

TEMA 4 –ORM,HIBERNATE

1. CONTENIDOS

- Herramientas ORM. Características
- Hibernate. Instalando, configurando y mapeando una base de datos.
- Codificando con Hibernate. Cargando, creando, actualizando y borrando objetos persistentes.
- Ejecutando Queries

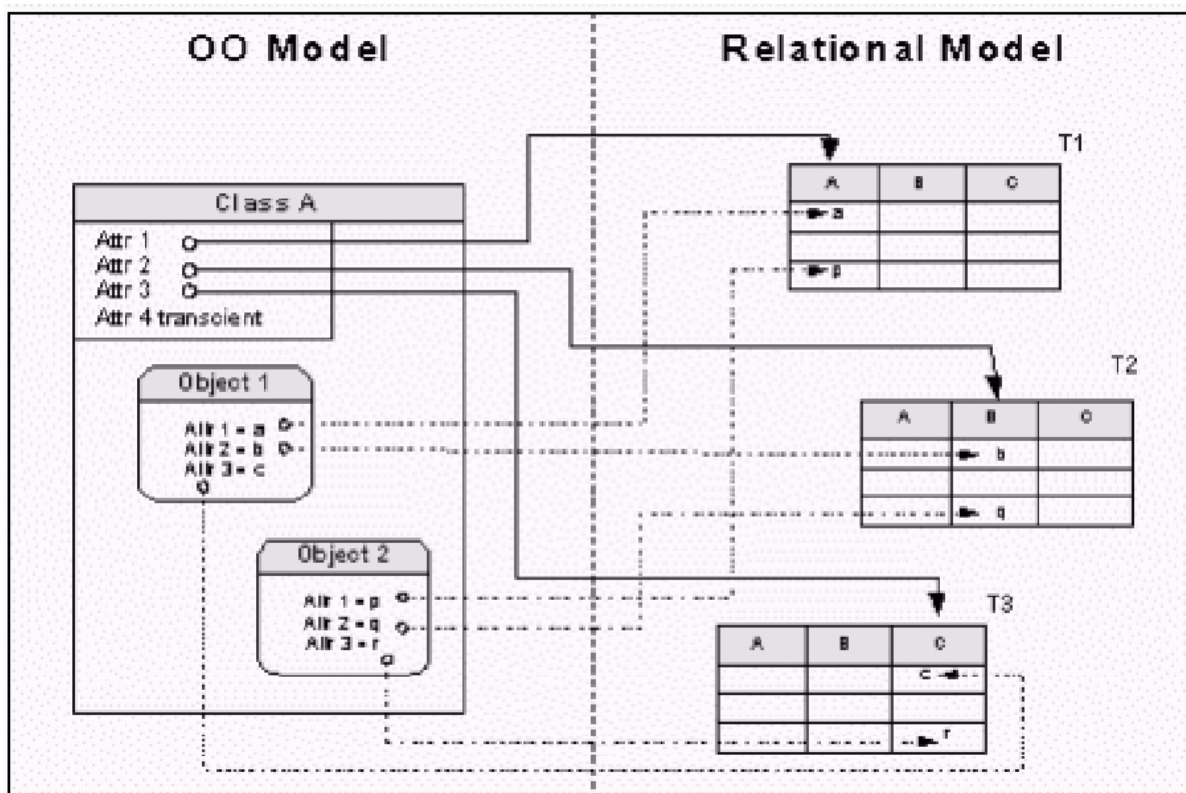
2. INTRODUCCIÓN, BASES DE DATOS RELACIONALES VS PROGRAMACIÓN ORIENTADA A OBJETOS

Los datos almacenados en bases de datos relacionales se almacenan en tablas, que se relacionan mediante lógica relacional, cuando recibimos los datos mediante una query, estos vienen en forma de tabla y las relaciones entre las tablas se modelan mediante las claves foráneas.

Mientras que en los programas que utilizan programación orientada a objetos, los datos los almacenamos en clases y objetos, que tienen propiedades y métodos, tenemos el concepto de herencia y unos objetos pueden incluir otros objetos como propiedades o incluso una colección de objetos

Un **ORM**, o **Object-Relational Mapping**, es una herramienta que nos traduce los datos de lógica relacional a lógica de objetos y viceversa:

Lógica Relacional	Lógica de Objetos
Tablas	Clases
Columnas	Propiedades
Filas	Objetos
Procedimientos	Métodos



En cuanto a las características clave, de los ORM, tenemos:

- Integración del acceso a datos relacionales con programación orientada a objetos
- Habilita el uso de características de la programación orientada a objetos como la herencia o el polimorfismo.
- El acceso a datos es transparente.
- Nos permite centrarnos en la lógica del programa.
- Reduce el tiempo de desarrollo y el número de errores.
- Hace que las aplicaciones sean independientes del sistema de datos.

Por el contrario, el mapeado necesita configuración y el rendimiento de la aplicación se puede ver reducido por la traducción de los datos.

3. HERRAMIENTAS ORM

En el mercado existen diferentes herramientas ORM dependiendo del lenguaje de programación, algunas de estas son:

- *Python*

- STORM. - <https://storm.canonical.com/>
- SQLOBJECT. - <http://www.sqlobject.org/>

- *Java*

- HIBERNATE <http://www.hibernate.org/>
- NHIBERNATE: HIBERNATE para .NET

- *PHP*

- Doctrine <http://www.doctrine-project.org/>
- Propel <http://propelorm.org/>

- *.NET*

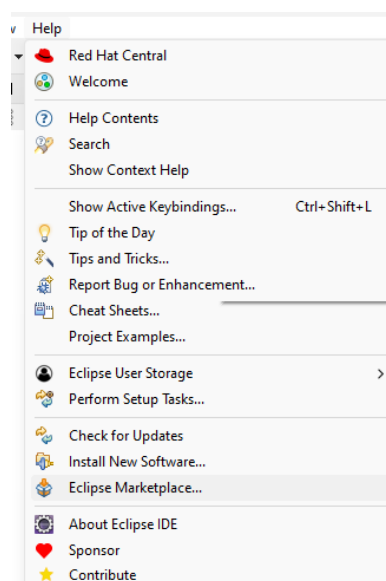
- NHIBERNATE
- LINQ [http://msdn.microsoft.com/es-es/library/bb397897\(v=vs.100\)](http://msdn.microsoft.com/es-es/library/bb397897(v=vs.100))

4. HIBERNATE

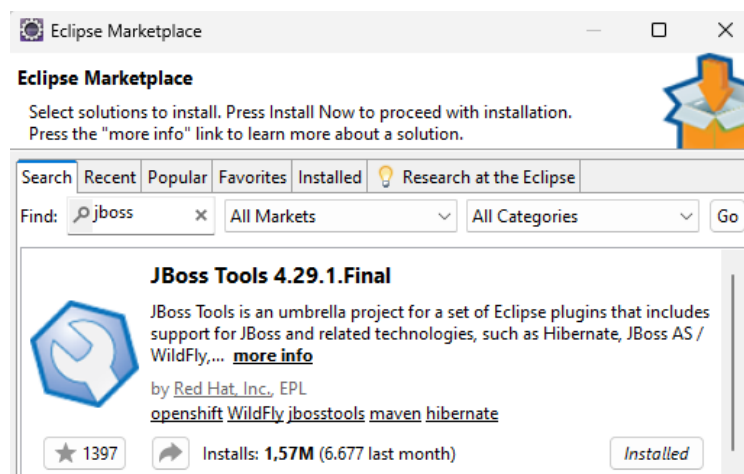
Hibernate es una Herramienta ORM para Java también disponible para .NET, mapea los datos entre tablas y objetos utilizando ficheros XML y anotaciones en las clases y los datos de la base de datos se mapean en POJOs (Plain Old Java Objects), estas clases son independientes de los frameworks.

CONFIGURACIÓN DE HIBERNATE

Para configurar Hibernate, primero accedemos al Marketplace de Eclipse e instalamos JBoss Tools, en el menú contextual de Help>Eclipse Marketplace:

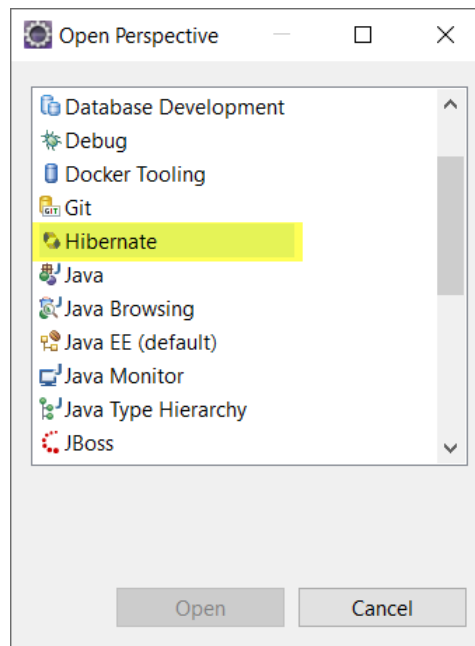


Buscamos el plugin JBoss y lo instalamos:



Una vez tenemos instalado el plugin debemos reiniciar eclipse.

Una vez instalado, podremos abrir la perspectiva de eclipse en el menú contextual de Ventana:
Ventana>Perspectivas>Abrir Perspectiva>Otros:



EJEMPLO 1 – ESTABLECER CONEXIÓN CON LA DB

Para la realización del ejemplo primero debemos crear la Base de datos con la cual vamos a conectar, para ello instalamos MySQL en caso de que no lo tengamos instalado y ejecutamos el siguiente script:

```
CREATE DATABASE biblio;

USE biblio;

CREATE TABLE authors (
cod VARCHAR(5) PRIMARY KEY,
name VARCHAR(60));

CREATE TABLE books (
id INTEGER PRIMARY KEY,
title VARCHAR(60),
codauthor VARCHAR(5),
FOREIGN KEY (codauthor) REFERENCES authors(cod));

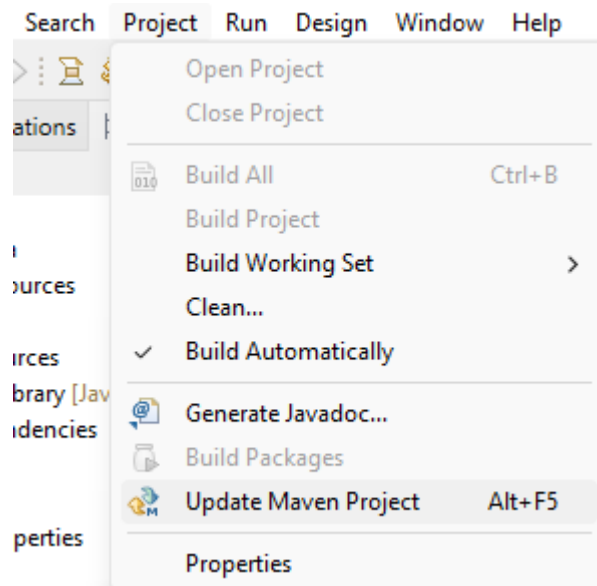
INSERT INTO authors VALUES ('WSHAK', 'William Shakespeare'),
('GMART', 'George R. R. Martin');

INSERT INTO books VALUES (1, 'Macbeth', 'WSHAK'),
(2, 'A Game of Thrones', 'GMART');10. Rise salary for those players of a
team who maxed stat in season 07/08
```

Una vez creada la DB, creamos un nuevo proyecto Maven y añadimos las dependencias necesarias al proyecto que en este caso serán las siguientes:

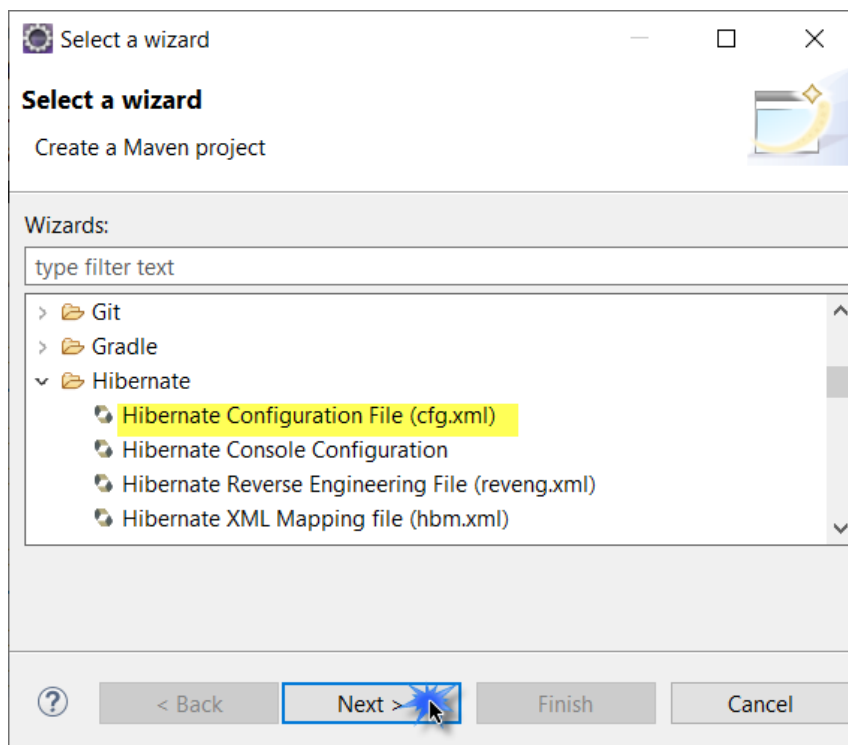
```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.28</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.4.10.Final</version>
        <type>pom</type>
    </dependency>
</dependencies>
```

Cuando tenemos las dependencias importadas, actualizamos el proyecto Maven:

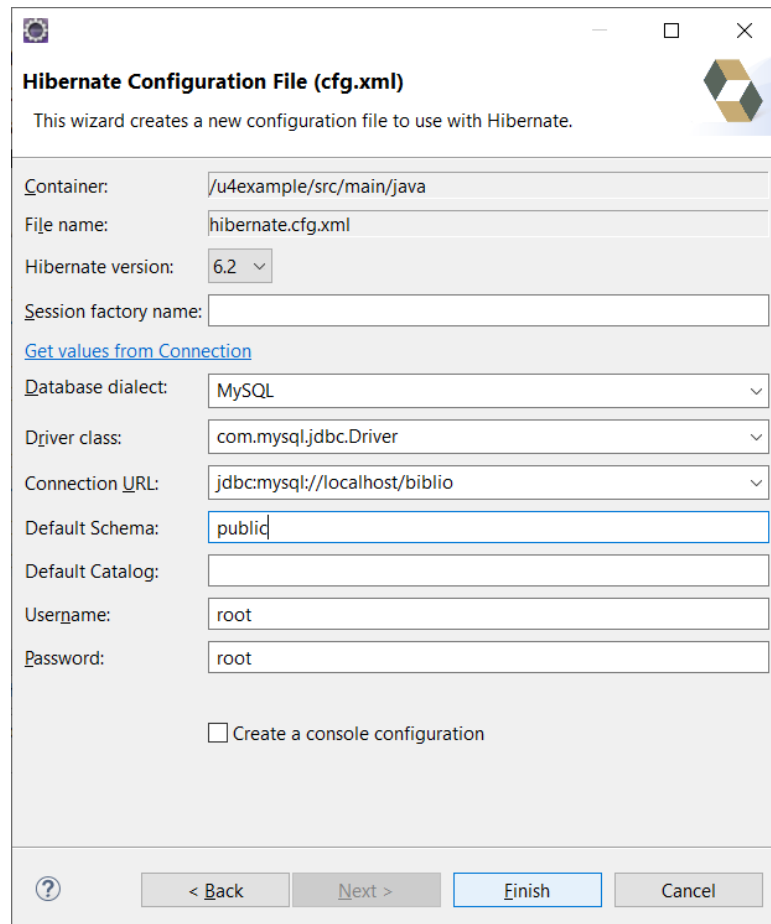


Y vamos a la carpeta **src/main/java** y con click derecho añadimos un archivo de configuración de hibernate:

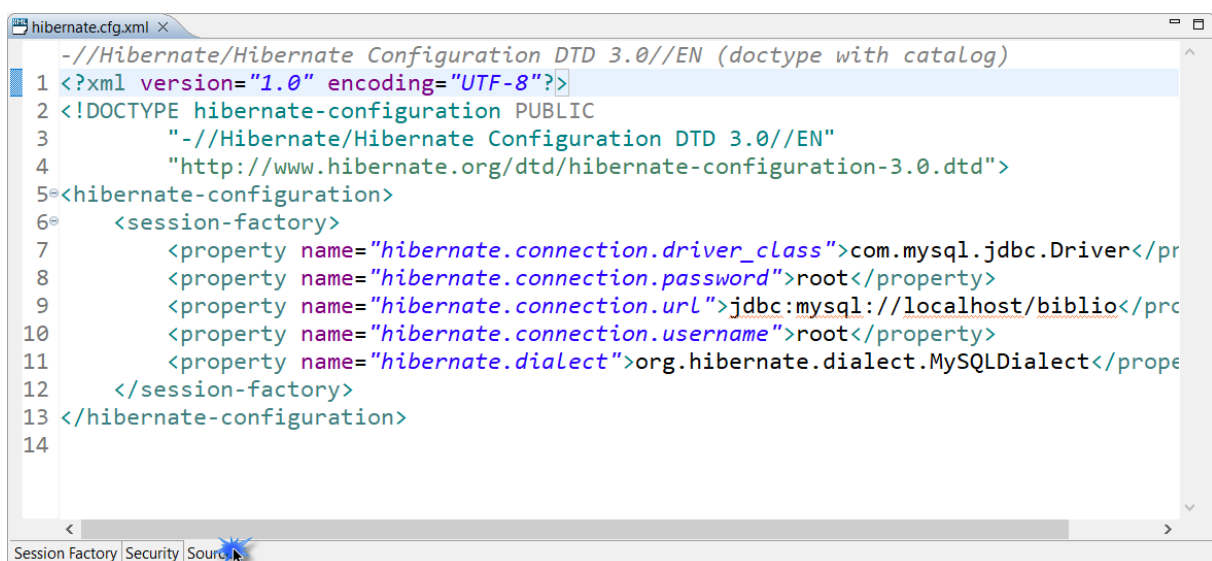
Nuevo > Otros > Hibernate > HibernateConfigurationFile



Rellenamos el formulario de configuración con los datos pertinentes y hacemos click en finish.

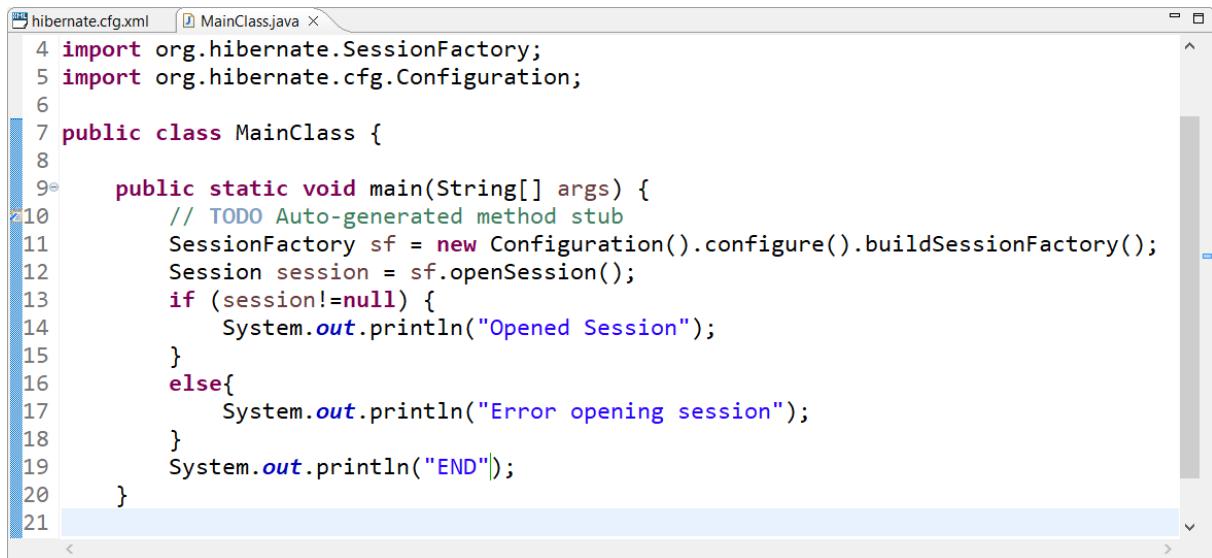


Al abrir el fichero, en la parte inferior tendremos 3 pestañas, si seleccionamos source, nos mostrará el contenido del fichero XML con la configuración creada:



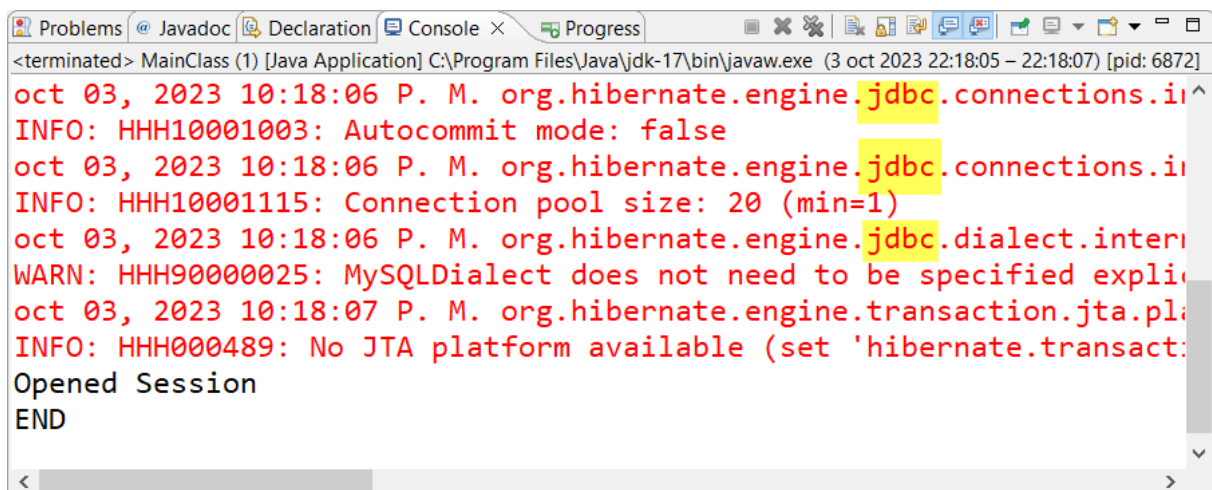
```
-//Hibernate/Hibernate Configuration DTD 3.0//EN (doctype with catalog)
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-configuration PUBLIC
3     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4     "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
5 <hibernate-configuration>
6     <session-factory>
7         <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
8         <property name="hibernate.connection.password">root</property>
9         <property name="hibernate.connection.url">jdbc:mysql://localhost/biblio</property>
10        <property name="hibernate.connection.username">root</property>
11        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
12    </session-factory>
13 </hibernate-configuration>
14
```


Una vez tenemos el fichero de configuración, creamos la clase Main con el siguiente código para establecer una conexión:



```
hibernate.cfg.xml MainClass.java x
4 import org.hibernate.SessionFactory;
5 import org.hibernate.cfg.Configuration;
6
7 public class MainClass {
8
9     public static void main(String[] args) {
10         // TODO Auto-generated method stub
11         SessionFactory sf = new Configuration().configure().buildSessionFactory();
12         Session session = sf.openSession();
13         if (session!=null) {
14             System.out.println("Opened Session");
15         }
16         else{
17             System.out.println("Error opening session");
18         }
19         System.out.println("END");
20     }
21 }
```

Si ejecutamos el código deberíamos tener la siguiente salida:



```
Problems @ Javadoc Declaration Console x Progress
<terminated> MainClass (1) [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (3 oct 2023 22:18:05 - 22:18:07) [pid: 6872]
oct 03, 2023 10:18:06 P. M. org.hibernate.engine.jdbc.connections.in
INFO: HHH10001003: Autocommit mode: false
oct 03, 2023 10:18:06 P. M. org.hibernate.engine.jdbc.connections.in
INFO: HHH10001115: Connection pool size: 20 (min=1)
oct 03, 2023 10:18:06 P. M. org.hibernate.engine.jdbc.dialect.intern
WARN: HHH90000025: MySQLDialect does not need to be specified explici
oct 03, 2023 10:18:07 P. M. org.hibernate.engine.transaction.jta.pla
INFO: HHH000489: No JTA platform available (set 'hibernate.transact:
Opened Session
END
```

Como podemos observar en la captura anterior, Hibernate utiliza JDBC para establecer la conexión con la base de datos.

CLASES PERSISTENTES

Cada clase persistente representa una tabla, donde cada columna representa una propiedad de la clase y cada objeto es una fila de la tabla.

Hibernate gestiona el emparejamiento entre tablas y objetos, de forma que guardará datos de objetos a tablas y recuperará datos de las tablas y creando o actualizando objetos.

Las clases deben tener constructores, getters, setters y propiedades privadas.

Al crear las clases se seguirá el estándar JavaBean (también llamado POJO), y será de la siguiente forma:

- Utilizaremos lowerCamelCase
- Las propiedades serán privadas y tendrán getters y setters
- Tendrán un constructor público sin argumentos
- Getters:
 - Deberán ser públicos, sin argumentos y con un valor de retorno.
 - Empezarán por **get** más el nombre de la propiedad: getDeptNO()
 - Para las propiedades booleanas, los getters pueden empezar por **is**: isMarried() o getMarried()
- Setters:
 - Deberán ser públicos, devolver un valor vacío y tener un argumento del tipo de la propiedad
 - Empezarán con **set** más el nombre de la propiedad: setDeptNO(int n)

Hibernate creará las clases por nosotros siguiendo los siguientes pasos:


1. Crear una consola de configuración
2. Crear el reverse engineering file (reveng.xml)
3. Creando el Code Generation Configuration y ejecutándolo
4. Corrigiendo posibles errores de la generación automática.

EJEMPLO 2 – CREANDO POJOS

Para crear los POJOs el primer paso es crear una consola de configuración para ello vamos al menú contextual de archivo y seleccionamos:

Archivo > Nuevo > Otro > Hibernate > Hibernate Console Configuration

En el formulario configuramos el nombre y el proyecto, seleccionaremos anotaciones y la versión de Hibernate 4.3 y seleccionamos el fichero de configuración que hemos creado anteriormente:

Edit launch configuration properties 

Select or configure a Console Configuration

Name:

Main Options Classpath Mappings Common

Type:
☐ Core ☒ Annotations (jdk 1.5+) ☐ JPA (jdk 1.5+)

Hibernate Version:

Project:

Database connection:

Property file:

Configuration file:

Persistence unit:

Una vez lleno, cambiamos a la pestaña de opciones y en database dialect configuramos MySQL:

Edit launch configuration properties

Select or configure a Console Configuration



Name:

☒ Main
 ☒ Options
 ☐ Classpath
 ☐ Mappings
 ☐ Common

Database dialect: MySQL

Naming strategy:

Entity resolver:

El segundo paso es crear el fichero de Reverse Engineering, para ello vamos a:

Archivo > Nuevo > Otro > Hibernate > Hibernate Reverse Engineering File

Una vez abierto el fichero, vamos a la pestaña de *Table Filters* y hacemos click en refresh, nos saldrá un popup pidiendo que seleccionemos la Console Configuration, seleccionamos la que hemos creado en el paso anterior y nos saldrá el listado de DB y tablas que hay en el equipo. Seleccionamos las que queremos y las incluimos:

hibernate.cfg.xml *Hibernate Reverse Engineering Editor

Table filters
Table filters defines which tables/views are included when performing reverse engineering.

Database schema:

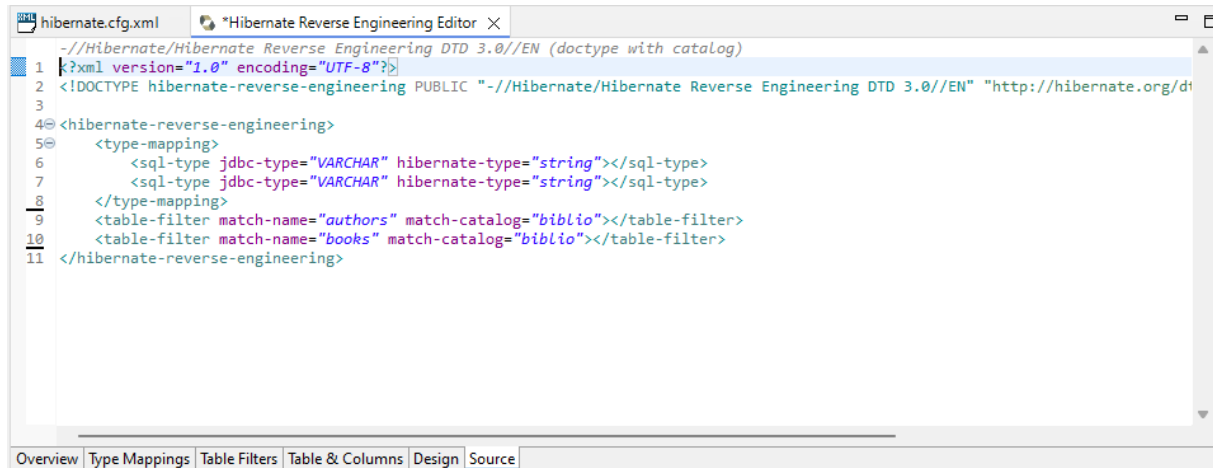
- > biblio
- ▼ personal
 - > departamentos
 - > empleados
- > sys

Table filters:

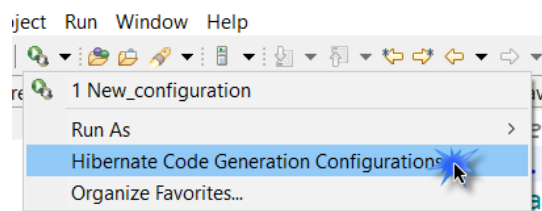
!	Catalog	Schema	Table
	personal	.*	departamentos
	personal	.*	empleados

Overview Type Mappings **Table Filters** Table & Columns Design Source

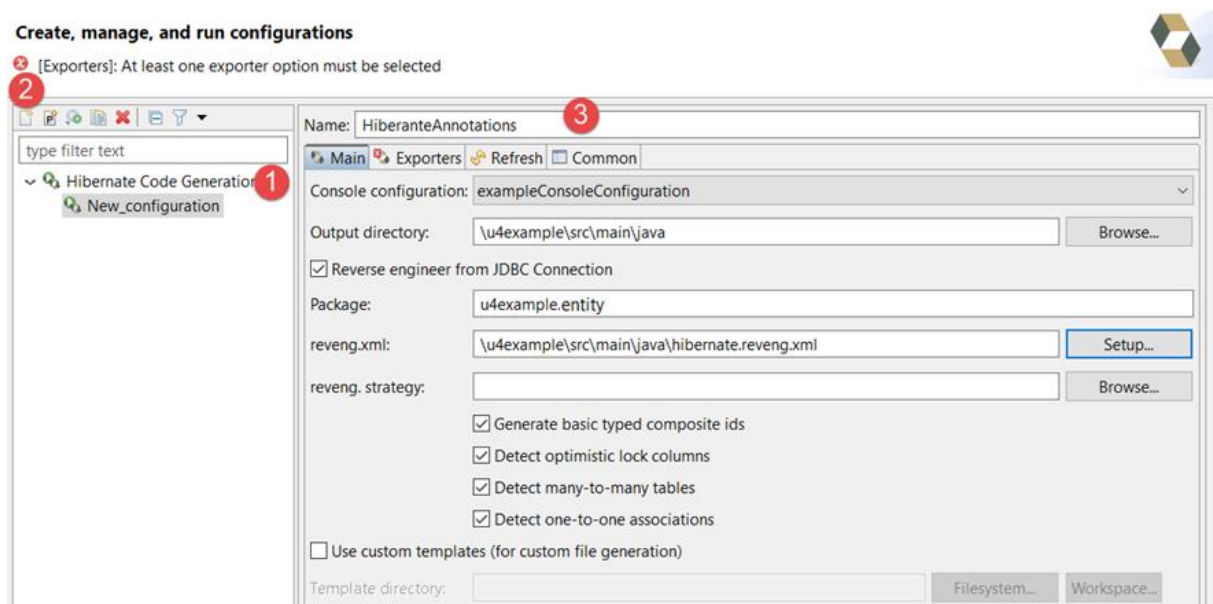
Una vez incluidas si vamos a la pestaña de source, deberíamos tener lo siguiente:



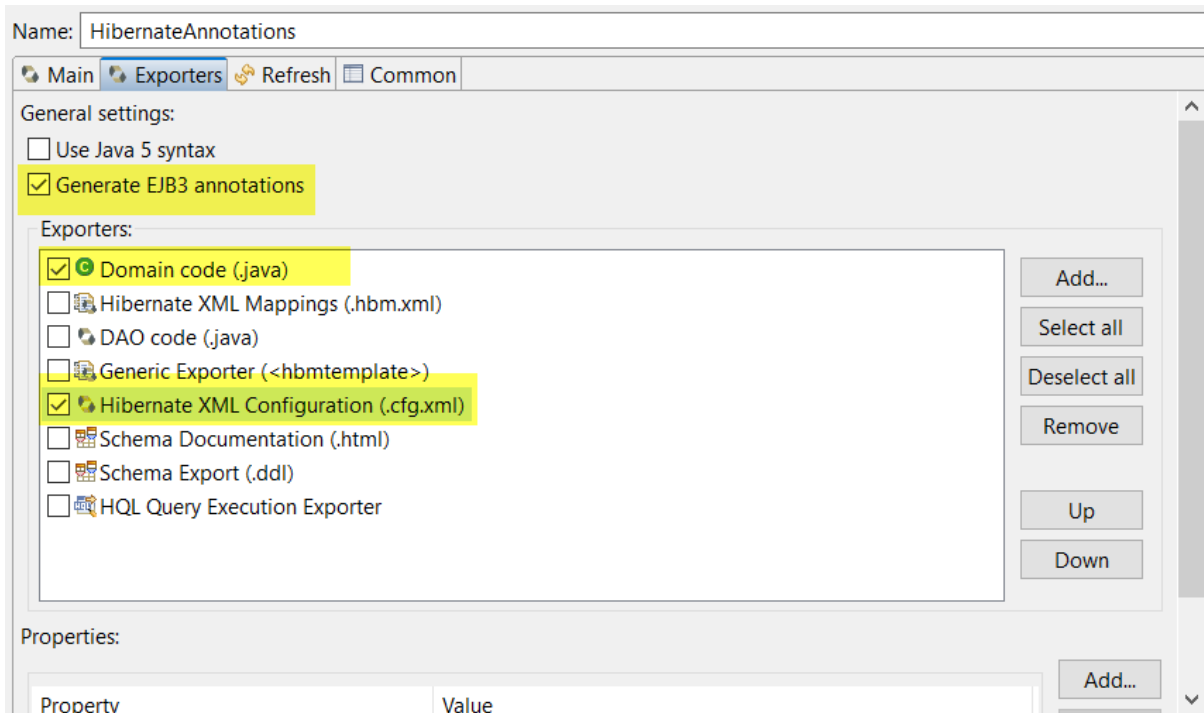
El siguiente paso es crear el Code Generation Configuration y ejecutarlo para ello, primero hacemos click en Hibernate Code Generation:



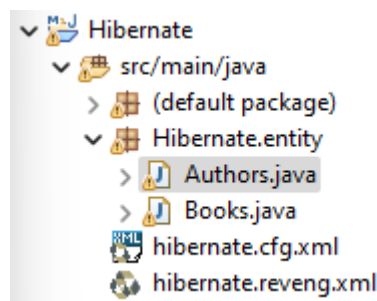
Seleccionamos nueva configuración y configuramos el nombre, elegimos el directorio de salida, que en nuestro caso será la carpeta raíz del código, elegimos el nombre del paquete (como convención, el nombre del paquete de clases persistentes se llama entity) y elegimos el fichero de reveng.xml:



Cuando tenemos el formulario completo, pasamos a la pestaña de “exporters” y seleccionamos *Generate EJB3 annotations, domain code* y *Hibernate XML Configuration*.



Ejecutamos la configuración y por último revisamos los errores de la generación automática:



Abrimos el fichero Authors y books, dependiendo de la versión de Oracle que usemos puede que ten los import, tengamos que cambiar javax por jakarta:

```

4 import java.util.HashSet;
5 import java.util.Set;
6 import javax.persistence.Column;
7 import javax.persistence.Entity;
8 import javax.persistence.FetchType;
9 import javax.persistence.Id;
10 import javax.persistence.OneToOne;
11 import javax.persistence.Table;
12

```

```
4 import jakarta.persistence.Column;  
5 import javax.persistence.Entity;  
6 import javax.persistence.Id;  
7 import javax.persistence.Table;  
8
```


En cada uno de los archivos tendremos que arreglar el warning de Serializable, haciendo click en la sugerencia de Add default versión ID:

4 quick fixes available:

+ [Add default serial version ID](#)

Y también corregimos la declaración de los sets:

```
private Set bookses = new HashSet(0);  
private Set<Books> bookses = new HashSet<Books>(0);
```



Una vez hecho esto ya tenemos la generación de POJOs completada.

ANOTACIONES JPA

Las anotaciones JPA mapean las relaciones entre las tablas y las clases, las principales son las siguientes:

- **@Entity**: especifica que la clase representa una entidad de DB
- **@Table**: define la table que se relaciona con la clase
- **@Id**: especifica que el atributo es una clave primaria
- **@Column**: especifica que el atributo es una columna de la DB
- **@OneToMany**: define una relación 1:M
 - En Authors, se especifica la table relacionada.
 - FetchType.Lazy indica que los libros de un autor solo se cargarán cuando se use y no antes
- **@ManyToOne**: define una relación M:1 (en Books)
- **@JoinColumn**: realciona un atributo con una columna que es FK

Cuando generamos el código de manera automática las anotaciones se ponen delante de los getters y setters. Es recomendable que las movamos a la declaración de los atributos, de esta forma será más fácil detectar errores:

```
@Entity
@Table(name = "authors", catalog = "biblio")
public class Authors implements java.io.Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "cod", unique = true, nullable = false, length = 5)
    private String cod;
    @Column(name = "name", length = 60)
    private String name;
    @OneToMany(fetch = FetchType.LAZY, mappedBy = "authors")
    private Set<Books> bookses = new HashSet<Books>();
```


EJEMPLO 3 – MOSTRAR LIBROS Y AUTORES

En el proyecto que hemos creado en los dos ejemplos anteriores, en la clase main añadimos las siguientes funciones:

```
public static void showBooks(Session session) {
    Query<Books> q = session.createQuery("from Books", Books.class);
    List<Books> results = q.getResultList();
    System.out.println("Showing books data: ");

    for (Books result : results) {
        System.out.println(result.getId() + ": " + result.getTitle() + ", by "
            + (result.getAuthors() != null ? result.getAuthors().getName() : "Anónimo"));
    }
}
```

```
public static void showAuthors(Session session) {
    Query<Authors> q = session.createQuery("from Authors", Authors.class);
    List<Authors> results = q.list();
    System.out.println("Showing Authors data: ");

    for (Authors result : results) {
        System.out.println("The author " + result.getCod() + " with name " +
            result.getName() + " ha escrito :");
        result.getBooks().forEach(e ->
            System.out.println("\t*" + ((Books) e).getTitle()));
    }
}
```

Y si las ejecutamos en el hilo principal, nos mostrarán la siguiente salida:

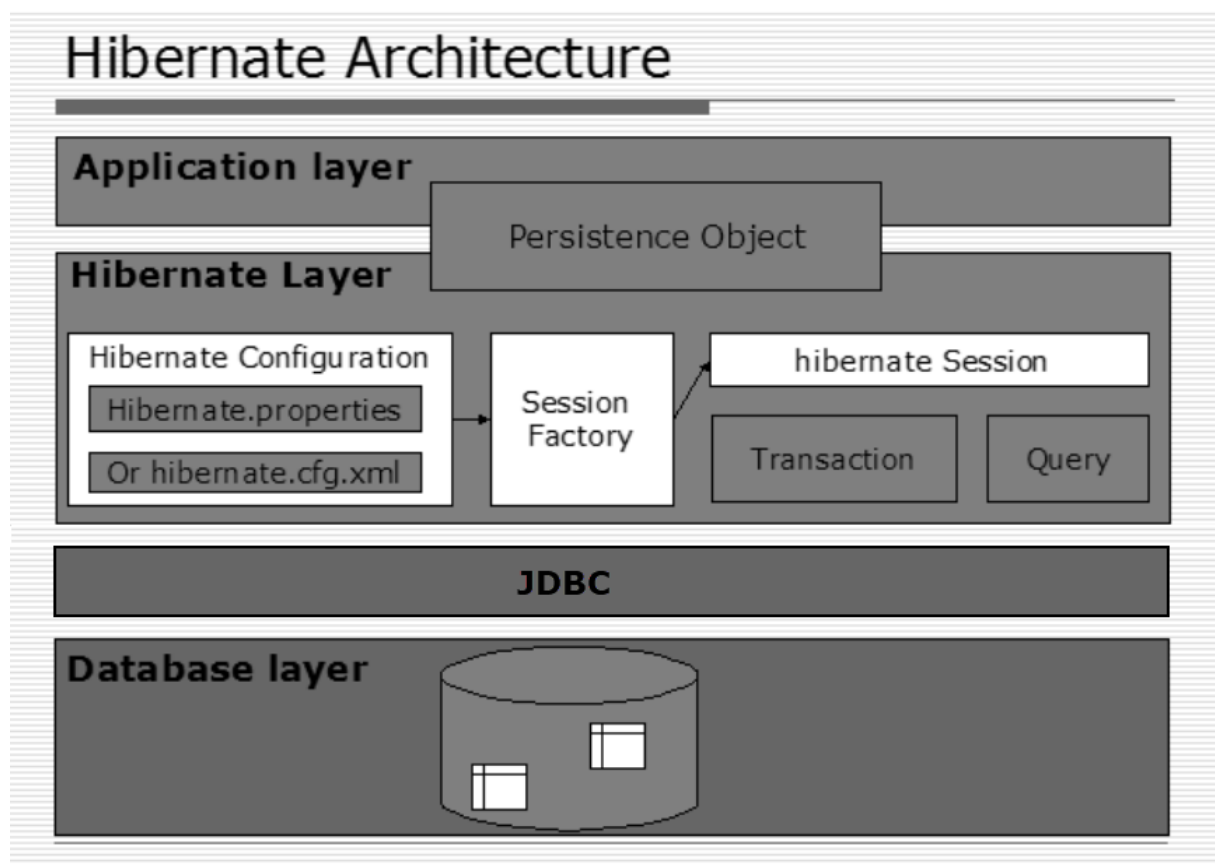
```
INFO: HHH10001001: Connection properties: {password=****, user=sa}
dic 05, 2024 11:29:47 A. M. org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001003: Autocommit mode: false
dic 05, 2024 11:29:47 A. M. org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl$PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
dic 05, 2024 11:29:47 A. M. org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect
dic 05, 2024 11:29:48 A. M. org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator initiateService
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
Opened Session
Showing books data:
1: Macbeth, by William Shakespeare
2: A Game of Thrones, by George R. R. Martin
Showing Authors data:
The author GMART with name George R. R. Martin ha escrito :
    *A Game of Thrones
The author WSHAK with name William Shakespeare ha escrito :
    *Macbeth
END
```

5. PROGRAMANDO CON HIBERNATE

Cuando estemos programando con hibernate las clases principales que utilizaremos serán las siguientes:

- **Session**: similar a la conexión JDBC, guarda objetos, hace queries y gestiona transacciones.
- **SessionFactory**: genera objetos Session, la creación de SessionFactory es cara en cuanto a recursos por los que solo utilizaremos una por programa, creada en la inicialización. Las conexiones a diferentes DBs, requieren diferentes SessionFactory.
- **Configuration**: se utiliza para crear una SessionFactory e incluye toda la configuración para conectar a la base de datos y mapear los objetos.
- **Transaction**: evita que cualquier error que ocurra entre el principio y el final de una transacción produzca un fallo en la DB.
- **Query**: permite hacer queries a la base de datos, las queries se escriben en HQL y controla como se ejecuta la query, por ejemplo, puede enlazar parámetros o limitar el número de resultados.

De esta forma, la arquitectura de Hibernate tendría el siguiente aspecto:



SessionFactory

SessionFactory se obtiene de un objeto *Configuration*:

```
SessionFactory sf = new Configuration()  
    .configure().buildSessionFactory();
```

Y una sesión se abre mediante un *SessionFactory*:

```
Session session = sf.openSession();
```

Como hemos visto anteriormente, la creación de una *SessionFactory* consume recursos por lo que solamente utilizaremos una por programa creada en la inicialización. Para ello utilizaremos un patrón de diseño **singleton**.

Un *Singleton* o instancia única, restringe la creación de objetos de una clase a una única instancia, se utiliza cuando un único objeto se utiliza para coordinar acciones a través del sistema, por lo que creamos un objeto nuevo únicamente en caso de no haberlo creado ya.

Para crearlo, utilizaremos:

- Un constructor privado.
- Una propiedad privada para guardar la instancia única de la clase.
- Un getter público que devuelve la instancia existente o crea una nueva en caso de no existir.

EJERCICIO PROPUESTO:

Crea una clase llamada *hibernateUtil* para proporcionar un objeto Singleton de *SessionFactory*, esta clase debe tener un método público llamado *getSessionFactory()*.

Solución:

```
import org.hibernate.SessionFactory;  
import org.hibernate.cfg.Configuration;  
public class HibernateUtil {  
    // Singleton template with lazy initialization  
    private static SessionFactory instance = null;  
    private HibernateUtil() {}  
    public static SessionFactory getSessionFactory() {  
        if (instance == null) {  
            instance = new Configuration().configure().buildSessionFactory();  
        }  
        return instance;  
    }  
}
```

TRANSACCIONES:

Una transacción simboliza una unidad de trabajos contra una base de datos, esta puede estar formada por múltiples operaciones, debe permitir recuperarse de fallos, mantener la base de datos consistente en caso de fallo y proveer de aislamiento entre los programas.

Las transacciones deben ser **ACID**:

- **Atomic**: todas las operaciones deben ser completadas o no tener ningún efecto
- **Consistent**: la DB debe permanecer en un estado consistente
- **Isolated**: una transacción no debe afectar a otras
- **Durable**: los cambios que haga se deben escribir en un almacenamiento persistente

Para utilizar una transacción la crearemos desde una sesión:

```
Transaction tx = sesion.beginTransaction();
```

Modificaremos los datos que sean necesarios y, en caso de no haber errores, validaremos la transacción con un commit:

```
tx.commit();
```

Y en el caso de que haya ocurrido algún error, desharemos la transacción con un rollback:

```
tx.rollback();
```

Tanto el commit como el rollback, cerrarán la transacción. Una vez cerrada una nueva transacción se puede abrir desde la misma sesión.

Siempre necesitaremos una transacción cuando se modifiquen los datos (insert, update, delete...)

EJEMPLO TRANSACTION:

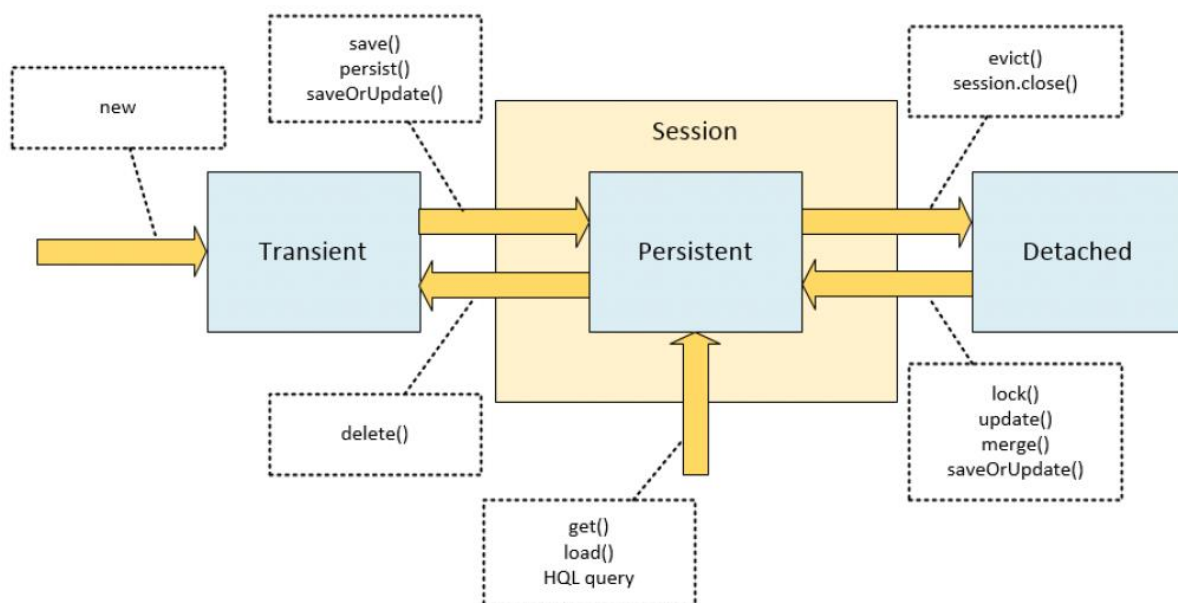
```
//get the session factory
SessionFactory sessionFactory = HibernateUtil.getSessionFactory();
//start the session
Transaction tx =null;

try (Session sess = sessionFactory.openSession()){
    tx = sess.beginTransaction();
    //do some work
    tx.commit();
}
catch (Exception e) {
    if (tx!=null)
        tx.rollback();
    throw e;
}
```

ESTADOS DE UN OBJETO:

Un objeto puede tener 3 estados diferentes:

- **Transient:** se acaba de inicializar, no se ha asociado a una sesión de Hibernate y no tiene ID ni representación en la base de datos.
- **Persistent:** el objeto se ha asociado a una sesión de Hibernate, por lo que tiene ID y una representación en la base de datos. Hibernate sincronizará cualquier cambio en la base de datos cuando se produzca el commit de una transacción.
- **Detached:** el objeto era persistente, pero se ha cerrado la sesión. El objeto existe en el entorno de java y se puede re-anexar a una sesión y hacerlo persistente de nuevo.



EJEMPLO – CREACIÓN DE UN DEPARTAMENTO:

Crea la siguiente base de datos y mapeala con Hibernate:

```
CREATE DATABASE ejemplo;
USE ejemplo;

CREATE TABLE departamentos (
    dept_NO INT NOT NULL PRIMARY KEY,
    dnombre VARCHAR (15),
    loc VARCHAR (15));

CREATE TABLE empleados (
    Emp_no INT NOT NULL PRIMARY KEY,
    apellido VARCHAR ( 20) ,
    oficio VARCHAR ( 15) ,
    dir INT,
    fecha_alta DATE,
    salario FLOAT (6,2),
    comision FLOAT(6,2),
    dept_NO INT,
    FOREIGN KEY(dept_NO) REFERENCES departamentos(dept_NO));

INSERT INTO departamentos VALUES (1, 'Ventas', 'VALENCIA');
INSERT INTO departamentos VALUES (2, 'Administracion', 'MADRID');
INSERT INTO departamentos VALUES (3, 'Ingenieria', 'BARCELONA');
INSERT INTO departamentos VALUES (4, 'Fabricacion', 'BARCELONA');

INSERT INTO empleados VALUES (1, 'Garcia', 'Comercial', 1, '2019-01-01', 1200, 20, 1);
INSERT INTO empleados VALUES (2, 'Martinez', 'Comercial', 1, '2020-01-01', 1800, 15, 1);
INSERT INTO empleados VALUES (3, 'Torres', 'Tecnico Com', 1, '2020-01-01', 1800, 15, 1);
INSERT INTO empleados VALUES (4, 'Pérez', 'Administrativo', 1, '2019-02-01', 1300, 0, 2);
INSERT INTO empleados VALUES (5, 'López', 'Ing Jefe', 1, '2019-01-01', 2200, 5, 3);
INSERT INTO empleados VALUES (6, 'Sánchez', 'Ingeniero', 1, '2019-01-01', 1800, 5, 3);
```

Creación de un nuevo departamento:

```
SessionFactory sf = HibernateUtil.getSessionFactory();

Transaction tx = null;
try (Session sess = sf.openSession() ){
    tx = sess.beginTransaction();
    //do some work
    Departamentos dep = new Departamentos(778);
    dep.setDnombre("new department");
    dep.setLoc("Valencia");

    sess.persist(dep);
    tx.commit();
}
catch (Exception e) {
    if (tx!=null)
        tx.rollback();
    throw e;
}
```

Una vez ejecutado el código comprobaremos la inserción con el cliente de MySQL.

PERSISTIENDO OBJETOS

Para hacer los objetos persistentes, utilizaremos el método *persist* de *Session*, este método añade una nueva instancia entidad al contexto de persistencia, la instancia cambia de un estado **transient** a un estado **persistent**.

Persist(Object object)

Los cambios no se insertan hasta que no se ha hecho un commit de la transacción:

```
Transaction tx = null;
try (Session session = sf.openSession()) {
    tx = session.beginTransaction();

    Departamentos dep = new Departamentos();
    dep.setDeptNo(6);
    dep.setDnombre("Nuevo Dep");
    dep.setLoc("locDept");

    Empleados emp = new Empleados(117, dep, "Garcia", "Comercial",
        7369, Date.valueOf("2023-01-01"), 1500f, 10f);

    session.persist(dep);
    session.persist(emp);
    tx.commit();
} catch (Exception e) {
    if(tx!=null) tx.rollback();
    throw e;
}
```

Persisting order is not important since insertion is done when committing

EJERCICIO

Modifica el código anterior para insertar nuevos empleados y departamentos mediante una entrada de teclado del usuario.

- Debes verificar la inserción mediante un cliente de MySQL
- Comprueba la diferencia entre *commit* y *rollback*
- Cambia el orden de persistencia
- Comprueba que los nuevos empleados aparecen en “*empleadoses*” cuando son persistentes

CARGANDO OBJETOS

¿Cómo creamos un nuevo empleado en un departamento existente?

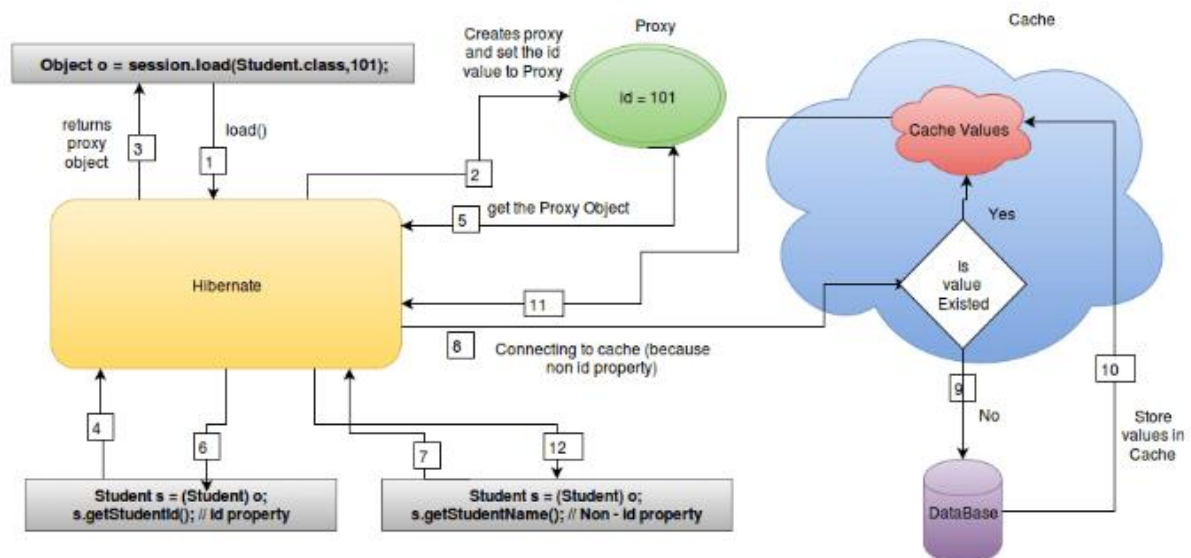
Es necesaria una carga desde la base de datos, para cargar desde la base de datos utilizamos los métodos *getReference* y *get*:

- **getReference(className, id)**
 - Devuelve la referencia a un objeto, pero no instancia el objeto
 - Si el objeto no existe lanza una excepción
- **get(className, id)**
 - Devuelve una instancia del objeto,
 - Es más lento
 - Si el objeto no existe devuelve valor *NULL*

Usaremos el método *getReference()* si estamos seguros de que el objeto existe, en caso de que no exista, obtendremos una excepción. Este método solamente devuelve un proxy y no accederá a la base de datos hasta que se invoque el objeto.

Un proxy significa que hibernate preparará un objeto con un identificador en la memoria sin acceder a la base de datos y la base de datos solo se utilizará cuando se utilice dicho objeto.

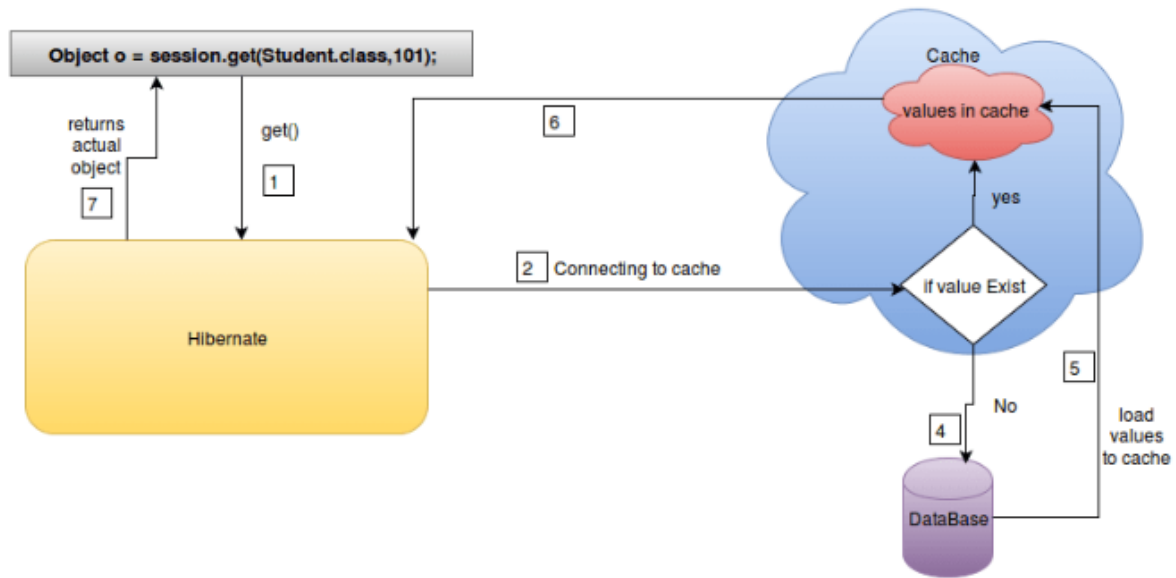
Este método se conoce como *Lazy Loading*



load() is now deprecated. Its function is performed by getReference()

En cambio, utilizaremos el método *get()* si no estamos seguros de que el objeto existe, obteniendo un *NULL* en caso contrario. Este método si accede directamente a la base de datos para crear la instancia del objeto.

Este método se conoce como *Early Loading*.



Ejemplo get():

```

Departamentos dep = session.get(Departamentos.class, 1);
System.out.println("Data for Department 1");
System.out.println("->Name: " + dep.getDeptNo());

System.out.println("->Employees:");
dep.getEmpleados().forEach(emp->
    System.out.println("\t->Surname: " + emp.getApellido()
        + "; Salary: " + emp.getSalario()));
//
// for (Empleados emp : dep.getEmpleados()) {
//     System.out.println("\t->Surname: " + emp.getApellido()
//         + "; Salary: " + emp.getSalario());
// }

```

.forEach() ejecuta una acción por cada elemento de un iterable. La acción se define mediante una expresión Lambda.

BORRADO

Para borrar un objeto, primero, tenemos que cargarlo de la base de datos y acto seguido emplearemos el método ***remove()*** de *Session*. Los cambios únicamente se aplicarán en la base de datos con el commit una transacción. Hay que ir con cuidado con las referencias al objeto borrado.

```
Transaction tx = null;
try (Session session = sf.openSession()) {
    tx = session.beginTransaction();

    Departamentos dep = session.get(Departamentos.class, 5);
    session.remove(dep);
    tx.commit();
} catch (Exception e) {
    if(tx!=null) tx.rollback();
    throw e;
}
```

UPDATE IMPLÍCITO

Hibernate detecta cambios en los objetos persistentes y automáticamente los guarda cuando se produce el commit de una transacción:

```
Session session = sf.openSession();
Transaction tx = session.beginTransaction();

Empleados emp = session.get(Empleados.class, 2);
System.out.println("Old salary: " + emp.getSalario());
System.out.println("Old comision: " + emp.getComision());
//Set new values
emp.setSalario(1800f);
emp.setComision(15f);
tx.commit();
session.close();
//Now reopen session and check values
session = sf.openSession();
emp = session.get(Empleados.class, 2);
System.out.println("New salary: " + emp.getSalario());
System.out.println("New comision: " + emp.getComision());
session.close();
```

ACTUALIZANDO OBJETOS DESANEXADOS "DETACHED"

Para actualizar objetos desanexados utilizaremos el método `merge()` de `Session`, este método crea un objeto persistente de uno detached, de este modo podemos guardar los cambios realizados en este objeto:

En el siguiente ejemplo de código vemos como se crea una sesión y se lee un empleado de la base de datos, se cierra la conexión y el objeto `emp`, pasa a estar *detached*, realizamos cambios en el objeto, y para que se actualicen en la base de datos, abrimos una nueva sesión, creamos un objeto persistente con los datos del objeto *detached* y hacemos un *commit* de los cambios:

```
Session session = sf.openSession();
Empleados emp = session.get(Empleados.class, 2);
session.close();//Now emp is detached

System.out.println("Old salary: " + emp.getSalario());
System.out.println("Old comision: " + emp.getComision());
//Set new values
emp.setSalario(3000f);
emp.setComision(25f);

session = sf.openSession();
Transaction tx = session.beginTransaction();
Empleados persistEmp = session.merge(emp);//returns a persistent obj
tx.commit();//database is modified

System.out.println("New salary: " + persistEmp.getSalario());
System.out.println("New comision: " + persistEmp.getComision());
session.close();
```

6. QUERIES CON HQL

HQL es un lenguaje inspirado en SQL para realizar queries contra objetos de Hibernate, hay que tener en cuenta que HQL funciona con colecciones de objetos y no con tablas y los comandos no son case sensitive.

Comandos básicos:

- **FROM:** lista de las clases a utilizar
- **SELECT:** obtiene una lista de propiedades, “*” no se puede utilizar como “todas las columnas”, si no ponemos select, se obtienen todas las propiedades.
- **Id:** palabra reservada, hace referencia a la clave primaria de la tabla, pero no se requiere el nombre de la columna

Ejemplo:

FROM Empleados: obtiene todas las instancias de la clase Empleados

SELECT id, apellido, emp.salario FROM Empleados emp: obtiene únicamente las propiedades listadas en SELECT

Empleados tiene una referencia a su objeto Departamento correspondiente, no siempre se necesita un JOIN.

Por ejemplo: para cada empleado, obtén el nombre del departamento:

```
SELECT E.apellido, E.departamentos.dnombre FROM Empleados E
```

También Podemos utilizar un FROM de la siguiente forma:

```
FROM Empleados AS em, Departamentos AS dep
```

- **WHERE:** para filtrar la lista de instancias:

```
FROM Empleados WHERE salario>1500
```

```
FROM Empleados E WHERE E.departamentos.dnombre ='Ingenieria'
```

```
FROM Empleados E WHERE E.departamentos.dnombre LIKE '%Ing%'
```

- **ORDER BY:** para ordenar los resultados de la query
- **GROUP BY:** para hacer grupos
- **HAVING:** para filtrar los grupos

- **Funciones de agregación y operadores:** igual que en SQL
 - avg(), sum(), min(), max()
 - count(*) , count(), count(distinct), count(all)
 - +, -, *, /, =, >, <, >=, <=, <>, like
 - and, or, not, in, not in, between, is null, is not null, etc

Para bases de datos que soportan subselects, Hibernate soporta subqueries dentro de queries.

Ejemplos:

1- Obtén el nombre y el salario medio de cada departamento:

```
SELECT departamentos.dnombre, avg(salario) FROM Empleados GROUP BY  
departamentos.dnombre
```

2- Obtén el nombre y el salario medio de cada departamento que tiene un salario medio superior a 1500:

```
SELECT departamentos.dnombre, avg(salario) FROM Empleados GROUP BY  
departamentos.dnombre HAVING avg(salario)>1500
```

3- Obtén todos los empleados cuyo salario sea más alto que la media del salario de la compañía, ordenado por salario:

```
FROM Empleados WHERE salario > (select avg(salario) FROM Empleados) ORDER BY salario
```

4 – Obtén todos los empleados cuyo salario sea más alto que la media de su departamento:

```
FROM Empleados e1 WHERE e1.salario > (select avg(e2.salario) FROM Empleados e2  
WHERE e2.departamentos.dnombre = e1.departamentos.dnombre)
```

CLASE QUERY

Crear un objeto Query es similar a un PreparedStatement:

```
Query<T> q = sess.createQuery("HQLquery", Class<T>);
```

Las Queries también proveen métodos para establecer parámetros en la query.

Las Query pueden ser de varios tipos:

- Retrieval Query
- Scalar Query

RETRIEVAL QUERY

Tiene un método **.list()** que devuelve una colección de todos los resultados de la query. Devuelve una lista de todos los elementos que pasan el filtro de la query y cada elemento corresponde a una fila del resultado y el tipo de la lista devuelta depende de la query.

```
List <T> l = q.list();
```

Para usar la query, primero definimos la query HQL y creamos el objeto Query:

```
Query<T> q = sess.createQuery("HQLquery", Class<T>);
```

Después ejecutamos la query y obtenemos el resultado:

```
List <T> l = q.list();
```

Por último, trabajamos con la lista, considerando que cada elemento como una fila del resultado:

```
for (T element: l) {
    //process the element
}
```

Hay que tener en cuenta que tipo de dato devuelve la query, esto lo vemos en el tipo de dato que ponemos en el FROM

```
Session session = sf.openSession();

Query<Empleados> q = session.createQuery("from Empleados", Empleados.class);
List<Empleados> l = q.list();
l.forEach(e-> System.out.println(e.toString()));

//Another way
System.out.println("---");
session.createQuery("from Empleados", Empleados.class)
    .list()
    .forEach(e -> System.out.println(e.toString()));

session.close();
```

Si tenemos mas de un tipo de objetos en el from (FROM A,B), cada elemento del resultado es una tupla con dos objetos, uno de la clase A y otro de la clase B. Una tupla es una colección de varios objetos, similar a Object[].

Esta clase se encuentra en *javax.persistence.Tuple*; o *jakarta.persistence.Tuple*; dependiendo de la versión que utilicemos:

```
//import jakarta.persistence.Tuple;

Query<Tuple> q = session.createQuery("from Departamentos D, Empleados E "
    + "where D.deptNo=E.departamentos.deptNo order by E.apellido", Tuple.class);

List<Tuple> l = q.list();

for (Tuple obj : l) {
    Departamentos d = (Departamentos) obj.get(0);
    Empleados e = (Empleados) obj.get(1);

    System.out.println(d.getDeptNo() + " | " + d.getDnombre() + " | " + e.getApellido());
}
```

Si utilizamos una query con un **SELECT** cada elemento del resultado es una tupla con tantos objetos como haya definidos en el select, cada uno de la propiedad seleccionada:

```
Query<Tuple> q = session.createQuery("select D.deptNo, D.dnombre, E.apellido"
    + " from Departamentos D, Empleados E"
    + " where D.deptNo=E.departamentos.deptNo order by E.apellido", Tuple.class);

List<Tuple> l = q.list();

for (Tuple obj : l) {
    System.out.println( obj.get(0)+ " | " + obj.get(1) + " | " + obj.get(2));
}
```


SCALAR QUERY

Algunas queries devuelven un único elemento, por ejemplo agregaciones sin agrupar o un WHERE en la clave primaria.

En estos casos utilizaremos el método `.uniqueResult()` que devuelve solamente una instancia que enlaza con la query. En caso que no haya ningún resultado, devolverá un NULL y si más de una instancia cuadra con la query, devolverá una excepción:

```
Query <Departamentos> q = session.createQuery(
    "from Departamentos d where d.deptNo=1", Departamentos.class);
Departamentos dep = q.uniqueResult();
System.out.println(dep.toString());
```

```
Query <Double> q1 = session.createQuery(
    "select avg(salario) from Empleados", Double.class);
Double avgSalary = q1.uniqueResult();
System.out.println("Average salary: " + avgSalary);
```

Si hay varias propiedades definidas en el SELECT, la query devolverá una tupla, pero en este caso no necesitaremos una lista:

```
Query <Tuple> q2 = session.createQuery(
    "select count(*), sum(salario), avg(salario) from Empleados",
    Tuple.class);
Tuple result = q2.uniqueResult();
System.out.println("Total employees: " + result.get(0)
    + "; Sum of salaries: " + result.get(1)
    + "; Average salary: " + result.get(2));
```

▼ ⓘ result	TupleImpl (id=83)
▼ ⓘ row	Object[3] (id=87)
> ▲ [0]	Long (id=91)
> ▲ [1]	Double (id=93)
> ▲ [2]	Double (id=94)

EJERCICIO:

Muestra en la consola el resultado de las siguientes queries:

1. La información de los empleados del departamento de Ventas
2. Una lista de los apellidos y la ciudad donde trabaja cada empleado
3. El empleado más veterano
4. El número de empleados en cada departamento

Solución:

```
SessionFactory sf = HibernateUtil.getSessionFactory();
Session sess = sf.openSession();

//      Get the information of the employees of the 'Ventas' department
Query<Empleados> q1 = sess.createQuery(
    "from Empleados E ",
    Empleados.class);
q1.list().forEach(e -> System.out.println(e.toString()));

//      Get a list of the surname and the city of work of each employee
Query<Tuple> q2 = sess.createQuery(
    "select E.apellido, E.departamentos.loc from Empleados E",
    Tuple.class);
q2.list().forEach(t ->
    System.out.println("Apellido:" + t.get(0) + "; Ciudad: " + t.get(1)));
//      Get the most veteran employee
Query<Empleados> q3 = sess.createQuery(
    "from Empleados E "
    + "where E.fechaAlta = "
    + "(select min(E1.fechaAlta) from Empleados E1)",
    Empleados.class);
// "from Empleados e order by e.fechaAlta" y nos quedamos el primero:
list().get(0)
// "from Empleados e where e.fechaAlta <= all "
//      + "(select e1.fechaAlta from Empleados e1)"
q3.list().forEach(e ->
    System.out.println(e.toString()));

//      Get the number of employees in each department
Query<Tuple> q4 = sess.createQuery(
    "select E.departamentos.dnombre, count(*) "
    + "from Empleados E "
    + "Group by E.departamentos.dnombre",
    Tuple.class);
q4.list().forEach(t ->
    System.out.println("Dep_name:" + t.get(0) + "; Num_emp: " + t.get(1)));
```

MUTATION QUERIES

HQL incorpora también los comandos INSERT INTO, UPDATE y DELETE. Estos comandos se ejecutan con el método `.executeUpdate()` de la clase *MutationQuery*

INSERT INTO...

Solo se soporta la forma de **INSERT INTO ... SELECT...**

```
INSERT INTO className(property_list) select_statement
```

Property List trabaja de la misma forma que un statement INSERT de SQL pero en este caso es obligatorio. Los statements SELECT deben ser correctos en HQL y los tipos devueltos deben coincidir con los esperados por el insert.

Ejemplo:

```
String hql = "insert into Partner (id, name) "  
            + "select p.id, p.name from Person p ";  
int insertedPartners = session.createMutationQuery(hql).executeUpdate();  
System.out.println("Inserted rows:" + insertedPartners);
```

UPDATE

Sintaxis:

```
UPDATE className SET property = newValue WHERE condition
```

Varias propiedades pueden ser actualizadas de una, separadas por comas, devuelve el número de filas afectadas.

Ejemplo:

```
Transaction tx = null;  
try {  
    tx=session.beginTransaction();  
    String hql = "UPDATE Departamentos dep SET dep.loc = 'VALENCIA'"  
                + " WHERE dep.loc = 'MADRID'";  
    int modifDeps = session.createMutationQuery(hql).executeUpdate();  
    System.out.println("Modified departments: "+modifDeps);  
    tx.commit();  
} catch (Exception e) {  
    if(tx !=null) tx.rollback();  
}
```

DELETE

Sintaxis:

DELETE className WHERE condition

Devuelve el número de filas modificadas.

Ejemplo:

```

Transaction tx = null;
try {
    tx=session.beginTransaction();
    String hql = "delete Empleados where salario = 1500";
    int deletedEmp = session.createMutationQuery(hql).executeUpdate();
    System.out.println("Deleted employees: "+deletedEmp);
    tx.commit();
} catch (Exception e) {
    if(tx !=null) tx.rollback();
}

```

QUERIES CON PARÁMETROS

Hibernate también admite queries con parámetros, son similares a los PreparedStatement

Hibernate utiliza un nombre para los parámetros:

- EL nombre del parámetro debe empezar por ":" (as :param_name)
- El orden no es relevante
- Un parámetro puede aparecer varias veces pero solo se puede establecer un valor

Ejemplo:

```

Query<Empleados> q = session.createQuery(
    "from Empleados emp where emp.departamentos.deptNo=:ndep"
    + "and emp.salario > :sueldo", Empleados.class);

q.setParameter("ndep", 1);
q.setParameter("sueldo", 1500);
q.list().forEach(e-> System.out.println(e.toString()));

```

Antes de ejecutar una query, los parámetros deben ser asignados de la siguiente forma:

```
.setParameter(String param_name, Object value)
```

```
Query<Empleados> q = session.createQuery(  
    "from Empleados emp where emp.departamentos.deptNo=:ndep"  
    + "and emp.salario > :sueldo", Empleados.class);  
  
q.setParameter("ndep", 1);  
q.setParameter("sueldo", 1500);  
q.list().forEach(e-> System.out.println(e.toString()));
```

También hay un método para setear un parámetro con una lista de valores:

```
.setParameterList(String param_name, Object[] list_of_values)  
or  
.setParameterList(String param_name, Collection list_of_values)
```

Ejemplo:

```
Query<Empleados> q1 = session.createQuery(  
    "from Empleados emp where emp.apellido in :empApellidos",  
    Empleados.class);  
  
q1.setParameterList("empApellidos",  
    new String[] {"Martínez", "Torres", "Pérez"});  
q1.list().forEach(e-> System.out.println(e.toString()));
```