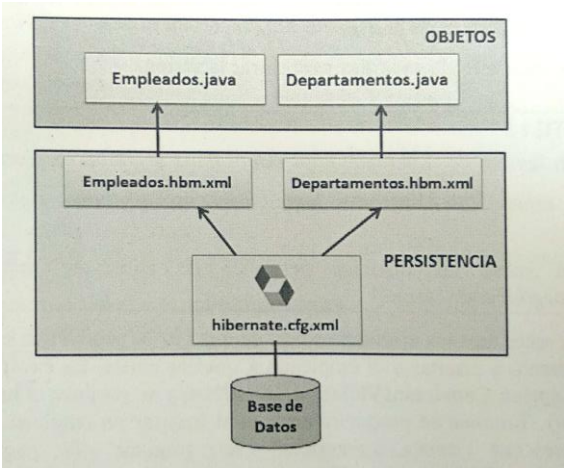


Ilustración 2. Correspondencia entre los ficheros de mapeo y las clases generadas

La estructura del fichero *Departamentos.hbm.xml* es la siguiente:

```
<?xml version="1.0"?>
<!-- Generated 12-nov-2018 3:20:24 by Hibernate Tools 5.2.3.Final --><!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-
mapping-3.0.dtd">
<!-- Generated 12-nov-2018 11:20:24 by Hibernate Tools 5.2.3.Final -->
<hibernate-mapping>
   <class name="primero.Departamentos" table="departamentos" catalog="ejemplo">
      <id name="deptNo" type="byte">
         <column name="dept_no" />
         <generator class="assigned" />
      </id>
      <property name="dnombre" type="string">
         <column name="dnombre" length="15" />
      </property>
      <property name="loc" type="string">
         <column name="loc" length="15" />
      </property>
      <set name="empleadoses" table="empleados" inverse="true" lazy="true" fetch="select">
         <key>
            <column name="dept_no" not-null="true" />
         </key>
         <one-to-many class="primero.Empleados" />
      </set>
   </class>
</hibernate-mapping>
```

Cada fichero de mapeo XML tiene las diferentes etiquetas, vamos a conocer su significado:

- **<hibernate-mapping>.** Delimita el documento XML de mapeo. Todos lo ficheros de mapeo comienzan y terminan con esta etiqueta.
- **<class>.** Engloba a la clase con sus atributos, indicando siempre el mapeo a la tabla de la base de datos:
  - *name.* Es el nombre de la clase.
  - *table.* Es el nombre de la tabla que representa ese objeto.
  - *catalog.* Es el nombre de la base de datos.

```
<class name="primero.Departamentos" table="departamentos" catalog="ejemplo">
```

- **<id>.** Representa el atributo clave en la clase, se encuentra por tanto dentro de *class*:
  - *name.* Campo que representa el nombre del atributo clave en la clase.
  - *type.* Es el tipo de datos del atributo.
  - *column.* Es el nombre de la columna de la tabla asociada al atributo.
  - *generator.* Indica la naturaleza del campo clave: *assigned* si el usuario introduce la clave o *increment* si es autogenerado por la base de datos.

En nuestro ejemplo, este atributo en la tabla Empleados correspondería a la columna *dept_no(* `dept_no tinyint(2) NOT NULL PRIMARY KEY):`

```
<id name="deptNo" type="byte">
   <column name="dept_no" />
   <generator class="assigned" />
</id>
```

- **<property>.** Especifica un atributo no clave en la clase, para el resto de atributos:
  - *name.* Es el nombre del atributo.
  - *type.* Es el tipo de datos del atributo.
  - *column.* Es el nombre del campo de la columna de la tabla asociada al atributo.

La columna *dnombre* en la tabla *departamentos* (`dnombre VARCHAR(15))` sería así:

```
<property name="dnombre" type="string">
  <column name="dnombre" length="15" />
</property>
```

La etiqueta **set** se utiliza para mapear colecciones. Dentro de **set** se definen varios atributos.

- **<set>.** Mapeo de colecciones:
  - *name.* Es el nombre del atributo generado en la clase Java.
  - *table.* Es el nombre de la tabla de done se tomará la colección.
  - *key.* Nombre de la columna identificadora en la asociación.
  - *one-to-many.* Define la relación con una asociación uno-a-muchos.
  - *class.* Indica de qué tipos son los elementos de la colección.

En el caso del ejemplo propuesto, *key* correspondería a la columna *dept_no* de la tabla *departamentos*; para *one-to-many* define que un departamento puede tener muchos empleados. Por lo que en nuestro ejemplo, se va a especificar que la clase *Departamento.java* tiene un atributo de nombre *empleadores* que es una lista de instancias de la clase *primero.Empleados:*

```
<set name="empleadoses" table="empleados" inverse="true" lazy="true" fetch="select">
  <key>
    <column name="dept_no" not-null="true" />
  </key>
  <one-to-many class="primero.Empleados" />
</set>
```

Los tipos declarados y utilizados en los ficheros de mapeo no son tipos de datos Java ni tipos de la base de datos, sino *que son tipos de mapeo Hibernate*, es decir, convertidores que pueden traducir de tipos de datos Java a tipos de la base de datos (SQL) y viceversa. Hibernate tratará de determinar de forma automática el tipo correcto de conversión y de mapeo por sí mismo.

Además, cada fichero de mapeo puede tener etiquetas que proporcionen información sobre las asociaciones o relaciones entre tablas. Puede aparecer una asociación unidireccional **muchos a uno** (*many-to-one,* una clave ajena en una tabla referencia la columna/columnas de la clave primaria de la tabla destino) como por ejemplo muchos empleados pertenecen a un departamento; también asociación **uno a muchos** (*one-to-many) o* **muchos a muchos** (*many-to-many*).

- **<many-to-one>.** Relación muchos a uno:
  - *name.* Es el nombre del atributo en la clase Java.
  - *class.* Indica la clase de referencia.
  - *column name.* Es el nombre de la columna de la tabla.

En nuestro ejemplo, el mapeo indica que la clase *Empleados.java* tiene un atributo de nombre *departamentos* que es una instancia de la clase *primero. Departamentos.*

```
<many-to-one class="primero.Departamentos" embed-xml="true" fetch="select" insert="true"
name="departamentos" not-found="exception" optimistic-lock="true" unique="false" update="true">
  <column name="dept_no" not-null="true"/>
</many-to-one>
```

El atributo *fetch (*por defecto *select*) escoge la recuperación de la unión exterior (outer-join) o la recuperación por selección secuencial.

- OneToOne Indica que el objeto es parte de una relación 1-1
- ManyToOne Indica que el objeto es parte de una relación N-1. En este caso el atributo sería el lado 1
- OneToMany Indica que el objeto es parte de una relación 1-N. En este caso el atributo sería el lado N
- ManyToMany Indica que el objeto es parte de una relación N-M. En este caso se indica la tabla que mantiene la referencia entre las tablas y los campos que hacen el papel de claves ajenas en la base de datos.

6. CLASES PERSISTENTES

Cada clase que se genera durante el mapeo se denomina **clase persistente**, equivale a una tabla de la base de datos relacional y tiene atributos y métodos de acceso (*getters* y *setters*). Cada registro o fila de una tabla se corresponde con un **objeto persistente** de esa clase.

Hemos visto anteriormente que entre las etiquetas `<hibernate-mapping> </hibernate-mapping>` de los ficheros XML se incluye un elemento **class** que hace referencia a un clase:

```
<class catalog="ejemplo" name="primero.Departamentos" table="departamentos">
<class catalog="ejemplo" name="primero.Empleados" table="empleados">
```

En nuestro proyecto se han generado las clases *Empleados.java y Departamentos.java.* A estas clases se les llaman **clases persistentes**¸ son las clases que implementan las entidades del problema y deben implementar la clase *Serializable.*

Utilizan convenciones de nombreado estándares de JavaBean para los métodos de propiedades *getter y setter* asi como también visibilidad provada para los campos. Al ser los atributos de los objetos privados se crean métodos públicos para retornar un valor de un atributo, método *getter*, o para cargar un valor a un atributo método *setter.* Por ejemplo *getDnombre()* devuelve el nombre de un departamento (atributo *dnombre*) y e método *setDnombre()*carga un valor en el atributo *dnombre*. A estas reglas también se las llama modelo de programación POJO (Plain Old Java Object).

Para completar el nombre de un método *getter o setter*, solo hay que poner la primera lestra que los une en mayúsculas. Si nos fijamos en el fichero *Departamento.hbm.xml*, el elemento *id* es la declaración de la propiedad identificadora, el atributo de mapeo *name="deptNo"* declara el nombre de la propiedad JavaBean y le dice a Hibernate que utilice los métodos *getDeptNo() y setDeptNo()* para acceder a la propiedad:

```
<id name="deptNo" type="byte">
```

Al igual con el elemento *id,* el atributo *name* del elemento *propierty* le dice a Hibernate qué métodos *getter y setter* utilizar. Así que en este caso, Hibernate buscara los métodos *getDnombre(), setDnombre, getLoc() y setLoc().*

```
<property name="dnombre" type="string">
<property name="loc" type="string">
```

7. SESIONES Y OBJETOS EN HIBERNATE

Página web de referencia:
http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/objectstate.html

Para poder utilizar los mecanismos de persistencia de Hibernate se debe inicializar el entorno Hibernate y obtener un objeto **Session** utilizando la clase **SessionFactory** de Hibernate. El siguiente fragmento de código muestra este proceso:

```
// inicializar el entorno Hibernate
Configuration cfg = new Configuration().configure();
//crear el ejemplar de Session Factory
SessionFactory sessionFactory = cfg.buildSessionFactory();
// obtener un objeto sesión
Session session = sessionFactory.openSession();
```

La llamada a *Configuration().configure()* carga el fichero de configuración **hibernate.cfg.xml** e inicializa el entorno de Hibernate. Una vez inicializada la configuración, se crea el ejemplar de *SessionFactory*, que normalmente solo se crea una vez y se utiliza para crear todas las sesiones relacionadas con un contexto dado. Esto es lo que hicimos en el proyecto de ejemplo creado *Proyecto* al crear la calse *HibernateUtil.java*, para crear una vez el ejemplar de SessionFactory.

7.1. 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. Cuando se haya completado todo el trabajo de una **transacción**, se debe cerrar la sesión. El siguiente fragmento de código muestra una sesión de persistencia de Hibernate:

```
// comenzar la sesión
Session session = sessionFactory.openSession();
// comenzar la transacción
Transaction tx = session.beginTransaction();
//
// código de persistencia
//
// validar la transacción
tx.commit();
// finalizar la sesión
session.close();
```

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

7.2. ESTADOS DE UN OBJETO

Hibernate define y soporta los siguientes estados de un objeto: **transitorio**, **persistente** y **separado**.

Un objeto es **transitorio (Transient)** si ha sido recién instanciado utilizando el operador *new* y no está asociado a una sesión de Hibernate. No tiene una representación persistente en la base de datos y no se le ha asignado un valor identificador. Las instancias transitorias serán destruidas por el recolector de basura si la aplicación no mantiene más una referencia. Utiliza la *Session* de Hibernate para hacer un objeto persistente (y deja que Hibernate se ocupe de las declaraciones SQL que necesitan ejecutarse para esta transacción).

Las instancias recién instanciadas de una clase persistente, Hibernate las considera como transitorias. Podemos hacer que una instancia transitoria persistente asociándola con una sesión:

```
//Inserto una fila en la tabla DEPARTAMENTOS
Departamentos dep = new Departamentos();
dep.setDeptNo((byte) 57);
dep.setDnombre("MARKETING");
dep.setLoc("GUADALAJARA");
session.save(dep); //save() hace que la instancia sea persistente
```

Un objeto **persistente (Persistent)** tiene una representación en la base de datos y un valor identificador. Puede haber sido guardado o cargado, sin embargo, por definición, se encuentra en el ámbito de una *Session*. Hibernate detectará cualquier cambio realizado a un objeto en estado persistente y sincronizará el estado con la base de datos cuando se complete la unidad de trabajo. Los desarrolladores no ejecutan sentencias *UPDATE* o sentencias *DELETE* cuando un objeto se deba poner como transitorio.

Es decir, los objetos transitorios solo existen en memoria y no en un almacén de datos, han sido instanciados por el desarrollador sin haberlos almacenado mediante una sesión. Los persistentes se caracterizan por haber sido ya creados y almacenados en una sesión o bien devueltos en una consulta realizada con la sesión.

Un objeto **separado** (**Detached)** es aquel que se ha hecho persistente, pero su *Session* ha sido cerrada. Un objeto esta en este estado cuando cerramos la sesión mediante el método *close()* de *Session.*

La referencia al objeto todavía es válida, por su puesto, y la instancia separada podría incluso ser modificada en este estado. Una instancia separada puede ser asociada a una nueva *Session* más tarde, haciéndola persistente de nuevo (con todas las modificaciones).

En resumen:

- *Objeto transitorio*: nunca persistente, no asociado a ninguna sesión.
- *Objeto persistente*: asociado con una única sesión.
- *Objeto separado*: anteriormente persistente, no asociado a ninguna sesión.

7.3. CARGA DE OBJETOS

La interfaz *Session* de Hibernate dispone de varios métodos para cargar objetos persistentes:

- **load().** Recupera un objeto persistente de la base de datos conociendo el identificador del registro. Lanza una excepción irrecuperable si la fila no existe. Si no se está seguro de la existencia de la fila se debe utilizar el método *get().*
- **get().** Recupera un objeto persistente de la base de datos conociendo el identificador del registro o devuelve nulo si la fila correspondiente no existe.

MÉTODO	DESCRIPCIÓN
`<T> T load(Class<T> Clase, Serializable id)`	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
`Object load(String nombreClase, Serializable id)`	Similar al método anterior, pero en este caso indicamos en el primer parámetro el nombre de la clase en formato de *String*
`<T> T get(Class<T> Clase, Serializable id)`	Devuelve la **instancia persistente** de la clase indicada con el identificador dado. Si la instancia no existe, devuelve *null*
`Object get(String nombreClase, Serializable id)`	Similar al método anterior, pero en este caso indicamos en el primer parámetro el nombre de la clase

El siguiente ejemplo utiliza el método *load()* para obtener los datos del departamento 20. En primer parámetro se indica la clase *Departamento* y en el segundo el número de departamento que se quiere recuperar, se hace un *cast* para convertirlo al tipo de dato definido en el atributo identificador de la clase (*deptNo*) que es un byte. Con el segundo formato del método *load()* también podemos obtener el departamento. Si la fila no existe se lanza la excepción *ObjectNotFoundException*:

```
        // Visualizo los datos del departamento 20
        Departamentos dep = new Departamentos();
        try {
            //dep = (Departamentos) session.load(Departamentos.class, (byte) 20);
            dep = (Departamentos) session.load("primero.Departamentos", (byte) 20);

            System.out.printf("Nombre Dep: %s%n", dep.getDnombre());
            System.out.printf("Localidad: %s%n", dep.getLoc());
        } catch (ObjectNotFoundException o) {
            System.out.println("NO EXISTE EL DEPARTAMENTO!!");
        }
```

El método *load()* lanza la excepción *ObjectNotFoundException* si la fila no existe. Si no tenemos la certea de que la fila exista debemos utilizar el método *get()*, que llama a la base de datos inmediatamente y devuelve *null* si no existe la fila correspondiente, vamos a ver un ejeplo que comprueba si el departamento 11 existe. Si existe se visualizan sus datos, y si no existe aparece un mensaje indicándolo:

```
Departamentos dep = (Departamentos) session.get(Departamentos.class, (byte) 10);
if (dep==null) {
    System.out.println("El departamento no existe");
}
else
{
    System.out.printf("Nombre Dep: %s%n",dep.getDnombre());
    System.out.printf("Localidad: %s%n",dep.getLoc());
}
```

En el próximo ejemplo obtenemos los datos del departamento 10 y el APELLIDO y SALARIO de sus empleados, para obtener los empleados usamos el método *getEmpleadoses()* de la clase *Departamentos* y usamos un *Iterador* para recorrer a lista de empleados.

```
public class ListadoDep {
    public static void main(String[] args) {
        SessionFactory sesion = HibernateUtil.getSessionFactory();
        Session session = sesion.openSession();

        System.out.println("==============================");
        System.out.println("DATOS DEL DEPARTAMENTO 10.");

        Departamentos dep = new Departamentos();
        dep = (Departamentos) session.load(Departamentos.class, (byte) 10);
        System.out.println("Nombre Dep:" + dep.getDnombre());
        System.out.println("Localidad:" + dep.getLoc());

        System.out.println("==============================");
        System.out.println("EMPLEADOS DEL DEPARTAMENTO 10.");

        Set<Empleados> listaemple = dep.getEmpleadoses();// obtenemos empleados

        Iterator<Empleados> it = listaemple.iterator();

        System.out.printf("Número de empleados: %d %n", listaemple.size());
        while (it.hasNext()) {
            // Empleados emple = new Empleados();
            Empleados emple = it.next();
            System.out.printf("%s * %.2f %n", emple.getApellido(), emple.getSalario());
        }
        System.out.println("==============================");
        session.close();
        System.exit(0);
    }
}
```

7.4. ALMACENAMIENTO, MODIFICACIÓN Y BORRADO DE OBJETOS

Hibernate considera a las instancias recién creadas de una clase persistente como transitorias. Se puede asociar una instancia transitoria *persistente* con una sesión:

```
// insertar el departamento 60 en la tabla DEPARTAMENTO
Departamento dep = new Departamento(); // instancia transitoria
dep.setIdDep((byte) 60);
dep.setNombre("Marketing");
dep.setLocalidad("Guadalajara");
session.save(dep); // instancia persistente
```

Si el departamento tiene un identificador auto-generado, el identificador se genera y se asigna al departamento cuando se invoca *save().* Si el departamento tiene un identificador *assigned* o una clave compuesta, el identificador se debe asignar a la instancia de departamento antes de llamar a *save().* También se puede usar *persist()* en lugar de *save():*

- **save().** Garantiza el retorno de un identificador.
- **persist().** Hace que una instancia transitoria sea persistente. Sin embargo, no garantiza que el valor identificador sea asignado a la instancia persistente inmediatamente.

La interfaz *Session* de Hibernate dispone de varios métodos para objetos persistentes:

- **save().** Primero asigna un identificador generado al objeto y después lo almacena en la base de datos.
- **update().** Modifica un objeto persistente de la base de datos a partir de su identificador conocido. Previamente se tiene que haber cargado con *load()* o *get().*
- **delete().** Elimina un objeto persistente de la base de datos a partir de su identificador conocido. Previamente se tiene que haber cargado con *load()* o *get()* y se tiene que asegurar que no existen referencias a otros objetos.

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 ***load()*** o ***get()***
`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 ***load()*** o ***get()***

Vamos a **guardar** un objeto *Departamentos* en la base de datos usando el método *save().* Hibernate se encarga de SQL y ejecuta un INSERT en la base de datos.

```
Departamentos dep = new Departamentos();
dep.setDeptNo((byte) 57);
dep.setDnombre("MARKETING");
dep.setLoc("GUADALAJARA");
```

En el caso del borrado de un objeto, primero debe ser cargado con el método *load()* o el método *get()* y a continuación podemos borrarlo con el método delete(), hacemos asegurarnos de que no existan referencias de otros objetos al que se va a borrar, de lo contrario se producirá una excepción. Veamos cómo borrar el empleado cuyo numero de empleado (*empNo*) es 7369.

```
Empleados em = new Empleados();
em = (Empleados) session.load(Empleados.class, (short) 7369);
sesion.delete(em); //elimina el objeto
```

Creamos ahora un ejemplo para la **eliminación** de un departamento controlando las distintas excepciones, que el departamentos no exista y que tenga empleados. La aplicación debe mostrar un mensaje: NO SE PUEDE ELIMINAR, TIENE EMPLEADOS:

```
public class BorradoDepGet {

    public static void main(String[] args) {
        SessionFactory sesion = HibernateUtil.getSessionFactory();
        Session session = sesion.openSession();
        Transaction tx = session.beginTransaction();

        Departamentos de = (Departamentos)
                session.get(Departamentos.class, (byte) 23);

        if (de==null) {
            System.out.println("El departamento no existe");
        }
        else
        {
            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...");
            }catch (Exception e) {
                System.out.println("ERROR NO CONTROLADO....");
                e.printStackTrace();
            }
        }
        session.close();
        System.exit(0);
    }
}
```

Para modificar un objeto, igual que par borrarlo, primero hemos de cargarlos, a continuación realizamos las modificaciones con los métodos *setter*, y por último, utilizamos el método *update()* para modificarlo. El siguiente ejemplo modifica el salario y el departamento del empleado 7369, sumamos 1000 al salario y le asignamos el departamento 30:

```
Empleados em = new Empleados();
try {
    em = (Empleados) session.load(Empleados.class, (short) 7369);
    System.out.printf("Modificación empleado: %d%n", em.getEmpNo());
    System.out.printf("Salario antiguo: %.2f%n", em.getSalario());
    System.out.printf("Departamento antiguo: %s%n", em.getDepartamentos().getDnombre());

    float NuevoSalario= em.getSalario()+1000;
    em.setSalario(NuevoSalario);

    Departamentos dep = (Departamentos) session.get(Departamentos.class, (byte) 30);
    if (dep == null) {
        System.out.println("El departamento NO existe");
    } else {
        em.setDepartamentos(dep);
        session.update(em); // modifica el objeto
        tx.commit();System.out.printf("Salario nuevo: %.2f%n",em.getSalario());
        System.out.printf("Departamento nuevo: %d%n",
                em.getDepartamentos().getDnombre());
    }
} catch (ObjectNotFoundException o) {
    System.out.println("NO EXISTE EL EMPLEADO...");
} catch (ConstraintViolationException c) {
    System.out.println("NO SE PUEDE ASIGNAR UN DEPARTAMENTO QUE NO EXISTE.....");
} catch (Exception e) {
    System.out.println("ERROR NO CONTROLADO....");
    e.printStackTrace();
}
```

8. CONSULTAS DE DATOS EN HIBERNATE

Páginas web de referencia:
http://docs.jboss.org/hibernate/orm/5.0/userguide/html_single/Hibernate_User_Guide.html#hql

http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#hql

Hibernate soporta un lenguaje de consulta orientado a objetos denominado HQL (Hibernate Query Language) fácil de usar y potente. Este lenguaje es una extensión orientada a objetos de SQL. Las consultas de HQL y SQL nativas son representadas como una instancia de la interfaz **Query**, que ofrece métodos para ligar parámetros, manejo del conjunto de resultados y ejecución de consultas. Siempre obtiene una *Query* utilizando el objeto *Session* actual. Algunos métodos de esta interfaz son:

Consulta la API de Hibernate: https://docs.jboss.org/hibernate/orm/current/javadocs

8.1. CONSULTAS SOBRE LAS CLASES MAPEADAS

Para realizar una consulta se utiliza el método *createQuery()* de la interface de un objeto **Session** instanciado previamente, se le pasará en un *String* la consulta HQL:

```
Query createQuery(String queryString);
```

Por ejemplo para una consulta sobre la tabla *departamentos,* mapeada con la clase Departamentos:

```
org.hibernate.query.Query q = session.createQuery("from Departamento");
```

Para recuperar los datos de la consulta se usa el método *list():*

```
List<Departamento> depList = query.list();
```

Y el método *iterate():*

```
Iterator<Empleados> iter = lista.iterator();
```

Los principales métodos de la interfaz *Query* son:

- **list().** Devuelve todos los resultados de la consulta en una colección, en la colección se encuentran instanciadas todas las entidades que corresponden con el resultado de la ejecución de la consulta. Realiza una única comunicación con la base de datos para traer todos los resultados y requiere que haya memoria suficiente para almacenar todos los objetos resultantes de la consulta. Si la cantidad de resultados es extensa, el retraso del acceso a la base de datos puede ser notorio.
- **iterate().** Devuelve un iterador Java para recuperar los resultados de una consulta. Hibernate ejecuta la consulta y obtiene solo los identificadores de las entidades, y en cada llamada, el método *Iterator.next()* ejecuta la consulta propia para obtener la entidad completa. Esto implica una mayor cantidad de accesos a la base de datos, y por tanto, mayor tiempo de procesamiento total. La ventaja es que no requiere que todas las entidades estén cargadas en memoria simultáneamente.

El siguiente programa Java realiza una consulta, utilizando el método *list() e iterator()* de la interfaz *Query*, de todas las filas de la tabla DEPARTAMENTO de una base de datos MySQL:

```
public class ListaDepartamentos {

    public static void main(String[] args) {
        SessionFactory sesion = HibernateUtil.getSessionFactory();
        Session session = sesion.openSession();

        org.hibernate.query.Query 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 = (Departamentos) iter.next();
           System.out.printf("%d, %s%n", depar.getDeptNo(), depar.getDnombre());
        }
        session.close();
        System.exit(0);
    }
}
```

Como resultado de la ejecución de este programa, se produce la siguiente salida en la pantalla:

```
Número de registros: 7
10, contabilidad
15, INFORMATICA
20, investigación
30, ventas
40, producción
55, FOL
60, CALIDAD
```

El siguiente ejemplo visualiza el apellido y salario de los empleados del departamento 20:

```
org.hibernate.query.Query q  = session.createQuery
            ("from Empleados as e where e.departamentos.deptNo = 20");
List<Empleados> lista = q.list();

// Obtenemos un Iterador y recorremos la lista
Iterator<Empleados> iter = lista.iterator();
while (iter.hasNext()) {
    Empleados emp = (Empleados) iter.next();// extraer el objeto
    System.out.printf("%s, %.2f %n", emp.getApellido(), emp.getSalario());
}
System.out.println("-----OTRO EJEMPLO---------");
List<Empleados> emples = session.createQuery
                    ("from Empleados as e where e.departamentos.deptNo = 20",
                        Empleados.class ).getResultList();
// Obtenemos un Iterador y recorremos la lista
Iterator<Empleados> itera = emples.iterator();
while (itera.hasNext()) {
    Empleados emp2 = (Empleados) itera.next();// extraer el objeto
    System.out.printf("%s, %.2f %n", emp2.getApellido(), emp2.getSalario());
}
```

Otra interface interface TypedQuery<X> que extiende de Query se utiliza para preparar una consulta para la ejecución y especificando el tipo de resultado de la consulta. El método getResultList de TypedQuery<T> se usa para ejecutar una consulta. Esta consulta devuelve una colección de entidades, el resultado se almacena en una lista.

```
TypedQuery <Empleados>q2 = session.createQuery
                ("from Empleados as e where e.departamentos.deptNo = 20");

List<Empleados> lista2 = q.getResultList();
// Obtenemos un Iterador y recorremos la lista
    Iterator<Empleados> iter2 = lista2.iterator();
    while (iter.hasNext()) {
        Empleados emp2 = (Empleados) iter2.next();// extraer el objeto
        System.out.printf("%s, %.2f %n", emp2.getApellido(), emp2.getSalario());
    }
```

https://javaee.github.io/javaee-spec/javadocs/javax/persistence/TypedQuery.html

8.2. PARÁMETROS EN LAS CONSULTAS

Hibernate soporta parámetros con nombre y anteriormente también parámetros de estilo JDBC(?) en las consultas HQL.

Los parámetros con nombre, que son identificadores de la forma **:nombre** en la cadena de consulta. Hibernate numera los parámetros desde cero (el primero que aparece estará en la posición 0, el siguiente en la posición 1, y así sucesivamente).

Los parámetros con nombre tienen las siguientes ventajas:

- Son insensibles al orden en que aparecen en la cadena de consulta.
- Pueden aparecer múltiples veces en la misma consulta.
- Son auto-documentados.

Para asignar valores a los parámetros se utilizan los métodos ***setXX*** vistos en la tabla anterior. La sintaxis más simple de utilizar estos parámetros es usando el método ***setParameter()***, por ejemplo, la siguiente consulta utiliza el parámetro nombrado *:numemple* y muestra el apellido y oficio del empleado con numero 7369:

```
String hql = "from Empleados where empNo = :numemple";
org.hibernate.query.Query q = session.createQuery(hql);
q.setParameter("numemple", (short) 7369);
List<Empleados> emples = q.list();
for (Empleados emple: emples) {
    System.out.printf("%s, %s %n", emple.getApellido(), emple.getOficio());
}
```

https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Query.html

El siguiente ejemplo Java realiza una consulta de los empleados que tienen un oficio determinado “DIRECTOR” y su número de departamento sea 10 de la tabla EMPLEADO de una base de datos MySQL:

```
// Empleados cuyo número de departamento sea 10 y el oficio DIRECTOR
Query q2 = session
        .createQuery("from Empleados emp where emp.departamentos.deptNo = :ndep
                                                + "and emp.oficio = :ofi");
q2.setParameter("ndep", (byte) 10);
q2.setParameter("ofi", "DIRECTOR");
List<Empleados> lista = q2.list();
Iterator<Empleados> iter = lista.iterator();
while (iter.hasNext()) {
    // extraer el objeto
    Empleados emp = (Empleados) iter.next();
    System.out.printf("%d, %s%n", emp.getEmpNo(), emp.getApellido());
}
```

El ejemplo de utilización con parámetros nombrados es el siguiente:

```
// empleados cuyo número de departamento sea 10 y el oficio director
org.hibernate.query.Query query = session.createQuery("from Empleado emp
      where emp.idDep = :ndep and emp.oficio = :ofi");
query.setParameter("ndep", (byte) 10);
query.setParameter("ofi", "DIRECTOR");
```

Un ejemplo de utilización con parámetros por posición de estilo JDBC (el uso de estos parámetros está obsoleto por lo que se debe utilizar los parámetros nombrados) quedaría de la siguiente forma:

```
// empleados cuyo número de departamento sea 10 y el oficio director
org.hibernate.query.Query query = session.createQuery("from Empleado emp
      where emp.idDep = ? and emp.oficio = ?");
query.setParameter(0, (byte) 10);
query.setParameter(1, "DIRECTOR");
```

Otro ejemplo sería asignar valor a un parámetro nombrado de tipo Date. Vamos a obtener los empleados cuya fecha de alta es 1991-12-03:

```
// Empleados cuya fecha de alta es 1991-12-03
System.out.println("-----------FECHA---------");
SimpleDateFormat formatoDelTexto = new SimpleDateFormat("yyyy-MM-dd");
String strFecha = "1991-12-03";
Date fecha = null;
try {
    fecha = formatoDelTexto.parse(strFecha);
} catch (ParseException ex) {
    ex.printStackTrace();
}
String hql6 = "from Empleados where fechaAlt = :fechalta ";

org.hibernate.query.Query q6 = session.createQuery(hql6);
q6.setParameter("fechalta", fecha);
System.out.println("Empleados cuya fecha de alta es 1991-12-03");
List<Empleados> lista5 = q6.list();
Iterator<Empleados> iter5 = lista5.iterator();
while (iter5.hasNext()) {
    // extraer el objeto
    Empleados emp = (Empleados) iter5.next();
    System.out.printf("%d, %d, %s%n", emp.getDepartamentos().getDeptNo(),
                                        emp.getEmpNo(), emp.getApellido());
}
```

Un ejemplo de utilización de una lista de parámetros por nombre es el siguiente donde se asigna al parámetro nombrado *:listadep* una colección de valores llamada números con los valores 10 y 20 para obtener aquellos empleados cuyo número de departamento sea 10 o 20; se usa el método ***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 ";
    org.hibernate.query.Query q5 = session.createQuery(hql5);
    q5.setParameterList("listadep", numeros);

    List<Empleados> lista4 = q5.list();
    Iterator<Empleados> iter4 = lista4.iterator();
    while (iter4.hasNext()) {
        // extraer el objeto
        Empleados emp = (Empleados) iter4.next();
        System.out.printf("%d, %d, %s%n", emp.getDepartamentos().getDeptNo(),
                                          emp.getEmpNo(), emp.getApellido());
    }
```

8.3. CONSULTAS SOBRE CLASES NO ASOCIADAS

Si queremos recuperar datos de una consulta en la que intervienen varias tablas y no tenemos asociada a ninguna clase los atributos que devuelve esa consulta podemos utilizar la clase **Object.** Los resultados se reciben en un array de objetos, donde el primer elemento del array se corresponde con la primera clase que ponemos a la derecha del FROM, el siguiente elemento con la siguiente clase y así sucesivamente. El siguiente ejemplo realiza una consulta para obtener los datos de los empleados y de sus departamentos. El resultado de la consulta se recibe en un array de objetos, donde el primer elemento del array pertenece a la clase *Empleados* y el segundo a la clase *Departamentos:*

```
String hql = "from Empleados e, Departamentos d where e.departamentos.deptNo = d.deptNo "
                + "order by apellido";
org.hibernate.query.Query cons = session.createQuery(hql);
List<Object> lista = cons.list();
Iterator<Object> iter = lista.iterator();
while (iter.hasNext()) {
    Object[] par = (Object[]) iter.next();
    Empleados em = (Empleados) par[0];
    Departamentos de = (Departamentos) par[1];
    System.out.printf("%s, %.2f, %s, %s %n", em.getApellido(), em.getSalario(),
                                              de.getDnombre(),de.getLoc());
}
```

*Para estos casos se podría crear una vista a partir de las tablas y realizar el mapeo de la vista con Hibernate.

8.4. FUNCIONES DE GRUPO EN LAS CONSULTAS

Los resultados devueltos por una consulta HQL o SQL en la que se ha utilizado una función de grupo como por ejemplo *avg(), sum(), count(),* etc. Se pueden recoger como un único valor utilizando el método ***uniqueResult()***. El siguiente ejemplo muestra el salario medio de los empleados:

```
// MOSTRAR SALARIO MEDIO DE LOS EMPLEADOS
        String hql = "select avg(em.salario) from Empleados as em";
        Query cons = session.createQuery(hql);
        Double suma = (Double) cons.uniqueResult();
```

Si en la consulta intervienen varias funciones de grupo y además devuelve varias filas, podemos utilizar objetos devueltos por las consultas. Por ejemplo, a continuación se muestra el salario medio y el número de empleados por cada departamento.

```
// mostrar el salario medio y el numero de empleados por departamento
hql = "select e.departamentos.deptNo, avg(salario), " + " count(empNo) from Empleados e "
          + " group by e.departamentos.deptNo ";

    cons = session.createQuery(hql);
    List<Object> lista = cons.list();
    Iterator<Object> iter = lista.iterator();
    while (iter.hasNext()) {
        Object[] par = (Object[]) iter.next();
        Byte depar = (Byte) par[0];
        Double media = (Double) par[1];
        Long cuenta = (Long) par[2];
        System.out.printf("Dep: %d, Media: %.2f, Nº emp: %d %n", depar, media, cuenta);
    }
```

8.5. OBJETOS DEVUELTOS POR LAS CONSULTAS

Anteriormente se ha comentado como se puden tratar los resultados obtenidos por una SELECT que no está asociada a ninguna entidad. Supongamos que a partir de las tablas *empleados* y *departamentos* quiero obtener una consulta en la que aparezcan el nombre del departamento, su número, el numero de empleados y el salario medio.

Para ello, por ejemplo creo la clase *Totales* en el paquete primero con cuatro atributos: número, departamentos, número de empleados, la media de salario y el nombre del departamento) y los constructores, *getter y setter* asociados. La clase *Totales.java* sería la siguiente:

```
package primero;
public class Totales {
private Long cuenta; //numero empleados
private Byte numero; //numero departamento
private Double media;//media salario
private String nombre;//nombre dep

public Totales( Byte numero,  Long cuenta,
                Double media,  String nombre ) {
    this.cuenta = cuenta;
    this.media = media;
    this.nombre = nombre;
    this.numero = numero;
}

public Totales(){}

public Long getCuenta() {return this.cuenta;}
public void setCuenta( Long cuenta) {this.cuenta = cuenta;}

public Byte getNumero() {return this.numero;}
public void setNumero( Byte numero) {this.numero = numero;}

public Double getMedia() {return this.media;}
public void setMedia(final Double media) {this.media = media;}

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

Para hacer uso de la clase anterior construimos la consulta HQL de la siguiente forma:

```
hql= "select new primero.Totales(" +
          " d.deptNo,  count(e.empNo), coalesce(avg(e.salario),0) , "+
          " d.dnombre ) "+
          " from Empleados as e right join  e.departamentos as d "+
          " group by  d.deptNo, d.dnombre ";

    Query cons = session.createQuery(hql);
    Iterator q = cons.iterate();
    while (q.hasNext()) {
       Totales tot =(Totales) q.next();
       System.out.printf(
          "Numero Dep: %d, Nombre: %s, Salario medio: %.2f, Nº emple: %d%n",
          tot.getNumero(), tot.getNombre(),tot.getMedia(),tot.getCuenta());
    }
```

También podemos recuperar los valores de una consulta que no está asociada a ninguna clase mediante un array de objetos, clase Object (un ejemplo similar visto anteriormente). Los resultados se reciben en un array de objetos, donde el primero elemento del array se corresponde con la primera fila, el siguiente con la segunda fila. Dentro de cada fila será necesario acceder a los atributos o columnas mediante otro array de objetos:

```
String hql = "select  d.deptNo,  count(e.empNo), coalesce(avg(e.salario),0) , "
        + " d.dnombre  "
        + " from Empleados as e right join  e.departamentos as d "
        + " group by  d.deptNo, d.dnombre ";

Query cons = session.createQuery(hql);

List<Object[]> filas = cons.list(); // Todas las filas
for (int i = 0; i < filas.size(); i++) {
    Object[] filaActual = filas.get(i); // Acceso a una fila
    System.out.printf("Numero Dep: %d, Nombre: %s, Salario medio: %.2f, Nº emple: %d%n",
    filaActual[0], filaActual[3], filaActual[2], filaActual[1]);
}
```

9. MANIPULACIÓN DE DATOS EN HIBERNATE

Página web de referencia:

http://docs.jboss.org/hibernate/orm/5.1/userguide/html_single/Hibernate_User_Guide.html#batch-bulk-hql

El lenguaje HQL también permite realizar operaciones de manipulación masiva de datos, como INSERT, UPDATE y DELETE.

La sintaxis para las sentencias **UPDATE** y **DELETE** es:

```
UPDATE [FROM] NombreEntidad [WHERE condición]

DELETE [FROM] NombreEntidad [WHERE condición]
```

Las principales consideraciones sobre estas sentencias UPDATE y DELETE son:

- En la cláusula FROM, la palabra clave FROM es opcional.
- En la cláusula FROM, solamente puede haber una entidad mencionada y puede tener un alias. Si la entidad tiene un alias, entonces cualquier referencia a alguna propiedad tendrá que ser calificada usando ese alias. Si la entidad no tiene un alias, entonces es ilegal calificar cualquier referencia de la propiedad.
- La cláusula WHERE también es opcional.
- No se puede especificar ninguna unión o asociación, ya sea implícita o explícita, en una consulta masiva de HQL. No obstante, se pueden utilizar subconsultas en la cláusula WHERE (estas subconsultas pueden contener uniones o asociaciones en sí mismas).

Para ejecutar una sentencia INSERT, UPDATE o DELETE de HQL, se utiliza el método **executeUpdate().** El valor entero, devuelto por este método, indica el número de entidades afectadas por la operación. No se debe olvidar realizar el commit para validar la transacción:

Vemos un ejemplo en el que dentro de la misma transacción modificamos un empleado y eliminamos los empleados del departamento 20:

```
// Modificamos el salario de GIL
String hqlModif = "update Empleados set salario = :nuevoSal where apellido = :ape";
Query q1 = session.createQuery(hqlModif);
q1.setParameter("nuevoSal", (float) 2500.34);
q1.setString("ape", "GIjL");
int filasModif = q1.executeUpdate();
System.out.printf("FILAS MODIFICADAS: %d%n", filasModif);

// Eliminamos los empleados del departamento 20
String hqlDel = "delete Empleados e where e.departamentos.deptNo = :dep";
Query q = session.createQuery(hqlDel);
q.setInteger("dep", 20);
int filasDel = q.executeUpdate();
System.out.printf("FILAS ELIMINADAS: %d%n", filasDel);

//tx.rollback(); // Deshace la transacción

tx.commit(); // valida la transacción
```

La sintaxis para la sentencia **INSERT** es:

```
INSERT INTO NombreEntidad (lista_propiedades) sentencia_select
```

Las principales consideraciones sobre esta sentencia INSERT son:

- Solamente se soporta la forma INSERT INTO … SELECT …, no la forma INSERT INTO … VALUES … del lenguaje SQL. Es decir, sólo se puede hacer con datos procedentes de otra tabla, que tendrá que estar mapeada en el proyecto de trabajo.
- La lista de propiedades es análoga a la lista de columnas de la sentencia INSERT de SQL. Para las entidades involucradas en la herencia mapeada, sólamente las propiedades definidas directamente en ese nivel de clase dado se pueden utlizar en la lista de propiedades. Las propiedades de la superclase no están permitidas y las propiedades de la subclase no tienen sentido.
- *sentencia_select* puede ser cualquier consulta SELECT de HQL válida, con la condición de que los tipos devueltos por la consulta coincidan con los esperados por la sentencia INSERT. Actualmente, esto se verifica durante la compilación de la consulta, en lugar de relegar esta verificación a la base de datos.
- Para la propiedad id, hay dos opciones:
  1) Se puede especificar en la lista de propiedades. En este caso, su valor se toma de la expresión de selección correspondiente.
  2) Se puede omitir de la lista de propiedades. En este caso, se utiliza un valor generado. Esta opción se encuentra disponible cuando se usan generadores de identificadores que operan en la base de datos (por ejemplo, cuando se usa AUTO_INCREMENT PRIMARY KEY en MySQL, la clave primaria se crea de forma automática sin necesidad de dar valor).

Ejemplo: desde SQL creo la siguiente tabla e inserto varias filas:

```
CREATE TABLE nuevos (
       dept_no TINYINT (2) NOT NULL PRIMARY KEY,
       dnombre VARCHAR (15),
       loc VARCHAR (15)
```

```
);

INSERT INTO nuevos VALUES (51, 'PERSONAL', 'MADRID');
INSERT INTO nuevos VALUES (52, 'NOMINAS', 'TOLEDO');
INSERT INTO nuevos VALUES (53, 'OCIO', 'BARCELONA');
```

A continuación añadimos la siguiente línea a fichero **hibernate.reveng.xml:**

```
<table-filter match-catalog="ejemplo" match-name="nuevos"/>
```

A continuación generamos la nueva clase desde **Hibernate Code Generation Configurations.** Se tiene que generar la clase mapeada y el fichero nuevos.hbm.xml. Por último, ejecuto el código Java para insertar los datos de esa tabla en *Departamentos:*

```
// Insertamos los departamentos de la tabla NUEVOS, la tabla tiene
// que estar mapeada en nuestro proyecto

String hqlInsert = "insert into Departamentos (deptNo, dnombre, loc) " +
                   " select n.deptNo, n.dnombre, n.loc from Nuevos n";
org.hibernate.query.Query cons = session.createQuery( hqlInsert );
int filascreadas = cons.executeUpdate();

System.out.printf("FILAS INSERTADAS: %d%n",filascreadas);
```

10. RESUMEN DEL LENGUAJE HQL

Páginas web de referencia:
http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/objectstate.html

http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/queryhql.html

Hibernate soporta un lenguaje de consulta orientado a objetos denominado HQL (Hibernate Query Language) fácil de usar y potente. Este lenguaje es una extensión orientada a objetos de SQL. Las consultas de HQL y SQL nativas son representadas como una instancia de la interfaz **Query**, que ofrece métodos para ligar parámetros, manejo del conjunto de resultados y ejecución de consultas.

Las consultas en HQL no son sensibles a mayúsculas a excepción de los nombres de las clases y propiedades Java. Podemos escribir FROM, from, SELECT, sELet, etc.

La cláusula más simple que existe en Hibernate es from, que obtiene todas las instancias de una clase, por ejemplo, **from Empleados** obtiene todas las instancias de la clase *Empleado* (en SQL, obtiene todas las filas de la tabla *empleados*). La cláusula **order by** ordena los resultados de la consulta.

La cláusula **where** permite refinar la lista de instancias retornadas y **order by** orderna la lista.

Ejemplos:

```
from Empleados where deptNo=10 order by apellido
from Empleados as em where em.deptNo=10 order by  1 desc
from Empleados as em where em.empNo=10
```

Se pueden asignar alias a las clases usando la cláusula **as**: *from Empleados as em*, o sin usar dicha cláusula: *from Empleados em.*

Pueden aparecer múltiples clases a la derecha de FROM, lo que causar un producto cartesiano o una unión “cruzada” (*cross join)*: *from Empleados as em, Departamentos as dep.*

Para obtener determinadas propiedades (columnas) en una consulta utilizamos la cláusula SELECT: *select apellido, salario from Empleados,* obtiene el apellido y el salario de las clase Empleado.

Las consultas pueden retornar múltiples objetos y/o propiedades como un array de tipo **Objet[]**, una lista, una clase, etc.

Las funciones de grupo soportadas son las siguientes, la semántica es similar a SQL:

- *avg(…), sum(…), min(…), max(…)*
- *coun(*)*
- *count(…), count(distinct…), count(all…)*

Se puede utilizar alias para nombrar los atributos y expresiones. Se pueden utilizar operadores aritméticos, de concatenación y funciones SQL reconocidas por la cláusula SELECT. Algún ejemplo:

```
select avg(salario) as med, count(empNo) as c from Empleados
select avg(salario), count(empNo) from Empleados
select avg(salario)+sum(salario), count(empNo) from Empleados
select apellido ||'*'||oficio as campo from Empleados
select count (distinct deptNo) from Empleados
```

Las expresiones utilizadas en la cláusula **where** incluyen lo siguiente:

- Operadores matemáticos: +,-,*,/
- Operadores de comparación binarios: *=, >=, <=, <>, !=, like*
- Operadores lógicos: *and, or, not*
- Paréntesis () que indican agrupación
- *in, not in, beetween, is null, in not null, is empty, in not empty, memer of y not member of…*
- Caso “simple”, *case…, when…then…, else…, end*, y caso “buscado”, *case when…then…else…end.*
- Concatenación de cadenas *..||.. o concat(…,…)*
- *current_date(), current_time() y current timestamp().*
- *second(…), minute(…), hour(…), day(…), month(…)* y *year(…).*
- Cualquier función u operador definido por EJB-QL 3.0: *substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length(), mod().*
- *coalesce() y nullif()*
- *str()* para convertir valores numéricos o temporales en una cadena legible.
- *cast(…as…),* donde el segundo argumento es el nombre de un tipo de Hibernate, y *extract(…from…)* si *cast() y extract()* es soportado por la base de datos subyacente.
- La función *index()* de HQL, que se aplica a alias de una colección indexada unida.
- Las funciones de HQL que tomen expresiones de ruta valuadas en colecciones: *size(), minelement(), maxelement(), minindex(), maxindex(),* junto con las funciones esenciales *elements()* e índices, las cuales se pueden cuantificar utilizando *some, all, exists, any, in.*
- Cualquier función escalar de SQL soportada por la base de datos como *sgn(), trunc(), rtrim() y sin().*
- Parámetros posicionales JDBC ?.
- Parámetros con nombre *:name, :start_date y :x1*

- Literales SQL *'foo', 69, 6,66E+2, '1970-01-01 10:00:01.0'*
- Constantes Java

```
from Empleados where departamentos.deptNo in (1,2)
from Empleados where departamentos.deptNo not in (1,2)
from Empleados where salario not between 2000 and 3000
from Empleados where comision is not null
from Empleados where comision is null
select lower(apellido),coalesce(comision,0) from  Empleados
select apellido from Empleados where apellido like 'L%'
```

Se pueden agrupar las consultas usando **group by** y **having.** Las funciones SQL y las funciones de agregación SQL están permitidas en las clausulas **having** y **order by**, si están soportadas por la base de datos subyacente. Ni la cláusula **group by** ni la cláusula **order by** pueden contener expresiones aritméticas.

```
 select de.dnombre, avg(em.salario)
from Empleados em, Departamentos
where em.departamentos.deptNo = de.deptNo
group by de.dnombre
having avg(em.salario) > 200
```

Para bases de datos que soportan subconsultas, Hibernate soporta subconsultas den tro de las consultas. Uns subconsulta se debe cerrar entre paréntesis. Incluso se permiten subconsultas correlacionadas (subconsultas que se refieren a un alias en la consulta exterior)

```
from Empleados as where em.salario >
          (select avg(em2.salario) from Emepleados em2
             where em2.deptNo=em.deptNo)

from Empleados as where em.salario >
          (select avg(em.salario) from Emepleados)
```

10.1. ASOCIACIONES Y UNIONES (JOINS)

En los mapeos realizados sobre las tablas las asociaciones de claves ajenas se generan de forma automática. Entre las tablas de una base de datos existen diferentes **tipos de relaciones binarias**, como uno a uno (1:1), uno a muchos (1:N) y muchos a muchos (N:M). El tipo de relación entre dos tablas viene definida por la cardinalidad máxima que hay entre sus tuplas.

Podemos realizar asociaciones de forma manual sobre tablas que no tienen clave ajena definida.

Por ejemplo, la tabla *empleados* tiene la columna *dir* que representa el director del empleado y es un número de empleado. Entonces podemos decir que un empleado que es director puede tener a cargo otros empleados. Tenemos pues, una relación de uno a muchos (*one-to-many*) entre dos clases persistentes *Empleados*, ya que en la clase *Departamentos* se tiene una colección de objetos de la clase *Empleados*, es decir, por cada departamento se puede tener muchos empleados.

Para mapear esta relación, debemos añadir al fichero generado Departamentos.hbm.xml debiendo contener lo siguiente al final, antes de la finalización del elemento de la clase (</class>):

```
<set name="empleacargo" table="empleados" >
        <key>
                <column name="dir" />
        </key>
        <one-to-many class="primero.Empleados" />
 </set>
```

La colección se indica mediante la etiqueta **set**, que da un nombre a la colección (*empleacargo*), y que indica el nombre de la tabla de donde se tomará esa colección de objetos (*empleados*), la columna de la tabla por la que se relacionan (dir), el tipo de relación (*one-to-many*) y la clase con la que se establece la relación (*primero.Empleados*).

Es necesario añadir a la clase *Empleados.java* la colección (*empeacargo)* con sus métodos de acceso:

```
private Set <Empleados> empleacargo = new HashSet()<Empleados> (0);
public Set <Empleados> getEmpleacargo() {
       return empleacargo;
}
public void setEmpleacargo(Set <Empleados> empleacargo) {
       this.empleacargo = empleacargo;
}
```

Una vez realizados los cambios, se pueden ejecutar consultas con **join**. Los tipos de join soportados son *inner join, left outer join (left join), right outer join (right join)* y *full join* (no es útil usualmente).

Aunque la forma de utilizarlos es diferente a la usada en SQL. En estos joins no es necesario especificar en la cláusula *from* las instancias (tablas) que se combinan, solo hay que hacer el jin con el atributo donde se define la asociación.

La consulta

```
from Empleados as emp join emp.empleacargo
```

devuelve tantas instancias de dos objetos Empleados, como resultante de combinar la tabla *empleados* consigo misma mediante las columnas *dir y emp_no.* El primer objeto resultante representa el director del empleado y el segundo el empleado. La orden anterio corresponde con la siguiente orden en SQL:

```
SELECT dire.emp_no, dire.apellido, em.emp_no, em.apellido
FROM empleados dire, empleados em
WHERE dire.emp_no = em.dir
```

En la consulta anterior faltan los empleados que no tienen director, para que se muestren ejecutamos la consulta con *right join*, además la salida se puede ordenar:

```
from Empleados as emp right join emp.empleacargo order by emp.empNo
```

En SQL quedaría así:

```
SELECT dire.emp_no, dire.apellido, em.emp_no, em.apellido
FROM empleados dire
RIGHT JOIN empleados em ON dire.emp_no = em.dir
```

El siguiente ejemplo Java muestra los datos de los empleados (numero de empleado y apellido) y los de su director, la salida se ordena por director:

```
String hql = "from Empleados as emp right join emp.empleacargo order by emp.empNo";
org.hibernate.query.Query cons = session.createQuery(hql);
Iterator q = cons.iterate();

while (q.hasNext()) {
    Object[] par = (Object[]) q.next();
    Empleados dir = (Empleados) par[0];//director
    Empleados em = (Empleados) par[1]; //empleado
    if(dir!=null)
    System.out.printf("Empleado: %d, %s, DIRECTOR: %d, %s %n",
            em.getEmpNo(), em.getApellido(),
            dir.getEmpNo(), dir.getApellido());
    else
        System.out.printf("Empleado %d, %s, SIN DIRECTOR.%n",
                em.getEmpNo(), em.getApellido());

}
```

El siguiente ejemplo se muestran los empleados, si el empleado es director muestra los que tiene a su cargo:

```
String hql = "from Empleados ";
Query cons = session.createQuery(hql);

List<Empleados> lis = cons.list();
Iterator<Empleados> ite = lis.iterator();

System.out.println("=======================================");
while (ite.hasNext()) {
    Empleados emple = (Empleados) ite.next();
    if (emple != null) {
        Set acargo = emple.getEmpleacargo(); //empleados a cargo

        if (acargo.size() == 0) {// no son directores
            System.out.printf("EMPLEADO: %d, %s %n", emple.getEmpNo(), emple.getApellido());
            System.out.println("=======================================");
        } else {
            System.out.printf("DIRECTOR: %d, %s %n",
                    emple.getEmpNo(), emple.getApellido());
            System.out.println("A cargo: " + acargo.size());

            Iterator it = acargo.iterator();
            while (it.hasNext()) { // recorro los empleados a cargo
                Empleados em = (Empleados) it.next();
                System.out.printf("\t %d, %s %n",
                        em.getEmpNo(), em.getApellido());
            }
            System.out.println("=======================================");
        }
    }
```

Otro ejemplo es el caso de la base de datos con las tablas DEPARTAMENTOS y EMPLEADOS, donde ya hay una relación de uno a muchos (*one-to-many*) entre las clases generadas *Departamentos* y *Empleados*, ya que en la clase *Departamentos* se tiene una colección de objetos de la clase *Empleados*, es decir, por cada departamento se puede tener muchos empleados.

Para mapear esta relación, el fichero generado Departamentos.hbm.xml debe contener lo siguiente al final, antes de la finalización del elemento de la clase (</class>):

```
<set name="empleadoses" table="empleados" inverse="true" lazy="true" fetch="select">
        <key>
                <column name="DEPT_NO" not-null="true" />
        </key>
        <one-to-many class="clases.Empleados" />
</set>
```

La colección se indica mediante la etiqueta **set**, que da un nombre a la colección (*empleadoses*), y que indica el nombre de la tabla de donde se tomará esa colección de objetos (*empleados*), la columna de la tabla por la que se relacionan (*dept_no*), el tipo de relación (*one-to-many*) y la clase con la que se establece la relación (*clases.Empleados*).

Es necesario añadir a la clase *Departamentos.java* la colección con sus métodos de acceso:

```
private Set empleadoses = new HashSet();
public Set getEmpleadoses() {
        return empleadoses;
}
public void setEmpleadoses(Set empleadoses) {
        this.empleadoses = empleadoses;
}
```

Una vez realizados estos cambios, se pueden ejecutar consultas con **join**. Los tipos de join soportados son *inner join, left outer join (left join), right outer join (right join)* y *full join* (no es útil usualmente).

La consulta

```
FROM Departamentos LEFT OUTER JOIN Empleados
```

devuelve tantas instancias de dos objetos como resulte de combinar las tablas EMPLEADOS y DEPARTAMENTOS, incluyendo los departamentos que no tengan empleados. Se obtiene por cada fila un objeto *Departamento* y un objeto *Empleado*.

El siguiente programa Java recupera los datos de esta consulta mediante dos objetos, el primero con los datos del departamento y el segundo con los datos del empleado:

```
import java.util.Iterator;
import org.hibernate.*;

public class ConsultaConJoin1 {
    public static void main(String[] args) {
        // establecer la sesión de trabajo
        SessionFactory sessionFactory = SessionFactoryUtil.getSessionFactory();
        Session session = sessionFactory.openSession();
        // imprimir en pantalla la cabecera del listado
        System.out.println("====================");
        System.out.println("CONSULTA CON JOIN 1");
        System.out.println("====================");
        // realizar la consulta left join
        String hql = "from Departamento as dep left join dep.empleadoses order by dep.idDep";
        Query query = session.createQuery(hql);
        // construir un iterador para recorrer los resultados de la consulta
        Iterator iterator = query.iterate();
        // recorrer los resultados de la consulta con el iterador
        Departamento dep = new Departamento();
        Empleado emp = new Empleado();
        while (iterator.hasNext()) {
            // extraer la siguiente pareja (departamento, empleado) del iterador
            Object[] objectArray = (Object[]) iterator.next();
            dep = (Departamento) objectArray[0];
            emp = (Empleado) objectArray[1];
            // departamento con empleados
            if(dep != null && emp != null) {
                // mostrar la información del departamento y del empleado
                System.out.println("Id del Departamento:     " + dep.getIdDep());
                System.out.println("Nombre del Departamento: " + dep.getNombre());
                System.out.println("Apellido del Empleado:   " + emp.getApellido());
                System.out.println("Salario del Empleado:    " + emp.getSalario());
                System.out.println("-------------------------------------");
            }
            // departamento sin empleados
            if(dep != null && emp == null) {
                System.out.println("El departamento " + dep.getIdDep() + " no tiene empleados.");
                System.out.println("-------------------------------------");
            }
        }
        // cerrar la sesión de trabajo
        session.close();
    }
}
```

La consulta

```
FROM Departamentos LEFT OUTER JOIN FETCH Empleados
```

devuelve tantas instancias de dos objetos como resulte de combinar las tablas EMPLEADOS y DEPARTAMENTOS, incluyendo los departamentos que no tengan empleados, pero solo del objeto *Departamento* con los datos de sus empleados cargados en la colección.

El siguiente programa Java recupera los datos de esta consulta mediante *fetch*, lo que indica que, para cada departamento, se carga una colección con los datos de los empleados de dicho departamento:

```
import java.util.Iterator;
import java.util.List;
import java.util.Set;
import org.hibernate.*;

public class ConsultaConJoin2 {
    @SuppressWarnings({"rawtypes", "unchecked"})
    public static void main(String[] args) {
        // establecer la sesión de trabajo
        SessionFactory sessionFactory = SessionFactoryUtil.getSessionFactory();
        Session session = sessionFactory.openSession();
        // imprimir en pantalla la cabecera del listado
        System.out.println("====================");
        System.out.println("CONSULTA CON JOIN 2");
        System.out.println("====================");
        // realizar la consulta left join fetch
        String hql = "from Departamento as dep left join fetch dep.empleadoses order by dep.idDep";
        Query query = session.createQuery(hql);
        // guardar los resultados de la consulta en una lista de departamentos
        List<Departamento> depList = query.list();
        // construir un iterador para recorrer la lista de departamentos
        Iterator<Departamento> depIterator = depList.iterator();
        // recorrer la lista de departamentos con el iterador
        Departamento dep = new Departamento();
        Empleado emp = new Empleado();
        while (depIterator.hasNext()) {
            // extraer el siguiente departamento del iterador
            dep = (Departamento) depIterator.next();
            // mostrar la información del departamento extraido
            System.out.println("Id del Departamento:     " + dep.getIdDep());
            System.out.println("Nombre del Departamento: " + dep.getNombre());
            // obtener el conjunto de empleados del departamento
            Set empSet = dep.getEmpleadoses();
            // departamento sin empleados
            if (empSet.size() == 0) {
                System.out.println("Número de Empleados:     0");
            }
            // departamento con empleados
            if (empSet.size() > 0) {
                System.out.println("Número de Empleados:     " + empSet.size());
                System.out.println("-----------------------------------");
                // construir un iterador para recorrer el conjunto de empleados
                Iterator empIterator = empSet.iterator();
                // recorrer el conjunto de empleados con el iterador
                while (empIterator.hasNext()) {
                    // extraer el siguiente empleado del iterador
                    emp = (Empleado) empIterator.next();
                    // mostrar la información del empleado extraido
                    System.out.println("Apellido del Empleado: " + emp.getApellido());
                    System.out.println("Salario del Empleado:  " + emp.getSalario());
                    System.out.println("-----------------------------------");
                }
            }
            System.out.println("===================================");
        }
        // cerrar la sesión de trabajo
        session.close();
    }
}
```