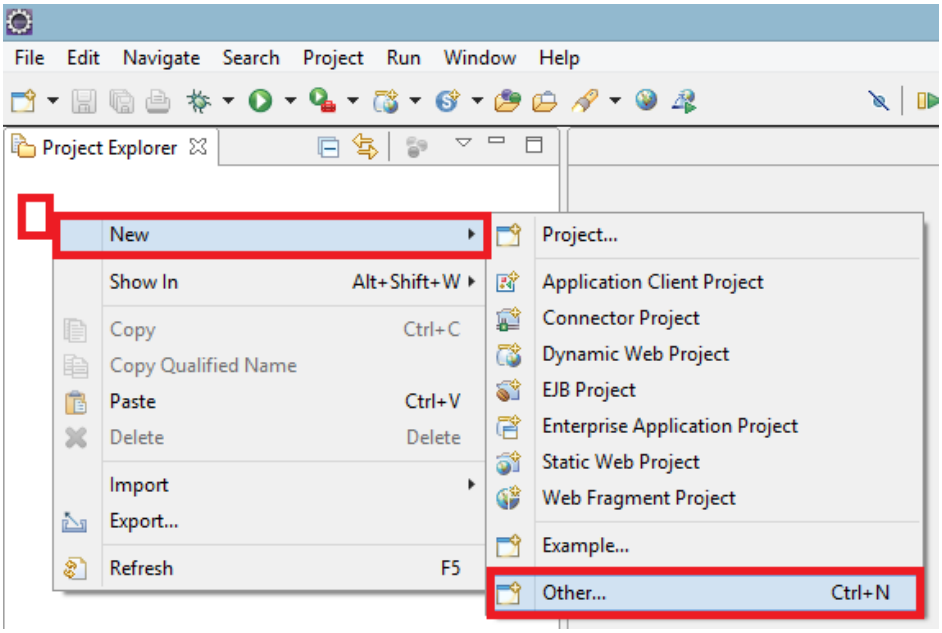
Contenido

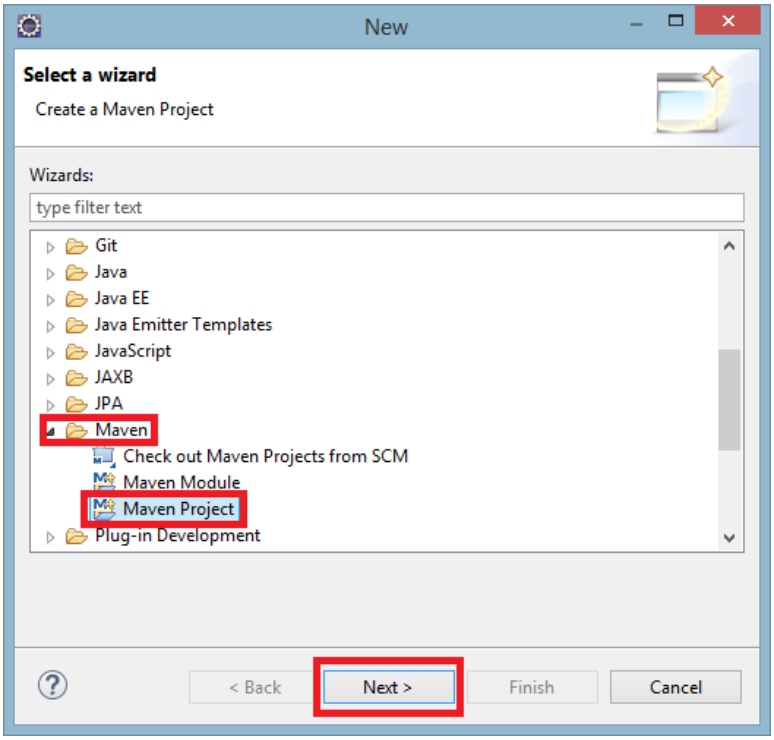
1. Configuración Maven

1.1.Creación de un proyecto con Maven

Creamos un nuevo proyecto Maven. Como se hace habitualmente en Eclipse, pulsamos con el botón derecho del ratón en el área en blanco llamada “Project Explorer”. Seleccionamos en el desplegable en “New” y luego en “Other…”.

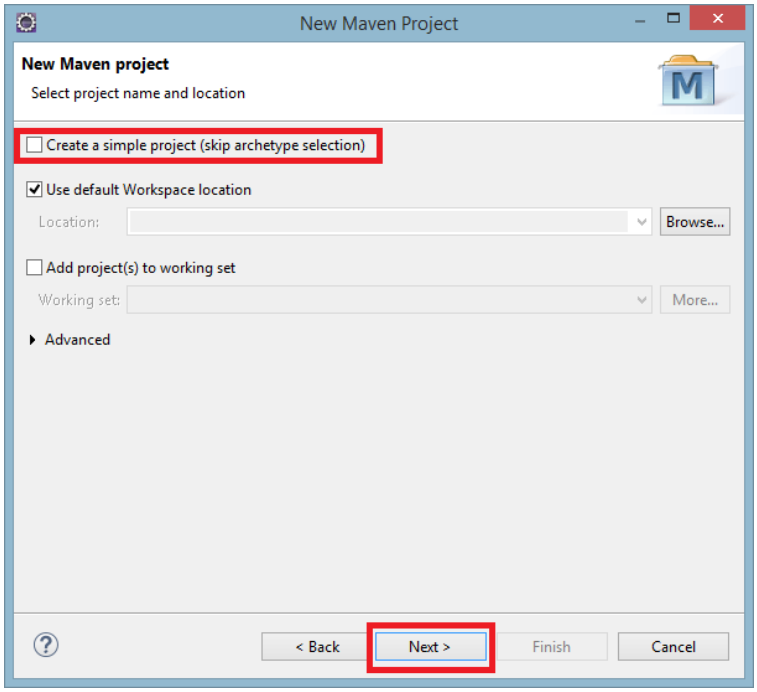


En la ventana que se abre buscamos la carpeta “Maven”, seleccionamos el tipo de proyecto “Maven Project” y pulsamos “Next”

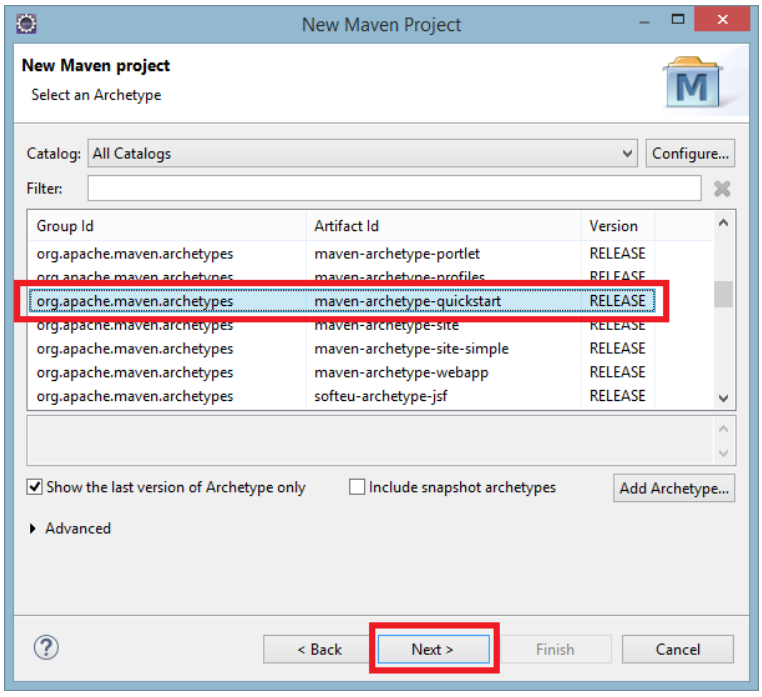


En la siguiente parte del asistente podemos seleccionar la casilla “Create a simple Project (skip archetype selection)”, así nos creará un proyecto simple automáticamente, sin tener que seleccionar un arquetipo. Si dejamos sin seleccionar la casilla, nos aparecerá un buscador para elegir el arquetipo.

Para este ejemplo vamos a descargar un arquetipo simple (el mismo que descargamos con la consola de comandos). Para ello dejamos la casilla desmarcada y pulsamos “Next”.

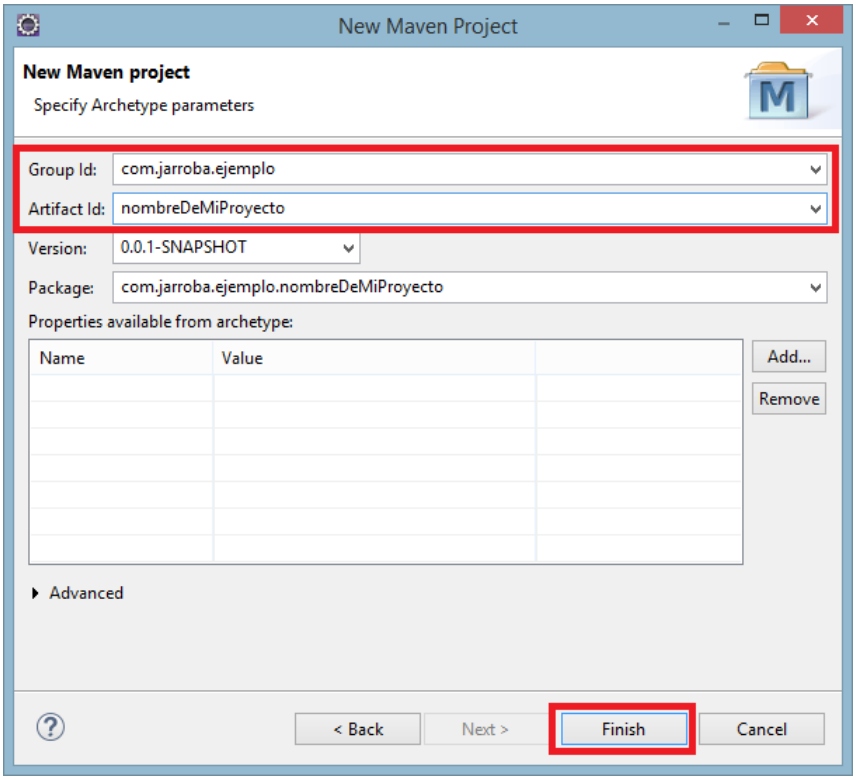


Podremos seleccionar un arquetipo de los que ya están por defecto o desde alguna ubicación. Para este ejemplo seleccionaremos el que tiene de “Group Id” como “org.apache.maven.archetypes”, y de “Artifact Id” como “maven-archetype-quickstart”. Pulsamos “Next”.

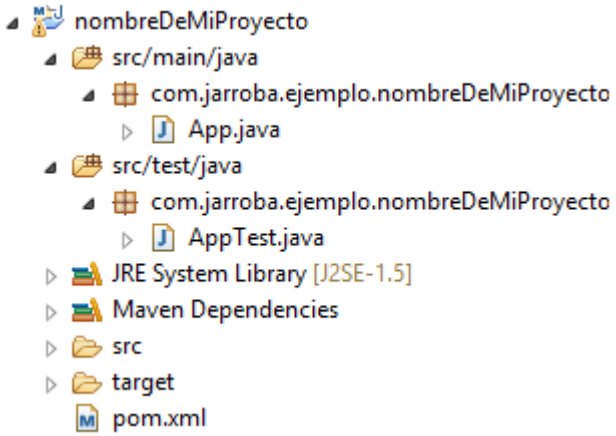


Ya solo nos queda configurar nuestro proyecto, poner el “Group Id” (yo he puesto de ejemplo “com.jarroba.ejemplo”) y el “Artifact Id” (he puesto de ejemplo “nombreDeMiProyecto”). Pulsamos sobre “Finish”.

Puede suscitar la duda de: ¿Si el arquetipo tiene un “Group Id” y un “Artifact Id”, ¿Por qué tengo que escribir otros nuevos? Los que sirvieron para filtrar son los datos que pusieron los creadores de ese arquetipo cuando hicieron el proyecto nuevo; como nosotros vamos a crear un nuevo proyecto, lo tenemos que poner con nuestros datos personales, no con los de otra persona (si nuestro proyecto se convirtiera algún día en arquetipo de otros desarrolladores, otros desarrolladores lo buscarían con nuestros datos, y los sustituirían con los suyos).



Ya tenemos nuestro proyecto creado:



1.2.Configurar las librerías de Hibernate con Maven

En el fichero pom.xml añadimos las siguientes dependencias:

La siguiente dependencia nos agregará las librerías necesarias para trabajar con Hibernate

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
	<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-core</artifactId>
	<version>5.4.9.Final</version>
	</dependency>
```

Para añadir el driver de MySql añadimos la siguiente dependencia:

```
	<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
	<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
	<version>5.1.39</version>
	</dependency>
```

Como podemos ver en las dependencias estamos agregando al proyecto la versión 5.4.9 de hibérnate y la versión 5.1.39 del driver de MySQL.

2. Configuración de hibérnate[1]

Como con cualquier otro framework es necesario configurarlo para su correcto funcionamiento. Ya hemos visto que se puede configurar usando los ficheros de mapeo o las anotaciones Mapeo de una Entidad. Sin embargo, aún quedan aspectos que debemos configurar como:

- Datos de conexión a la base de datos
- Ubicación de las clases a persistir.
- Ubicación de los ficheros `.hbm.xml`.
- Nivel de Log
- Pool de conexiones a usar.
- Etc, etc.

2.1.hibernate.cfg.xml

La forma de configurar hibernate es usando el fichero XML de configuración llamado `hibernate.cfg.xml`. Este fichero deberemos guardarlo en el paquete raíz de nuestras clases Java, es decir fuera de cualquier paquete.Si estamos usando NetBeans deberá ser en la carpeta `src` de nuestro proyecto.

La información que contiene es la siguiente:

- Propiedades de configuración.
- Las clases que se quieren mapear.

[1] http://www.cursohibernate.es/doku.php

El fichero tiene la siguiente estructura:

```
1:<?xml version="1.0" encoding="UTF-8"?>
 2:<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration
DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
 3:<hibernate-configuration>
 4:<session-factory>
 5:<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
 6:<property name="connection.url">jdbc:mysql://localhost/hibernate1</property>
 7:<property name="connection.username">hibernate1</property>
 8:<property name="connection.password">hibernate1</property>
 9:<property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
10:<property name="hibernate.show_sql">true</property>
11:
12:<mapping resource="ejemplo01/Profesor.hbm.xml"/>
13:<mapping class="ejemplo01.Profesor"/>
14:
15:</session-factory>
16:</hibernate-configuration>
```

Podemos ver que el fichero `hibernate.cfg.xml` es un típico fichero xml.

- En la línea 1 vemos la declaración de que es un fichero XML.
- En la línea 2 se aprecia la declaración del DOCTYPE junto con la referencia al documento http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd DTD que permite validarlo. Es decir que si nos descargamos el fichero hibernate-configuration-3.0.dtd podremos saber todos los elementos que hay en un fichero de mapeo.
- El tag raiz del documento xml se llama `<hibernate-configuration>` y se encuentra en la línea 3.
- Dentro del tag `<hibernate-configuration>` encontramos,en la línea 4, el tag `<session-factory>` que contendrá la configuración de hibernate.
- Las siguientes 5 líneas (desde la 5 a la 10) contienen propiedades de configuración mediante el tag `<property>`
- El atributo `name` contiene el nombre de la propiedad de configuración.
- El contenido del tag `<property>` define el valor de la propiedad de configuración.
- Las líneas 12 y 13 contienen el tag `<mapping>` que se usa para indicarle a hibernate las clases que queremos usar desde hibernate.
- El atributo `resource` contiene el nombre de un fichero `.hbm.xml` asociada a la clase que queremos persistir. En nuestro caso del fichero `Profesor.hbm.xml`.
- El atributo `class` contiene la FQCN [1)] de la clase que queremos persistir. En nuestro ejemplo será la clase `ejemplo01.Profesor`.

Para una misma clase solo es necesario indicar una única vez el tag `<mapping>` con el atributo `resource` o `class`.

En el fichero se ha incluido dos veces para indicar las dos posibilidades **excluyentes**.

Pasemos ahora a explicar en detalle los tag `<property>` y `<mapping>`.

2.1.1. Tag <property>

El tag *<property>* se usa para definir cada una de las propiedades de configuración [3)] de hibernate. Como ya hemos indicado consta del atributo *name* con el nombre de la propiedad de configuración. Dentro del tag *<property>* incluiremos el valor de dicha propiedad de configuración.

Volvamos ahora a ver las propiedades de configuración del fichero

```
1:<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
2:<property name="connection.url">jdbc:mysql://localhost/hibernate1</property>
3:<property name="connection.username">hibernate1</property>
4:<property name="connection.password">hibernate1</property>
5:<property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
6:<property name="hibernate.show_sql">true</property>
```

Estas propiedades se usan para poder conectarse mediante JDBC a la base de datos.

Propiedad de configuración	Explicación
connection.driver_class	Contiene la FQCN del driver de la base de datos a usar.
connection.url	La URL de conexión a la base de datos tal y como se usa en JDBC
connection.username	El usuario de la base de datos
connection.password	La contraseña de la base de datos
dialect	El lenguaje de SQL que usará Hibernate contra la base de datos. Este parámetro es opcional ya que hibernate lo puede intentar deducir a partir de los datos de la conexión [4]. Los posibles valores de esta propiedad de configuración son la FQCN de una clase Java que extienda de la clase `org.hibernate.dialect.Dialect`. La lista de dialectos que soporta hibernate se encuentra en el paquete `org.hibernate.dialect`.
hibernate.show_sql	Propiedad opcional que indica si se mostrará por la consola la SQL que lanza Hibernate contra la base de datos. Su posibles valores son `true` o `false`. Esta propiedad es muy útil mientras programamos ya que nos ayudará a entender cómo está funcionanado Hibernate
connection.datasource	Indica el nombre del `DataSource` con el que se conectará Hibernate a la base de datos. En caso de estar esta propiedad no debería estar ninguna de las 4 primeras propiedades sobre la conexión o viceversa. Esta propiedad se usa en aplicaciones Web ya que los datos de la conexión se definen en el servidor de aplicaciones y se accede a la base de datos a través del `DataSource`.

Al utilizar la propiedad `connection.datasource` recuerda que antes del nombre que se le haya dado al datasource hay que añadir:

```
java:/comp/env/
```

Es decir que si nuestro datasource se llama `jdbc/hibernate1` el valor de `connection.datasource` deberá ser:

```
java:/comp/env/jdbc/hibernate1
```

Los dialectos de hibenate son importantes ya que indican en qué base de datos podemos usarlo. En caso de que una base de datos no esté soportada , se puede definir una nueva clase que extienda de `org.hibernate.dialect.Dialect` para poder soportar dicha base de datos.

2.1.2. Tag <mapping>

El tag `<mapping>` se usa para indicarle a Hibernate las clases queremos mapear. Este tag se usa de 2 formas distintas:

- Indicando el nombre del fichero de mapeo ".hbm.xml"
- Indicando la FQCN de la clase Java que hemos anotado

Fichero de mapeo .hbm.xml

Si hemos creado un fichero de mapeo `.hbm.xml` para poder persistir una clase Java a la base de datos , deberemos utilizar el atributo `resource` en el tag `<mapping>`.

```
<mappingresource="ejemplo01/Profesor.hbm.xml"/>
```

El atributo `resource` contiene el *path* dentro de los paquetes Java al fichero `.hbm.xml`.

Nótese que para separar el *path* se usa la barra y no el punto.

Estamos indicando un path dentro del sistema de paquetes de Java y **NO** una ruta en el sistema de ficheros del disco duro.

Nota importante: si el nombre del paquete es una jerarquía debemos separar cada paquete con "/" en vez de con "."

Por ejemplo si tenemos el fichero miclase.hbm.xml dentro del paquete com.miempresa.ejemplo01 tendremos el siguiente tag `<mapping>`:

```
<mapping resource="com/miempresa/ejemplo01/Profesor.hbm.xml"/>
```

FQCN de la clase Java

Si la clase Java la hemos anotado para poder persistirla en vez de usar el fichero `.hbm.xml` deberemos utilizar el atributo *`class`* en el tag *`<mapping>`*.

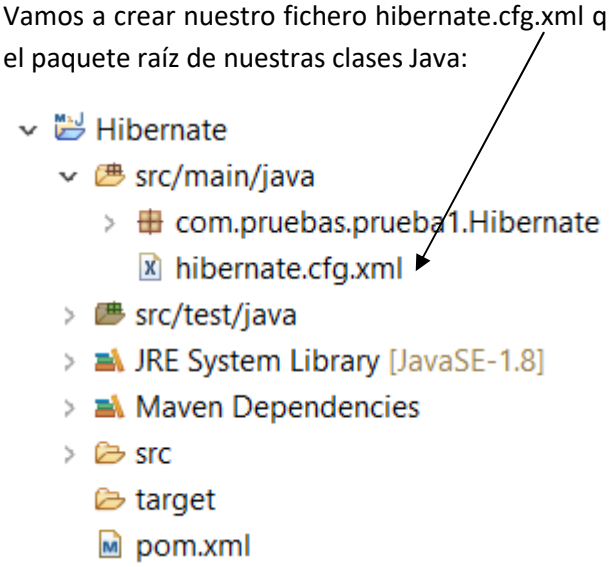
```
<mapping class="ejemplo01.Profesor"/>
```

El atributo *`class`* contiene la FQCN de la clase Java que deseamos que se pueda persistir.

Nótese que para separar los paquetes ahora se usa el punto en vez de la barra.

2.2.Crear nuestro fichero hibernate.cfg.xml

Vamos a crear nuestro fichero hibernate.cfg.xml que como se dijo anteriormente debe guardarse en el paquete raíz de nuestras clases Java:



Creamos el fichero de configuración en la localización indicada e introducimos el siguiente contenido de configuración personalizando los datos de conexión de forma que concuerden con los datos de nuestra base de datos

```
<?xmlversion="1.0"encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<propertyname="connection.driver_class">com.mysql.jdbc.Driver</property>
<propertyname="connection.url">jdbc:mysql://localhost/hibernate1</property>
<propertyname="connection.username">hibernate1</property>
<propertyname="connection.password">hibernate1</property>
<propertyname="dialect">org.hibernate.dialect.MySQL5Dialect</property>
<propertyname="hibernate.show_sql">true</property>

</session-factory>
</hibernate-configuration>
```

Nota: De momento no introducimos la etiqueta mapping ya que lo haremos cuando veamos las dos formas de mapear una clase en el siguiente apartado.

3. Mapeo de entidades

Una entidad va a ser una simple clase Java que deseamos persistir en la base de datos.

Las clases Java que deseamos que persistan deberán tener las siguientes características:

- Deben tener un constructor público sin ningún tipo de argumentos [1)].
- Para cada propiedad que queramos persistir debe haber un método get/set asociado.

- Implementar el interfaz `Serializable` (No es obligatorio que se implemente el interfaz Serializable pero sí recomendable)

Ejemplo de clase:

```
1:public class Profesor implementsSerializable{
 2:private int id;
 3:private String nombre;
 4:private String ape1;
 5:private String ape2;
 6:
 7:public Profesor(){
 8:}
 9:
10:public Profesor(int id, String nombre, String ape1, String ape2){
11:this.id= id;
12:this.nombre= nombre;
13:this.ape1= ape1;
14:this.ape2= ape2;
15:}
16:
17:public int getId(){
18:return id;
19:}
20:
21:public void setId(int id){
22:this.id= id;
23:}
24:
25:public String getNombre(){
26:return nombre;
27:}
28:
29:public void setNombre(String nombre){
30:this.nombre= nombre;
31:}
32:
33:public String getApe1(){
34:return ape1;
35:}
36:
37:publicvoidsetApe1(String ape1){
38:this.ape1= ape1;
39:}
40:
41:public String getApe2(){
42:return ape2;
43:}
44:
45:public void setApe2(String ape2){
46:this.ape2= ape2;
47:}
48:}
```

Vemos en el código fuente cómo la clase `Profesor` tiene un constructor sin ningún tipo de argumentos (Linea 7). Además para las propiedades `id`,`nombre`,`ape1` y `ape2` están el par de métodos `get` y `set` y por último implementa el interfaz `Serializable` (Línea 1).

¿Ya podemos persistir la clase `Profesor` usando hibernate? Pues **NO**, debemos indicarle a hibernate toda la metainformación relativa a esta clase. Hay que *explicarle* como se mapeará el objeto en una base de datos relacional [3)] , indicando para ello en que tabla de base de datos se debe guardar cuál es la clave primaria de la tabla, las columnas que tiene, etc.

3.1.Fichero de mapeo ".hbm.xml"

Para cada clase que queremos persistir se creará un fichero xml con la información que permitirá mapear la clase a una base de datos relacional. Este fichero estará en el mismo paquete que la clase a persistir.

En nuestro caso, si queremos persistir la clase `Profesor` deberemos crear el fichero *`Profesor.hbm.xml`* en el mismo paquete que la clase Java.

Nada impide que el fichero *`.hbm.xml`* esté en otro paquete distinto al de la clase Java. En este sentido suele haber 2 posibilidades:

1. Almacenar el fichero *`.hbm.xml`* en el mismo paquete que la clase Java a la que hace referencia.

2. Crear un *árbol* alternativo de paquetes donde almacenar los ficheros *`.hbm.xml`*. Por ejemplo, si tenemos el paquete raíz *`com.miempresa.proyecto.dominio`* donde se guardan todas las clases Java a persistir, crear otro paquete llamado *`com.miempresa.proyecto.persistencia`* donde almacenar los ficheros *`.hbm.xml`*.

La ventaja de la segunda opción es que en caso de que no queramos usar Hibernate, simplemente hay que borrar toda la carpeta *`com.miempresa.proyecto.persistencia`* y ya está, mientras que la ventaja de la primera opción es que la clase Java y su correspondiente fichero de mapeo están más juntos facilitando en caso de algún cambio en la clase Java el cambio en el fichero de mapeo.

```
1:<?xml version="1.0" encoding="UTF-8"?>
 2:<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
 3:<hibernate-mapping>
 4:<class name="ejemplo01.Profesor" table="Profesor">
 5:<id column="Id" name="id" type="integer"/>
 6:<property name="nombre"/>
 7:<property name="ape1"/>
 8:<property name="ape2"/>
 9:</class>
10:</hibernate-mapping>
```

Podemos ver cómo el fichero *`Profesor.hbm.xml`* es un típico fichero xml.

- En la linea 1 vemos la declaración de que es un fichero XML.
- En la linea 2 se aprecia la declaración del DOCTYPE junto con la referencia al documento http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd DTD que permite validarlo. Es decir que si nos descargamos el fichero hibernate-mapping-3.0.dtd podremos saber todos los elementos que hay en un fichero de mapeo.
- El nodo raiz del documento xml se llama *`<hibernate-mapping>`* y se encuentra en la línea 3.
- La parte interesante de este fichero empieza en la línea 4. Vemos el tag *`<class>`* que nos indica que vamos a mapear una clase.
  - En el atributo *`name`* deberemos poner el FQCN [4] de la clase que queremos mapear.Es decir el nombre de la clase incluyendo el paquete en el que se encuentra.
  - El atributo *`table`* nos indica el nombre de la tabla en la que vamos a mapear la clase. Este atributo es opcional si el nombre de la clase Java y el de la tabla coinciden.
- El tag *`<id>`* de la línea 5 se usa para indicar la propiedad de la clase que es la clave primaria.
  - El atributo *`name`* es el nombre de la propiedad Java que contiene la clave primaria.

    - El atributo *column* contiene el nombre de la columna de la base de datos asociado a la propiedad. Este atributo es opcional si el nombre de la propiedad Java y el nombre de la columna coinciden.
    - El atributo *type* indica el tipo de la propiedad Java. Este atributo no es necesario puesto que Hibernate por defecto ya usa el tipo de la propiedad Java. Más información en Tipos básicos.
- El tag *<property>* de las líneas 6 a la 8 se usa para declarar más propiedades Java para ser mapeadas en la base de datos. Si no declaramos las propiedades Java mediante este tag no se leerán o guardarán en la base de datos.
    - El atributo *name* es el nombre de la propiedad Java que queremos mapear a la base de datos.
    - El atributo *column* contiene el nombre de la columna de la base de datos asociado a la propiedad. Este atributo es opcional si el nombre de la propiedad Java y el nombre de la columna coinciden.

Recuerda que usando el atributo *column* puedes especificar un nombre de columna en la tabla distinto del nombre de la propiedad en la clase Java.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="ejemplo01.Profesor" table="Profesor">
<id column="Id" name="id" type="integer"/>
<property name="nombre"/>
<property name="ape1" column="primer_apellido"/>
<property name="ape2" column="segundo_apellido"/>
</class>
</hibernate-mapping>
```

Como podemos apreciar en el DTD http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd, hay muchos más tags y atributos en los ficheros `.hbm.xml` pero por ahora simplemente hemos visto lo mas básico. Durante el resto del curso iremos viendo muchas mas opciones de este fichero.

Si buscamos documentación sobre hibernate podemos encontrar que el `DOCTYPE` antes de la versión 3.6 era [5] :

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

Hemos comentado que necesitamos los métodos `get/set` para que hibernate acceda a los campos. Sin embargo nos puede interesar que no estén alguno de esos métodos para que el usuario no pueda cambiar o leer los valores. En ese caso le deberemos decir a Hibernate que acceda directamente a las propiedades privadas, ya que por suerte Hibernate sabe hacerlo.

Para ello modificaremos el fichero `.hbm.xml` añadiendo el atributo `access="field"` a la propiedad sobre la que queremos que acceda directamente.
Por ejemplo si no quisieramos tener un `getNombre()` o `setNombre()` de la clase `Profesor` deberíamos cambiar el fichero `Profesor.hbm.xml` añadiendo en la definición de la columna profesor el texto `access="field"`, quedando en ese caso de la siguiente forma:

```
<property name="nombre" access="field"/>
```

La propiedad `access="field"` también puede aplicarse al tag `<id>` como a los diversos tag que definen una propiedad en Hibernate.
Realmente como norma general siempre deberíamos utilizar al tributo `access="field"` ya que así, podremos decidir *tranquilamente* si poner o no los métodos `get/set` y además dichos métodos `get/set` podrían tener reglas o calculos que hicieran que se generaran errores en nuestra aplicación al ser cargados desde Hibernate.Más información sobre este tema en Avoiding Anemic Domain Models with Hibernate
Sin embargo, durante el resto del curso no haremos uso de esta característica para simplificar las explicaciones/ejemplos.

3.2.Mapeo con anotaciones

En vez de configurar el mapeo de clases utilizando ficheros xml como hemos visto en el apartado anterior, se pueden introducir anotaciones en el código de la clase que queremos mapear.

InicialmenteHibernate creó sus propia anotaciones en el paquete `org.hibernate.annotations` pero a partir de la versión 4 de Hibernate la mayoría de dichas anotaciones han sido `java.lang.Deprecated` y ya no deben usarse. Las anotaciones que deben usarse actualmente son las del estándar de JPA que se encuentran en el paquete `javax.persistence`. Sin embargo hay características específicas de Hibernate que no posee JPA lo que hace que aun sea necesario usar alguna anotación del paquete `org.hibernate.annotations` pero en ese caso Hibernate 4 no las ha marcado como `java.lang.Deprecated`.

Veamos ahora el ejemplo de la clase `Profesor` pero mapeada con anotaciones.

```
1: @Entity
 2:@Table(name="Profesor")
 3:public class Profesor implementsSerializable{
 4:
 5:     @Id
 6:@Column(name="Id")
 7:private int id;
 8:
 9:@Column(name="nombre")
10:private String nombre;
11:
12:@Column(name="ape1")
13:private String ape1;
14:
15:@Column(name="ape2")
16:private String ape2;
17:
18:
19:public Profesor(){
20:}
21:
22:public Profesor(int id, String nombre, String ape1, String ape2){
23:this.id= id;
24:this.nombre= nombre;
25:this.ape1= ape1;
26:this.ape2= ape2;
27:}
28:
29:public int getId(){
30:return id;
31:}
32:
33:public void setId(int id){
34:this.id= id;
35:}
36:
37:public String getNombre(){
38:return nombre;
39:}
40:
```

```
41:public void setNombre(String nombre){
42:this.nombre= nombre;
43:}
44:
45:public String getApe1(){
46:return ape1;
47:}
48:
49:public void setApe1(String ape1){
50:this.ape1= ape1;
51:}
52:
53:public String getApe2(){
54:return ape2;
55:}
56:
57:public void setApe2(String ape2){
58:this.ape2= ape2;
59:}
60:}
```

Las anotaciones que se han usado son las siguientes:

`@Entity`: Se aplica a la clase e indica que esta clase Java es una entidad a persistir. Es una anotación estándar de JPA. En nuestro ejemplo estamos indicando que la clase `Profesor` es una entidad que se puede persistir.

`@Table(name="Profesor")`: Se aplica a la clase e indica el nombre de la tabla de la base de datos donde se persistirá la clase. Es opcional si el nombre de la clase coincide con el de la tabla. Es una anotación estándar de JPA. En nuestro ejemplo estamos indicando que la clase `Profesor` se persistirá en la tabla `Profesor` de la base de datos.

`@Id`: Se aplica a una propiedad Java e indica que este atributo es la clave primaria. Es una anotación estándar de JPA. En nuestro ejemplo estamos indicando que la propiedad Java `id` es la clave primaria.

`@Column(name="Id")`: Se aplica a una propiedad Java e indica el nombre de la columna de la base de datos en la que se persistirá la propiedad. Es opcional si el nombre de la propiedad Java coincide con el de la columna de la base de datos. Es una anotación estándar de JPA. En nuestro ejemplo estamos indicando que la propiedad Java `id` se persistirá en una columna llamada `Id`.

`@Column(name="nombre")`: Se aplica a una propiedad Java e indica el nombre de la columna de la base de datos en la que se persistirá la propiedad. Es opcional si el nombre de la propiedad Java coincide con el de la columna de la base de datos. Es una anotación estándar de JPA. En nuestro ejemplo estamos indicando que la propiedad Java `nombre` se persistirá en una columna llamada `nombre`.

`@Column(name="ape1")`: Es igual al caso anterior pero para la propiedad `ape1`.

`@Column(name="ape2")`: Es igual al caso anterior pero para la propiedad `ape2`.

Una diferencia importante entre usar el fichero de mapeo `.hbm.xml` y las anotaciones es que en el fichero es **obligatorio** indicar todas las propiedades que queremos que se persistan en la base de datos, mientras que usando las anotaciones esto no es necesario. Usando anotaciones se persisten todas las propiedades que tengan los métodos `get/set`.

Ya hemos comentado en el apartado anterior sobre como Hibernate accede a los datos al usar el fichero `.hbm.xml`, si mediante el uso de los métodos `get/set` o mediante el acceso a las propiedades.Veamos como se especifica ésto mediante anotaciones:

Si colocamos las anotaciones sobre las propiedades, el acceso será a las propiedades y no serán necesarios los métodos `get/set`.

Si colocamos las anotaciones sobre los métodos `get()`, el acceso será mediante los métodos `get/set`.

4. Utilizando Hibernate

Hasta ahora hemos visto cómo configurar hibernate usando los ficheros XML de configuración o usando notaciones pero no hemos visto nada de código Java para usarlo realmente. Ahora veremos finalmente cómo usar Java para persistir una clase.

La clase que más usaremos en Hibernate es `org.hibernate.Session`. Esta clase contiene métodos para leer, guardar o borrar entidades sobre la base de datos.

Pero antes de poder usarla hace falta crear el objeto `SessionFactory` que mediante el método `SessionFactory.openSession()` nos dará acceso a `Session`.

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

Veamos ahora cómo crear el objeto SessionFactory.

4.1.SesionFactory

La forma de crear el objeto `SessionFactory` es mediante un objeto `org.hibernate.cfg.Configuration` que leerá el fichero de configuración de hibernate `hibernate.cfg.xml` que se encuentra en el directorio raíz de las clases Java.

```
// Inicializa el entorno Hibernate
Configuration cfg = new Configuration().configure();

// Crea el ejemplar de SessionFactory
SessionFactory sessionFactory = cfg.buildSessionFactory(
        new StandardServiceRegistryBuilder().configure().build() );
```

La llamada a **Configuration().configure** carga el fichero de configuración **hibérnate.cfg.xml** e inicializa el entorno de Hibernate. Se necesita crear un objeto del tipo **StandardServiceRegistry** que contiene la lista de servicios que utiliza Hibernate para crear el ejemplar de **SessionFactory**; este normalmente sólo se crea una vez y se utiliza para crear todas las sesiones relacionadas con un contexto dado.

Para conseguir tener sólo una instancia de **SessionFactory** se puede utilizar el patrón de diseño **Singleton**. Este patrón está diseñado para restringir la creación de objetos pertenecientes a una clase. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella (así tenemos un único objeto creado de una clase)

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

Nuestro **Singleton** será una clase de ayuda que accede a **SessionFactory** para obtener una sesión, hay una única SessionFactory que recoge el objeto **SessionFactory** devuelvo por el método **buildSessionFactory**(); este objeto se crea a partir del fichero de configuración (**hibérnate.cfg.xml**). El método **getSessionFactory**() devuelve el valor de la variable estática definida , o lo que es lo mismo, devuelve el objeto **SessionFactory** creado. El nombre de la clase es HibernateUtil.java.

Este es el código de la clase HibernateUtil.java (disponible en recursos de la Unidad 3)

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

publicclass HibernateUtil {
      privatestaticfinal SessionFactory sessionFactory =
buildSessionFactory();

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

      publicstatic SessionFactory getSessionFactory(){
            return sessionFactory;
      }
}
```

4.2.Session

Ahora que ya tenemos el objeto `SessionFactory` podemos obtener la `Session` para trabajar con Hibernate. Como ya hemos visto crear la `Session` es tan sencillo como llamar al método `openSession()`:

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

Una vez obtenida la sesión trabajaremos con Hibernate persistiendo las clases y una vez finalizado se deberá cerrar la sesión con el método `close()`:

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

4.3.Transacciones

Un objeto Session de Hibernate representa una única unidad de trabajo para un almacén de datos dado, y lo abre un ejemplar SessionFactory. Al crear la sesión se crea la transacción para dicha sesión. Se deben cerrar las sesiones cuando se haya completado todo el trabajo de una transacción. El siguiente código ilustra una sesión de persistencia de Hibernate:

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

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

4.4.Estados de un objeto Hibernate

Hibernate define y soporta los siguientes estados de un objeto:

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

  ```
  //Inserto el departamento 60 en la tabla de DEPARTAMENTOS
  Departamentos dep =new Departamentos();
  dep.setDepNo((byte)60);
  dep.setDnombre("MARKETING");
  dep.setLoc("GUADALAJARA");
  session.save(dep);//save() hace que la instancia sea persistente
  ```

- Persistente (Persistent). Un objeto estará en este estado cuando ya está almacenado en la base de datos. Puede haber sido guardado o cargado, sin embargo, por definición, se encuentra en el ámbito de una **Session**. Hibernate detectará cualquier cambio realizado a un objeto en estado persistente y sincronizará el estado con la base de datos cuando se complete la unidad de trabajo. En definitiva, los objetos transitorios solo existen en memoria y no en un almacén de datos,

4.5.CRUD

Ya hemos llegado al punto en que tenemos todo preparado para poder trabajar con Hibernate en las operaciones fundamentales de una base de datos, las operaciones CRUD.

Create: Guardar un nuevo objeto en la base de datos.

Read: Leer los datos de un objeto de la base de datos.

Update: Actualizar los datos de un objeto de la base de datos.

Delete:Borrar los datos de un objeto de la base de datos.

Estas 4 operaciones será tan sencillas de usar desde hibernate como llamar a un único método para cada uno de ellos.

4.5.1. Guardar

Usaremos el método ***save(Object object)*** de la sesión pasándole como argumento el objeto a guardar.

```
//Obtenemos el SessionFactory
SessionFactory sessionFactory=HibernateUtil.getSessionFactory();

//Abrimos la sesión mediante el SessionFactory
Session session =sessionFactory.openSession();

//Creamos el objeto
Profesor profesor=new Profesor(101,"Juan","Perez","García");
      Transaction tx=session.beginTransaction();

session.save(profesor);//<|--- Aqui guardamos el objeto en la base de
datos.

tx.commit();
session.close();
sessionFactory.close();
```

Como vemos, guardar una clase Java en la base de datos solo implica usar una única línea.

4.5.2. Leer

El método que debemos usar es ***get(Class,Serializable)***, al que le deberemos pasar la clase que queremos leer y su clave primaria.

```
Profesor profesor=(Profesor)session.get(Profesor.class,102);
```

El método ***get(Class,Serializable)*** permite leer un único objeto de la base de datos a partir de su clave primaria.

El uso de este método tiene 2 peculiaridades:

- Uso del cast:Es necesario hacer un cast añadiendo `(Profesor)` en el retorno de la función. Esto es así ya que el método `get()` se usa para cualquier tipo de entidad así que Java desconoce qué tipo de datos va a retornar , por lo que debemos decírselo nosotros mediante el cast para “asegurarle” el tipo que retorna.
- El uso de la propiedad `.class`:Ésta es la forma que se ha definido en en lenguaje Java para pasar un objeto de la clase ***java.lang.Class***. Véase Class Literals.

Ejemplo completo 1:

```
	//Obtenemos el SessionFactory
	SessionFactory sessionFactory=HibernateUtil.getSessionFactory();

	//Abrimos la sesión mediante el SessionFactory
	Session session =sessionFactory.openSession();

	Profesor profesor=(Profesor)session.get(Profesor.class,102);

	System.out.println("Profesor:"+profesor.getNombre());

	session.close();
	sessionFactory.close();
```

Nota: El método get devuelve null cuando no se encuentra la instancia buscada

También podemos usar el método load:

```
Profesor profesor=(Profesor)session.load(Profesor.class,102);
```

La diferencia con el método **get** es que si no se encuentra la instancia buscada se genera una excepción **ObjetcNotFoundException**, que por lo tanto debemos controlar en el código

Ejemplo completo 2:

```
	//Obtenemos el SessionFactory
	SessionFactory sessionFactory=HibernateUtil.getSessionFactory();

	//Abrimos la sesión mediante el SessionFactory
	Session session =sessionFactory.openSession();


	Transaction tx=session.beginTransaction();
	Profesor profesor=newProfesor();

	profesor=(Profesor)session.load(Profesor.class,101);
	System.out.println("Profesor:"+profesor.getNombre());

	tx.commit();
	session.close();
	sessionFactory.close();
```

4.5.3. Actualizar

El método a usar es ***update(Object object)***, al que le deberemos pasar el objeto a actualizar en la base de datos. Para poder actualizar un objeto será necesario haberlo obtenido previamente con **load** o con **get**

Ejemplo:

```
	//Obtenemos el SessionFactory
	SessionFactory sessionFactory=HibernateUtil.getSessionFactory();

	//Abrimos la sesión mediante el SessionFactory
	Session session =sessionFactory.openSession();


	Transaction tx=session.beginTransaction();

	Profesor profesor=(Profesor)session.get(Profesor.class,102);
```

```
        System.out.println("Profesor:"+profesor.getNombre());

        profesor.setNombre("Pedro");

        session.update(profesor);

        tx.commit();
        session.close();
        sessionFactory.close();
```

4.5.4. Borrar

Ahora pasemos a borrar un objeto desde la base de datos.El método que debemos usar es `delete(Object object)`, al que le deberemos pasar el objeto a borrar de la base de datos

```
        //Obtenemos el SessionFactory
        SessionFactory sessionFactory=HibernateUtil.getSessionFactory();

        //Abrimos la sesión mediante el SessionFactory
        Session session = sessionFactory.openSession();


        Transaction tx=session.beginTransaction();

        Profesor profesor=(Profesor)session.get(Profesor.class,102);

        System.out.println("Profesor:"+profesor.getNombre());

        session.delete(profesor);

        tx.commit();
        session.close();
        sessionFactory.close();
```

El método `delete(Object object)` simplemente borra el objeto de la base de datos.

4.5.5. Guardar o actualizar

Muchas veces resulta cómodo al programar no tener que estar pendiente de si un objeto va a insertarse o actualizarse. Para ello Hibernate dispone del método `saveOrUpdate(Object object)` que inserta o actualiza en la base de datos en función de si ya existe o no dicha fila.

```
        //Obtenemos el SessionFactory
        SessionFactory sessionFactory=HibernateUtil.getSessionFactory();

        //Abrimos la sesión mediante el SessionFactory
        Session session =sessionFactory.openSession();

        Profesor profesor=newProfesor(102,"Lucas","Perez","García");

        //Creamos el objeto
        Transaction tx=session.beginTransaction();

        session.saveOrUpdate(profesor);

        tx.commit();
        session.close();
        sessionFactory.close();
```

En este caso dependiendo de si existe o no la fila en la base de datos con `Id=102` se realizará un `UPDATE` o un `INSERT` contra la base de datos.

4.6.Claves primarias

4.6.1. Generación automática de valores

Podemos configurar el campo identificador de una clase para que se generen sus valores automáticamente.

Hay varias formas de configurarlo vamos a ver dos de ellas:

Estrategia increment

```
@Entity
@Table(name="Incidencias")
public class IncidenciaAula {

	@Id
	@GeneratedValue(generator="migenerador")
	@GenericGenerator(name="migenerador", strategy = "increment")
	@Column(name="Id")
	int id;
	String fecha;
	String descripcion;
```

En este ejemplo tenemos la clase IncidenciaAula la vamos a asociar a la tabla Incidencias como vemos en la anotación @Table

El atributo id está marcado con la anotación @Id que indica que será la clave primaria.

La anotación `@GeneratedValue` indica que el valor de este campo va a ser generado por un generador llamado "migenerador" que definimos en la línea siguiente.

Con la anotación `@GenericGenerator` definimos el generador:

Con el **atributo name** le asignamos un nombre en este caso "migenerador"

Con el **atributo strategy** definimos el tipo de generación. En este caso "**increment**". Este tipo de generador genera identificadores de tipo long, short o int que son únicos cuando solo un proceso está generando valores.

Ejemplo de uso:

```
//Abrimos la sesión mediante el SessionFactory
Session session =sessionFactory.openSession();

//Creamos el objeto
IncidenciaAula incidencia=new IncidenciaAula("2020-01-1","Retraso");

Transaction tx=session.beginTransaction();

session.save(incidencia);
tx.commit();
```

Como vemos no hace falta inicializar el atributo en el constructor ya que el id se generará de forma automática.

Estrategia uuid

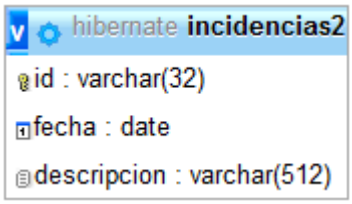
Esta estrategia genera un identificador de 128 bits utilizando un algoritmo. El valor generado como clave se representa como un String con 32 dígitos hexadecimales.

```
@Entity
@Table(name="Incidencias2")
public class IncidenciaAula2 {

	@Id
	@GeneratedValue(generator="migenerador")
	@GenericGenerator(name="migenerador", strategy = "uuid")
	@Column(name="Id")
	String id;
	String fecha;
	String descripcion;
```

Como la clave generada es una cadena el atributo id es de tipo String.

En la base de datos el campo id tendrá que ser un Varchar de 32 bytes para poder almacenar el identificador generado:



Ejemplo de uso:

```
//Abrimos la sesión mediante el SessionFactory
Session session =sessionFactory.openSession();

//Creamos el objeto
IncidenciaAula2 incidencia=new IncidenciaAula2("2020-01-11" ,"Retraso");

Transaction tx=session.beginTransaction();

session.save(incidencia);
tx.commit();
```

4.6.2. Claves primarias de más de un atributo

Hay varias estrategias para mapear claves primarias de más de un atributo vamos a ver una de ellas:

Crearemos una clase que contenga los atributos de la clave primaria y añadiremos una instancia de esta clase como clave primaria de la clase que queríamos diseñar inicialmente.

Por ejemplo supongamos que queremos crear una clase alumno en la que los atributos nombre y apellidos pueden usarse como clave primaria al no haber alumnos en los coincidan el nombre y los apellidos.

Crearemos por tanto una clase que contenga nombre y apellidos:

```java
@Embeddable
class AlumnoId implements Serializable {
  String nombre;
  String apellidos;
```

Como vemos le añadiremos a esta clase la anotación `@Embeddable` que indica que se puede incrustar en otra clase.
La clase alumno quedaría de la siguiente manera:

```java
@Entity
@Table(name="alumno")
public class Alumno {
	@EmbeddedId
	AlumnoId id;
	int edad;
```

Como vemos el atributo id del tipo AlumnoId lo anotamos con la etiqueta `@EmbeddedId` que indica que es un atributo incrustado.

Ejemplo de uso:

```java
	//Creamos el objeto
	AlumnoId miId=new AlumnoId("Andrés","Fernández");
	Alumno al=new Alumno(miId,25);
	Transaction tx=session.beginTransaction();

	session.save(al);
	tx.commit();
```

Creamos un AlumnoId y se lo pasamos al constructor de Alumno.

Si quisiéramos indicar la columna en que mapear un atributo de la clase compuesta podríamos hacerlo así:

```java
	@EmbeddedId
	@AttributeOverride(name="nombre", column=@Column(name="nombreAlumno"))
	AlumnoId id;
```

Con la anotación `@AttributeOverride` indicamos que al atributo **nombre** se va a mapear con el campo **nombreAlumno** de la tabla de la base de datos.

5. Relaciones

En este apartado vamos a explicar cómo realizar las distintas relaciones entre entidades de Hibernate.

5.1.Uno a uno (unidireccional)

La relación uno a uno en Hibernate consiste simplemente en que un objeto tenga una referencia a otro objeto de forma que al persistirse el primer objeto también se persista el segundo.

La relación va a ser unidireccional es decir que la relación *uno a uno* va a ser en un único sentido.

5.1.1. Clases Java

Antes de entrar en cómo se implemente en Hibernate , veamos las clases Java y las tablas que definen la relación uno a uno.

Para nuestro ejemplo vamos a usar las clases:

- Profesor
- Direccion

Estas dos clases van a tener una relación uno a uno.

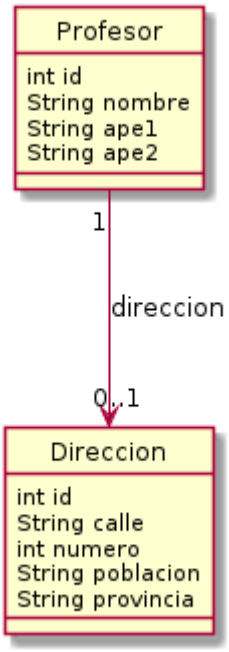
```
1:public class Profesor implements Serializable{
2:private int id;
3:private String nombre;
4:private String ape1;
5:private String ape2;
6:private Direccion direccion;
7:
8:public Profesor(){
9:}
10:
11:public Profesor(int id, String nombre, String ape1, String ape2){
12:this.id= id;
13:this.nombre= nombre;
14:this.ape1= ape1;
15:this.ape2= ape2;
16:}
17:}
18:
19:public class Direccion implements Serializable{
20:private int id;
21:private  String calle;
22:private int numero;
23:private String poblacion;
24:private String provincia;
25:
26:public Direccion(){
27:}
28:
29:public Direccion(int id, String calle, int numero, String poblacion, String
provincia){
30:this.id= id;
31:this.calle= calle;
32:this.numero= numero;
33:this.poblacion= poblacion;
34:this.provincia= provincia;
35:}
36:}
```

Listado 1.Relación 1 a 1

En el listado 1 podemos ver cómo la clase *Profesor* tiene una propiedad llamada *direccion* de la clase *Direccion* (línea 6), mientras que la clase *Direccion* no posee ninguna referencia a Profesor ya que hemos definido una direccionalidad desde *Profesor* hacia *Direccion* pero no al revés.

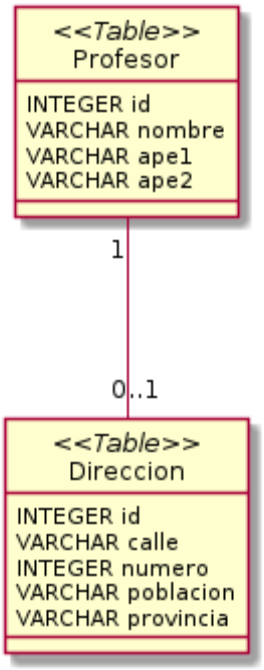
En la clases Java *Profesor* y *Direccion* no se han incluido los métodos get/set de cada propiedad para facilitar la lectura pero deben estar en la clase Java.

En el siguiente diagrama UML se ve que la relación es solo desde `Profesor` hacia `Direccion`.



5.1.2. Tablas

Las tablas de base de datos quedarían de la siguiente forma:



Podemos apreciar que en el diseño de las tablas de la base de datos ya no existe una columna en `Profesor` con la clave primaria de `Direccion` ya que si la hubiera sería una relación `muchos a uno`. Entonces ¿cómo se establece la relación entre las dos filas? Simplemente porque tanto `Profesor` como `Direccion` **deben tener** la misma clave primaria y de esa forma se establece la relación.

5.1.3. Fichero de mapeo ".hbm.xml"

Al persistir dos clases serán necesarios dos ficheros de persistencia:

- `Profesor.hbm.xml`
- `Direccion.hbm.xml`

Profesor.hbm.xml

El fichero `Profesor.hbm.xml` quedará de la siguiente forma

```
 1:<?xml version="1.0" encoding="UTF-8"?>
 2:<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
 3:<hibernate-mapping>
 4:<class name="ejemplo01.Profesor">
 5:<id column="Id" name="id"type="integer"/>
 6:<property name="nombre"/>
 7:<property name="ape1"/>
 8:<property name="ape2"/>
 9:
10:<one-to-one name="direccion" cascade="all"/>
11:
12:</class>
13:</hibernate-mapping>
```

Fichero Profesor.hbm.xml

El fichero básicamente contiene lo que se ha explicado en las lecciones anteriores excepto por el tag `<one-to-one>` de la línea 10.

Tag one-to-one

El tag `<one-to-one>` se utiliza para definir una relación *uno a uno* entre las dos clases Java. En su forma más sencilla contiene sólamente dos atributos:

- `name`:Este atributo contiene el nombre de la propiedad Java con la referencia al otro objeto con el que forma la relación *uno a uno*. En nuestro ejemplo es el atributo `direccion`.
- `cascade`: Este atributo indicará a hibernate cómo debe actuar cuando realicemos las operaciones de persistencia de guardar, borrar, leer, etc. En el ejemplo el valor debe ser `all` indicando que deberemos realizar la misma operación en `Profesor` que en `Direccion`.

Más información sobre el atributo `cascade` en Cascade

Direccion.hbm.xml

El fichero `Direccion.hbm.xml` quedará de la siguiente forma:

```
 1:<?xml version="1.0"encoding="UTF-8"?>
 2:<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
 3:<hibernate-mapping>
 4:<class name="ejemplo01.Direccion">
 5:<idcolumn="Id" name="id" type="integer"/>
 6:<property name="calle"/>
 7:<property name="numero"/>
 8:<property name="poblacion"/>
 9:<property name="provincia"/>
10:
11:</class>
12:</hibernate-mapping>
```

Fichero Direccion.hbm.xml

En el fichero `Direccion.hbm.xml` no hay ninguna información relativa a la relación *uno a uno* puesto que en nuestro ejemplo la relación *uno a uno* tiene una direccionalidad desde `Profesor` hasta `Direccion` por lo tanto `Direccion` no sabe nada sobre `Profesor` y por ello en su fichero de persistencia no hay nada relativo a dicha relación.

5.1.4. Anotaciones

Para usar notaciones deberemos modificar el código fuente de las clases Java y *no* usar los ficheros *.hbm.xml*.

El código fuente de la clase *Profesor* queda de la siguiente forma:

```
 1:import java.io.Serializable;
 2:import javax.persistence.*;
 3:
 4:
 5: @Entity
 6:@Table(name="Profesor")
 7:public class Profesor implements Serializable{
 8:
 9:@Id
10:@Column(name="Id")
11:private int id;
12:
13:@Column(name="nombre")
14:private String nombre;
15:
16:@Column(name="ape1")
17:private String ape1;
18:
19:@Column(name="ape2")
20:private String ape2;
21:
22:@OneToOne(cascade=CascadeType.ALL)
23:     @PrimaryKeyJoinColumn
24:private Direccion direccion;
25:}
```

Clase Profesor anotada

A la propiedad *direccion* (línea 24) se han añadido dos anotaciones para indicar la relación *uno a uno* y que ésta relación se implemente mediante la clave primaria.

- **@OneToOne(cascade=CascadeType.ALL)**: Esta anotación indica la relación *uno a uno* de las 2 tablas. Tambien indicamos el valor de `cascade`: `cascade`: este atributo indicará a hibernate cómo debe actuar cuando realicemos las operaciones de persistencia de guardar, borrar, leer, etc. En el ejemplo el valor debe ser `all` indicando que deberemos realizar la misma operación en `Profesor` que en `Direccion`.
- **@PrimaryKeyJoinColumn**: Indicamos que la relación entre las dos tablas se realiza mediante la clave primaria.

En caso de no incluir la anotación *@PrimaryKeyJoinColumn* se producirá un error indicando que falta la columna *direccion_Id* en la tabla *Profesor*.

Nótese que si utilizamos anotaciones en necesario usar *@PrimaryKeyJoinColumn* mientras que usando el fichero *.hbm.xml* no es necesario indicarlo.

Más información sobre el atributo *cascade* en Cascade

En código de *Direccion* no es necesario indicar nada sobre la relación tal y como hemos explicado en el caso del fichero de hibernate.

En el código de *Direccion* no es necesario indicar nada sobre la relación uno a uno puesto que en nuestro ejemplo la relación uno a uno tiene una direccionalidad desde *Profesor* hasta *Direccion* por lo tanto *Direccion* no sabe nada sobre *Profesor* y por ello en su fichero de persistencia no hay nada relativo a dicha relación.

```
 1:import java.io.Serializable;
 2:import javax.persistence.Column;
 3:import javax.persistence.Entity;
 4:import javax.persistence.Id;
 5:import javax.persistence.Table;
 6:
 7:
 8: @Entity
 9:@Table(name="Direccion")
10:public class Direccion implements Serializable{
11:
12:     @Id
13:@Column(name="Id")
14:private int id;
15:
16:@Column(name="calle")
17:private String calle;
18:
19:@Column(name="numero")
20:private int numero;
21:
22:@Column(name="poblacion")
23:private String poblacion;
24:
25:@Column(name="provincia")
26:private String provincia;
27:}
```

Clase Direccion anotada

5.1.5. Código Java

Ahora que ya tenemos preparadas las clase Java para que puedan persistirse veamos el código necesario para persistirlas.

```
 1:Direccion direccion=new Direccion(1, "Plaza del ayuntamiento", 8, "Xativa",
"Valencia");
 2:         Profesor profesor=new Profesor(1, "Juan", "Perez", "García");
 3:profesor.setDireccion(direccion);
 4:
 5:         Session session=sessionFactory.openSession();
 6:session.beginTransaction();
 7:
 8:session.save(profesor);
 9:
10:session.getTransaction().commit();
11:session.close();
```

Persistiendo la clase Profesor

Como podemos ver no hay *nada* nuevo en el código Java para persistir una relación *uno a uno*, simplemente creamos las 2 clases (Líneas 1 y 2) y establecemos la relación entre ambas asignando al objeto la referencia al objeto *Direccion* (Línea 3). Por último, simplemente persistimos la clase `Profesor` tal y como se ha explicado anteriormente.

Al ejecutar el ejemplo Hibernate vemos cómo se han creado las filas en las tablas *Profesor* y *Direccion* mientras que desde Java solo se ha persistido la clase *Profesor*.

Si vemos el log que se genera al persistir los 2 objetos, podemos ver que se realiza primero una orden *SELECT* contra la tabla *Direccion* para comprobar si ya existe la dirección en la base de datos. Esto lo realiza hibernate ya que si ya existe no es necesario insertar la fila de la dirección pero si la fila ya existe pero los datos son distintos, hibernate lanzará un *UPDATE* para modificarlos.

Recuerda que la clave primaria de `Direccion` y `Profesor` debe ser la misma para que se establezca correctamente la relación.

5.2.Uno a uno (bidireccional)

Este apartado es muy similar al anterior, pero en éste caso la relación entre las clases *Profesor* y *Direccion* va a ser bidireccional.

5.2.1. Clases Java

Antes de entrar en cómo se implemente en Hibernate, veamos las clases Java y las tablas que definen la relación *uno a uno*.

Para nuestro ejemplo vamos a usar las clases:

- `Profesor`
- `Direccion`

Estas dos clases van a tener una relación uno a uno.

```
 1:public class Profesor implements Serializable{
 2:private int id;
 3:private String nombre;
 4:private String ape1;
 5:private String ape2;
 6:private Direccion direccion;
 7:
 8:public Profesor(){
 9:}
10:
11:public Profesor(int id, String nombre, String ape1, String ape2){
12:this.id= id;
13:this.nombre= nombre;
14:this.ape1= ape1;
15:this.ape2= ape2;
16:}
17:}
18:
19:public class Direccion implementsSerializable{
20:private int id;
21:private String calle;
22:private int numero;
23:private String poblacion;
24:private String provincia;
25:private Profesor profesor;
26:
27:public Direccion(){
28:}
29:
30:public Direccion(int id, String calle, int numero, String poblacion, String
provincia){
31:this.id= id;
32:this.calle= calle;
33:this.numero= numero;
34:this.poblacion= poblacion;
35:this.provincia= provincia;
36:}
```

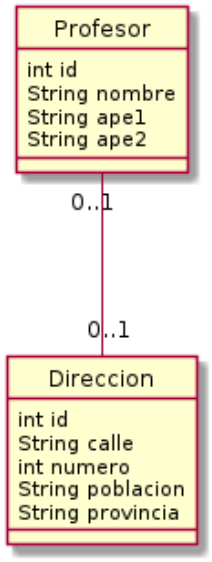
```
37: }
```

Listado 1.Relación 1 a 1

En el listado 1 podemos ver cómo la clase `Profesor` tiene una propiedad llamada `direccion` de la clase `Direccion` (línea 6) y además la clase `Direccion` también posee referencia a `Profesor` ya que hemos definido que la relación es bidireccional desde `Profesor` hacia `Direccion` y viceversa.

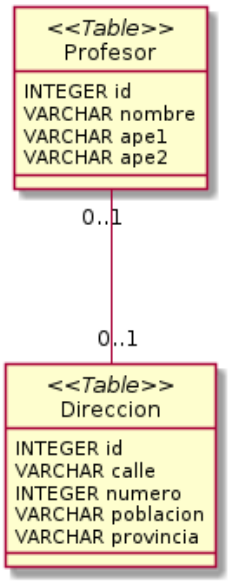
En la clases Java `Profesor` y `Direccion` no se han incluido los métodos get/set de cada propiedad para facilitar la lectura, pero deben estar en la clase Java.

En el siguiente diagrama UML se ve la relación desde `Profesor` hacia `Direccion` y viceversa.



5.2.2. Tablas

La tablas de base de datos quedarían de la siguiente forma:



Podemos apreciar que en el diseño de las tablas de la base de datos ya no existe una columna en `Profesor` con la clave primaria de `Direccion` o viceversa ya que si la hubiera sería una

relación *muchos a uno*. Entonces ¿cómo se establece la relación entre las dos filas? Simplemente porque tanto *Profesor* como *Direccion deben tener* la misma clave primaria y de esa forma se establece la relación.

5.2.3. Fichero de mapeo ".hbm.xml"

Al persistir dos clases serán necesarios dos ficheros de persistencia:

- *Profesor.hbm.xml*
- *Direccion.hbm.xml*

5.2.4. Profesor.hbm.xml

El fichero *Profesor.hbm.xml* quedará de la siguiente forma

```
 1:<?xml version="1.0" encoding="UTF-8"?>
 2:<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
 3:<hibernate-mapping>
 4:<class name="ejemplo01.Profesor">
 5:<id column="Id" name="id"type="integer"/>
 6:<property name="nombre"/>
 7:<property name="ape1"/>
 8:<property name="ape2"/>
 9:
10:<one-to-one name="direccion" cascade="all"/>
11:
12:</class>
13:</hibernate-mapping>
```

Fichero Profesor.hbm.xml

El fichero básicamente contiene lo que se ha explicado en las lecciones anteriores excepto por el tag *<one-to-one>* de la línea 10.

Tag one-to-one

El tag *<one-to-one>* se utiliza para definir una relación *uno a uno* entre las dos clases Java. En su forma más sencilla contiene sólamente dos atributos:

- **name**: Este atributo contiene el nombre de la propiedad Java con la referencia al otro objeto con el que forma la relación *uno a uno*. En nuestro ejemplo es el atributo `direccion`.
- **cascade**: Este atributo indicará a hibernate cómo debe actuar cuando realicemos las operaciones de persistencia de guardar, borrar, leer, etc. En el ejemplo el valor debe ser `all` indicando que deberemos realizar la misma operación en `Profesor` que en`Direccion`.

Más información sobre el atributo *cascade* en Cascade

5.2.5. Direccion.hbm.xml

El fichero `Direccion.hbm.xml` quedará de la siguiente forma:

```
 1:<?xml version="1.0" encoding="UTF-8"?>
 2:<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
 3:<hibernate-mapping>
 4:<class name="ejemplo01.Direccion">
 5:<id column="Id" name="id" type="integer"/>
 6:<property name="calle"/>
 7:<property name="numero"/>
 8:<property name="poblacion"/>
 9:<property name="provincia"/>
10:
11:<one-to-one name="profesor" cascade="all"/>
```

```
12:
13:</class>
14:</hibernate-mapping>
```

Fichero Direccion.hbm.xml

La clase `Direccion` también tiene la relación *uno a uno* hacia profesor.

5.2.6. Anotaciones

Para usar anotaciones deberemos modificar el código fuente de las clases Java y *no* usar los ficheros *.hbm.xml*.

El código fuente de la clase *Profesor* queda de la siguiente forma:

```
 1:import java.io.Serializable;
 2:import javax.persistence.*;
 3:
 4:
 5: @Entity
 6:@Table(name="Profesor")
 7:public class Profesor implements Serializable{
 8:
 9:      @Id
10:@Column(name="Id")
11:private int id;
12:
13:@Column(name="nombre")
14:private String nombre;
15:
16:@Column(name="ape1")
17:private String ape1;
18:
19:@Column(name="ape2")
20:private String ape2;
21:
22:@OneToOne(cascade=CascadeType.ALL)
23:      @PrimaryKeyJoinColumn
24:private Direccion direccion;
25:}
```

Clase Profesor anotada

A la propiedad *direccion* (línea 24) se han añadido dos anotaciones para indicar la relación *uno a uno* y que esta relación se implementa mediante la clave primaria.

- **@OneToOne(cascade=CascadeType.ALL)**: Esta anotación indica la relación *uno a uno* de las 2 tablas. También indicamos el valor de `cascade` al igual que en el fichero de hibernate.
- **@PrimaryKeyJoinColumn**: Indicamos que la relación entre las dos tablas se realiza mediante la clave primaria.

En caso de no incluir la anotación `@PrimaryKeyJoinColumn` se producirá un error indicando que falta la columna `direccion_Id` en la tabla `Profesor`.

Notar que si utilizamos anotaciones es necesario usar `@PrimaryKeyJoinColumn` mientras que usando el fichero `.hbm.xml` no es necesario indicarlo.

Más información sobre el atributo `cascade` en Cascade

El código de *Direccion* es similar al anterior.

```
 1:import java.io.Serializable;
 2:import javax.persistence.Column;
 3:import javax.persistence.Entity;
 4:import javax.persistence.Id;
 5:import javax.persistence.Table;
 6:
 7:
```

```
 8: @Entity
 9:@Table(name="Direccion")
10:public class Direccion implements Serializable{
11:
12:     @Id
13:@Column(name="Id")
14:private int id;
15:
16:@Column(name="calle")
17:private String calle;
18:
19:@Column(name="numero")
20:private int numero;
21:
22:@Column(name="poblacion")
23:private String poblacion;
24:
25:@Column(name="provincia")
26:private String provincia;
27:
28:@OneToOne(cascade=CascadeType.ALL)
29:     @PrimaryKeyJoinColumn
30:private Profesor profesor;
31:}
```

Clase Direccion anotada

Como en el caso anterior vemos cómo debemos incluir los tags `@OneToOne` y `@PrimaryKeyJoinColumn` (Líneas 22 y 23) para establecer la relación entre `Direccion` y `Profesor`.

5.2.7. Código Java

Ahora que ya tenemos preparadas las clase Java para que puedan persistirse veamos el código necesario para persistirlas.

```
 1: Direccion direccion1=newDireccion(3, "Calle de la sarten", 23, "Manises",
"Valencia");
 2: Profesor profesor1=newProfesor(3, "Sergio", "Mateo", "Ramis");
 3:profesor1.setDireccion(direccion1);
 4:direccion1.setProfesor(profesor1);
 5:
 6: Direccion direccion2=newDireccion(4, "Calle Luis lamarca", 45, "Torrente",
"Valencia");
 7: Profesor profesor2=newProfesor(4, "Paco", "Moreno", "Díaz");
 8:profesor2.setDireccion(direccion2);
 9:direccion2.setProfesor(profesor2);
10:
11:
12: Session session=sessionFactory.openSession();
13:session.beginTransaction();
14:
15:session.save(profesor1);
16:session.save(direccion2);
17:
18:session.getTransaction().commit();
19:session.close();
```

Persistiendo la clase Profesor

El ejemplo incluye dos casos:

- Crear un objeto `direccion1` y otro `profesor1` (líneas 1 y 2), crear las relaciones (líneas 3 y 4) y finalmente en la línea 16 guardar el objeto `profesor1`.
- Crear un objeto `direccion2` y otro `profesor2` (líneas 6 y 7), crear las relaciones (líneas 8 y 9) y finalmente en la línea 17 guardar el objeto `direccion2`.
-

En ambos casos el resultado aparente es el mismo , se guarda tanto el objeto *Direccion* como el objeto *Profesor* al ser la relación bidireccional aunque realmente los 2 casos *no* son iguales; veamos ahora el porqué.

En el primer caso, si persistimos un objeto *Profesor* se inserta directamente dicho objeto en la base de datos por lo que *no* puede existir ya la fila pero sí que se permite que la *Direccion* ya exista, actualizándose en dicho caso. Pero en el segundo caso si persistimos el objeto *Direccion* lo que ocurre es lo contrario, no podrá existir la fila de la dirección pero sí podrá existir la fila del *Profesor*.

5.3.Uno a muchos bidireccional (Usando el interfaz Set)

La relación uno a muchos consiste en que un objeto *padre* tenga una lista de otros objetos *hijo* de forma que al persistirse el objeto principal también se persista la lista de objetos *hijo*. Esta relación también suele llamarse *maestro-detalle* o *padre-hijo*.

5.3.1. Clases Java

Antes de entrar en cómo se implementa en Hibernate , veamos las clases Java y las tablas que definen la relación uno a muchos.

Para nuestro ejemplo vamos a usar las clases:

- Profesor
- CorreoElectronico

Estas dos clases van a tener una relación uno a muchos.

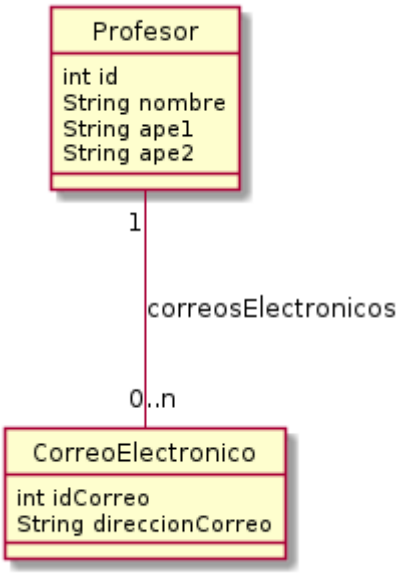
```
1:public class Profesor implements Serializable{
2:private int id;
3:private String nombre;
4:private String ape1;
5:private String ape2;
6:private Set<CorreoElectronico> correosElectronicos;
7:
8:
9:public Profesor(){
10:}
11:
12:public Profesor(int id, String nombre, String ape1, String ape2){
13:this.id= id;
14:this.nombre= nombre;
15:this.ape1= ape1;
16:this.ape2= ape2;
17:}
18:
19:}
20:
21:public class CorreoElectronico implements Serializable{
22:private int idCorreo;
23:private String direccionCorreo;
24:private Profesor profesor;
25:
26:publicCorreoElectronico(){
27:
28:}
29:
30:public CorreoElectronico(int idCorreo,String direccionCorreo,Profesor profesor){
31:this.idCorreo=idCorreo;
32:this.direccionCorreo=direccionCorreo;
33:this.profesor=profesor;
34:}
35:}
```

Listado 1.Relación 1 a n

En el listado 1 podemos ver cómo la clase `Profesor` tiene una propiedad llamada `correosElectronicos` de la clase `CorreoElectronico` (línea 6) y además la clase `CorreoElectronico` también posee referencia a `Profesor` ya que hemos definido que la relación es bidireccional desde `Profesor` hacia `Direccion` y viceversa.

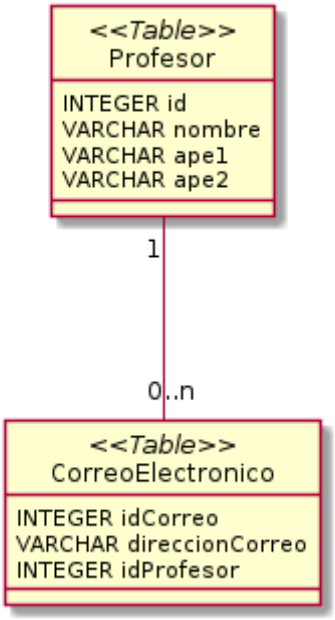
En la clases Java `Profesor` y `CorreoElectronico` no se han incluido los métodos get/set de cada propiedad para facilitar la lectura pero deben estar en la clase Java.

En el siguiente diagrama UML se ve que la relación desde `Profesor` hacía `CorreoElectronico` y viceversa.



5.3.2. Tablas

La tablas de base de datos quedarían de la siguiente forma:



Podemos ver cómo la tabla *CorreoElectronico* contiene como clave ajena la clave primaria de la tabla *Profesor* y de esa forma se establece la relación uno a muchos.

5.3.3. Fichero de mapeo ".hbm.xml"

Al persistir dos clases serán necesarios dos ficheros de persistencia:

- Profesor.hbm.xml
- CorreoElectronico.hbm.xml

5.3.4. Profesor.hbm.xml

El fichero Profesor.hbm.xml quedará de la siguiente forma:

```
 1:<?xmlversion="1.0"encoding="UTF-8"?>
 2:<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
 3:<hibernate-mapping>
 4:<classname="ejemplo05.Profesor">
 5:<idcolumn="Id"name="id"type="integer"/>
 6:<propertyname="nombre"/>
 7:<propertyname="ape1"/>
 8:<propertyname="ape2"/>
 9:
10:<setname="correosElectronicos"cascade="all"inverse="true">
11:<key>
12:<columnname="idProfesor"/>
13:</key>
14:<one-to-manyclass="ejemplo05.CorreoElectronico"/>
15:</set>
16:</class>
17:</hibernate-mapping>
```

Fichero Profesor.hbm.xml

El fichero básicamente contiene lo que se ha explicado en las lecciones anteriores excepto por el tag <set> (líneas 10 a 15).

Tag set

El tag <set> se utiliza para definir una relación *uno a muchos* desordenada entre las dos clases Java.

Atributos

- **name**: Es el nombre de la propiedad Java del tipo Set en la cual se almacenan todos los objetos *hijos*.En nuestro ejemplo el valor es correosElectronicos ya que es la propiedad que contiene el Set.
- **cascade**: Como ya hemos explicado en anteriores lecciones, este atributo indica que se realizan las mismas operaciones con el objeto *padre* que con los objetos *hijos*, es decir si uno se borra los otros también, etc. Su valor habitual es all.
- **inverse**: Este atributo se utiliza para minimizar las SQLs que lanza Hibernate contra la base de datos.En este caso concreto debe establecerse a true para evitar una sentencia SQL de UPDATE por cada *hijo*. Al final de esta sesión se indican enlaces donde intentan explicar (desde mi punto de vista con poco éxito) el funcionamiento del atributo inverse.

Mas información sobre el atributo cascade en Cascade

La documentación de hibernate no ayuda mucho a entender el atributo `inverse`.Se muestran a continuación algunas explicaciones que se dan sobre ello:

Marks this collection as the "inverse" end of a bidirectional association.

For you, and for Java, a bi-directional link is simply a matter of setting the references on both sides correctly. Hibernate, however, does not have enough information to correctly arrange SQL INSERT and UPDATE statements (to avoid constraint violations). Making one side of the association inverse tells Hibernate to consider it a mirror of the other side. That is all that is necessary for Hibernate to resolve any issues that arise when transforming a directional navigation model to a SQL database schema. The rules are straightforward: all bi-directional associations need one side as inverse. In a one-to-many association it has to be the many-side, and in many-to-many association you can select either side.

Tags anidados

- **key** Este tag contiene otro anidado llamado `column` con el atributo `name` que indica el nombre de una columna de la base de datos. Esta columna debe ser de la tabla *hijo* y ser el nombre de la clave ajena a la tabla *padre*. En nuestro ejemplo es `idProfesor` ya que es el nombre de la clave ajena en la tabla `CorreoElectronico`.
- **one-to-many** Este tag contiene el atributo `class` con el FQCN de la clase Java *hija*. En nuestro ejemplo es el nombre de la clase `CorreoElectronico` cuyo FQCN es `ejemplo05.CorreoElectronico`.

Podemos pensar que el valor del atributo `class` en el tag `one-to-many` sea opcional ya que hibernate podría ser capaz de deducirlo como ha hecho en otras ocasiones, sin embargo no es así. En caso de no indicarlo se producirá la siguiente excepción:

```
org.hibernate.MappingException: Association references unmapped class: null
```

5.3.5. CorreoElectronico.hbm.xml

El fichero `CorreoElectronico.hbm.xml` quedará de la siguiente forma:

```
 1:<?xmlversion="1.0"encoding="UTF-8"?>
 2:<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
 3:<hibernate-mapping>
 4:<classname="ejemplo05.CorreoElectronico">
 5:<idcolumn="IdCorreo"name="idCorreo"type="integer"/>
 6:<propertyname="direccionCorreo"/>
 7:
 8:
 9:<many-to-onename="profesor">
10:<columnname="idProfesor"/>
11:</many-to-one>
12:
13:
14:</class>
15:</hibernate-mapping>
```

Fichero CorreoElectronico.hbm.xml

El fichero básicamente contiene lo que se ha explicado en las lecciones anteriores excepto por el tag `<many-to-one>` (líneas 9 a 11).

Tag many-to-one

El tag `<many-to-one>` se utiliza para definir una relación *muchos a uno* entre las dos clases Java.

Atributos:

name: Es el nombre de la propiedad Java que enlaza con el objeto *padre*. En nuestro ejemplo el valor es `profesor` ya que es la propiedad que contiene la referencia a la clase `Profesor`.

Tags anidados:

- **column** Este tag contiene el atributo `name` que indica el nombre de una columna de la base de datos. Esta columna debe ser de la tabla *hijo* y ser el nombre de la clave ajena a la tabla *padre*. En nuestro ejemplo es `idProfesor` ya que es el nombre de la clave ajena en la tabla `CorreoElectronico`.

5.3.6. Anotaciones

Para usar notaciones deberemos modificar el código fuente de las clases Java y no usar los ficheros *`.hbm.xml`*.

El código fuente de la clase *`Profesor`* queda de la siguiente forma:

```
 1: @Entity
 2:@Table(name="Profesor")
 3:public class Profesor implements Serializable{
 4:
 5:     @Id
 6:@Column(name="Id")
 7:private int id;
 8:
 9:@Column(name="nombre")
10:private String nombre;
11:
12:@Column(name="ape1")
13:private String ape1;
14:
15:@Column(name="ape2")
16:private String ape2;
17:
18:@OneToMany(mappedBy="profesor",cascade= CascadeType.ALL)
19:private Set<CorreoElectronico> correosElectronicos;
20:
21:
22:public Profesor(){
23:}
24:
25:public Profesor(int id, String nombre, String ape1, String ape2){
26:this.id= id;
27:this.nombre= nombre;
28:this.ape1= ape1;
29:this.ape2= ape2;
30:}
31:}
```

Clase Profesor anotada

A la propiedad correosElectronicos se ha añadido una anotación para indicar la relación uno a muchos.

- **OneToMany**:Como su nombre indica le dice a Hibernate que esta propiedad contendrá la lista de *hijos*.
  - **mappedBy**: Este atributo contendrá el nombre de la propiedad Java de la clase *hija* que enlaza con la clase *padre*. En nuestro ejemplo es el nombre de la propiedad `profesor` que se encuentra en la clase `CorreoElectronico`.
  - **cascade**: Este atributo tiene el mismo significado que el del fichero de mapeo de Hibernate. Mas información en Cascade.

El código de la clase CorreoElectronico es el siguiente:

```
 1: @Entity
 2:@Table(name="CorreoElectronico")
 3:public class CorreoElectronico implements Serializable{
 4:
 5:     @Id
```

```
 6:@Column(name="IdCorreo")
 7:private int idCorreo;
 8:
 9:@Column(name="DireccionCorreo")
10:private String direccionCorreo;
11:
12:     @ManyToOne
13:@JoinColumn(name="IdProfesor")
14:private Profesor profesor;
15:
16:public CorreoElectronico(){
17:
18:}
19:
20:public CorreoElectronico(int idCorreo,String direccionCorreo,Profesor profesor){
21:this.idCorreo=idCorreo;
22:this.direccionCorreo=direccionCorreo;
23:this.profesor=profesor;
24:}
25:}
```

Clase CorreoElectronico anotada

A la propiedad *profesor* se han añadido dos anotaciones para indicar la relación:

- **ManyToOne**:Al ser el otro lado de la relación indicamos que desde este lado es una relación *muchos a uno*.
- **JoinColumn**: Indicaremos el nombre de la columna que en la tabla *hija* contiene la clave ajena a la tabla *padre*. En nuestro ejemplo es la columna de la base de datos `IdProfesor` que se encuentra en la tabla `CorreoElectronico` la cual enlaza con la tabla `Profesor`.

5.3.7. Código Java

Ahora que ya tenemos preparadas las clase Java para que puedan persistirse veamos el código necesario para persistirlas.

```
 1: Profesor profesor=newProfesor(7, "Sara", "Barrrera", "Salas");
 2: Set<CorreoElectronico> correosElectronicos=new HashSet<>();
 3:correosElectronicos.add(new                          CorreoElectronico(3,
"sara@yahoo.com",profesor));
 4:correosElectronicos.add(new                          CorreoElectronico(2,
"sara@hotmail.com",profesor));
 5:correosElectronicos.add(new                          CorreoElectronico(1,
"sara@gmail.com",profesor));
 6:
 7:profesor.setCorreosElectronicos(correosElectronicos);
 8:
 9: Session session=sessionFactory.openSession();
10:session.beginTransaction();
11:
12:session.save(profesor);
13:
14:
15:session.getTransaction().commit();
16:session.close();
```

Persistiendo la clase Profesor

La explicación del código es la siguiente:

- En la línea 1 se crea el objeto `Profesor`

- En la segunda línea se crea el objeto `HashSet` que implementa el interfaz `Set` el cual contendrá la lista de hijos.
- Desde las líneas 3 a la 5 se crean los objetos `CorreoElectronico` y se añaden al `Set`.
- En la línea 7 se establece la relación entre la lista de hijos (`CorreoElectronico`) y el padre (`Profesor`).
- En la línea 12 se guarda el objeto `Profesor` y automáticamente se guardan también sus hijos.

5.4.Uno a muchos bidireccional (Usando el interfaz List)

Usando la mismas anotaciones que en caso anterior podemos implementar la relación uno a muchos simplemente cambiando el interfaz Set por el interfaz List.

5.5. Recomendaciones para elegir Set o List en relaciones uno a muchos

Si la versión de Hibernate que se va a usar es anterior a la 5.0.8 se recomienda usar siempre Set ya que existe un bug al usar List.

Para versiones posteriores Se recomienda el uso de la interfaz List al proporcionar un mejor rendimiento.

5.6.Muchos a muchos

La relación muchos a muchos consiste en que un objeto A tenga una lista de otros objetos B y también que el objeto B a su vez tenga la lista de objetos A.De forma que al persistirse cualquier objeto también se persista la lista de objetos que posee.

5.6.1. Clases Java

Antes de entrar en cómo se implementa en Hibernate, veamos las clases Java y las tablas que definen la relación uno a muchos.

Para nuestro ejemplo vamos a usar las clases:

- Profesor
- Modulo

Estas dos clases van a tener una relación muchos a muchos.

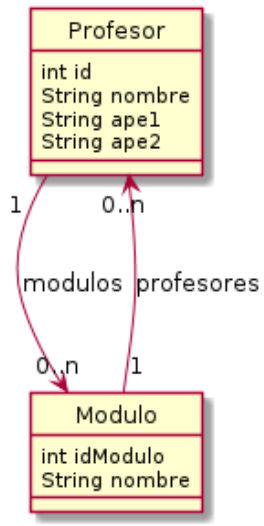
```
1: public class Profesor implements Serializable  {
 2:     private int id;
 3:     private String nombre;
 4:     private String ape1;
 5:     private String ape2;
 6:     private Set<Modulo> modulos=new HashSet();
 7:
 8:
 9:     public Profesor(){
10:     }
11:
12:     public Profesor(int id, String nombre, String ape1, String ape2) {
13:         this.id = id;
14:         this.nombre = nombre;
15:         this.ape1 = ape1;
16:         this.ape2 = ape2;
17:     }
```

```
18:
19: }
20:
21: public class Modulo implements Serializable {
22:     private int idModulo;
23:     private String nombre;
24:     private Set<Profesor> profesores=new HashSet();
25:
26:     public Modulo() {
27:
28:     }
29:
30:     public Modulo(int idModulo, String nombre) {
31:         this.idModulo = idModulo;
32:         this.nombre = nombre;
33:
34:     }
35: }
```

En el código anterior podemos ver cómo la clase Profesor tiene una propiedad de tipo Set llamada modulos de la clase Modulo (línea 6) y además la clase Modulo también posee un Set de objetos Profesor (línea 24).

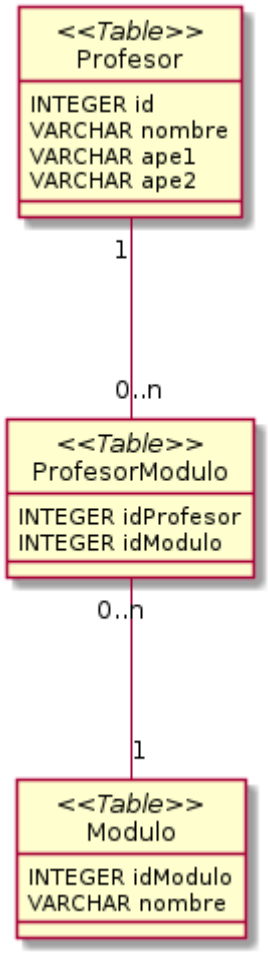
El mecanismo que usamos en Java para almacenar la lista de objetos es el interfaz Set. No vamos a usar el interfaz List o un array ya que dichas formas implican un orden de los objetos mientras que usando un Set no hay ningún tipo de orden.

En el siguiente diagrama UML se ve la relación entre Profesor y Modulo.



5.6.2. Tablas

Podemos ver cómo en este caso las tablas Profesor y Modulo se relacionan mediante la nueva tabla ProfesorModulo que contiene las claves primarias de ambas tablas.



5.6.3. Anotaciones

El código fuente de la clase Profesor queda de la siguiente forma:

```
1: @Entity
2: @Table(name="Profesor")
3: public class Profesor implements Serializable  {
4:
5:     @Id
6:     @Column(name="Id")
7:     private int id;
8:
9:     @Column(name="nombre")
10:    private String nombre;
11:
12:    @Column(name="ape1")
13:    private String ape1;
14:
15:    @Column(name="ape2")
16:    private String ape2;
17:
18:    @ManyToMany(cascade = {CascadeType.ALL})
19:                                          @JoinTable(name="ProfesorModulo",
joinColumns={@JoinColumn(name="IdProfesor")},
inverseJoinColumns={@JoinColumn(name="IdModulo")})
20:    private Set<Modulo> modulos=new HashSet();
21:
22:
23:    public Profesor(){
24:    }
25:
26:    public Profesor(int id, String nombre, String ape1, String ape2) {
27:        this.id = id;
28:        this.nombre = nombre;
```

```
29:         this.ape1 = ape1;
30:         this.ape2 = ape2;
31:     }
32: }
```

A la propiedad módulos (línea 20) se han añadido dos anotaciones para indicar la relación muchos a muchos.

- **ManyToMany**: Como su nombre indica le dice a Hibernate que la propiedad contendrá una lista de objetos que participa en una relación muchos a muchos.
    - **cascade**: Este atributo tiene el mismo significado que el del fichero de mapeo de Hibernate. Más información en Cascade.
- **JoinTable**: Esta anotación contiene la información sobre la tabla que realiza la relación muchos a muchos
    - **name**: Nombre de la tabla que realiza la relación muchos a muchos. En nuestro ejemplo es ProfesorModulo.
    - **joinColumns**: Contiene cada una de las columnas que forman la clave primaria de esta clase que estamos definiendo. Cada columna se indica mediante una anotación **@JoinColumn** y en el atributo name contiene el nombre de la columna.
    - **inverseJoinColumns**: Contiene cada una de las columnas que forman la clave primaria de la clase clase con la que tenemos la relación. Cada columna se indica mediante una anotación **@JoinColumn** y en el atributo name contiene el nombre de la columna.

El código de la clase Modulo es el siguiente:

```
1: @Entity
2: @Table(name="Modulo")
3: public class Modulo implements Serializable {
4:
5:     @Id
6:     @Column(name="IdModulo")
7:     private int idModulo;
8:
9:     @Column(name="nombre")
10:    private String nombre;
11:
12:    @ManyToMany(cascade = {CascadeType.ALL},mappedBy="modulos")
13:    private Set<Profesor> profesores=new HashSet();
14:
15:    public Modulo() {
16:
17:    }
18:
19:    public Modulo(int idModulo, String nombre) {
20:        this.idModulo = idModulo;
21:        this.nombre = nombre;
22:
23:    }
24: }
```

A la propiedad profesores (línea 13) se han añadido dos anotaciones para indicar la relación muchos a muchos.

- **ManyToMany**:Indica que la propiedad contiene una lista de objetos que participan en una relación muchos a muchos.

- **cascade**: Este atributo tiene el mismo significado que el del fichero de mapeo de Hibernate. Mas información en Cascade.
- **mappedBy**: Contiene el nombre de la propiedad Java de la otra clase desde la cual se relaciona con ésta. En nuestro ejemplo es la propiedad modulos.

Al poner el atributo `mappedBy` ya no es necesario incluir la anotación `@JoinTable` ya que dicha información ya se indica en el otro lado de la relación.

5.6.4. Código Java

Ahora que ya tenemos preparadas las clase Java para que puedan persistirse veamos el código necesario para persistirlas.

```
1: Profesor profesor1=new Profesor(11, "Isabel", "Fuertes", "Gascón");
2: Profesor profesor2=new Profesor(12, "Jose", "Valenciano", "Gimeno");
3:
4: Modulo modulo1=new Modulo(1, "Sistemas Operativos en Red");
5: Modulo modulo2=new Modulo(2, "Entornos de desarrollo");
6: Modulo modulo3=new Modulo(3, "Sistemas Informáticos");
7:
8: profesor1.getModulos().add(modulo1);
9: profesor1.getModulos().add(modulo2);
10: profesor2.getModulos().add(modulo3);
11:
12: modulo1.getProfesores().add(profesor1);
13: modulo2.getProfesores().add(profesor1);
14: modulo3.getProfesores().add(profesor2);
15:
16:
17: Session session=sessionFactory.openSession();
18: session.beginTransaction();
19:
20: session.save(profesor1);
21: session.save(profesor2);
22:
23: session.getTransaction().commit();
24: session.close();
```

La explicación del código es la siguiente:

- En las líneas 1 y 2 se crean dos objetos Profesor
- En las líneas 4, 5 y 6 se crean tres objetos Modulo.
- De las líneas 8 a 10 se añaden los módulos a los profesores.
- De las líneas 12 a 14 se añaden los profesores a los módulos.
- En las líneas 20 y 21 se guardan los dos objetos Profesor y automáticamente se guardan también los módulos.

Apreciar cómo el código Java sigue siendo sencillo y no se complica prácticamente nada al guardarlo en la base de datos. Sólo estamos añadiendo la complejidad en el ficheros de mapeo de Hibernate o en las anotaciones.

5.7. Recomendación sobre que interfaz usar para implementar relaciones muchos a muchos

Para las relaciones muchos a muchos se recomienda el uso del interfaz Set ya que presenta un mejor rendimiento.

6. Clase Query

Hasta ahora nos hemos dedicado a ver las diversas formas de persistencia que soporta hibernate en función de nuestro modelo de objetos de negocio en Java. Pero una característica fundamental de cualquier ORM es la necesidad de leer dichos objetos de la base de datos.

Hibernate tiene el objeto *Query* que nos da acceso a todas las funcionalidades para poder leer objetos desde la base de datos. Veamos ahora un sencillo ejemplo de cómo funciona y posteriormente explicaremos más funcionalidades de la clase *Query*.

```
1: Query<Profesor> query =session.createQuery("SELECT p FROM Profesor p",
Profesor.class);
2: List<Profesor> profesores =query.list();
3:for(Profesor profesor : profesores){
4:System.out.println(profesor.getNombre());
5:}
```

Lanzar una consulta con Hibernate es bastante simple. Usando la *session* llamamos al método *createQuery(String queryString)* con la consulta en formato HQL y nos retorna un objeto *Query* (Línea 1). Después, sobre el objeto *Query* llamamos al método *list()* que nos retorna una lista de los objetos que ha retornado (Línea 2).

Por último en las líneas de la 3 a la 5 podemos ver cómo usar la lista de objetos *Profesor* aunque este código ya es simplemente el uso de la clase *java.util.List* que no tiene nada que ver con Hibernate.

Ejemplo completo:

```
      //Obtenemos el SessionFactory
      SessionFactory sessionFactory=HibernateUtil.getSessionFactory();

      //Abrimos la sesión mediante el SessionFactory
      Session session =sessionFactory.openSession();

      Query<Profesor> query =session.createQuery("SELECT p FROM Profesor
p",Profesor.class);
      List<Profesor> profesores =query.list();
      for(Profesor profesor : profesores){
      System.out.println(profesor.getNombre());
      }

      session.close();
      sessionFactory.close();
```

6.1.Listas de array de objetos

En las consultas se pueden devolver sólo algunos atributos en vez de clases completas-

```
SELECT p.id,p.nombreFROM Profesor p
```

En la consulta podemos ver cómo en vez de retornar un objeto `Profesor` se retorna únicamente el código del profesor y su nombre.

En estos casos el método `list()` retorna una lista con una Array de objetos con tantos elementos como propiedades hayamos puesto en la `SELECT`.

Veamos ahora un ejemplo.

```
1: Query query =session.createQuery("SELECT p.id,p.nombre FROM Profesor p");
2: List<Object[]> listDatos = query.list();
3:for(Object[]datos : listDatos){
4:System.out.println(datos[0]+"--"+ datos[1]);
5:}
```

Consulta con datos escalares

- En la línea 1 vemos cómo se crea el objeto `Query` con la consulta de datos escalares.
- En la línea 2 se ve que el método `list()` retorna una lista de array de Objetos.Es decir 'List<Object[]>'.
- En la línea 4 se inicia el bucle para recorrer cada una de las filas de datos escalares.
- En la línea 5 finalmente se accede a los 2 datos de cada fila mediante `datos[0]` y `datos[1]`.

Lista de Objeto

6.2.Lista de Objetos

Hay otro caso cuando hay una única columna en el `SELECT` de datos escalares. Es ese caso, como el array a retornar dentro de la lista solo tendría un elemento , no se retorna una lista de arrays *`List<Object[]>`* sino únicamente una lista de elementos *`List<Object>`*.

Si modificamos la anterior consulta de forma que sólo se retorne el nombre, el código quedará de la siguiente forma:

```
1: Query query =session.createQuery("SELECT p.nombre FROM Profesor p");
2: List<Object> listDatos =query.list();
3:for(Object datos : listDatos){
4:System.out.println(datos);
5:}
```

Consulta con sólo un único dato escalar

- En la línea 1 ahora la consulta sólo tiene un único dato escalar.
- En la línea 2 el método `list()` ya no retorna un `List<Object[]>` sino un `List<Object>`.
- En la línea 4 se inicia el bucle para recorrer cada una de las filas de datos escalares pero ahora el tipo es `Object` en vez de `Object[]`.
- En la línea 5 finalmente se accede al dato sin el índice del array ya que ha dejado de serlo.

6.3.uniqueResult()

En muchas ocasiones una consulta únicamente retornará cero o un resultado. En ese caso es poco práctico que nos retorne una lista con un único elemento. Para facilitarnos dicha tarea Hibernate dispone del método *`uniqueResult()`*.

Este método retornará directamente el único objeto que ha obtenido la consulta. En caso de que no encuentre ninguno se retornará `null`.

```
1: Profesor profesor =(Profesor)session.createQuery("SELECT p FROM Profesor p WHERE
id=1001").uniqueResult();
2:System.out.println("Profesor con Id 101="+ profesor.getNombre());
```

uniqueResult()

Vemos cómo, gracias al método *`uniqueResult()`* , se simplifica el código aunque siempre se debe comprobar si ha retornado o no *`null`*.

Al igual que ocurre con `list()` el método `uniqueResult()` puede retornar tanto un objeto de una entidad , un array de objetos escalares `Object[]` o un único objeto escalar `Object`.

Si el método *`uniqueResult()`* retorna más de un resultado se producirá la excepción:

```
org.hibernate.NonUniqueResultException: query did not return a unique result
```

Ejemplo completo:

```
      //Obtenemos el SessionFactory
      SessionFactory sessionFactory=HibernateUtil.getSessionFactory();

      //Abrimos la sesión mediante el SessionFactory
      Session session =sessionFactory.openSession();

      Query query =session.createQuery("SELECT p FROM Profesor p WHERE
id=1001");
      Profesor profesor =(Profesor)query.uniqueResult();
      if(profesor!=null)
            System.out.println("Profesor con Id 101="+
profesor.getNombre());
      else
            System.out.println("Error: no existe el profesor buscado");

      session.close();
      sessionFactory.close();
```

6.4.Paginación

La paginación es parte fundamental de cualquier aplicación ya que una consulta puede tener miles de resultados y no queremos mostrarlos todos a la vez.

Para conseguir paginar el resultado de una consulta la clase `Query` dispone de los siguientes métodos:

- `setMaxResults(int maxResults)`: Establece el nº máximo de objetos que van a retornarse.
- `setFirstResult(int firstResult)`: Establece el primer de los objetos que se van a retornar.

Al realizar la paginación son necesarios al menos 2 valores:

- El tamaño de la página
- El nº de la página a mostrar

Con esos 2 valores hemos creado el siguiente código Java que muestra una única página en función del `tamanyoPagina` y `paginaAMostrar`.

```
 1:int tamanyoPagina =10;
 2:int paginaAMostrar =7;
 3:
 4: Query query =session.createQuery("SELECT p FROM Profesor p Order By p.id");
 5:query.setMaxResults(tamanyoPagina);
 6:query.setFirstResult(paginaAMostrar * tamanyoPagina);
 7: List<Profesor> profesores =query.list();
 8:
 9:for(Profesor profesor : profesores){
10:System.out.println(profesor.toString());
11:}
```

Mostrar el contenido de una única página

- Las líneas 1 y 2 establecen los valores necesarios para poder mostrar la página, que son el tamaño de la página y el nº de la página a mostrar.
- En la línea 4 se crea la *Query*.
- En la línea 5 se llama al método `setMaxResults(int maxResults)` para indicar que sólo se retornen como máximo tantos objetos como tamaño tiene la página.
- En la línea 6 se llama al método `setFirstResult(int firstResult)` para indicarle cuál es el primer objeto a retornar. Este valor coincidirá con el primer objeto de la página que se quiere mostrar. Para calcular dicho valor se multiplica el nº de la página a mostrar por el tamaño de página. Para que esta fórmula funcione el número de página debe empezar por 0, es decir, que la primera página deberá ser la nº 0, la segunda la nº 1, y así sucesivamente.
- Por fín, en la línea 7 se obtienen únicamente los resultados de la página solicitada.
- Por último en las líneas 9,10 y 11 se muestran los datos.

Ejemplo completo:

```
      //Obtenemos el SessionFactory
      SessionFactory sessionFactory=HibernateUtil.getSessionFactory();

      //Abrimos la sesión mediante el SessionFactory
      Session session =sessionFactory.openSession();

      int tamanyoPagina =10;
      int paginaAMostrar =0;

       Query query =session.createQuery("SELECT p FROM Profesor p Order By
p.id");
      query.setMaxResults(tamanyoPagina);
      query.setFirstResult(paginaAMostrar * tamanyoPagina);
       List<Profesor> profesores =query.list();

      for(Profesor profesor : profesores){
      System.out.println(profesor.getNombre());
      }

      session.close();
      sessionFactory.close();
```

El mayor problema que tiene la paginación es determinar el nº de páginas para poder mostrárselo al usuario, para saber el nº de páginas es necesario conocer el nº de objetos que retorna la consulta y la forma más rápida y sencilla es mediante una consulta que las cuente.

El siguiente código Java calcula el nº de páginas de la consulta.

```
1:long  numTotalObjetos  =(Long)session.createQuery("SELECT  count(*)  FROM
Profesor p").uniqueResult();
2:int           numPaginas           =(int)Math.ceil((double)numTotalObjetos
/(double)tamanyoPagina);
```

- En la línea 1 realizamos la consulta de `count(*^)` para obtener el nº de objetos que retorna la consulta
- En la línea 2 de divide el nº total de objetos entre el tamaño de la página obteniéndose el nº total de páginas.

Al hacer la división para calcular el nº de páginas es necesario hacer el cast de los 2 valores a `double` ya que si no Java automáticamente redondea el resultado a un valor entero con lo que el valor que se le pasa a `ceil` ya no será el correcto.

6.5.Consultas con nombre

En cualquier libro sobre arquitectura del software siempre se indica que las consultas a la base de datos no deberían escribirse directamente en el código sino que deberían estar en un fichero externo para que puedan modificarse fácilmente.

Hibernate provee una funcionalidad para hacer ésto mismo de una forma sencilla. En cualquier fichero de mapeo de Hibernate se puede incluir el tag `<query>` con la consulta HQL que deseamos lanzar.

En el siguiente ejemplo podemos ver cómo se ha definido una query en el fichero `Profesor.hbm.xml`.

```
 1:<?xmlversion="1.0"encoding="UTF-8"?>
 2:<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
 3:<hibernate-mapping>
 4:<classname="ejemplo01.Profesor">
 5:<idcolumn="Id"name="id"type="integer"/>
 6:<propertyname="nombre"/>
 7:<propertyname="ape1"/>
 8:<propertyname="ape2"/>
 9:</class>
10:
11:
12:<queryname="findAllProfesores"><![CDATA[
13:         SELECT p FROM Profesor p
14:     ]]></query>
15:</hibernate-mapping>
```

query con nombre "findAllProfesores"

Vemos cómo en las líneas 12, 13, y 14 se ha definido la consulta llamada `findAllMunicipios` mediante el tag `<query>`.

6.5.1. Tag <query>

Este tag tiene los siguientes datos:

- `name`:Este atributo define el nombre de la consulta.Es el nombre que posteriormente usaremos desde el código Java para acceder a la consulta.
- *contenido*:El contenido del tag `<query>` es la consulta en formato HQL que ejecutará Hibernate.

Hacer notar que la consulta se ha incluido dentro de la instrucción `CDATA` de XML para evitar que algun simbolo de ”>” o ”<” de la consulta se pueda interpretar como cierre del tag `<query>`.

`CDATA` forma parte de la especificación de los ficheros XML no siendo algo que se ha definido en Hibernate.Más información en CDATA.

6.5.2. Código Java

Para hacer uso de una consulta con nombre usaremos el método *getNamedQuery(String queryString)* en vez de *createQuery(String queryString)* para obtener el objeto `Query`. Por lo tanto sólo se ha modificado la línea 1 y el resto del código queda exactamente igual.

```
1: Query query =session.getNamedQuery("findAllProfesores");
2: List<Profesor> profesores =query.list();
3:for(Profesor profesor : profesores){
4:System.out.println(profesor.toString());
5:}
```

Uso de una query con nombre

Como podemos ver el uso de consultas con nombre es muy sencillo al usar Hibernate.

7. Lenguaje HQL

El Hibernate Query Languaje (HQL) es el lenguaje de consultas que usa Hibernate para obtener los objetos desde la base de datos. Su principal particularidad es que las consultas se realizan sobre los objetos java que forman nuestro modelo de negocio, es decir, las entidades que se persisten en Hibernate. Ésto hace que HQL tenga las siguientes características:

- Los tipos de datos son los de Java.
- Las consultas son independientes del lenguaje de SQL específico de la base de datos
- Las consultas son independientes del modelo de tablas de la base de datos.
- Es posible tratar con las colecciones de Java.
- Es posible navegar entre los distintos objetos en la propia consulta.

Vuelvo a insistir sobre al apartado anterior. En Hibernate las consultas HQL se lanzan (o se ejecutan) sobre el modelo de entidades que hemos definido en Hibernate, esto es, sobre nuestras clases de negocio.
De forma poco ortodoxa se podría ver cómo que nuestro *modelo de tablas* en HQL son las clases Java y **NO** las tablas de la base de datos. Es decir que cuando hagamos "`SELECT columna FROM nombreTabla`" , el "`nombreTabla`" será una clase Java y "`columna`" será una propiedad Java de dicha clase y **nunca** una tabla de la base de datos ni una columna de una tabla.

7.1.Modelo para los ejemplos

Para los ejemplos que iremos viendo a continuación, usaremos la siguiente base de datos:

Y las clases java asociadas este modelo de tablas son:

Clase Equipos.java:

```
publicclass Equipos implements java.io.Serializable {

	private String nombre;
	private String ciudad;
	private String conferencia;
	private String division;
	private Set<Jugadores> jugadoreses =new HashSet<Jugadores>(0);
	private Set<Partidos> partidosesForEquipoVisitante =new
HashSet<Partidos>(0);
	private Set<Partidos> partidosesForEquipoLocal =new
HashSet<Partidos>(0);

	public Equipos(){
	}

	public Equipos(String nombre){
		this.nombre = nombre;
	}
```

Clase Jugadores.java:

```
Public class Jugadores implements java.io.Serializable {

	privateint codigo;
	private Equipos equipos;
	private String nombre;
	private String procedencia;
	private String altura;
	private Integer peso;
	private String posicion;
	private Set<Estadisticas> estadisticases =new
HashSet<Estadisticas>(0);

	public Jugadores(){
	}
```

Clase Partidos.java:

```
publicclass Partidos implements java.io.Serializable {

	privateint codigo;
	private Equipos equiposByEquipoVisitante;
	private Equipos equiposByEquipoLocal;
	private Integer puntosLocal;
	private Integer puntosVisitante;
	private String temporada;

	public Partidos(){
	}
```

Clase Estadisticas.java:

```
publicclass Estadisticas implements java.io.Serializable {

	private EstadisticasId id;
	private Jugadores jugadores;
	private Float puntosPorPartido;
	private Float asistenciasPorPartido;
	private Float taponesPorPartido;
	private Float rebotesPorPartido;

	public Estadisticas(){
	}
```

Clase EstadisticasId:

```
publicclass EstadisticasId implements java.io.Serializable {

	private String temporada;
	privateint jugador;

	public EstadisticasId(){
	}
```

Ejemplo sencillo:

```
SELECT e FROM Equipos e ORDERBY nombre
```

¿Qué diferencias podemos ver entre HQL y SQL?

- `Equipos` hace referencia a la clase Java `Equipos` y **NO** a la tabla `equipos`.
- Es necesario definir el alias `e` de la clase Java `Equipos`.
- Tras la palabra `SELECT` se usa el alias en vez del "`*`".
- Al ordenar los objetos se usa la propiedad `nombre` de la clase `Equipos` en vez de la columna `nombre` de la tabla `equipos`.

Recuerda incluir el alias en la consulta HQL. Si no se hace y se deja la consulta de la siguiente forma:

```
SELECT Equipos FROM Equipos
```

se producirá la siguiente excepción:

```
java.lang.NullPointerException
```

Hibernate soporta **no** incluir la parte del `SELECT` en la consulta HQL, quedando entonces la consulta de la siguiente forma:

```
FROM Equipos
```

pero en la propia documentación se recomienda no hacerlo [1)] ya que de esa forma se mejora la portabilidad en caso de usar el lenguaje de consultas de JPA [2)].
Se ha hecho mención de esta característica ya que en muchos tutoriales que se encuentran por Internet se hace uso de ella.

Código completo:

```
	publicstaticvoid consultaSencilla(){
	//Obtenemos el SessionFactory
	SessionFactory sessionFactory=HibernateUtil.getSessionFactory();

	//Abrimos la sesión mediante el SessionFactory
	Session session = sessionFactory.openSession();

	Query<Equipos> query = session.createQuery("SELECT Equipos FROM
Equipos",primero.Equipos.class);
	List<Equipos> equipos = query.list();
	for(Equipos equipo : equipos){
		System.out.println(equipo.getNombre());
	}

	session.close();
	sessionFactory.close();
	}
```

7.2.Mayusculas

Respecto a la sensibilidad de las mayúsculas y minúsculas, el lenguaje HQL sí que lo es, pero con matices.

- Las palabras clave del lenguaje **NO** son sensibles a las mayúsculas o minúsculas.
  - Las siguientes 2 consultas son equivalentes.

```
select count(*) from Equipos
SELECT COUNT(*) FROM Equipos
```

- El nombre de las clases Java y sus propiedades **SI** son sensibles a las mayúsculas o minúsculas.
  - La siguiente consulta HQL es correcta

```
SELECT e.nombre FROM Equipos e WHERE nombre='Dallas Mavericks'
```

  - La siguiente consulta HQL es **errónea** ya que la propiedad `nombre` está escrita con la "N" en mayúsculas.

```
SELECT e.Nombre FROM Equipos e WHERE Nombre='Dallas Mavericks'
```

  - La siguiente consulta HQL es **errónea** ya que el nombre de la clase Java `Equipos` está escrita con la "e" en minúsculas.

```
SELECT c.nombre FROM equipos e WHERE nombre='Dallas Mavericks'
```

- Al realizar comparaciones con los valores de las propiedades, éstas **NO** son sensibles a las mayúsculas o minúsculas.
  - Las siguientes 2 consultas retornan los mismos objetos

```
SELECT e.nombre FROM Equipos e WHERE nombre='Dallas Mavericks'
SELECT e.nombre FROM Equipos e WHERE nombre='DALLAS MAVERICKS'
```

Ejemplo:

```
      //Obtenemos el SessionFactory
      SessionFactory sessionFactory=HibernateUtil.getSessionFactory();

      //Abrimos la sesión mediante el SessionFactory
      Session session = sessionFactory.openSession();

      Object obj = session.createQuery("select count(*) from
Equipos").uniqueResult();
      System.out.println(obj);

      session.close();
      sessionFactory.close();
```

7.3.Filtrando

Al igual que en SQL en HQL también podemos filtrar los resultados mediante la cláusula `WHERE`. La forma de usarla es muy parecida a SQL.

```
SELECT e FROM Equipos e WHERE conferencia='Este' AND división <>
'Atlántico'
```

Al igual que con el nombre de la clase, el nombre de los campos del `WHERE` siempre hace referencia a las propiedades Java y nunca a los nombres de las columnas.De esa forma seguimos independizando nuestro código Java de la estructura de la base de datos.

7.3.1. Literales

Texto

El carácter para indicar un literal de texto es la comilla simple no pudiéndose usar la doble comilla.

```
SELECT p FROM Profesor p WHERE nombre='juan'
```

Si se quiere usar la comilla dentro de un literal deberemos duplicarla.

```
SELECT p FROM Profesor p WHERE ape1='perez l''andreu'
```

Integer

Para incluir un número del tipo `integer` simplemente se escribe dicho número.

```
SELECT tb FROM TiposBasicos tb WHERE inte=4
```

Long

Para incluir un número del tipo `long` se escribe dicho número y se añade una `L` mayúscula al final.

```
SELECT tb FROM TiposBasicos tb WHERE long1=4L
```

double

Para representar un `double` se escribe el número separarando la parte decimal con un punto o se puede usar la notación científica.

```
SELECT tb FROM TiposBasicos tb WHERE double1=1.45
SELECT tb FROM TiposBasicos tb WHERE double1=1.7976931348623157E308
```

float

Para representar un `float` se escribe el número separarando la parte decimal con un punto o se puede usar la notación científica pero se le añade el carácter `F` en mayúscula al final.

```
SELECT tb FROM TiposBasicos tb WHERE float1=1.45F
SELECT tb FROM TiposBasicos tb WHERE float1=3.4028235E38F
```

Fecha

Para indicar una fecha la incluiremos entre comillas simples con el formato `yyyy-mm-dd`

```
SELECT tb FROM TiposBasicos tb WHERE dateDate='2012-07-25'
```

Hora

Para indicar una hora la incluiremos entre comillas simples con el formato `hh:mm:ss`

```
SELECT tb FROM TiposBasicos tb WHERE dateTime='02:05:10'
```

Fecha y hora

Para indicar una fecha y hora la incluiremos entre comillas simples con el formato `yyyy-mm-dd hh:mm:ss.millis`,siendo optativos el último punto y los milisegundos .

```
SELECT tb FROM TiposBasicos tb WHERE dateTime='2012-07-25 02:05:10'
```

Consultas a probar

1) La siguiente consulta funciona correctamente

```
SELECT e FROM Equipos e WHERE conferencia='Este' AND division<>'Atlántico'
```

2) La siguiente consulta falla al poner un valor de texto con comillas dobles

```
SELECT e FROM Equipos e WHERE conferencia=”Este” AND division<>'Atlántico'
```

3) La siguiente consulta funciona correctamente

SELECT p FROM Partidos p WHERE puntosLocal=89

4) La siguiente consulta funciona correctamente

SELECT e FROM Estadisticas e WHERE asistenciasPorPartido=12.0

7.3.2. Operadores de comparación

Para comparar los datos en una expresión se pueden usar las siguientes Operadores:

- Signo igual ”=”: La expresión será verdadera si los dos datos son iguales. En caso de comparar texto, la comparación **no** es sensible a mayúsculas o minúsculas.
- Signo mayor que ”>”: La expresión será verdadera si el dato de la izquierda es mayor que el de la derecha.
- Signo mayor que ”>=”: La expresión será verdadera si el dato de la izquierda es mayor o igual que el de la derecha.
- Signo mayor que ”<”: La expresión será verdadera si el dato de la izquierda es menor que el de la derecha.
- Signo mayor que ”<=”: La expresión será verdadera si el dato de la izquierda es menor o igual que el de la derecha.
- Signo desigual ”<>”: La expresión será verdadera si el dato de la izquierda es distinto al de la derecha.

- Signo desigual "`!=`": La expresión será verdadera si el dato de la izquierda es distinto al de la derecha.
- Operador "`between`": La expresión será verdadera si el dato de la izquierda está dentro del rango de la derecha.

```
SELECT tb FROM TiposBasicos tb WHERE inte BETWEEN 1 AND 10
```

- Operador "`in`": La expresión será verdadera si el dato de la izquierda está dentro de la lista de valores de la derecha.

```
SELECT tb FROM TiposBasicos tb WHERE inte IN(1,3,5,7)
```

- Operador "`like`": La expresión será verdadera si el dato de la izquierda coincide con el patrón de la derecha. Se utilizan los mismos signos que en SQL "`%`" y "_".

```
SELECT tb FROM TiposBasicos tb WHERE stri LIKE 'H_la%'
```

- Operador "`not`": Niega el resultado de una expresión.
- expresión "`is null`": Comprueba si el dato de la izquierda es `null`.

```
SELECT tb FROM TiposBasicos tb WHERE dataDate ISNULL
```

Consultas a probar

1) La siguiente consulta funciona correctamente

```
SELECT e FROM Equipos e WHERE conferencia='Este' AND división!='Atlántico'
```

2) La siguiente consulta funciona correctamente

SELECT p FROM Partidos p WHERE puntosLocalBETWEEN80AND100

3) La siguiente consulta funciona correctamente

```
SELECT e FROM Equipos e WHERE division in ('Atlántico','Central')
```

4) La siguiente consulta funciona correctamente

```
SELECT e FROM Equipos e WHERE division not in ('Atlántico','Central')
```

5) La siguiente consulta funciona correctamente

SELECT e FROM Equipos e WHERE ciudad like 'C%'

7.3.3. Operadores Lógicos

Se puede hacer uso de los típicos operadores lógicos como en SQL:

- AND
- OR
- NOT

Ejemplo: `SELECT e FROM Equipos e WHERE e.nombre like 'P%' OR ciudad='Chicago'`

7.3.4. Operadores Aritméticos

Se puede hacer uso de los típicos operadores aritméticos:

- suma +
- resta -
- multiplicación *
- division /

Ejemplo:
```
SELECT e FROM Estadisticas e WHERE ( (e.puntosPorPartido+
e.asistenciasPorPartido+ e.taponesPorPartido+e.rebotesPorPartido)/4)>10
```

7.3.5. Funciones de agregación

Las funciones de agregación que soporta HQL son:

- `AVG()`: Calcula el valor medio de todos los datos.
- `SUM()`: Calcula la suma de todos los datos.
- `MIN()`: Calcula el valor mínimo de todos los datos.
- `MAX()`: Calcula el valor máximo de todos los datos.
- `COUNT()`: Cuanta el nº de datos.

Ejemplo:
```
SELECT AVG(j.altura),AVG(j.peso),MIN(j.altura),MAX(j.altura),COUNT(*) FROM
Jugadores j
```

7.3.6. Funciones sobre escalares

Algunas de las funciones que soporta HQL sobre datos escalares son:

- `UPPER(s)`: Transforma un texto a mayúsculas.
- `LOWER(s)`: Transforma un texto a minúsculas.
- `CONCAT(s1, s2)`: Concatena dos textos
- `TRIM(s)`:Elimina los espacio iniciales y finales de un texto.
- `SUBSTRING(s, offset, length)`: Retorna un substring de un texto.El `offset` empieza a contar desde 1 y no desde 0.
- `LENGTH(s)`: Calcula la longitud de un texto.
- `ABS(n)`: Calcula el valor absoluto de un número.
- `SQRT(n)`: Calcula la raíz cuadrada del número
- Operador "`||`" : Permite concatenar texto.

7.4. Ordenación

Como en SQL también es posible ordenar los resultados usando `ORDER BY`. Su funcionamiento es como en SQL.

`Ejemplo:`
```
SELECT e From Equipos e order by e.nombre DESC
```

Las palabras `ASC` y `DESC` son opcionales al igual que en SQL.

El uso de funciones escalares y funciones de agrupamiento en la cláusula `ORDER BY` sólo es soportado por Hibernate si es soportado por el lenguaje de SQL de la base de datos sobre la que se está ejecutando.

No se permite el uso de expresiones aritméticas en la cláusula `ORDER BY`.

7.5.Agrupaciones

Al igual que en SQL se pueden realizar agrupaciones mediante las palabras claves `GROUP BY` y `HAVING`

```
SELECT e.conferencia,count(e.conferencia) From Equipos e Group by e.conferencia
```

Esta consulta obtiene el número de equipos que hay en cada conferencia

El uso de funciones escalares y funciones de agrupamiento en la cláusula `HAVING` sólo es soportado por Hibernate si es soportado por el lenguaje de SQL de la base de datos sobre la que se está ejecutando.

No se permite el uso de expresiones aritméticas en la cláusula `GROUP BY`.

7.6.Subconsultas

HQL también soporta subconsultas como en SQL.

```
SELECT e From Equipos e where e.nombre in(Select j.equipos from Jugadores j Group by j.equipos Having count(j.equipos)>2)
```

Esta consulta obtiene los equipos de los que hemos introducido en la base de datos más de dos jugadores

7.7.Consultas de actualización

```
Query query = session.createQuery("update Equipos e set e.division='Pacífico' where division='Central'");

Int numActualizaciones=query.executeUpdate();
```

Mediante la invocación del método executeUpdate() obtenemos el número de registros actualizados.

7.8.Consultas de borrado

```
Query query = session.createQuery("delete from Equipos e where e.division = 'Pacífico'");

Int numBorrados=query.executeUpdate();
```

Mediante la invocación del método executeUpdate() obtenemos el número de registros borrados.