

# 4.- Mapeo Objeto – Relacional.

## Hibernate

---

### 1 Persistencia de objetos.

El concepto de Persistencia está relacionado con el almacenamiento secundario de instancias de objetos. Podemos encontrar distintas definiciones del término *persistencia*, según distintos puntos de vista y autores. Veamos dos de ellas:

*“Es la capacidad del programador para conseguir que sus datos sobrevivan a la ejecución del proceso que los creo, de forma que puedan ser reutilizados en otro proceso. Cada objeto, independientemente de su tipo, debería poder llegar a ser persistente sin traducción explícita. También, debería ser posible que el usuario no tuviera que mover o copiar los datos expresamente para ser persistentes”*

*“Es la capacidad de un lenguaje de programación o entorno de desarrollo para almacenar y recuperar el estado de los objetos de forma que sobrevivan a los procesos que los manipulan”*

Estas definiciones señalan que ...

- ... es tarea del programador, determinar cuándo y cómo una instancia pasa a ser persistente o deja de serlo, o cuando, debe ser nuevamente reconstruida.
- ... la transformación de un objeto en su imagen persistente y viceversa, debe ser transparente para el programador.
- ... todos los tipos y clases deberían tener la posibilidad de que sus instancias perduren.
- ... el programador no debería preocuparse por el mecanismo interno que hace a un objeto ser persistente, sea este mecanismo soportado por el propio lenguaje de programación o por utilidades de programación para sustentar la persistencia, como librerías o frameworks.

En definitiva, el programador debe disponer de algún medio para poder convertir el estado de un objeto, a una representación adecuada sobre un soporte permanente y, posteriormente, poder reconstruir el objeto. Todo ello sin tener que preocuparse de cómo la operación se lleva a cabo.

Hasta ahora hemos visto formas de persistir los objetos que no cumplen con estos requisitos:

- Serialización de objetos (ObjectOutputStream y ObjectInputStream) y almacenamiento en ficheros (FileInputStream, FileOutputStream, RandomAccessFile, ...).

- Almacenamiento de la información de objetos en bases de datos relacionales a través de JDBC.

## 2 El desfase objeto – relacional.

Cuando se trata de almacenar (y procesar posteriormente) información, acudimos a los Sistemas Gestores de Bases de Datos Relacionales (SGBDR). Desde que surgieron en los años 70, las organizaciones las han adoptado masivamente, han invertido en implantarlas, se ha investigado en ellas y el lenguaje SQL se ha convertido en un estándar. En sus bases de datos las organizaciones dejan constancia, no solo de su estado actual, sino de la información que se ha ido generando a lo largo del tiempo y que cosecha grandes volúmenes de datos.

En los años 80 aparece el paradigma de programación orientado a objetos y el mundo empresarial comienza a adoptarlo de forma lenta pero constante. Su implantación supone beneficios en lo que respecta a modularidad, reducción de la complejidad y reusabilidad del código de las aplicaciones (entre otras ventajas). Los objetos están pensados para reflejar el estado actual de las entidades a las que modelan.

En la actualidad el uso de ambas tecnologías (SGBDR y POO) está ampliamente extendido. Esto ha conducido a que, en nuestras aplicaciones manejemos objetos y los mecanismos de la POO (abstracción, asociaciones, herencia, ..) mientras que, por otra parte, la información se almacene en bases de datos relacionales en forma de tablas, campos y filas.

Cuando llega el momento de almacenar (o recuperar) la información de nuestros objetos en una base de datos relacional, se producen una serie de dificultades técnicas derivadas de la estructura que subyace a cada una de las tecnologías. Al conjunto de estas dificultades o diferencias se le llama **desfase objeto-relacional** o **impedancia objeto-relacional**.

El desfase objeto-relacional consiste en la diferencia de aspectos que existen entre la programación orientada a objetos y las bases de datos relacionales. Estos aspectos se puede presentar en cuestiones como:

- **Lenguaje de programación.** El programador debe conocer el lenguaje de programación orientada a objetos (POO) y el lenguaje de acceso a datos.
- **Tipos de datos:** En las bases de datos relacionales los tipos de datos que se utilizan son más simples, están más limitados, mientras que la programación orientada a objetos utiliza tipos de datos más complejos.
- **Paradigma de programación.** En el proceso de diseño y construcción del software se tiene que hacer una traducción del modelo orientado a objetos de clases al modelo Entidad-Relación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas (o filas), lo que implica que se tengan que diseñar dos diagramas diferentes para el diseño de la aplicación.

Nosotros nos centraremos en el problema que supone tener que persistir los objetos de nuestro modelo en una base de datos relacional. El modelo relacional trata con relaciones y conjuntos debido a su naturaleza matemática. Sin embargo, el modelo de Programación Orientada a Objetos trata con objetos y las asociaciones entre ellos.

La escritura (y de manera similar la lectura) mediante JDBC implica: abrir una conexión, crear una sentencia en SQL y copiar todos los valores de las propiedades de un objeto en la sentencia, ejecutarla y así almacenar el objeto. Esto es sencillo para un caso simple, pero trabajoso si el objeto posee muchas propiedades, o bien se necesita almacenar un objeto que a su vez posee una colección de otros elementos.

Supongamos por ejemplo que disponemos de una clase *Cliente*, con su identificador, su nombre y su teléfono. Trasladar a una base de datos relacional resulta sencillo: basta con almacenar los datos del cliente en una tabla que tenga una columna del tipo adecuado para cada atributo de la clase.

Sin embargo, si nuestra clase *Cliente* tuviera un atributo *teléfonos*, consistente en un List con los teléfonos de que dispone el cliente, la traducción no sería ya tan sencilla. Persistir el objeto implicaría la intervención de varias tablas en el SGBD y recuperar la información de un cliente, la ejecución de varias sentencias SQL.

A esto podemos añadir la dificultad que puede suponer representar las asociaciones entre clases o la herencia.

### 3 Herramientas de mapeo objeto - relacional.

El **mapeo objeto-relacional** (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilizando un motor de persistencia.

Las **herramientas de mapeo objeto-relacional** ayudan a manejar las dificultades que produce el desfase objeto-relacional.

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

Desde el punto de vista de Java se han desarrollado varios estándares:

- **EJB (Enterprise JavaBeans)** son una de las API que forman parte del estándar de construcción de aplicaciones empresariales Java EE. Su especificación detalla cómo los servidores de aplicaciones proveen objetos desde el lado del servidor.
- **JDO (Java Data Objects)** Es una API para persistencia de objetos en Java. Los objetos que se persisten son **POJO's**<sup>1</sup>. Los vendors proporcionan optimizadores que modifican los bytecodes de forma que la serialización se realice de forma transparente y más eficiente.
- **JPA (Java Persistence API)**, es la API de persistencia desarrollada para la plataforma Java EE. Es un framework del lenguaje de programación Java que

---

<sup>1</sup> Un **POJO** (acrónimo de Plain Old Java Object) es una sigla utilizada por programadores Java para enfatizar el uso de clases simples y que no dependen de un framework en especial. Este acrónimo surge como una reacción en el mundo Java a los frameworks cada vez más complejos. En particular surge en oposición al modelo planteado por los estándares EJB anteriores al 3.0, en los que los "Enterprise JavaBeans" debían implementar interfaces especiales.

maneja datos relacionales en aplicaciones usando la Plataforma Java en sus ediciones Standard (Java SE) y Enterprise (Java EE). El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional) y permitir usar objetos regulares (*POJOs*).

JPA ha sido incluida como parte de EJB3. JPA define un estándar del que existen varios distribuidores:

- **Hibernate:** se trata del más utilizado y dispone de una gran cantidad de documentación y foros de ayuda. Según los testeos se trata del que mejor rendimiento tiene, puesto que se trata de una solución que lleva tiempo en funcionamiento, en concreto desde 2001. Es el framework que utilizaremos.
- **TopLink:** se trata de un producto de similares características en cuanto a funcionalidades y rendimiento, sin embargo se trata de un producto de pago y que pertenece a Oracle.
- **EclipseLink:** se trata del ORM de Eclipse Foundation basado en TopLink. JPA es uno de los estándares que soporta. Se trata de un producto reciente y algo inmaduro.
- **OpenJPA:** se trata del ORM de Apache Foundation. Es open source. Se trata de una solución algo inmadura.

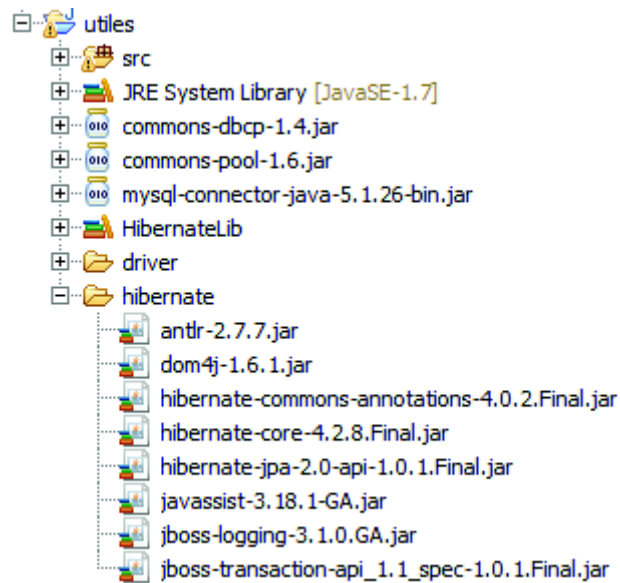
## 4 Hibernate

### 4.1 Instalación de las librerías de hibernate

Descargaremos los archivos de hibernate y crearemos con ellos una *userlibrary* que añadiremos a los proyectos en los que usemos hibernate:

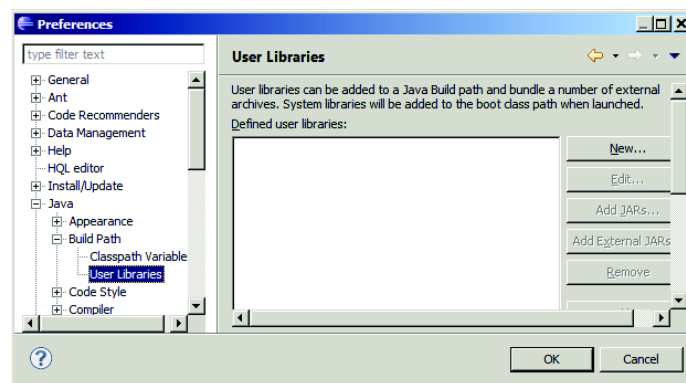
1. Descargar la última versión estable de Hibernate (en nuestro caso, la 4.2) desde el sitio de su distribuidor y descomprimir el contenido en una carpeta.
2. Copiar las librerías de Hibernate en el proyecto utiles, de modo que estén disponibles en todos nuestros proyectos. En realidad los archivos podrían estar en cualquier ubicación, pero situarlos en útiles hará que los archivos acompañen siempre al workspace y facilitará el trabajo sobre los proyectos desde distintos ordenadores.
  - 2.1. En el proyecto utiles crea un nuevo folder llamado *hibernate*.
  - 2.2. Copia en el folder *hibernate* las librerías de Hibernate que encontrarás en la carpeta *lib/required* de la carpeta descargada en el paso 1.

El resultado es el siguiente:

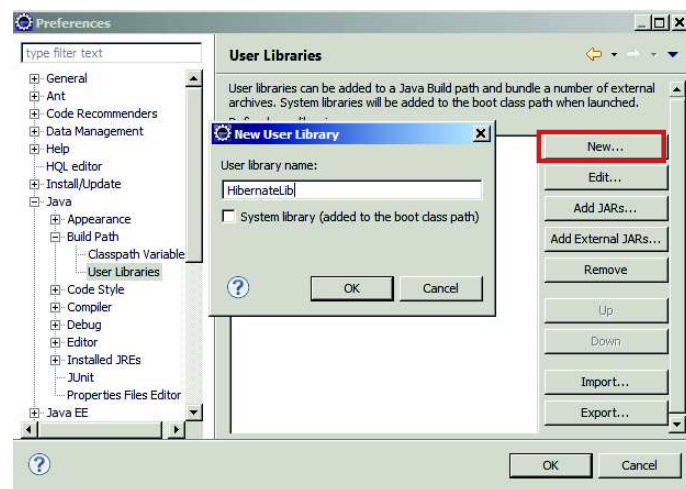


3. En eclipse, crear una userLibrary con los archivos de Hibernate que hemos copiado en la carpeta *hibernate*. Las user libraries no dependen de cada proyecto, sino del workspace en general:

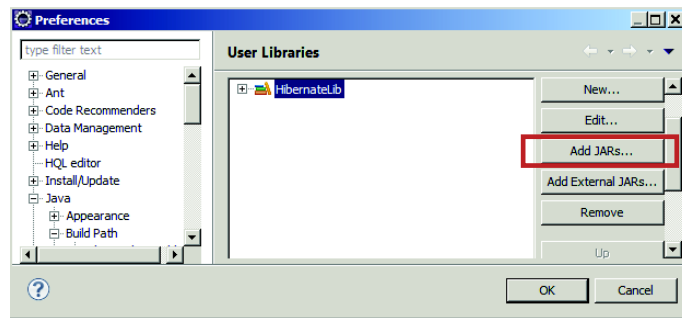
3.1. Ir al menú **Window – Preferences** y localizar **Java – BuildPath – UserLibraries**



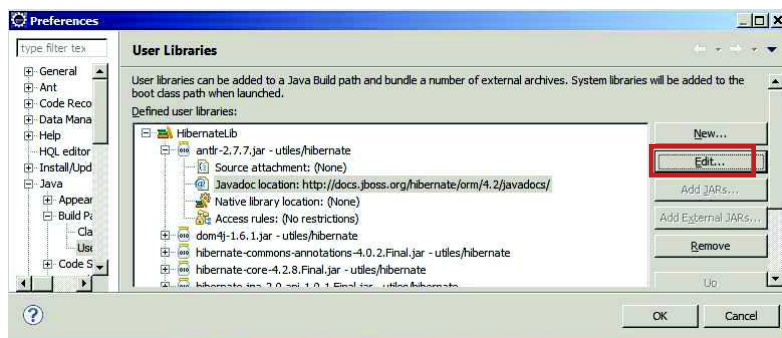
3.2. Crear una nueva *user Library* de nombre **Hibernate4.1**



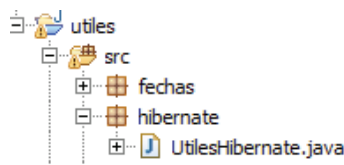
3.3. Selecciona la librería creada y Add JAR's en el folder *hibernate* del proyecto utiles .



3.4. Edita el Javadoc location de **todos** los archivo .JAR con para que se pueda acceder a la documentación en línea. (<http://docs.jboss.org/hibernate/orm/4.2/javadocs/>)



4. Añade al proyecto *utiles* un paquete *hibernate* y crea en él una clase *UtilesHibernate* con el siguiente contenido.



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

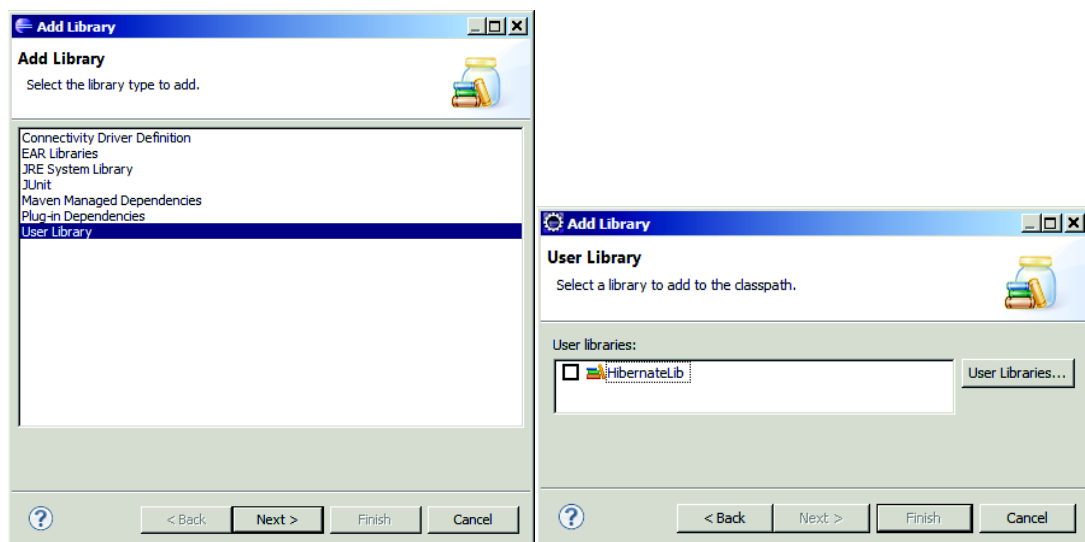
public class UtilesHibernate {
    private static final SessionFactory sessionFactory;
    static {
        try {
            Configuration configuration = new Configuration().configure();
            StandardServiceRegistryBuilder builder = new StandardServiceRegistryBuilder().
                applySettings(configuration.getProperties());
            sessionFactory = configuration.buildSessionFactory(builder.build());
        } catch (Throwable ex) {
            // Log the exception.
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

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

- Esta clase es lo que se llama un **singleton**. Es una clase que se encarga de crear un único objeto de determinada clase para utilizar durante toda la aplicación. El bloque de código `static{}` se ejecuta, una sola vez, cuando la clase se usa por primera vez. En ese fragmento de código, se crea un objeto de la clase **SessionFactory** y se almacena en la variable privada **sessionFactory**. Para acceder al objeto las clases utilizarán el único método público, **getSessionFactory()**, que devuelve el objeto creado.
- El objeto de tipo **SessionFactory** será el encargado de proporcionar al programa los objetos de tipo **Session** sobre los que se realizarán las transacciones que afectan a la base de datos.

## 4.2 Crear un proyecto y mapear una clase.

1. Crea un nuevo Java Project.
2. Añade al nuevo proyecto el proyecto *utiles*.
3. Añade al proyecto la userlibrary creada (*hibernate*). Para ello, haz clic con el botón derecho sobre el proyecto y selecciona **BuildPath – Configure Buil Path - AddLibraries**. Selecciona **userlibraries** y marca la casilla de verificación correspondiente a HibernateLib



4. Para realizar su tarea, Hibernate utiliza un fichero general de configuración del framework. Por defecto, se llama **hibernate.cfg.xml**. Este fichero contiene información sobre los datos de conexión (driver, url, usuario y contraseña). También tiene información sobre diferentes propiedades de Hibernate. Añade a la carpeta src del proyecto un fichero xml de nombre hibernate.cfg.xml con el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost:3306/ciclismo</property>
    <property name="connection.username">root</property>
    <property name="connection.password">mysql</property>

    <!-- JDBC connection pool (use the built-in) -->
    <property name="connection.pool_size">1</property>

    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>

    <!-- Disable the second-level cache -->
    <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider
    </property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>
  </session-factory>
</hibernate-configuration>
```

5. Añadir a la carpeta src del proyecto el fichero **log4j.properties** que se te ha proporcionado
  - La interacción con Hibernate proporciona una serie de mensajes al usuario. Los mensajes se muestran a través de **log4j(Log for Java)**. **Log4j** es una biblioteca open source que permite a los desarrolladores de software elegir la salida y el nivel de granularidad de los mensajes o “logs” en tiempo de ejecución y no en tiempo de compilación como se hace habitualmente.. La configuración de salida y granularidad de los mensajes es realizada a tiempo de ejecución mediante el uso de archivos de configuración. Log4j.properties es el fichero de configuración.
6. Vamos a añadir al proyecto una clase básica (POJO) para representar los equipos de la BD de ciclismo. Los requisitos son:
  - Atributos privados.
  - Métodos *getter* y *setter* para acceder a los atributos.
  - La clase debe disponer de constructor vacío.
  - Debe implementar *Serializable*

Crea el paquete **pojos** y dentro la clase **Equipo** que se muestra a continuación

```
package ciclistas;
public class Equipo implements Serializable {
    private String nombre;
    private String director;

    public Equipo() {}
    public Equipo(String nombre, String director) {
        this.nombre = nombre;
        this.director = director;
    }
    public String getNombre() {...}
    public void setNombre(String nombre) {...}
    public String getDirector() {...}
}
```



7. Hibernate necesita recibir indicaciones de cómo trasladar la información almacenada en objetos a tablas de la base de datos. Estas indicaciones puede recibirlas de dos maneras:
  - A través de ficheros xml, llamados **ficheros de mapeo**.
  - A través de **anotaciones** (Annotations) embebidas en el código de la clase.

En este caso vamos a utilizar annotations. Completaremos el código con las anotaciones que se muestran a continuación:

```
package ciclistas;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Equipo implements Serializable{
    @Id
    @Column (name="nomeq")
    private String nombre;

    private String director;

    public Equipo() {}
    public Equipo(String nombre, String director){
        this.nombre = nombre;
        this.director = director;
    }
    public String getNombre() {...}
```

- Una entidad es un objeto Java regular (POJO) que Hibernate se encargará de persistir. Para marcar un objeto como entidad usando anotaciones se usa la anotación **@Entity** en la línea anterior al comienzo de la definición de la clase. Si no se indica otra cosa, la entidad se mapeará a una tabla de la base de datos con el mismo nombre que la entidad, en este caso **equipo**
  - Las clases mapeadas deben indicar cuál es la clave primaria de la tabla correspondiente. El atributo **nombre** actúa como clave principal. Esto se indica anteponiendo la anotación **@Id** al atributo.
  - Mediante **@Column** indicamos que el atributo **nombre** se corresponde con la columna **nomeq** de la tabla equipos.
  - El atributo **director**, en cambio, se mapea a la columna del mismo nombre, es decir **director** y por ello no es necesario utilizar ninguna anotación.
8. Indicaremos que para mapear la clase `ciclistas.Equipo`, deben utilizarse las Annotations que se han incluido en la clase. Para ello añadimos la siguiente línea al fichero de configuración de Hibernate (`hibernate.cfg.xml`)

**`<mapping class="pojos.Equipo"/>`**

Ya tenemos listo el proyecto para trabajar con Hibernate. A continuación vamos a realizar tres pequeños programas que realicen operaciones contra la base de datos a través de Hibernate:

- Grabar un equipo.

- Recuperar los datos de un equipo.
- Recuperar todos los equipos.

Por el momento no vamos a hacer separación en capas para poder entender mejor todo el código en su conjunto.

#### 4.2.1 Guardar los datos de una clase mapeada (Guardar un equipo).

El siguiente programa guarda un equipo en la base de datos.

Una vez ejecutado podremos comprobar que en la base de datos aparece un nuevo equipo. Por otra parte, si lo ejecutamos más de una vez se producirá una violación de la clave primaria al tratar de insertar un equipo que ya existe.

En paquete `_02ejemplos` crea la siguiente clase `AnyadirEquipo`:

```
package _02ejemplos;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import hibernate.UtilesHibernate;
import pojos.Equipo;

public class AnyadirEquipo {
    public static void main(String[] args) {
        SessionFactory factory = UtilesHibernate.getSessionFactory();

        Session sesion = factory.getCurrentSession();
        sesion.beginTransaction();

        Equipo eq = new Equipo("dam1", "Javier García");
        sesion.save(eq);

        sesion.getTransaction().commit();
    }
}
```

- `SessionFactory factory = UtilesHibernate.getSessionFactory();`
  - Obtiene el objeto `SessionFactory` creado en `UtilesHibernate`.
  - `UtilesHibernate` configura `Hibernate` usando por defecto el fichero de configuración **`hibernate.cfg.xml`**
  - Una `SessionFactory` es un objeto seguro entre hilos y **costoso de crear** pensado para que todos los hilos de la aplicación lo compartan. Se crea una sola vez, usualmente en el inicio de la aplicación. En nuestro caso la creación la hace el código estático del *singleton* que hemos creado en la clase `UtilesHibernate` (Apartado 4.1, paso 4) .
- `Session sesion = factory.getCurrentSession();`
  - Obtiene un objeto de tipo **`Session`**.
  - **`Session`** es el objeto principal del API de persistencia. Actúa de intermediario entre Java e `Hibernate`, proporcionando operaciones de creación, lectura y borrado sobre las clases mapeadas.
  - Una `Session` es un objeto de **bajo costo**, inseguro entre hilos que se debe utilizar una sola vez luego se debe descartar: para un sólo pedido, una sola conversación o una sola unidad de trabajo. Una `Session` no obtendrá una `Connection JDBC` o un `Datasource` a menos de que sea necesario. No consumirá recursos hasta que se utilice.

- `sesion.save(eq);`
  - Con esta instrucción encargamos a Hibernate que almacene el objeto en la base de datos. Hibernate se encarga de generar la sentencia INSERT necesaria y de ejecutarla con los datos adecuados.

#### 4.2.2 Leer datos de una clase mapeada.

En este otro ejemplo realizamos la operación inversa, es decir, recuperamos los datos de un equipo.

```
package _02ejemplos;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import hibernate.UtilesHibernate;
import pojos.Equipo;

public class CargarEquipo {
    public static void main(String[] args) {
        SessionFactory factory = UtilesHibernate.getSessionFactory();

        Session sesion = factory.getCurrentSession();
        sesion.beginTransaction();

        Equipo e = (Equipo) sesion.get(Equipo.class, "dam1");
        System.out.println(e.getDirector());

        sesion.getTransaction().commit();
    }
}
```

- `Equipo e = (Equipo) sesion.get(Equipo.class, "dam1");`
  - El método `get` de la clase `Session` devuelve un objeto recuperado de la base de datos **a partir de su identificador**. Al método hay que indicarle la clase a la que pertenece el pojo para que sepa la tabla a atacar y el identificador del objeto en la tabla, en este caso el nombre del equipo, para que sepa qué fila recuperar.
  - El método devuelve un `Object` y por tanto hay que hacer un casting a `Equipo`.
  - `Get` devuelve `null` si no hay ningún elemento con el identificador indicado.
  - Existe otro método para recuperar una fila por su identificador: `Session.load`. Éste método lanza una excepción si no hay ningún elemento con el identificador indicado.

#### 4.2.3 Listar los datos de una tabla.

En tercer lugar vamos a recuperar más de un elemento de la base de datos. En concreto vamos a recuperar todos los equipos de la tabla de equipos.

```

package _02ejemplos;

import java.util.List;
import hibernate.UtilesHibernate;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import pojos.Equipo;

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

        SessionFactory factory = UtilesHibernate.getSessionFactory();

        Session session = factory.getCurrentSession();
        session.beginTransaction();

        Query q = session.createQuery("SELECT e FROM Equipo e");
        List<Equipo> lista = (List<Equipo>) q.list();

        for(Equipo eq: lista){
            System.out.format("%-15s %s%n",eq.getNombre(),eq.getDirector());
        }

        session.getTransaction().commit();
    }
}

```

- `Query q = session.createQuery("SELECT e FROM Equipo e");`
  - Crea un query, una consulta, a partir del objeto `Session`. No se trata de una consulta de SQL como haríamos con `jdbc`, sino de una consulta de HQL (Hibernate Query Language). Más adelante en el tema veremos con más detalle el lenguaje HQL. De momento, hacer notar que las consultas no se realizan sobre tablas de una base de datos sino sobre `pojos`, es por eso que “*Equipo*” aparece necesariamente en mayúsculas, pues se refiere a la clase *Equipo* y no a la tabla *equipo*.
- `List<Equipo> lista = (List<Equipo>) q.list();`
  - El método `list` de la clase `Query` devuelve un `List` con todos objetos resultantes de la consulta.

## 4.3 Asociaciones/Relaciones entre tablas

### 4.3.1 Relaciones “Uno a uno”

Una relación uno a uno se da cuando una entidad(origen) tiene un atributo que hace referencia a otra entidad(destino). Si la entidad destino tuviera también una referencia de vuelta a la de origen, entonces ésta sería también una relación uno a uno.

En JPA y Java todas las relaciones son unidireccionales, es decir que si un objeto de origen hace referencia a un objeto de destino no existe una garantía de que el objeto de destino también haga referencia al objeto de origen. Esto es diferente en las bases de datos relacionales, en el que las relaciones se definen a través de las claves ajenas y, de existir una relación, la consulta siempre se puede hacer en las dos direcciones.

En Hibernate una relación de uno a uno se define a través de la anotación `@OneToOne`.

Si la entidad en la que aparece la anotación es la que contiene la clave ajena (entidad origen), entonces indicaremos un `@JoinColumn` (o `@JoinColumns` si la clave primaria fuera compuesta) para indicar qué columna de la tabla es la clave ajena.

Si, por el contrario, confeccionamos la relación inversa, es decir, la entidad de destino referencia a la de origen (la que en la tabla contiene la clave ajena), entonces tendremos que utilizar el atributo `mappedBy` para definir la relación.

Veamos un ejemplo

[http://en.wikibooks.org/wiki/Java\\_Persistence/OneToOne](http://en.wikibooks.org/wiki/Java_Persistence/OneToOne)

#### 4.3.2 Relaciones “uno a muchos” y “muchos a uno”

Una relación uno a muchos se da cuando una entidad (origen) tiene un atributo que almacena una **colección** de objetos de otro tipo (destino). La entidad de destino también puede contener una referencia de vuelta a la de origen, en cuyo caso la nombramos como “muchos a uno”.

Como hemos indicado anteriormente, En JPA y Java todas las relaciones son unidireccionales, es decir que si un objeto de origen hace referencia a un objeto de destino no existe una garantía de que el objeto de destino también haga referencia al objeto de origen. Esto es diferente en las bases de datos relacionales, en el que las relaciones se definen a través de las claves ajenas y, de existir una relación, la consulta siempre se puede hacer en las dos direcciones.

En Hibernate una relación de uno a muchos se define a través de la anotaciones `@OneToMany` y `@ManyToOne`

Si la entidad en la que aparece la anotación es la que contiene la clave ajena (entidad origen), entonces utilizaremos `@ManyToOne` e indicaremos un `@JoinColumn` (o `@JoinColumns` si la clave primaria fuera compuesta) para indicar qué columna de la tabla es la clave ajena.

Si, por el contrario, la entidad en la que aparece la anotación es la de destino, la referenciada, entonces tendremos que utilizar `@OneToMany` junto con el atributo `mappedBy` para definir la relación.

Observa el siguiente ejemplo

##### 4.3.2.1 En el lado “muchos” (Ciclista → Equipo).

Ciclista contiene un atributo de tipo Equipo

```
@Entity
//No es necesario en este caso, pero si el nombre de la entidad y el nombre de la
//tabla fuesen diferentes podríamos utilizar la annotation @Table para indicar el
//nombre de la tabla:
@Table (name = "Ciclista")
public class Ciclista implements Serializable{
    @Id
    private int dorsal;
    private String nombre;
    private int edad;
    @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
```

```

@JoinColumn (name = "nomeq")
private Equipo equipo;

public Ciclista() {}
public Ciclista(int dorsal, String nombre, int edad, Equipo equipo) {...}
public int getDorsal() {...}
public void setDorsal(int dorsal) {...}
public String getNombre() {...}
public void setNombre(String nombre) {...}
public int getEdad() {...}
public void setEdad(int edad) {...}
public Equipo getEquipo() {...}
public void setEquipo(Equipo e) {...}
}

```

- `private Equipo equipo`: La entidad `Ciclista` tiene un atributo **de tipo Equipo**
- `@ManyToOne`: Indica que hay una relación uno a muchos entre `Ciclista` y `Equipo`. `Ciclista`, que es la entidad en la que aparece la anotación, representa al lado "muchos" de la relación.
- `cascade = CascadeType.ALL`: Indica que las operaciones realizadas sobre `Ciclista` (inserciones, actualizaciones, borrados, ...) se propagarán a la entidad `Equipo`.
- `fetch = FetchType.LAZY`: Los datos del equipo al que pertenece el ciclista solo se recuperarán si es necesario, es decir, cuando se acceda a ellos
- `@JoinColumn (name = "nomeq")`: Indica que en la tabla `ciclista`, el campo que hace referencia al equipo es `nomeq`.

#### 4.3.2.2 En el lado "uno" (Equipo → Ciclista)

Equipo tiene un atributo que es una **colección** de ciclistas

```

@Entity
public class Equipo {

    @Id
    @Column (name = "nomeq")
    private String nombre;
    private String director;
    @OneToMany (cascade=CascadeType.ALL, fetch=FetchType.LAZY, mappedBy="equipo")
    private List<Ciclista> ciclistas = new ArrayList<Ciclista>();

    public Equipo() {}
    public Equipo(String nombre, String director) {...}
    public String getNombre() {...}
    public void setNombre(String nombre) {...}
    public String getDirector() {...}
    public void setDirector(String director) {...}
    public List<Ciclista> getCiclistas() {...}
    public void setCiclistas(List<Ciclista> ciclistas) {...}
}

```

- `private List<Ciclista> ciclistas = new ArrayList<Ciclista>()`: `Equipo` contiene un atributo que es una **lista** de los ciclistas que pertenecen a él. En este caso se trata de un `List`, pero podría ser también un `Set`. Es necesario crear la colección vacía. Para crear el objeto no se puede utilizar un nombre de interfaz (`List`), hay que usar un nombre de clase (`ArrayList` o `LinkedList`)
- `@OneToMany`: Indica que hay una relación uno a muchos entre `Ciclista` y `Equipo`, que es la entidad en la que aparece la anotación, representa al lado "uno" de la relación
- `cascade = CascadeType.ALL`: Indica que las operaciones realizadas sobre `Equipo` (inserciones, actualizaciones, borrados, ...) se propagarán a la entidad `Ciclista`.
- `fetch = FetchType.LAZY`: Los datos los ciclistas pertenecientes al equipo solo se recuperarán si es necesario, es decir, cuando se acceda a ellos
- `mappedBy="equipo"`: Se refiere al atributo `Ciclista.equipo`. Es decir, se indica qué propiedad del lado "muchos" hace referencia a la entidad actual.

### 4.3.3 Relaciones “Muchos a muchos”

Una relación muchos a muchos se da cuando una entidad (origen) tiene un atributo que almacena una **colección** de objetos de otro tipo (destino) y existe una relación inversa en la que la entidad de destino también contiene un atributo que almacena una colección de objetos origen. Como hemos indicado anteriormente, En JPA y Java todas las relaciones son unidireccionales, y por tanto no se garantiza la bidireccionalidad, como sí ocurre en las BD Relacionales.

En los SGBDR, las relaciones muchos a muchos se representan utilizando una tabla intermedia que contiene claves ajenas de las dos tablas referenciadas:

EMPLOYEE (table)			EMP_PROJ (table)		PROJECT (table)	
ID	FIRSTNAME	LASTNAME	EMP_ID	PROJ_ID	ID	NAME
1	Bob	Way	1	1	1	GIS
2	Sarah	Smith	1	2	2	SIG
			2	1		

En Hibernate se pueden representar:

- Definiendo dos relaciones uno a muchos, similares a las que hemos visto con anterioridad, utilizando las anotaciones `@OneToMany` y `@ManyToOne`
  - Relación uno a muchos entre Employee y Emp\_Proj
  - Relación uno a muchos entre Project y Emp\_Proj
- Definiendo una relación muchos a muchos utilizando la anotación `@ManyToMany`

En el siguiente ejemplo están definida mediante `@ManyToMany`:

Aunque la relación es simétrica, una de las entidades actúa como origen y la otra define la relación inversa a partir de la primera:

En la entidad que actúa como origen indicamos qué tabla sostiene la relación muchos a muchos : `@JoinTable`

```
@Entity
public class Employee {
    @Id
    @Column(name="ID")
    private long id;
    ...
    @ManyToMany
    @JoinTable(
        name="EMP_PROJ",
        joinColumns={@JoinColumn(name="EMP_ID", referencedColumnName="ID")},
        inverseJoinColumns={@JoinColumn(name="PROJ_ID", referencedColumnName="ID")}
    )
    private List<Project> projects;
    ...
}
```

En la otra entidad:

```
@Entity
public class Project {
    @Id
    @Column(name="PROJ_ID")
    ...
}
```

```

private long id;
...
@ManyToMany(mappedBy="projects")
private List<Employee> employees;
...
}

```

#### 4.3.4 Relaciones “Muchos a muchos” con atributos adicionales

### 4.4 La clase Session.

Session es el interfaz principal de ejecución entre la aplicación Java e Hibernate. Contiene el conjunto de métodos que se encargan de la persistencia y nos abstraen de ella. El ciclo de vida de una sesión está marcado por el comienzo y final de una transacción lógica.

La función principal de Session es ofrecer operaciones de creación, lectura y borrado de instancias de clases mapeadas. Estas instancias pueden encontrarse en **tres estados**:

- *Transient* – Un objeto es transient si únicamente se ha creado con el operador new pero no ha sido asociado con una Session de Hibernate. Hibernate no se ocupará de persistir los cambios del objeto.

○ *Ejemplo:*

```

public class _04ObjetoTransient {
    public static void main(String[] args) {
        SessionFactory factory = UtilesHibernate.getSessionFactory();

        Session sesion = factory.getCurrentSession();
        sesion.beginTransaction();

        Equipo eq = new Equipo("dam1", "Javier García");
        eq.setDirector("Miguel Aparicio");

        sesion.getTransaction().commit();
    }
}

```

Aunque el objeto se ha creado dentro del ciclo de vida de la Session, es un objeto Transient e Hibernate no se encarga de almacenarlo inicialmente en la base de datos ni de reflejar los cambios (cambio de director)

- *Persistent* – Un objeto es Persistent cuando se ha asociado a una Session. La asociación se consigue simplemente guardando el objeto (save()) o cargándolo (load() o get()) dentro del ámbito de la Session. Hibernate detectará cualquier cambio hecho en el objeto y sincronizará su estado en la base de datos.

○ *Ejemplos:*

```

public class _05ObjetoPersistent1 {
    public static void main(String[] args) {
        SessionFactory factory = UtilesHibernate.getSessionFactory();

        Session sesion = factory.getCurrentSession();
        sesion.beginTransaction();

        Equipo eq = new Equipo("dam3", "Javier García"); //eq es transient
        sesion.save(eq); // eq es persistent
        eq.setDirector("Miguel Aparicio"); //Hibernate reflejará este cambio en la bd

        sesion.getTransaction().commit();
    }
}

public class _05ObjetoPersistent2 {
    public static void main(String[] args) {
        SessionFactory factory = UtilesHibernate.getSessionFactory();
    }
}

```



```

Session session = factory.getCurrentSession();
session.beginTransaction();

Equipo eq = (Equipo) session.get(Equipo.class, "dam1"); //eq es persistent
eq.setDirector("Miguel Aparicio"); //Hibernate reflejará este cambio en la bd

session.getTransaction().commit();
}
}

```

El objeto se ha persistido (en un caso utilizando save() y en el otro utilizando get()). A partir de ese momento los cambios sufridos por el objetos serán actualizados sobre la BD por Hibernate

- *Detached* – Un objeto es detached si ha sido persistente pero su Session se ha cerrado. La referencia al objeto es válida y el objeto puede cambiar su estado, pero los cambios no se reflejarán en la BD. Un objeto detached se puede “reconectar” y hacerlo de nuevo persistente en una nueva session, tal y como se puede ver en el ejemplo.

- *Ejemplos:*

```

public class _06ObjetoDeattached {
    public static void main(String[] args) {
        SessionFactory factory = UtilesHibernate.getSessionFactory();

        Session session = factory.getCurrentSession();
        session.beginTransaction();

        Equipo eq = (Equipo) session.get(Equipo.class, "dam1"); //eq es persistent
        session.getTransaction().commit();
        // eq es detached
        // el siguiente cambio no se reflejará en la BD
        eq.setDirector("Miguel Aparicio");

        //En una nueva sesion ...
        session = factory.getCurrentSession();
        session.beginTransaction();
        //El objeto se hace de nuevo persistente y es ahora cuando los cambios se reflejan
        en la BD
        session.update(eq);

        session.getTransaction().commit();
    }
}

```

#### 4.4.1 Métodos de la clase Session

Transaction <b>beginTransaction()</b>	Empieza una unidad de trabajo y devuelve el objeto Transaction correspondiente.
Transaction <b>getTransaction()</b>	Devuelve el objeto Transaction asociado con la Session.
Query <b>createQuery</b> (String queryString)	Devuelve un objeto Query correspondiente a la sentencia HQL indicada.
Object <b>get</b> (Class clase, Serializable id)	Devuelve el objeto persistente con el identificador dado, o null si dicho objeto no existe.
Serializable <b>save</b> (Object object)	Persiste el objeto dado asignándole previamente un identificador (si la entidad no lo tiene). Devuelve el identificador generado.
void <b>delete</b> (Object object)	Elimina de la BD la fila correspondiente al objeto persistente.
void <b>update</b> (Object object)	Actualiza el objeto persistente correspondiente al identificador del objeto detached dado. Es decir, que este método se usará para actualizar los datos de un objeto detached.
void <b>saveOrUpdate</b> (Object object)	Si se trata de un nuevo objeto lo guarda en la BD y si se trata de uno ya existente lo actualiza.

<code>void refresh(Object object)</code>	Lee de nuevo desde la BD el estado del objeto dado.
--	---

## 4.5 La clase Query.

Query es la representación en forma de objeto de una consulta HQL.

- El query se construye con `Session.createQuery(String q)`
- El String que se pasa como parámetro contiene la consulta hql.
- El query se ejecuta utilizando:
  - `Query.list()`, que devuelve un List con los resultados.
  - `Query.iterate()`, que devuelve un iterator con los resultados.
  - `Query.scroll()`, que devuelve un ScrollableResults con los resultados.
- La consulta puede estar parametrizada:
  - Con **parámetros posicionales**, al estilo de jdbc (?). En Hibernate el primer parámetro es el 0 y no el 1.

```
Query q = sesion.createQuery("SELECT e FROM Equipo e WHERE director like ?");
q.setString(0, "%miguel%");
List<Equipo> lista = (List<Equipo>) q.list();
```

- Con **parámetros con nombre**. Los parámetros tienen la forma *:nombre* dentro del query. Esto permite que un mismo parámetro pueda aparecer varias veces dentro del mismo query

```
Query q = sesion.createQuery("SELECT e FROM Equipo e WHERE director like :texto or nombre like :texto");
q.setString("texto", "%on%");
List<Equipo> lista = (List<Equipo>) q.list();
```

- No se puede mezclar los dos tipos de parámetros en la misma consulta.
- Otras cuestiones interesantes sobre la ejecución de Queries
  - [http://www.cursoshibernate.es/doku.php?id=unidades:05\\_hibernate\\_query\\_language:01\\_query](http://www.cursoshibernate.es/doku.php?id=unidades:05_hibernate_query_language:01_query)

## 4.6 El ámbito de las unidades de trabajo.

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

### 4.6.1 Unidades de trabajo

Llamamos *unidad de trabajo* a un grupo de operaciones de acceso a datos. Por lo general nos referimos a una *Session* de Hibernate como una unidad de trabajo porque el ámbito de una *Session* es exactamente eso en la mayoría de los casos (una *Session* es otras muchas cosas, como una caché o un API).

Para comenzar una unidad de trabajo se abre una *Session*. Para terminar una unidad de trabajo se cierra la *Session*. También se hace *flush* sobre la *Session* al final de la unidad de trabajo para ejecutar las operaciones de actualización de la base de datos (*UPDATE*, *INSERT*, *DELETE*) para sincronizar el estado en memoria de la *Session* con el de la base de datos.

Una *Session* también ejecuta consultas cada vez que el desarrollador lanza un *query* a través del API o cuando carga los datos de algún objeto (carga perezosa o *lazy*). En este caso pensamos en la *Session* como la manera de acceder a nuestra base de datos, un conjunto de

entidades mapeadas contra la base de datos y una serie de sentencias SQL y DML que se crean y ejecutan automáticamente.

#### 4.6.2 Transacciones

Las transacciones también agrupan operaciones de acceso a datos, de hecho, cada sentencia SQL, sean queries o sentencias DML, se tiene que ejecutar dentro de una transacción con la base de datos. No puede haber comunicación con la base de datos fuera de una transacción.

Existe la posibilidad de trabajar en modo auto-commit, en el que cada sentencia SQL equivale a una transacción. Este modo nunca es apropiado para una aplicación, solo tendría cabida en una ejecución de sentencias SQL desde una consola por parte de un operador. Hibernate deshabilita por defecto el modo auto-commit y recomienda no utilizarlo.

La aproximación adecuada consistiría en definir claramente el ámbito de las transacciones comenzando y terminando cada transacción por programa. Si durante una transacción se produce una excepción, la transacción se deshace (rollback) convenientemente.

#### 4.6.3 El ámbito de una unidad de trabajo.

¿Cuál debe ser el ámbito de una unidad de trabajo?, es decir, ¿cuándo conviene comenzar y terminar una sesión o una transacción? La respuesta no es sencilla y dependerá del tipo y características de la aplicación que estemos realizando: una aplicación web, una aplicación de escritorio monousuario, una aplicación de escritorio multiusuario, etc.

Existen distintos **patrones**:

- **Sesión-por-petición.**

El patrón más común en una aplicación multiusuario cliente/servidor es *sesión-por-petición*. En este modelo, una petición del cliente se envía al servidor, en donde se ejecuta la capa de persistencia de Hibernate. Se abre una nueva Session de Hibernate y todas las operaciones de la base de datos se ejecutan en esta unidad de trabajo. Una vez completado el trabajo, y una vez se ha preparado la respuesta para el cliente, se limpia la sesión y se cierra. Hay una sola transacción de la base de datos para servir la petición del cliente. Transacción y sesión tienen el mismo ámbito y este modelo es adecuado para muchas aplicaciones

- **El antipatrón sesión-por-operación**

En primer lugar decir que no se debe usar. Consiste en abrir y cerrar una Session, iniciar y terminar una transacción para cada llamada simple a la base de datos en un solo hilo.

- **Patrones para conversaciones largas**

El patrón sesión-por-petición no es la única forma de diseñar unidades de trabajo. Muchos procesos empresariales requieren una serie completa de interacciones con el usuario intercaladas con accesos a la base de datos. En aplicaciones empresariales y web no siempre es aceptable que una transacción de la base de datos abarque la interacción de un usuario. Considera el siguiente ejemplo:

- Se abre la primera pantalla de un diálogo. Los datos que ve el usuario han sido cargados en una Session en particular y en una transacción de la base de datos. El usuario es libre de modificar los objetos.
- El usuario hace click en "Guardar" después de 5 minutos y espera que sus modificaciones se hagan persistentes. También espera que él sea la única persona editando esta información y que no ocurra ningún conflicto en la modificación.

Desde el punto de vista del usuario, llamamos a esta unidad de trabajo, una larga *conversación larga* o *transacción de aplicación*. Hay muchas formas de implementar esto en una aplicación.

- Una primera implementación ingenua podría mantener abierta la Session y la transacción de la base de datos durante el tiempo en el que el usuario piensa, bloqueando registros en la base de datos para prevenir la modificación simultánea. Esto es un antipatrón, ya que la utilización de bloqueos dificultaría que la aplicación crezca en número de usuarios simultáneos.
- La segunda forma de implementación consistiría en no bloquear registros de la base de datos y tener en cuenta la posibilidad de que los datos actualizados entren en conflicto con alguna otra actualización simultánea, de distintas formas
  - *Mediante versionado automático*: Hibernate realiza un control automático de concurrencia optimista (supone que no va a haber conflicto). Al final del proceso se puede chequear si ha ocurrido una modificación simultánea durante el tiempo en que el usuario pensaba.
  - *Mediante objetos separados*: Consistiría en utilizar el patrón sesión-por-petición. Mientras el usuario piensa, todos los objetos cargados estarán en estado "separado". Hibernate permite volver a "conectar" los objetos y hacer persistentes las modificaciones. El patrón se llama sesión-por-petición-con-objetos-separados. Se usa el versionado automático para aislar las modificaciones simultáneas.

## 5 HQL

Hibernate nos ofrece un lenguaje con el que poder realizar consultas a la base de datos. Este lenguaje es similar a SQL, y se denomina HQL (HibernateQueryLanguage).

HQL es independiente del SGBD que se esté utilizando. Su principal característica y diferencia con respecto a SQL es que las consultas se realizan sobre el conjunto de entidades (objetos Java) y no directamente sobre las tablas y columnas de la base de datos.

Para ejecutar consultas en HQL se utiliza `createQuery()`:

```
session.createQuery("from Ciclista")
```

Pero también es posible ejecutar consultas SQL usando `createSQLQuery()`:

```
session.createSQLQuery("SELECT nombre FROM ciclista")
```

- La referencia principal se encuentra en la página oficial de Hibernate. Esta información se encuentra en castellano
  - <http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/queryhql.html>
- Un resumen:
  - [http://www.cursohibernate.es/doku.php?id=unidades:05\\_hibernate\\_query\\_language:02\\_hql](http://www.cursohibernate.es/doku.php?id=unidades:05_hibernate_query_language:02_hql)

### 5.1 HQL Editor de JBoss – Hibernate Tools

Las Hibernate Tools de JBoss disponen de un editor de HQL sobre el que se pueden ejecutar las consultas visualizar y navegar por sus resultados.

Para poder utilizar el editor, previamente hay que realizar unas tareas de **configuración**:

1. Abre la perspectiva “Hibernate”. Aparecerá una pestaña “Hibernate Configurations” vacía.
2. Con el botón derecho, accede al menú contextual y selecciona “Add configuration”
3. Completa los datos del formulario (Name, Project, Property file y Configuration file) tal y como se indica en la siguientes figuras:

**Edit Configuration**

**Edit launch configuration properties**

Select or configure a Console Configuration

Name: 03ORM\_Ciclismo

Main Options Classpath Mappings Common

Type:

☐ Core ☒ Annotation (JDK 1.5+) ☐ JPA (jdk 1.5+)

Hibernate Version: 3.5

Project:

adjavi.03ORM.ciclismo Browse...

Database connection:

[Hibernate configured connection] New... Edit...

Property file:

adjavi.03ORM.ciclismo\src\log4j.properties Setup...

Configuration file:

adjavi.03ORM.ciclismo\src\hibernate.cfg.xml Setup...

Persistence unit:

Browse...

Apply Revert

OK Cancel

**Edit Configuration**

**Edit launch configuration properties**

Select or configure a Console Configuration

Name: 03ORM\_Ciclismo

Main Options Classpath Mappings Common

Database dialect:

MySQL

Naming strategy:

Browse...

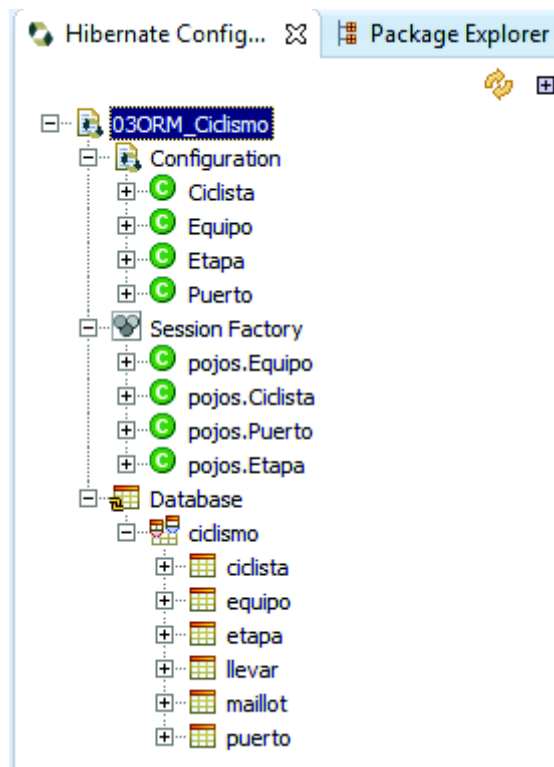
Entity resolver:

Browse...

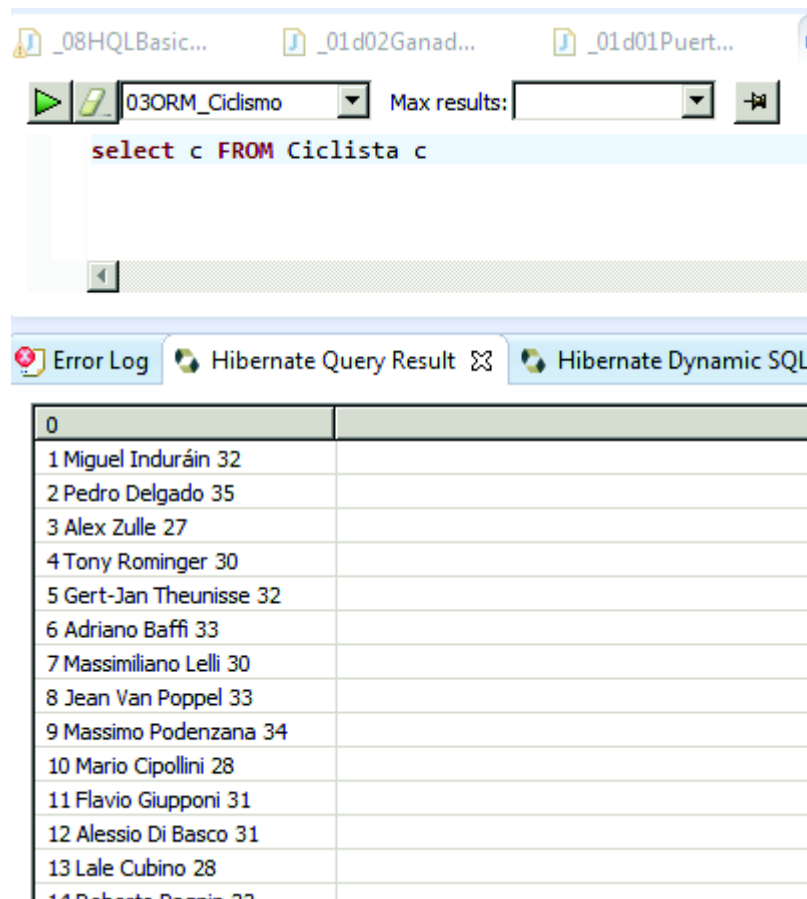
Apply Revert

OK Cancel

- La nueva configuración aparecerá en la perspectiva “Hibernate”



- Haz clic con el botón derecho y selecciona HQL Editor
- Escribe una sentencia hql y ejecútala



```
select c FROM Ciclista c
```

0	
1	Miguel Induráin 32
2	Pedro Delgado 35
3	Alex Zülle 27
4	Tony Rominger 30
5	Gert-Jan Theunisse 32
6	Adriano Baffi 33
7	Massimiliano Lelli 30
8	Jean Van Poppel 33
9	Massimo Podenzana 34
10	Mario Cipollini 28
11	Flavio Giupponi 31
12	Alessio Di Basco 31
13	Lale Cubino 28
14	Roberto Heras 33