

Introducción a JPA

JPA o Java Persistence API es uno de los APIs más conocidos y utilizados en el mundo de Java EE . Sin embargo, en muchas ocasiones cuando la gente quiere comenzar a trabajar con él, pronto comienzan los problemas y las dudas. En este libro vamos a intentar aprender a cómo usar este API partiendo de cero.

ENTORNO DE DESARROLLO

Vamos a empezar con lo más básico que no tiene porque ser lo más sencillo: configurar un entorno de desarrollo y una base de datos para trabajar con JPA. Lo primero que necesitamos es instalar el JDK en nuestro ordenador que podemos descargar de la siguiente URL.

<http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html>

Dependiendo del sistema operativo utilizado el procedimiento puede variar ligeramente ¡. En el entorno de windows es tan sencillo como dar doble click al ejecutable que nos hemos descargado y proceder por defecto con la instalación por defecto.

Una vez instalado el JDK ,el siguiente paso es descargar el Eclipse JEE como entorno de desarrollo . Este IDE se puede descargar desde la siguiente URL:

<https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplersr2>

En este caso la instalación es todavía más sencilla ya que simplemente nos vale con descomprimir el zip que nos descarguemos y dar doble click en el icono de Eclipse para que nos abra el entorno de desarrollo.

Ya tenemos el JDK instalado y el Eclipse abierto, aún así nos quedan algunos pasos. El siguiente paso importante es instalar una base de datos MySQL. Para ello accederemos a la siguiente URL y nos bajaremos el servidor que se adecue a nuestra plataforma.

<http://dev.mysql.com/downloads/mysql/>

Pulsamos doble click sobre el programa de instalación e instalamos nuestro servidor de base de datos pulsando las opciones por defecto . Realizada esta operación el último programa que necesitamos para poder trabajar con los ejemplos del libro es el **MySQL WorkBench** ,cliente gráfico de MySQL que nos da acceso al servidor de forma sencilla y clara **permitiéndonos crear bases de datos , tablas , modelos etc.** Este programa podemos descargarlo de la siguiente URL.

<http://dev.mysql.com/downloads/tools/workbench/>

Realizados estos pasos tenemos todo el software instalado (**JDK,Eclipse,MySQL Server,MySQLWorkbench**). Es el momento de comenzar a trabajar con JPA desde cero.

Configuración de JPA

Partiremos de conceptos sencillos para trabajar con JPA en los distintos capítulos. En este capítulo vamos a presentar el concepto de **Alumno (DNI,nombre,apellidos y edad)** a partir del cuál construiremos una tabla en la base de datos con las columnas necesarias y nos apoyaremos en el para configurar JPA.

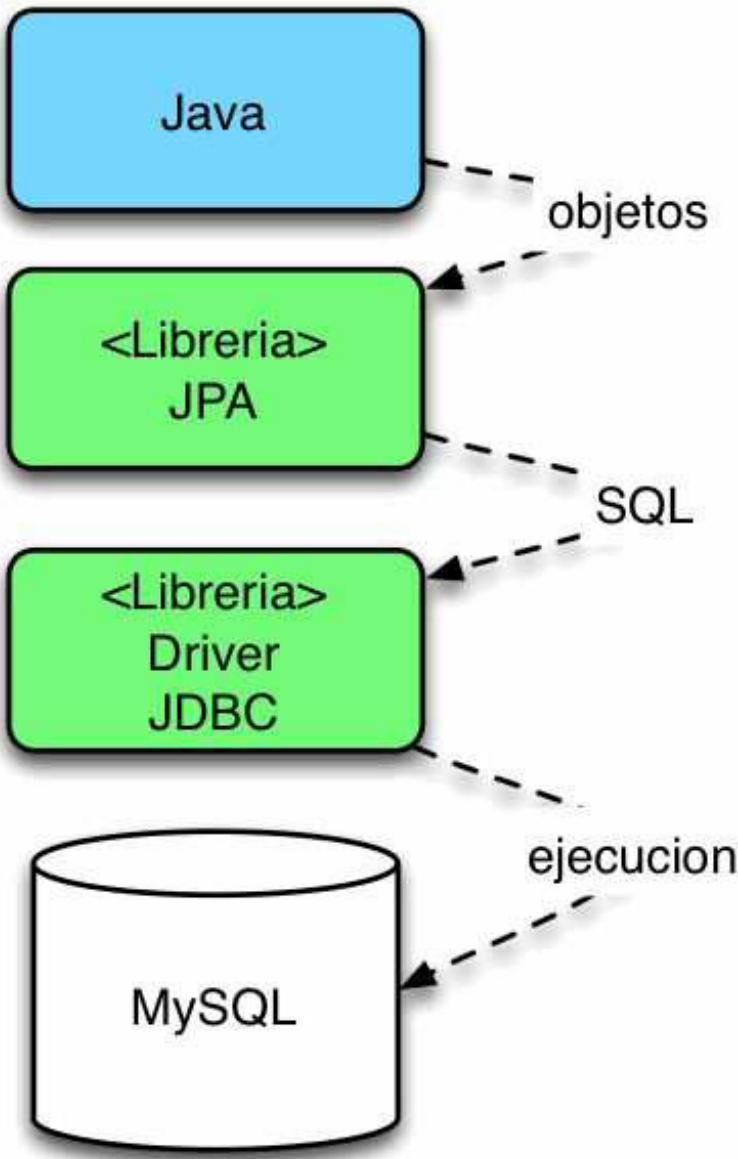
Alumno	
dni	varchar(10)
nombre	varchar(20)
apellidos	varchar(20)
edad	int

JPA Y LIBRERÍAS

Construida la tabla en la base de datos ,el siguiente paso **es acceder a la tabla Alumno** desde Java utilizando JPA y almacenar datos en ella . Este paso no es sencillo y para poder abordarlo necesitamos entender dos cosas: **el concepto de driver JDBC y el concepto de framework de persistencia JPA**.

Driver JDBC: Librería Java (jar) que implementa el protocolo de comunicación con una base de datos concreta, permitiendo ejecutar consultas SQL desde Java contra dicha base de datos

Framework de persistencia JPA: Conjunto de librerías Java que permiten gestionar la persistencia de objetos en una base de datos de forma sencilla .El framework se encarga de generar las consultas SQL y lanzarlas a través del Driver JDBC.



Así pues antes de comenzar a trabajar con JPA deberemos obtener dos librerías .En primer lugar el Driver JDBC que nos permite lanzar consultas y en segundo lugar el propio framework de persistencia JPA que las genera (**En este caso usaremos Hibernate**). Para ello deberemos acceder a las siguientes dos URLs.

Driver JDBC : <https://dev.mysql.com/downloads/connector/j/>

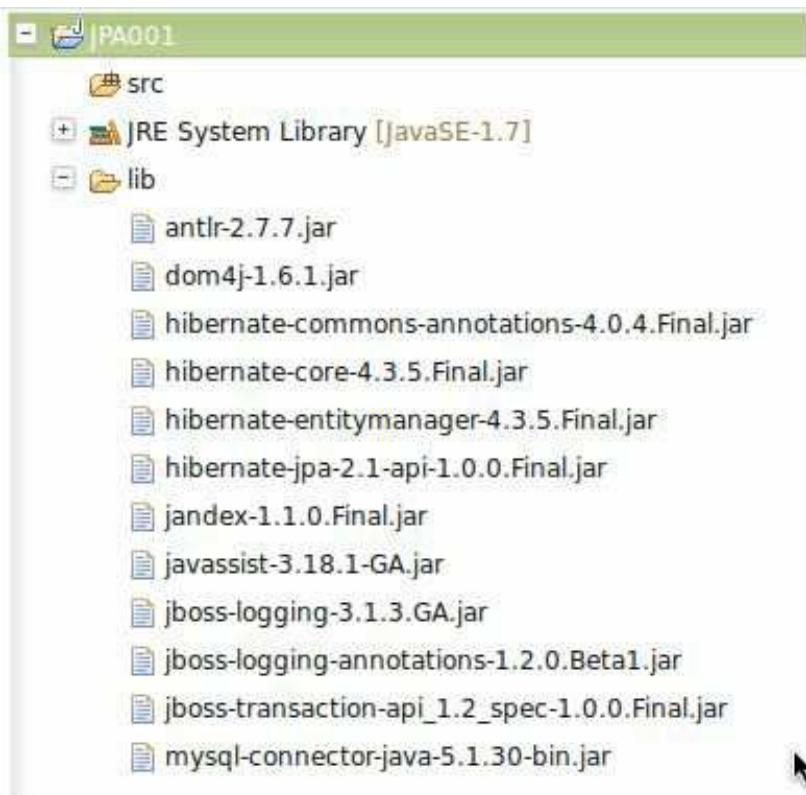
Hibernate : <http://hibernate.org/orm/downloads/>

Una vez descargadas los dos ficheros los descomprimiremos y **obtendremos los jars o librerías Java necesarias para nuestra aplicación**. En el caso del driver JDBC solo hay una (**mysql-connector-java-5.1.X-bin.jar**) y en Hibernate necesitaremos todas las de la carpeta **REQUIRED** mas las de la carpeta **JPA**.

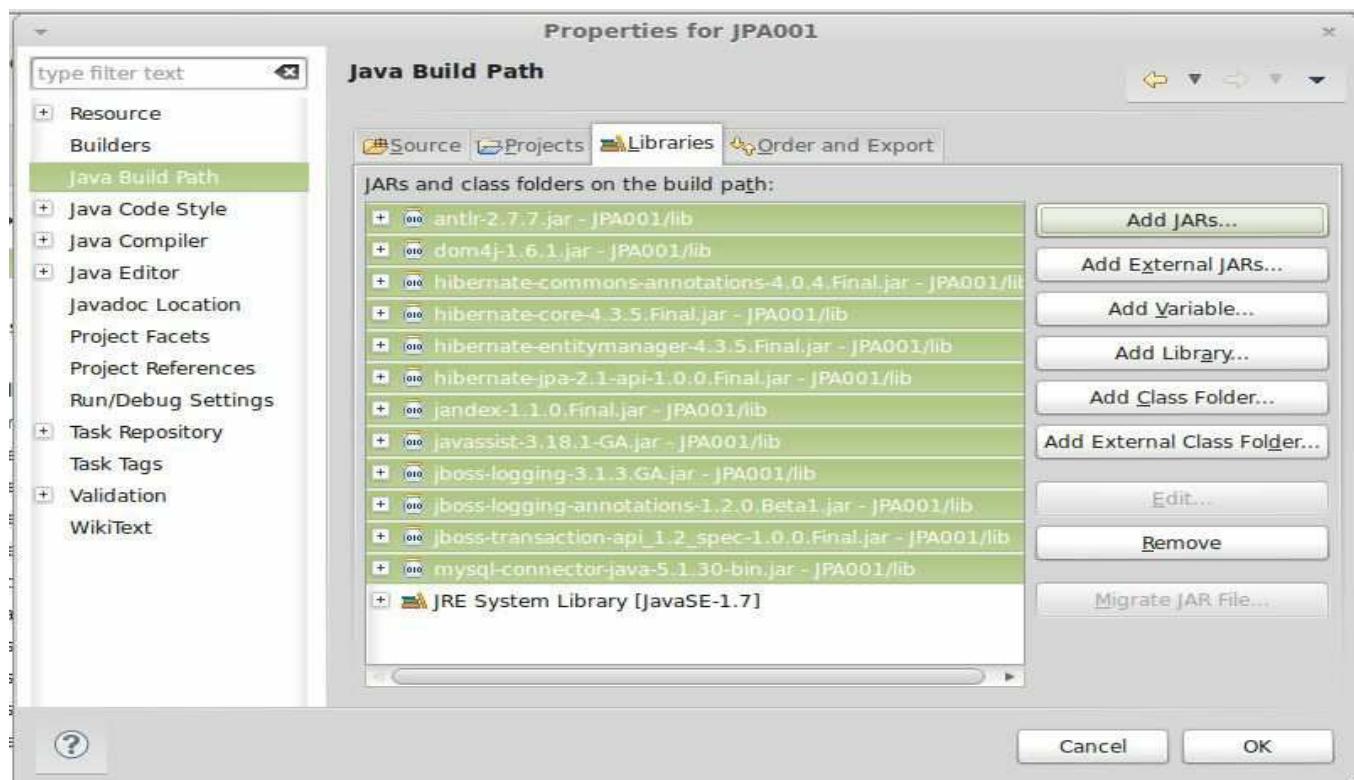
ECLIPSE Y JPA

Es momento de construir un proyecto con Eclipse JEE que nos permita trabajar con JPA . Podríamos usar los asistentes de JPA pero no vamos a hacerlo **¿Por qué razón?** , porque el contenido del libro debe ser válido para gente que use otros IDEs. Así pues vamos a crear un sencillo proyecto Java en Eclipse. (File—>New-> JavaProject) Nada mas crear el proyecto añadimos una nueva carpeta (**lib**) al mismo y copiamos todas las librerías que necesitamos en él. Para mayor claridad ,a

continuación se muestra una imagen con las librerías necesarias.



El siguiente paso es **añadir todas estas librerías Java al classpath de la aplicación** para que funcionen correctamente. Para realizar esta operación pulsaremos botón derecho—> properties en el proyecto—>**Java BuildPath** y **añadimos todos los jars de la carpeta lib (en la pestaña de librerías)**. El concepto de **classpath** hace referencia a los directorios o librerías jar en las que Java buscará las clases para cargarlas. La siguiente imagen muestra como quedan configuradas las librerías.



RESUMEN

Hechos configurado el proyecto Java con las librerías necesarias para trabajar con JPA. Es momento de comenzar a trabajar con JPA y el concepto de Alumno.

Alumno y JPA

Hemos definido en el capítulo anterior el concepto de Alumno y hemos visto qué campos tiene a nivel de base de datos .Ahora abordaremos el mismo concepto en el mundo Java y veremos cómo las anotaciones de JPA facilitan su persistencia en una base de datos . Lo primero que vamos a crear es la clase Alumno , que contendrá las siguientes anotaciones de JPA.



1. **@Entity** : Identifica una clase Java como clase a persistir contra la base de datos dentro del estándar de JPA
2. **@Id** : Define cuál de los campos de la clase es la clave primaria de la tabla en nuestro caso el DNI.

Además, la clase estará obligada por la especificación de JPA a **implementar el interface Serializable y disponer de un constructor por defecto**. Veamos su código fuente para despejar posibles dudas existentes.

```
package com.arquitecturajava;

import java.io.Serializable;

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

@Entity
public class Alumno implements Serializable{
```

```
    private static final long serialVersionUID = 1L;
```

```
    @Id
```

```
    private String dni;
```

```
    private String nombre;
```

```

private String apellidos;
private int edad;

public Alumno() {
    super();
}

public Alumno(String dni, String nombre,
String apellidos, int edad) {
    super();
    this.dni = dni;
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
}

public String getDni() {
    return dni;
}

public void setDni(String dni) {
    this.dni = dni;
}

public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public String getApellidos() {
    return apellidos;
}

public void setApellidos(String apellidos) {
    this.apellidos = apellidos;
}

public int getEdad() {
    return edad;
}

public void setEdad(int edad) {
    this.edad = edad;
}
}}
```

Aunque hemos terminado de construir la clase Alumno , en estos momentos no tenemos ninguna relación construida entre

nuestro programa Java y la base de datos de tal forma que Java sepa como acceder al servidor lanzar consultas ,etc . El siguiente paso obligatorio será **construir el fichero persistente.xml** que define dicha relación y es parte del standard.

PERSISTENCE.XML

Este fichero es el encargado de **definir la cadena de conexión, drivers, usuario y password que JPA utilizará para persistir los diferentes objetos**.El fichero ha de ser ubicado en la carpeta META-INF del proyecto ,si la carpeta no existe deberemos crearla.



El contenido del fichero será el siguiente :

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

  <persistence-unit name="UnidadCurso">
    <properties>
      <property name = "hibernate.show_sql"
        value = "true" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLDialect" />
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.user"
        value="root" />
      <property name="javax.persistence.jdbc.password"
        value="jboss" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost/LibroHibernate" />
    </properties>
  </persistence-unit>
</persistence>
```

Vamos a revisar cada uno de los parámetros del fichero:

1. **persistence-unit** : El concepto de persistence-unit está asociado a una conexión contra una base de datos concreta y se encarga de almacenar toda la información asociada a ésta.
2. **Hibernate-dialect**: Parámetro de Hibernate que define el tipo de motor de base de datos utilizado ,en este caso MySQL.

3. **javax.persistence.jdbc.driver:** Driver de acceso a datos que instalamos anteriormente para MySQL en la carpeta lib.
4. **javax.persistence.jdbc.user:** Usuario que tiene los permisos para acceder a la base de datos
5. **javax.persistence.jdbc.password:** La clave de acceso del Usuario a la base de datos.
6. **javax.persistence.jdbc.url :** Url de acceso a la base de datos , cada motor de base de datos tiene la suya. En el caso de MySQL se especifica el servidor “**localhost**” y la base de datos a la que deseamos acceder “**LibroHibernate**”.

Acabamos de configurar todos los parámetros que JPA necesita para acceder a la base de datos. Es momento de comenzar a trabajar con el standard para **insertar, borrar, actualizar y seleccionar objetos de tipo Alumno**.

INSERTAR ALUMNO (PERSIST)

La inserción de un objeto es nuestra primera operación a nivel de JPA . Aunque se trata de una operación relativamente sencilla hay que explicar una serie de conceptos .Veamos un ejemplo a nivel de código fuente y a partir de él clarificaremos los conceptos fundamentales.

```
package com.arquitecturajava.main;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import com.arquitecturajava.Alumno;

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

        EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("UnidadCurso");
        EntityManager em = emf.createEntityManager();
        EntityTransaction transaccion= em.getTransaction();
        transaccion.begin();
        Alumno pedro= new Alumno("1","pedro","gomez",30);
em.persist(pedro);
        transaccion.commit();
        em.close();
    }
}
```

El código no es muy extenso pero aparecen bastantes conceptos nuevos que vamos a ir aclarando poco a poco.

1. **Persistence :**Clase que se encarga de leer el fichero de **persistente.xml** y seleccionar una **unidad de persistencia** concreta con la cuál quedaremos enlazados a una base de datos . En este caso la unidad de persistencia se denomina “**UnidadCurso**” (ver **persistence.xml**). Una vez seleccionada la unidad se genera un **EntityManagerFactory** que asignamos a la variable **emf**.

2. **EntityManagerFactory :** Clase que tiene un rol de factoría y se encarga de crear **EntityManagers** que serán los

que en última instancia gestionen la persistencia de los distintos objetos contra la base de datos.

3. EntityManager : Clase fundamental de JPA que se encarga de todas las operaciones que afectan a los objetos que deseamos persistir . Es encargada de seleccionar objetos de la base de datos , de insertar nuevos , de actualizarlos, borrarlos etc .Se encarga además de mantener el estado de los objetos que tenemos en memoria con los cambios que en ellos se han producido.

4. EntityTransaction:Clase que genera una abstracción sobre el concepto general de Transacción a nivel de base de datos.

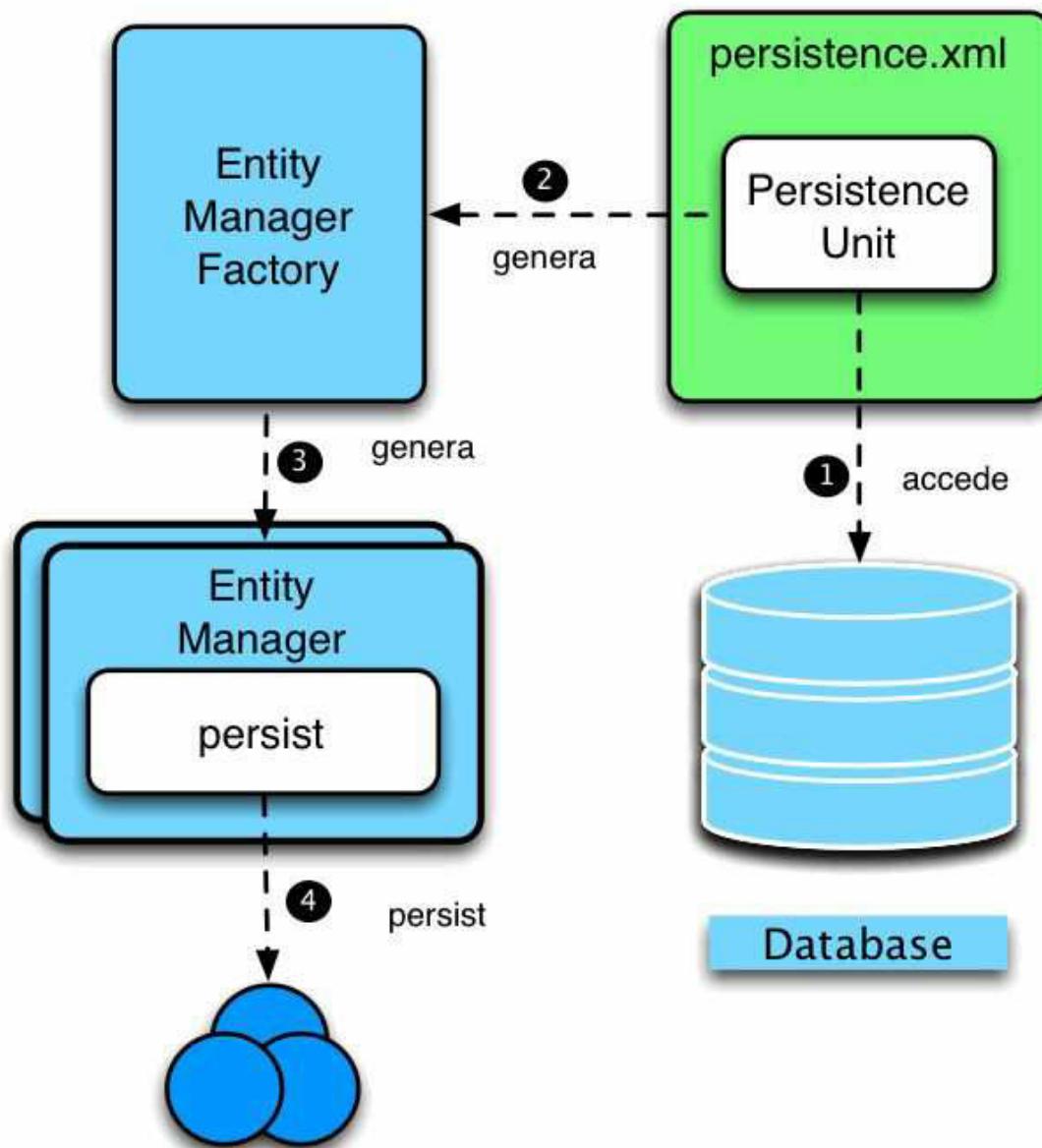
Explicados los conceptos fundamentales podemos ver como el código se adapta a ellos sin problema creando y utilizando el objeto **EntityManager** para invocar al **método persist** y pasarlo como parámetro un Alumno que será almacenado en la base de datos.

```
Alumno pedro=
```

```
new Alumno("1","pedro","gomez",30);
```

```
em.persist(pedro);
```

El siguiente diagrama muestra un resumen de la relación entre los diversos conceptos que hemos expuesto.



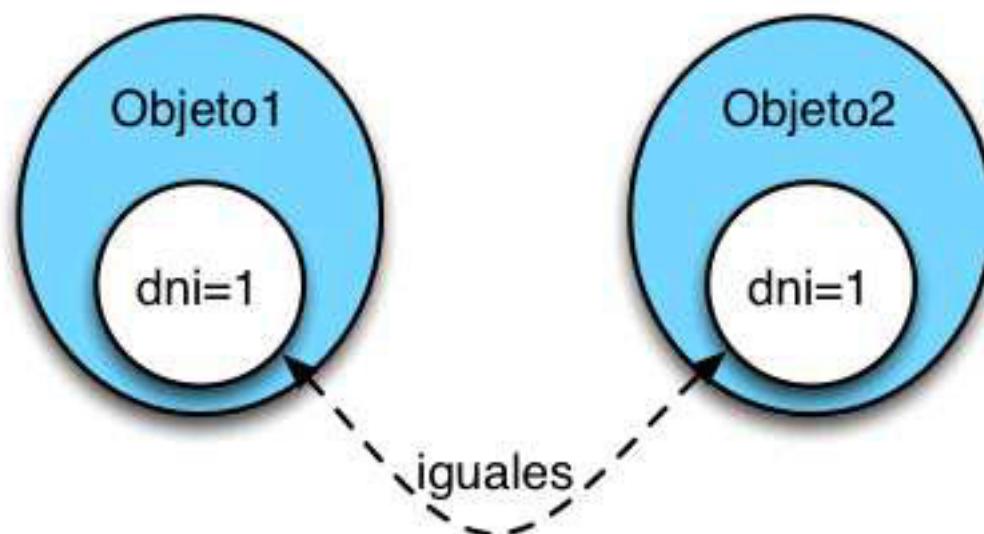
Acabamos de realizar la operación mas sencilla de JPA la de inserción a través del método persist que nos añadirá un

nuevo registro en la base de datos.

#	dni	nombre	apellidos	edad
1	1	pedro	gomez	30
*	NULL	NULL	NULL	NULL

OBJETOS E IDENTIDAD

El siguiente paso natural a la hora de trabajar con **JPA** es **realizar una búsqueda del objeto** que acabamos de insertar en la base de datos .Ahora bien para ello podemos buscar el objeto por DNI . A estas alturas la clase Alumno no tiene implementada esa regla de lógica de negocio. **Dos Alumnos deben ser iguales si su DNI es idéntico.**



Para implementar esta regla a nivel de Java **deberemos sobrecargar los métodos equals y hashCode de la clase Alumno de tal forma que dos Alumnos serán iguales si su DNI coincide**. Así cumpliremos con las reglas de negocio que existen en la base de datos y que definen la clave primaria.

Vamos pues a implementar ambos métodos y comentarlos:

```
@Override  
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result +  
        ((dni == null) ? 0 : dni.hashCode());  
    return result;  
}
```

```

@Override
public boolean equals(Object obj) {
    if(this == obj)
        return true;
    if(obj == null)
        return false;
    if(getClass() != obj.getClass())
        return false;
    Alumno other = (Alumno) obj;
    if(dni == null) {
        if(other.dni != null)
            return false;
    } else if(!dni.equals(other.dni))
        return false;
    return true;
}

```

Pueden parecer métodos complejos pero se generan de forma automática con el asistente de Eclipse →Botón Derecho en Alumno→Source→Generate Equals y hashCode. Una vez hecho seleccionamos únicamente el campo de DNI y Eclipse hará el trabajo por nosotros generando el código . Es momento de comenzar a buscar Alumnos.

SELECCIONAR ALUMNO (FIND)

Vamos a seleccionar el único Alumno que tenemos en la base de datos y que acabamos de insertar y que se llama “pedro” . En este caso volveremos a usar el EntityManager y el **método find** que es el encargado de buscar un alumno por clave primaria (**en nuestro caso DNI**). He aquí veamos el código

```

package com.arquitecturajava.main;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.arquitecturajava.Alumno;

public class Principal001Buscar {

    public static void main(String[] args) {

        EntityManagerFactory emf =
        Persistence.
        createEntityManagerFactory("UnidadCurso");

        EntityManager em =
        emf.createEntityManager();

Alumno mialumno=e.m.find(Alumno.class, "1");

System.out.println(mialumno.getNombre());

```

```
em.close();  
}  
}
```

La operación a realizar es muy sencilla **pasamos al método find la clase de los objetos que deseamos buscar (tipo Alumno.class) y la clave primaria del objeto en cuestión (“1”)**. Hecho ésto EntityManager nos devolverá el objeto Alumno que buscábamos e imprimiremos su nombre por consola.

pedro

Realizada la búsqueda mas sencilla procederemos a continuación al borrado.

BORRAR ALUMNO (REMOVE)

Una operación que es similar a la de insertar es la operación de borrar un Alumno de la base de datos .A continuación mostramos su código:

```
package com.arquitecturajava.main;  
  
import javax.persistence.EntityManager;  
  
import javax.persistence.EntityManagerFactory;  
  
import javax.persistence.EntityTransaction;  
  
import javax.persistence.Persistence;  
  
import com.arquitecturajava.Alumno;  
  
public class Principal004Borrar {  
  
    public static void main(String[] args) {  
        EntityManagerFactory emf =  
            Persistence.  
            createEntityManagerFactory("UnidadCurso");  
        EntityManager em = emf.  
            createEntityManager();  
        EntityTransaction transaccion=  
            em.getTransaction();  
        transaccion.begin();  
Alumno mialumno=em.find(Alumno.class, "1");  
em.remove(mialumno);  
        transaccion.commit();  
        em.close();  
    }  
}
```

En este caso operamos de una forma algo diferente . En primer lugar usamos JPA para **seleccionar el Alumno de la base de datos** . Obtenido éste invocamos **al método remove que se encarga de eliminarlo** , Ahora bien, recordemos que lo tenemos que realizar de forma transaccional.

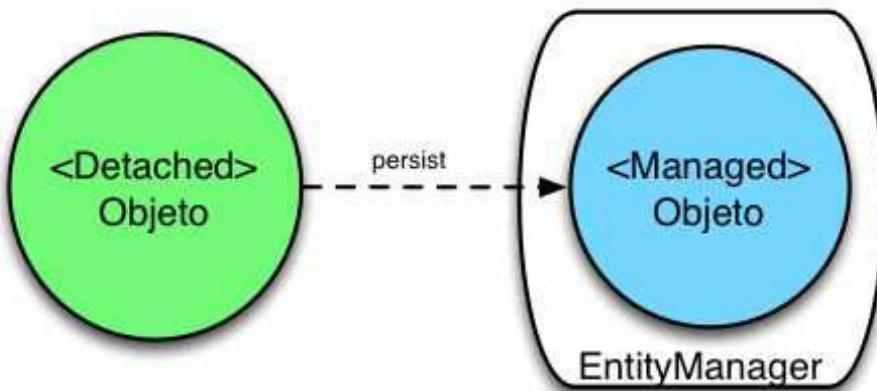
MANAGED ENTITIES

Tras ver las operaciones las operaciones que menos dificultad compartan, nos queda por comentar algo clave a la hora de trabajar con JPA . **El concepto de EntityManager y cómo éste controla el estado de las entidades .**

De los estados en que pueden estar las entidades , dos son los más importantes:

Detached : En este estado la entidad acaba de ser construida como objeto y no está bajo el control de un EntityManager.

Managed: En este estado la entidad ha pasado a estar controlada por un EntityManager . Ésto sucede por ejemplo cuando invocamos **el método persist**. A partir de estos momentos cualquier cambio en la Entidad es almacenado por el EntityManager al que está asociado.



ACTUALIZAR ALUMNO (MERGE)

Hay situaciones que no son tan sencillas como las que hemos presentado . Por ejemplo si nosotros creamos un objeto Alumno en Java con los datos que ya existen en la base de datos e invocamos al método persist este fallará. **El método persist únicamente es válido si el Alumno no existe** . Así pues tendremos un problema ya que **el Alumno que acabamos de construir ya existe**. Para solventar este tipo de casuísticas existe a nivel de EntityManager **el método merge** . Vamos a ver como se trabaja con él.

```
package com.arquitecturajava.main;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import com.arquitecturajava.Alumno;

public class Principal003Merge {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.
            createEntityManagerFactory("UnidadCurso");
        EntityManager em = emf.
            createEntityManager();
        EntityTransaction transaccion= em.
            getTransaction();
        transaccion.begin();
```

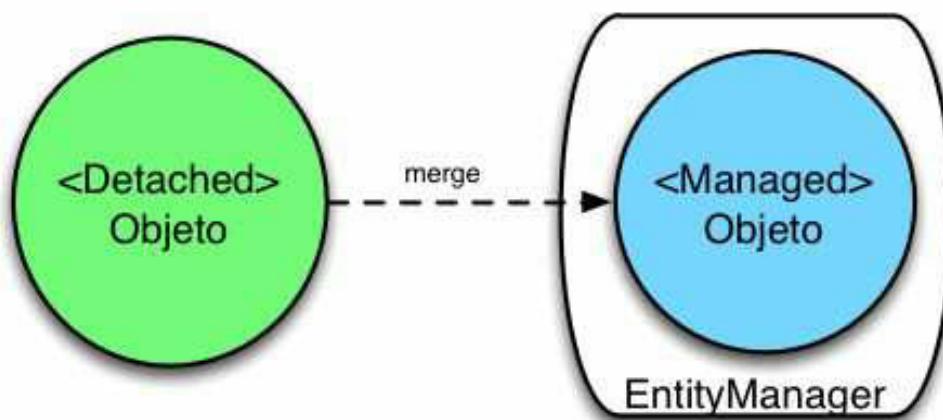
```

Alumno mialumno=new Alumno ();
mialumno.setDni("1");
mialumno.setNombre("pedro");
mialumno.setApellidos("sanchez");
mialumno.setEdad(30);

Alumno nuevo=em.merge(mialumno);
nuevo.setNombre("juan");
transaccion.commit();
em.close();
}
}

```

Podemos observar cómo el método merge cambia el estado del objeto nuevo a “Managed” devolviéndonos **una instancia nueva**. Una vez realizada esta operación podemos variar la información que necesitamos y terminar la transacción nuestro Entity Manager será capaz de mantener los cambios.



RESUMEN

Hemos visto como utilizar los métodos fundamentales de la clase EntityManager tales como **persist()**, **find()**, **remove()** y **merge()** en el siguiente capítulo comenzaremos a relacionar clases.

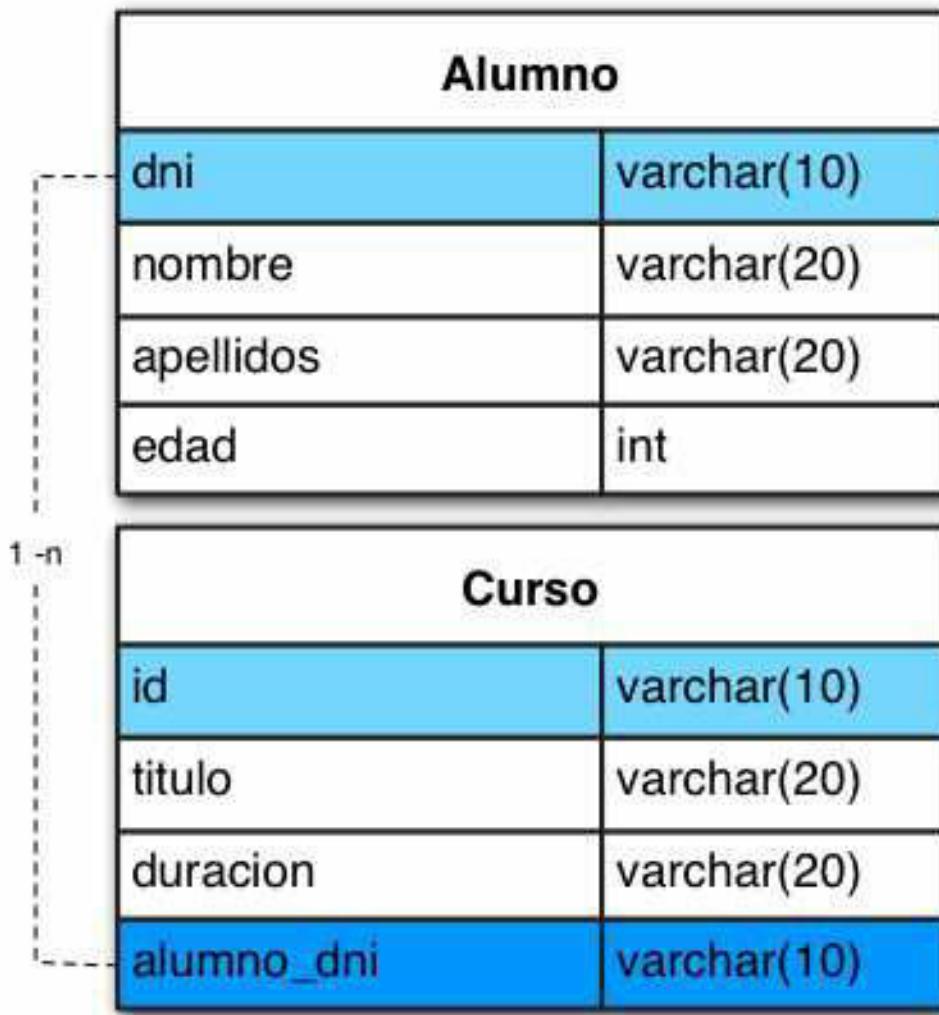
DESCARGAR EJEMPLOS

http://www.arquitecturajava.com/wp-content/uploads/001_AlumnoJPA.zip

http://www.arquitecturajava.com/wp-content/uploads/002_AlumnoJPAOperaciones.zip

Relaciones @ManyToOne y @OneToMany

En estos momentos únicamente tenemos la tabla Alumno con la que podemos trabajar. Ahora se debe ampliar el modelo y añadir nuevos conceptos con los cuales vayamos profundizando en JPA. En este caso vamos a construir la **tabla Curso** (**id, titulo, duración**) y vamos a partir de la situación en la que un Alumno puede hacer varios cursos pero un Curso concreto pertenece a un Alumno. La clásica relación de 1 a n entre dos conceptos (más adelante veremos cosas más complejas).



@MANYToOne (CURSO)

Es necesario ahora trasladar la relación existente en la base de datos al modelo de dominio. Para ello deberemos crear la clase **Curso** y asignarle las propiedades (**id, titulo, duración y alumno**). Para ello usaremos una nueva anotación **@ManyToOne**.



Esta **nueva anotación @ManyToOne** se encarga de relacionar el concepto de **Curso** con el concepto de **Alumno** en el modelo de dominio. Vamos a ver el código fuente de la clase **Curso** y comentarlo.

```

package com.arquitecturajava;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Serializable;

@Entity
public class Curso implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private String id;

    private String titulo;

    private int duracion;

    private double precio;

    @ManyToOne
    @JoinColumn(name="alumno_dni")
    private Alumno alumno;
}

```

```
public Curso() {
    super();
}

public Curso(String id, String titulo,
    int duracion, double precio,
    Alumno alumno) {
    super();
    this.id = id;
    this.titulo = titulo;
    this.duracion = duracion;
    this.precio = precio;
    this.alumno = alumno;
}

public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

public String getTitulo() {
    return titulo;
}

public void setTitulo(String titulo) {
    this.titulo = titulo;
}

public int getDuracion() {
    return duracion;
}

public void setDuracion(int duracion) {
    this.duracion = duracion;
}

public double getPrecio() {
    return precio;
}

public void setPrecio(double precio) {
    this.precio = precio;
}

public Alumno getAlumno() {
    return alumno;
}

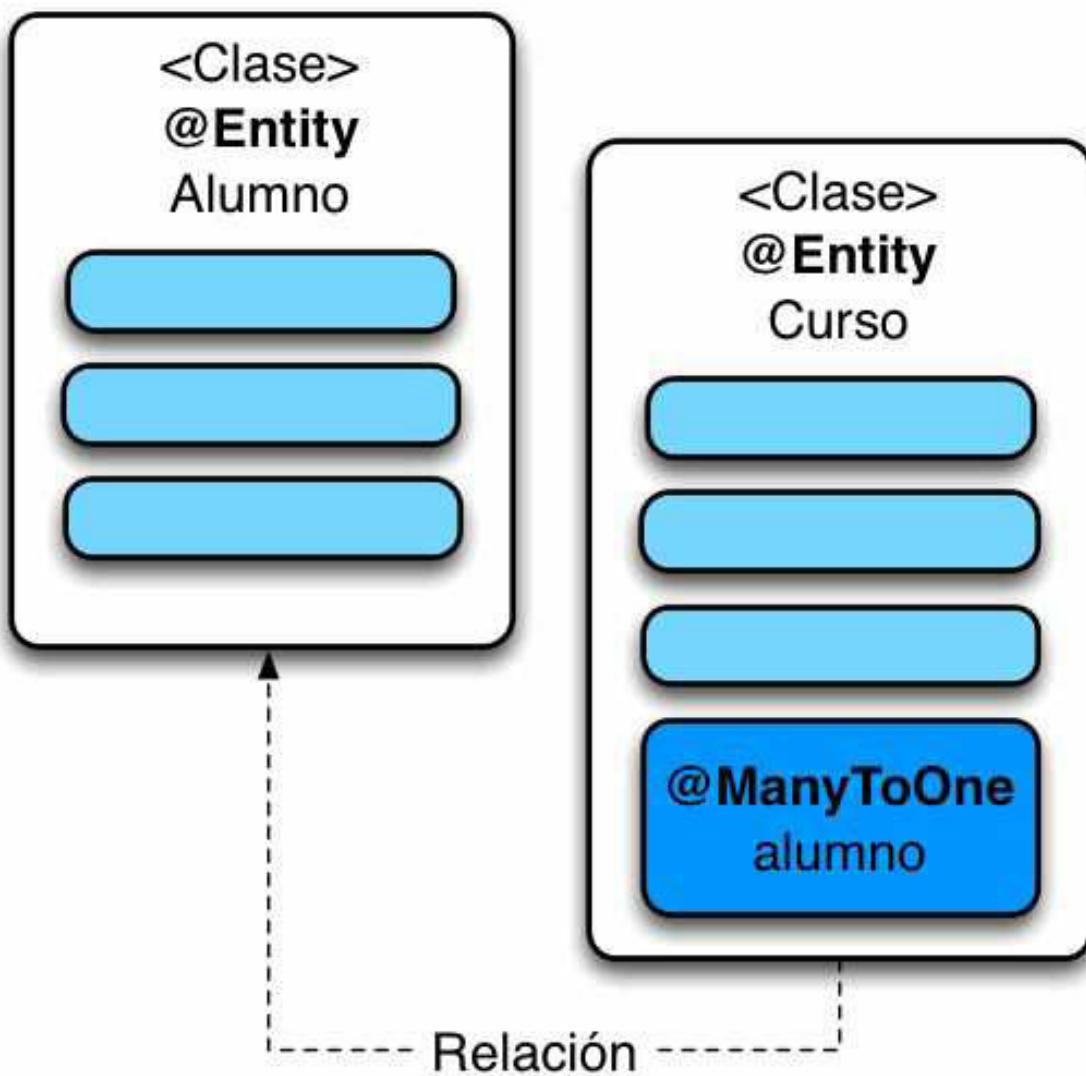
public void setAlumno(Alumno alumno) {
```

```
this.alumno = alumno;
}
```

Si nos centramos en la relación, veremos el siguiente código:

```
@ManyToOne
@JoinColumn(name="alumno_dni")
private Alumno alumno;
```

A parte de tener la anotación de @ManyToOne se añade la anotación @JoinColumn que define la columna que realiza las tareas de clave externa. Acabamos de construir nuestra primera relación a nivel de JPA.



Realizada esta operación , las dos clases quedan relacionadas. Vamos a ver el código del programa **main** que relaciona ambos conceptos y nos permite persistir Alumnos y Cursos en la base de datos simultáneamente.

```
//omitimos imports
public class Principal {
    public static void main(String[] args) {
        Alumno pedro= new Alumno("2","pedro","gomez",30);
```

```

Alumno maria= new Alumno("1","maria","perez",25);

EntityManagerFactory emf =
Persistence.createEntityManagerFactory("UnidadCurso");

EntityManager em = emf.createEntityManager();

Curso cursoJava=
new Curso("JAVA2","Introduccion Java 2",20,300, pedro);

Curso cursoNET=
new Curso("NET2","Introduccion NET 2",20,300,pedro);

Curso cursoPHP =
new Curso("PHP","Introducción a PHP",15,250,maria);

em.getTransaction().begin();

em.persist(pedro);

em.persist(maria);

em.persist(cursoJava);

em.persist(cursoNET);

em.persist(cursoPHP);

em.getTransaction().commit();

em.close();

System.out.println("termino");

}

}

```

La operativa es similar a los ejemplos anteriores solo que en este caso debemos salvar cada uno de los objetos construidos. Es momento de elaborar la otra parte de la relación y relacionar **Alumno con Curso convirtiendo la relación en bidireccional.**

@ONETOMANY (ALUMNO)

Esta relación es algo más compleja de construir que la anterior, **ya que un Alumno puede tener varios Cursos.** Estamos ante una relación de 1 a n .Por lo tanto tendremos, que gestionar un ArrayList o List de Cursos a nivel de la clase Alumno. Para ellos nos apoyaremos en la anotación **@OneToMany**.



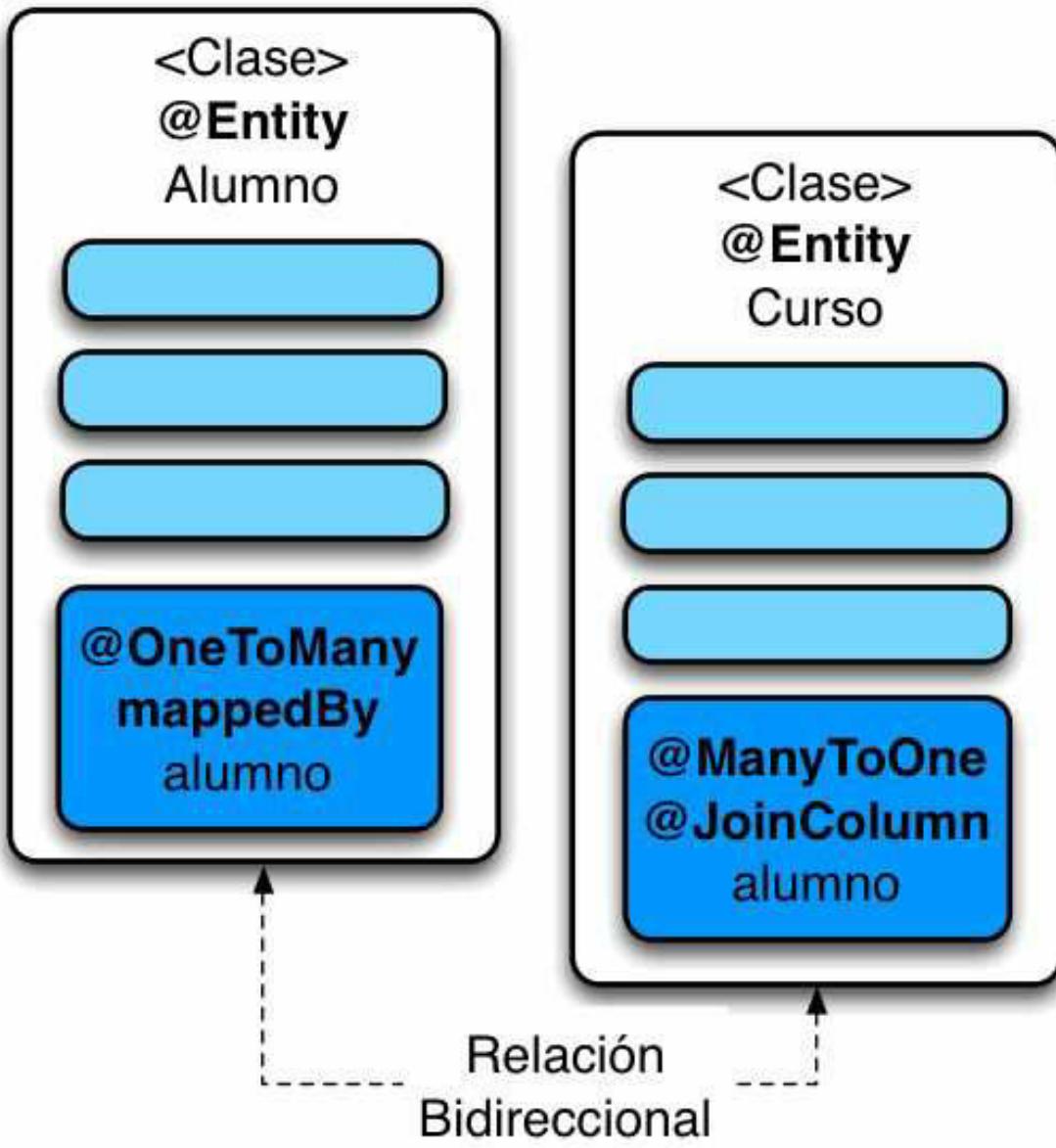
Esta anotación se encargará de **relacionar el Alumno con los Cursos que ha realizado** vamos a ver qué tenemos que cambiar de la clase Alumno para soportar dicha relación.

```

@Entity
public class Alumno implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    private String dni;
    private String nombre;
    private String apellidos;
    private int edad;
    @OneToMany(mappedBy="alumno")
    private List<Curso> cursos=
    new ArrayList<Curso>();
}

```

La anotación **@OneToMany** es muy similar a la anotación **@ManyToOne** en cuanto a su uso. Ahora bien ,si revisamos la relación detalladamente, nos podremos dar cuenta **la anotación JoinColumn no aparece** . Ésto es debido a que nuestra relación **@OneToMany delega en la relación @ManyToOne** construida en la clase Curso y que almacena la información sobre las clases externas.



Una vez definida la anotación **@OneToMany** nos encontramos con que la relación existente en ambas direcciones es **bidireccional**. Nos queda modificar el constructor de la clase **Curso** y el método **setAlumno** para reforzar la bidireccionalidad.

```

public Curso(String id, String titulo,
int duracion, double precio,
Alumno alumno) {
super();
this.id = id;
this.titulo = titulo;
this.duracion = duracion;
this.precio = precio;
setAlumno(alumno);
}

public void setAlumno(Alumno alumno) {
this.alumno = alumno;
alumno.getCurso().add(this);
}

```

RESUMEN

Hemos terminado de construir las primeras relaciones entre nuestras clases .En el siguiente capítulo abordaremos una introducción al mundo de las consultas con JPA.

DESCARGAR EJEMPLOS

<http://www.arquitecturajava.com/wp-content/uploads/003RelacionesOneToMany.zip>

Java Persistence Query Language

Vamos a realizar algunas consultas básicas a nuestro modelo. Para ello usaremos **JPQL (Java Persistence Query Language)** que es el lenguaje de consultas soportado por JPA. Empezaremos seleccionando todos los Alumnos.

```
package com.arquitecturajava.main;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;
import com.arquitecturajava.Alumno;
import com.arquitecturajava.Curso;

public class Principal01Alumnos {

    public static void main(String[] args) {

        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("UnidadCurso");
        EntityManager em = emf.createEntityManager();

        TypedQuery<Alumno>
        consultaAlumnos= em.
        createQuery("select a from Alumno a",
        Alumno.class);

        for (Alumno a :
        consultaAlumnos.getResultList()) {
        System.out.println(a.getNombre());
        }

        em.close();
        System.out.println("termino");
    }
}
```

Como podemos ver, las consultas tienen un aire similar a SQL. En este caso se utiliza el método **createQuery** del EntityManager para construir una consulta que selecciona todos los Alumnos utilizando un alias.

```
TypedQuery<Alumno> consultaAlumnos=
em.createQuery("select a from Alumno a ",Alumno.class);
```

Es evidente que la consulta hace referencia a la clase con la que trabajamos (**Alumno**) y le asocia un alias “**a**” para trabajar con ella . Construida la consulta, invocamos al método **.getResultSet()** que nos devolverá una lista con todos los Alumnos. Únicamente nos queda recorrerla con un bucle for.

```
for (Alumno a : consultaAlumnos.getResultList()) {  
    System.out.println(a.getNombre())  
}
```

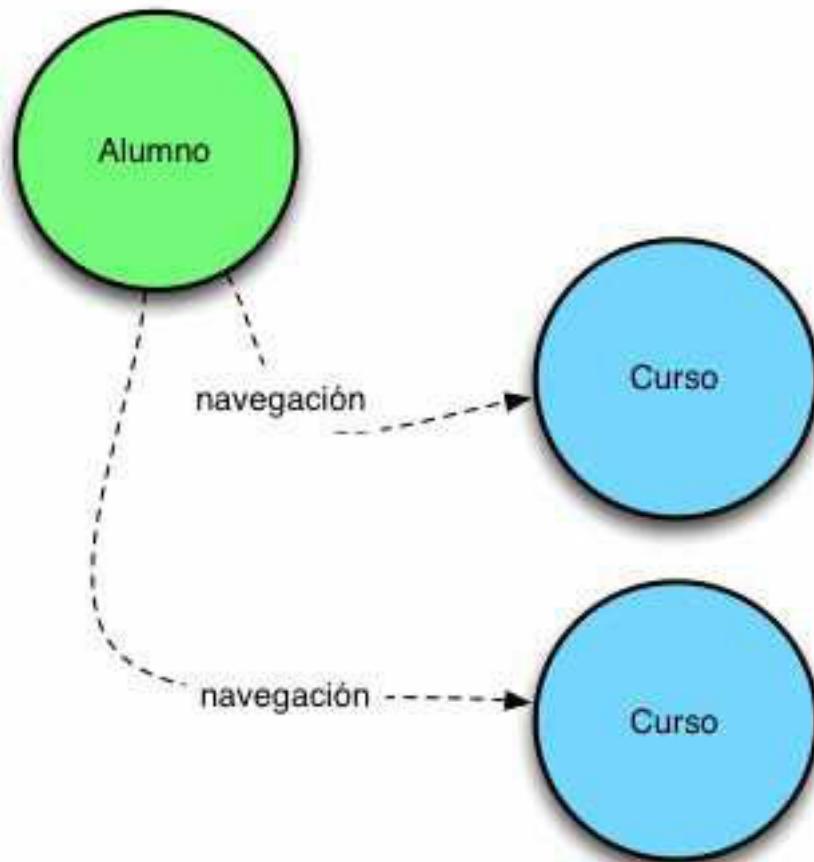
Java se encargará de imprimirmos por pantalla los nombres que en el capítulo anterior habíamos insertado en la base de datos.

maria

pedro

NAVEGACIÓN

Una de las características que tiene JPA es que mientras el EntityManager está abierto, tenemos la posibilidad de navegar entre las relaciones (**él se encargará de realizar las consultas que vaya necesitando**). Esto en algunas ocasiones será muy útil y en otras puede causar problemas .



Vamos a verlo en código :

```
TypedQuery<Alumno> alumnos= em.createQuery("select a from Alumno a ",Alumno.class);  
for (Alumno a :alumnos.getResultList()) {  
    System.out.println(a.getNombre());  
    for(Curso c: a.getCursos()) {  
        System.out.println(c.getTitulo());
```

```
}
```

```
}
```

En este caso primero, seleccionaremos los Alumnos vía JPA y más adelante Hibernate se encargara de realizar las consultas necesarias para obtener los cursos .Una vez hecho esto , imprimimos la información.

maria

Introduccion Java 2

Introduccion NET 2

pedro

Introducción a PHP

WHERE

La mayor parte de las cláusulas comunes SQL están soportadas por JPQL. Podemos seleccionar a los Alumnos cuyo nombre sea “pedro” utilizando la cláusula **where** y **asignando un parámetro**.

```
TypedQuery<Alumno> consultaAlumnos= em.  
createQuery("select a from Alumno a where           a.nombre=:nombre",Alumno.class);  
consultaAlumnos.setParameter("nombre", "pedro");
```

Esto nos seleccionará únicamente a pedro .

pedro

JOINS

JPA soporta también conceptos tales como el de Join .Podríamos ejecutar la siguiente consulta seleccionando los Alumnos realizando un Join con los Cursos .

```
TypedQuery<Alumno> alumnos=  
em.createQuery("select a from Alumno a join a.cursos c",  
Alumno.class);
```

Se imprimirá por pantalla lo siguiente :

maria

maria

pedro

El resultado no parece del todo correcto . Al realizar un join, JPA realiza un producto cartesiano y por lo tanto nos

aparecen 3 registros, cada uno de los cuales se mapea a un objeto

The screenshot shows a database grid with the following columns: #, dni1_0_, apellido2_0_, edad3_0_, and nombre4_0_. The data is as follows:

#	dni1_0_	apellido2_0_	edad3_0_	nombre4_0_
1	1	perez	25	maria
2	1	perez	25	maria
3	2	gomez	30	pedro

Para evitar este problema aplicaremos la cláusula distinct de JPA

```
TypedQuery<Alumno> alumnos=em.createQuery("select distinct a from Alumno a join a.cursos c",Alumno.class);
```

Ahora los datos se imprimirán de forma correcta

maria

pedro

Ahora bien **¿En que casos nos puede ser útil un Join?** . Esto es similar al acceso a una base de datos tradicional . Si queremos seleccionar a todos los alumnos que han realizado el curso que tiene de id “JAVA2” haremos la siguiente consulta.

```
TypedQuery<Alumno> alumnos=em.createQuery("select distinct a from Alumno a join a.cursos c where c.id=:id",Alumno.class);alumnos.setParameter("id", "JAVA2");
```

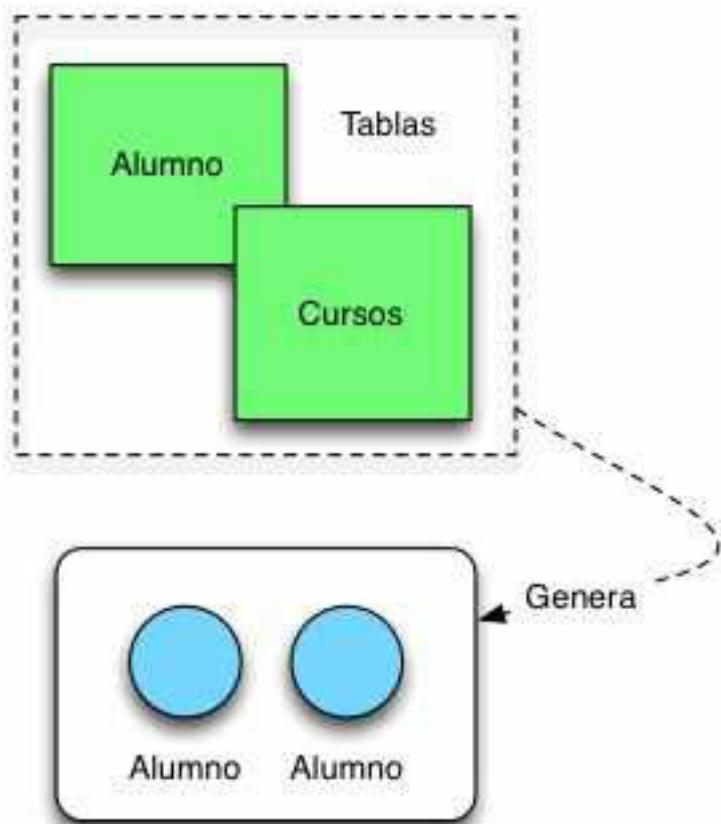
Y el resultado será :

maria

JOINS Y OBJETOS

Con frecuencia se cree que cuando se realiza un Join en JPA se ejecuta una consulta contra las dos tablas y se seleccionan todos los objetos de negocio .**Esto no es cierto** . Cuando se realiza un Join se realiza una consulta contra varias tablas pero al final únicamente se cargan en memoria los objetos indicados en la selección **en este caso los Alumnos**.

JPA JOIN



Habrá situaciones en las que esto nos sea útil y en otras en las cuales no tanto y queramos un enfoque diferente.

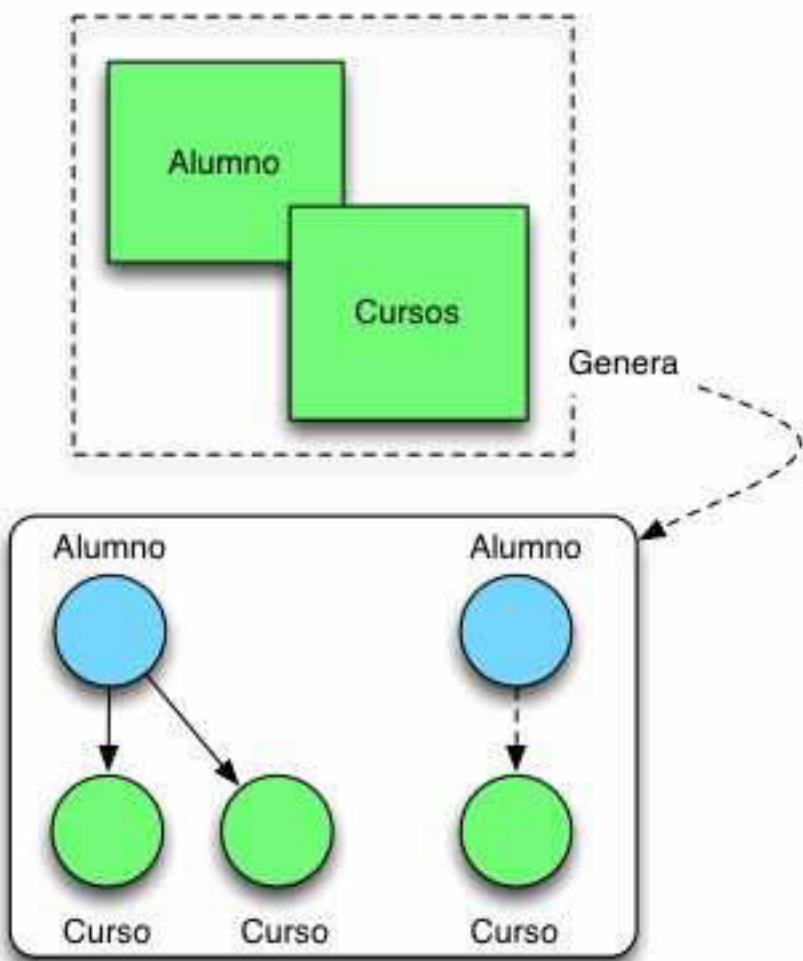
FETCH JOIN

Para solventar el problema anterior y que al realizar una operación de Join se seleccionen los objetos tanto de los Alumnos como de los Cursos, deberemos realizar un **fetch join**. He aquí un ejemplo:

```
TypedQuery<Alumno> alumnos= em.createQuery("select distinct a from Alumno a join fetch a.cursos c",Alumno.class);
```

En este caso la selección incluirá tanto los Alumnos como los Cursos en una estructura de grafo relacionada. De esta forma, con una única consulta podremos traernos todos los datos que necesitamos.

JPA FETCH JOIN



Esto nos devolverá el siguiente resultado para las tablas que tenemos con la ventaja de que únicamente habremos realizado una consulta.

maria

Introduccion Java 2

Introduccion NET 2

pedro

Introducción a PHP

AGREGADOS

En algunas ocasiones podemos encontrarnos con situaciones en las cuales tenemos que realizar consultas de agregación sobre nuestros objetos de negocio .En este caso podríamos por ejemplo necesitar saber cuál es el gasto total que ha habido en los cursos .

```
Query suma=
em.createQuery("select sum
(c.precio) from Curso c");
```

```
Double total=(Double) suma.getSingleResult();
```

```
System.out.println(total.intValue());
```

Como podemos ver, la consulta es muy similar a la de SQL clásica simplemente se usa la función SUM y nos devolverá la suma de todos los cursos.

850

RESUMEN

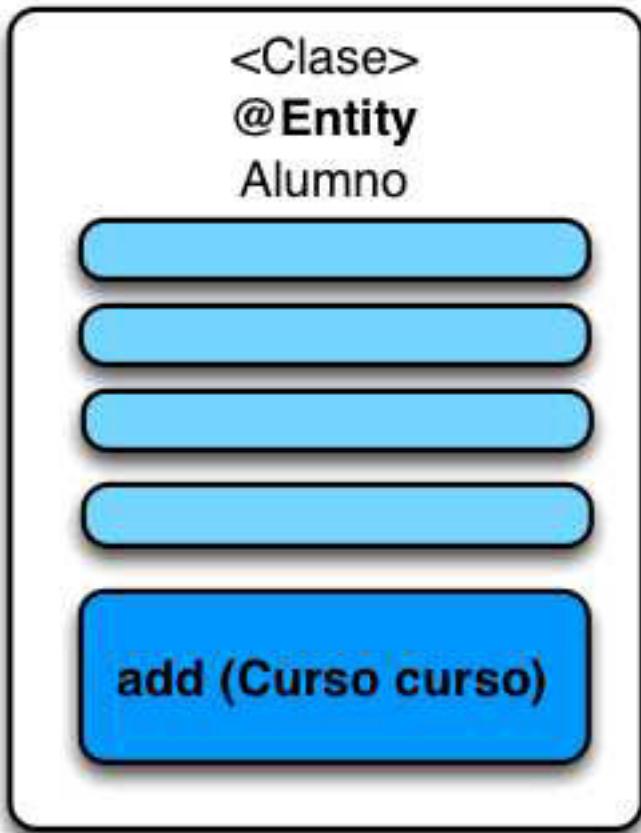
Hemos terminado de revisar a lo mas básico de JPQL a través de las consultas elementales , los Joins etc . Vamos a seguir avanzando con otros temas y abordar las reglas de negocio.

DESCARGAR EJEMPLOS:

<http://www.arquitecturajava.com/wp-content/uploads/004OneToManyManyToOneJPQL.zip>

Reglas de Negocio

Tenemos construida una relación entre dos clases pero en estos momentos se parece mucho a una relación entre dos tablas de la base de datos . **No hemos añadido ninguna regla de negocio que nos resulte útil a nuestro modelo de dominio.** Vamos a comenzar con ello y enriquecer nuestro modelo. Para ello vamos a ha añadir un método **add** a la clase Alumno que nos permite de forma sencilla añadirle Cursos.



Vamos a ver su código fuente :

```
public void add(Curso c) {  
    cursos.add(c);  
    c.setAlumno(this);  
}
```

La regla de negocio es sencilla , simplemente añadimos el Curso a la colección que tiene la clase Alumno y realizamos la operación inversa: asignamos al curso el Alumno también **ya que la relación es bidireccional** . Ahora bien, aquí se presentan los problemas: al añadir esta regla de negocio **estamos indicando al programador que puede usar el método para añadir Cursos al Alumno** . Sin embargo, cuando salvemos el Alumno en la base de datos, los **Cursos nuevos asociados no se salvarán** . Debemos corregir este problema.

CASCADE PERSIST

Para solucionar esta incidencia vamos a usar el atributo **Cascade** que soporta la anotación **@OneToMany** . Vamos a asignarle el valor de **CascadeType.Persist** que indica a JPA que no solo se salve el objeto de negocio actual sino que se también los objetos Curso pertenecientes a la relación.

```
@OneToMany(mappedBy="alumno",
```

```
cascade={CascadeType.PERSIST})
```

```
private List<Curso> cursos=
```

```
new ArrayList<Curso>();
```

De esta forma, será mas sencillo persistir los Objetos de negocio cuando están relacionados ,apoyándonos en nuestro nuevo método de negocio .Veámoslo:

```
Alumno pedro= new Alumno("1","pedro","gomez",30);
```

```
Alumno maria= new Alumno("2","maria","perez",25);
```

```
EntityManagerFactory emf=
```

```
Persistence.
```

```
createEntityManagerFactory("UnidadCurso");
```

```
EntityManager em=
```

```
emf.createEntityManager();
```

```
Curso cursoJava=
```

```
new Curso("JAVA2",
```

```
"Introduccion Java 2",20,300,pedro);
```

```
Curso cursoNET=
```

```
new Curso("NET2",
```

```
"Introduccion NET 2",20,300,pedro);
```

```
Curso cursoPHP=
```

```
new Curso("PHP",
```

```
"Introducción a PHP",15,250,maria);
```

```
pedro.add(cursoJava);
```

```
pedro.add(cursoNET);
```

```
maria.add(cursoPHP);
```

```
em.getTransaction().begin();
```

```
em.persist(pedro);
```

```
em.persist(maria);
```

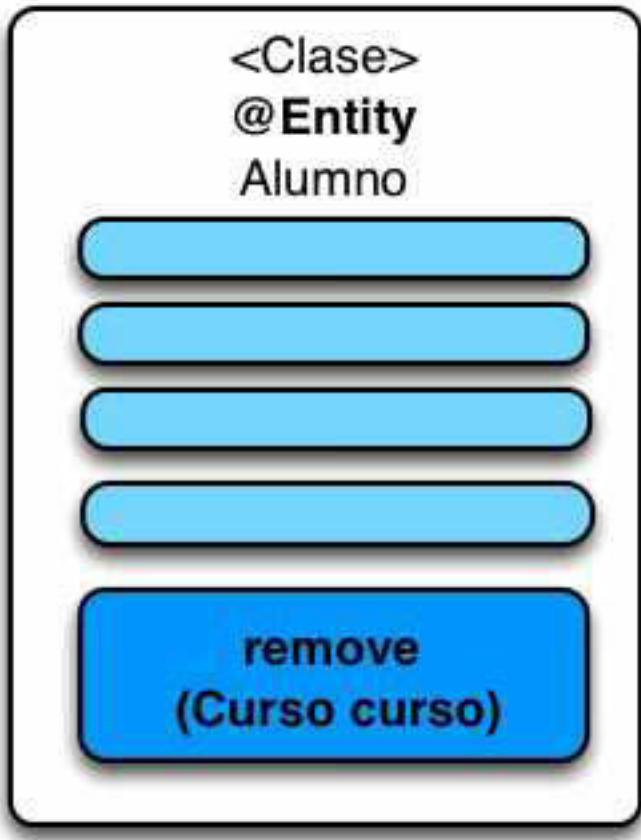
```
em.getTransaction().commit();
```

```
em.close();
```

Persistiendo los datos de Pedro y María los cursos asociados a ellos quedan automáticamente guardados.

CASCADE REMOVE

De igual forma que hemos añadido un método para añadir Cursos a un Alumno podemos hacer lo mismo con la funcionalidad de borrar y añadir un método remove en la clase.



Vamos a ver su código fuente:

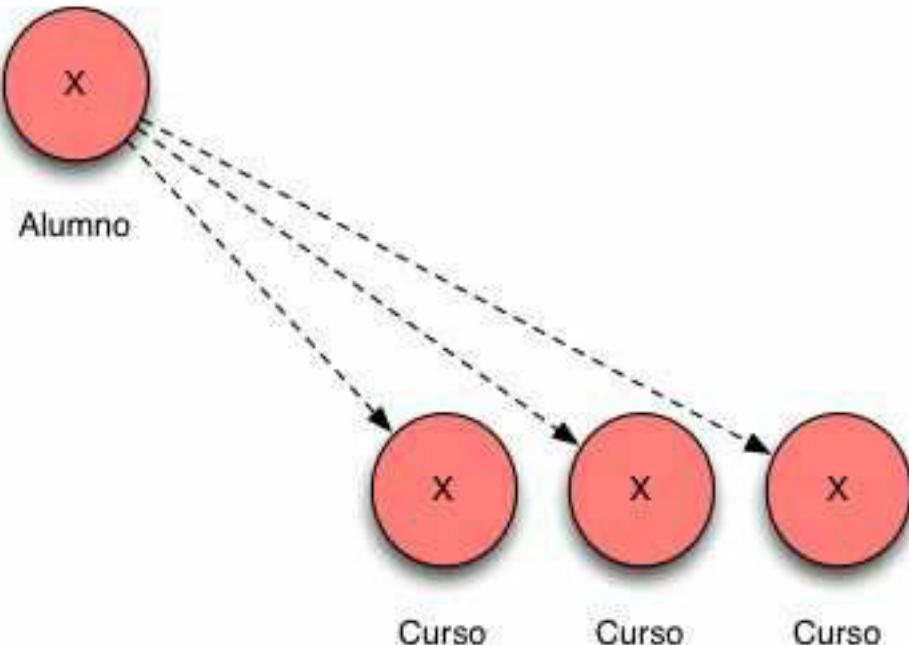
```
public void remove(Curso c)
{
    cursos.remove(c);
    c.setAlumno(null);
}
```

Al igual que en la situación anterior con insertar, en este caso deberemos modificar la relación para hacernos eco de la capacidad de borrar. Usaremos el atributo Cascade pero asignaremos otra propiedad adicional **CascadeType.REMOVE**., la cual indica que cuando borremos un Alumno, automáticamente deben ser borrados todos sus Cursos.

```
@OneToMany(mappedBy="alumno",
cascade={CascadeType.PERSIST,
CascadeType.REMOVE})
private List<Curso> cursos=
new ArrayList<Curso>();
```

ORPHAN REMOVAL

Ya disponemos de la funcionalidad necesaria a la hora de añadir y eliminar Cursos para un Alumno .Sin embargo, nuestro diseño tiene algunas limitaciones . **¿Podemos eliminar un único Curso de la lista de Cursos que el Alumno tiene? . La respuesta es NO.** Lo que podemos hacer es eliminar el Alumno y sus Cursos serán eliminados como se muestra en la siguiente figura.



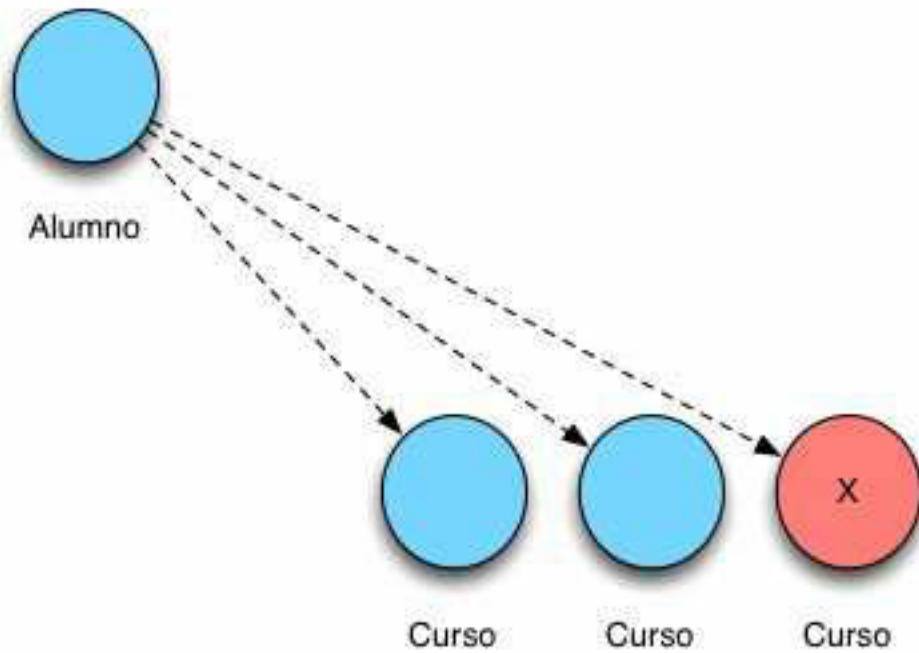
¿Qué utilidad tiene el método **remove** que acabamos de construir? . Pues en estos momentos **POCA**. Frecuentemente nos encontraremos con la necesidad necesidad de tener un Alumno y poder borrar uno de los Cursos que tiene asignados.

Si queremos eliminar algún elemento de la lista deberemos apoyarnos en otro de los atributos de JPA denominado **OrphanRemoval** y que permite eliminar todos los elementos que se hayan quedado huérfanos en una relación al ser eliminados de la lista.

Vamos a ver el código :

```
@OneToOne(mappedBy="alumno",
cascade={CascadeType.PERSIST,
CascadeType.REMOVE},
orphanRemoval=true)
private List<Curso> cursos=
new ArrayList<Curso>();
```

Ahora podremos eliminar elementos de la lista y serán borrados de la base de datos.



CASCADE.PERSIST vs CASCADE.DELETE

Hemos terminado de definir la relación entre **Alumno** y **Curso**. Sin embargo, nos queda terminar de definir la relación en el sentido contrario. Es aquí donde se necesita más reflexión previa ya que mientras que al eliminar un **Alumno** debemos eliminar también sus **Cursos**, **cuando eliminamos el Curso el Alumno no debiera ser eliminado**. Así pues en el **Curso** es muy apropiado **usar Cascade.Persist pero no Cascade.Delete**. Vamos a ver el código :

```

public class Curso implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private String id;
    private String titulo;
    private int duracion;
    private double precio;

    @ManyToOne(cascade=CascadeType.PERSIST)
    @JoinColumn(name="alumno_dni")
    private Alumno alumno;
}

```

RESUMEN

Ahora sí podemos dar por construida la relación entre **Alumno** y **Curso**. En el siguiente capítulo progresaremos en la construcción del modelo.

DESCARGAR EJEMPLOS

http://www.arquitecturajava.com/wp-content/uploads/005ReglasDeNegocio_1.zip

http://www.arquitecturajava.com/wp-content/uploads/006ReglasDeNegocio_2_Huerfanos.zip

Relaciones @ManyToMany

Ya comenté en los primeros ejemplos que el modelo que habíamos construido era realmente básico . Una relación entre Curso y Alumno es mas habitual que sea de n a n . Un Alumno realiza varios Cursos y un Curso es realizado por varios Alumnos. Así pues, vamos a modificar el modelo de tablas de la base de datos para poder construir algo más flexible.

Alumno	
dni	varchar(10)
nombre	varchar(20)
apellidos	varchar(20)
edad	int

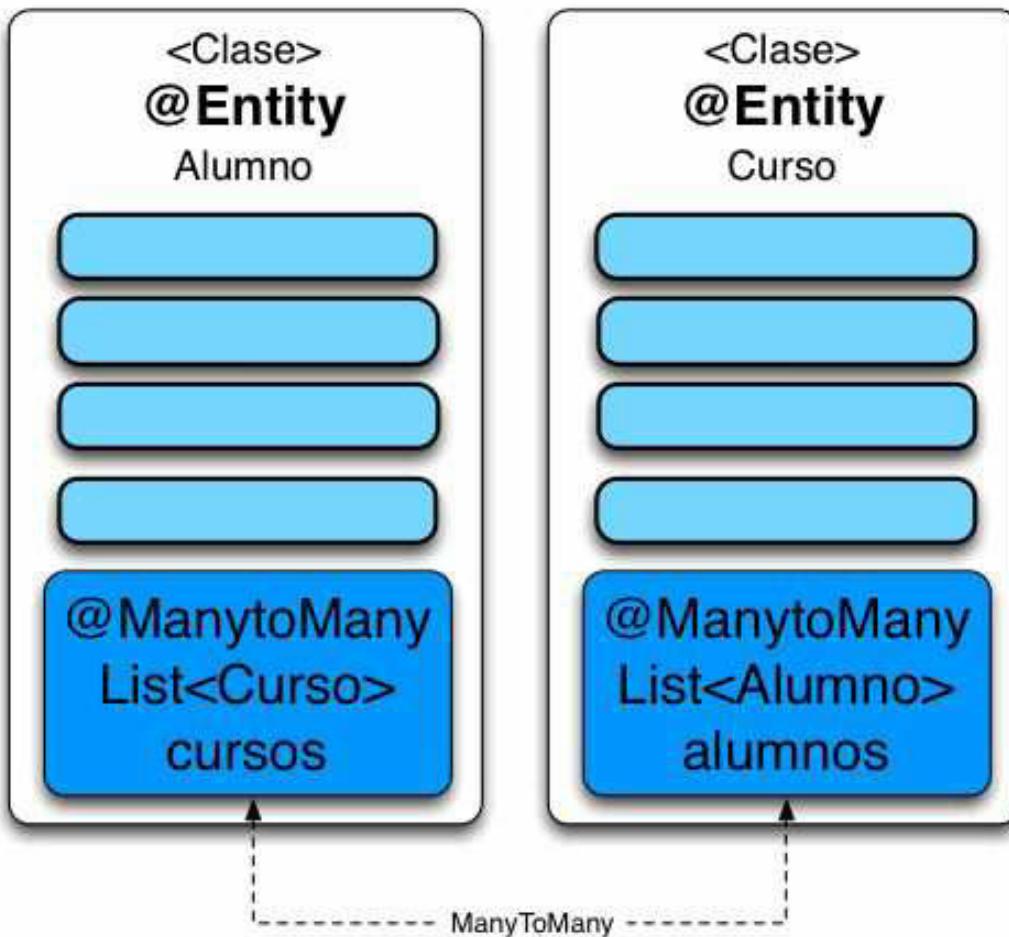
AlumnoCurso	
alumno_dni	varchar(10)
curso_id	varchar(10)

Curso	
id	varchar(10)
titulo	varchar(20)
duracion	varchar(20)
alumno_dni	varchar(10)

Una vez tenemos clara la relación entre las tablas a nivel de la base de datos, vamos a modificar nuestras clases Java para que puedan soportar una relación de este tipo.

@MANYToMANY

En este caso deberemos hacer uso de una nueva anotación que se encarga de definir las relaciones n a n . Esta anotación es **@ManyToMany**. No obstante, el modelo de Java será diferente al modelo de la base de datos ya que la tabla intermedia (AlumnoCurso) no aparecerá.



Vamos a comenzar a ver las modificaciones que tenemos que hacer al código de la clase Curso y cómo ha de ser modificada la relación para soportar las nuevas necesidades.

```

@ManyToMany
@JoinTable(name = "AlumnoCurso",
joinColumns = @JoinColumn(name = "curso_id"),
inverseJoinColumns =
@JoinColumn(name = "alumno_dni"))
private List<Alumno> alumnos
= new ArrayList<Alumno>();
    
```

@ManyToMany : Anotación que define una relación de muchos a muchos e incluye referencia a la tabla intermedia (**AlumnoCurso**) . Aparte de marcar la tabla, debe referenciar a las columnas que se encargan de marcar la relación en ambos sentidos (joinColumns, inverseJoinColumns). Por último, pasamos de tener una variable de tipo Alumno a un **List<Alumno>** con sus métodos set/get ya que la relación es de n a n. Es momento de ver la relación en el sentido contrario que es bastante más sencilla ya que como pasaba en casos anteriores, delegamos en la relación ya creada en la clase Curso .

```

@ManyToMany(mappedBy="alumnos")
private List<Curso> cursos=
new ArrayList<Curso>();
    
```

Realizadas estas modificaciones y añadidos los métodos set y get correspondientes, podemos crear un programa sencillo que las use:

```
package com.arquitecturajava.main;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.arquitecturajava.Alumno;
import com.arquitecturajava.Curso;

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

        Alumno pedro=
        new Alumno("1","pedro","gomez",30);

        Alumno maria=
        new Alumno("2","maria","perez",22);

        Curso cursoJava=
        new Curso("JAVA2","Introduccion Java",20,300);

        Curso cursoNET=
        new Curso("NET2","Introduccion NET",20,300);

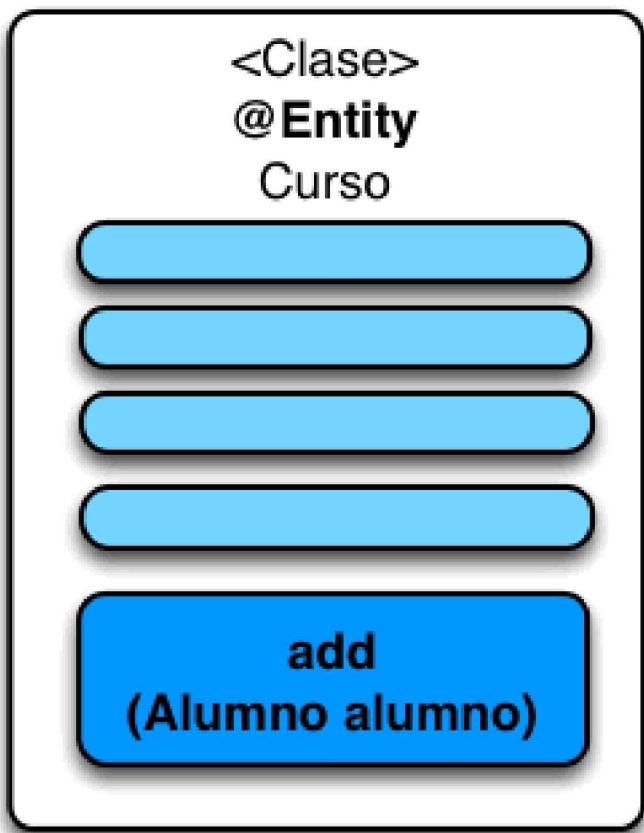
        pedro.getCurros().add(cursoJava);
        cursoJava.getAlumnos().add(pedro);
        pedro.getCurros().add(cursoNET);
        cursoNET.getAlumnos().add(pedro)
        maria.getCurros().add(cursoNET);
        cursoNET.getAlumnos().add(maria);

        EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("UnidadCurso");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        em.persist(pedro);
        em.persist(maria);
        em.persist(cursoJava);
        em.persist(cursoNET);
        em.getTransaction().commit();
        em.close();
        System.out.println("termino");
    }
}
```

Podemos ver como hemos relacionado de forma **bidireccional** ambos conceptos y persistido en la base de datos . Eso sí ,el código es bastante engorroso y hay que hacer demasiadas operaciones . Vamos a ver cómo podemos simplificarlo aplicando reglas de negocio.

REGLAS DE NEGOCIO Y @MANYToMANY

Vamos a modificar la clase Curso y Alumno para que puedan añadir métodos orientados a la gestión del negocio. Lo primero que haremos es añadir un método **add (Alumno a)** a la clase Curso que permite de una forma sencilla añadir Alumnos.



Vamos a ver el código fuente de este método a nivel de la clase Curso:

```
public void add(Alumno alumno) {  
    alumnos.add(alumno);  
    alumno.getCurso().add(this);  
}
```

Al tratarse de una relación bidireccional, deberemos asegurarnos de que el Alumno queda referenciado desde ambos lados de la relación.

@MANYToMANY CASCADE.PERSIST

Una vez hemos añadido este método tendremos que modificar la anotación **@ManyToMany** añadiendo la propiedad **CascadeType.CascadePersist** para asegurarnos de que, cuando salvemos un Curso todos los Alumnos asociados a él son salvados.

```
@ManyToMany(cascade=CascadeType.PERSIST)
```

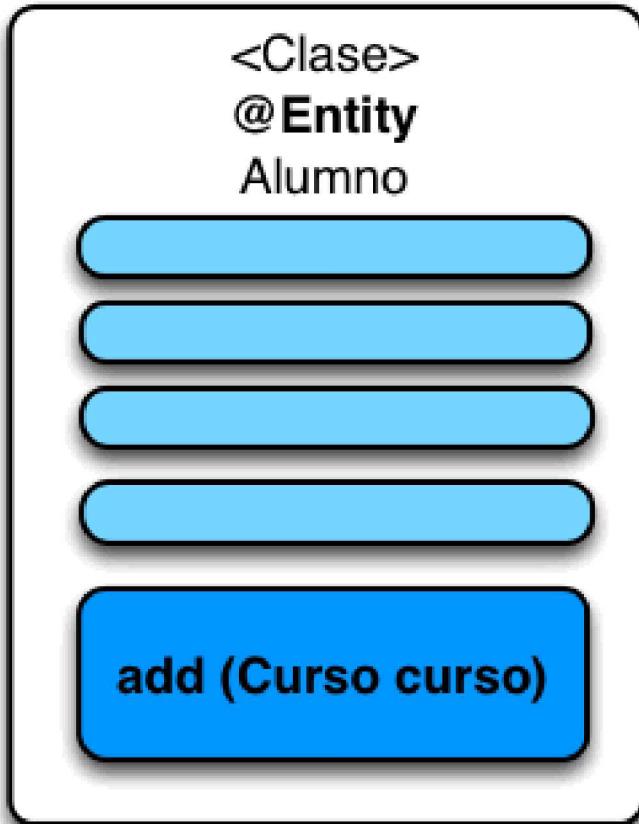
```
@JoinTable(name = "AlumnoCurso",
```

```

joinColumns =
    @JoinColumn(name = "curso_id"),
inverseJoinColumns =
    @JoinColumn(name = "alumno_dni"))
private List<Alumno> alumnos=new ArrayList<Alumno>();

```

Hecho esto deberemos realizar las mismas operaciones del lado de la clase Alumno que contendrá un método para añadir Cursos.



Añadimos una regla de negocio similar a la que teníamos anteriormente, solo que en este caso se encargará de añadir Cursos a la clase Alumno.

```

public void addCurso(Curso curso) {
    cursos.add(curso);
    curso.getAlumnos().add(this);
}

```

De la misma forma, modificaremos la anotación para que soporte de forma correcta la regla de negocio.

```

@ManyToMany(mappedBy="alumnos",
cascade=CascadeType.PERSIST)
private List<Curso> cursos=
new ArrayList<Curso>();

```

Hemos terminado de gestionar los métodos encargados de añadir Cursos y Alumnos a ambas clases. Veamos ahora como quedaría un resumen del código inicial del capítulo con los cambios aplicados.

```
package com.arquitecturajava.main;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.arquitecturajava.Alumno;
import com.arquitecturajava.Curso;

public class PrincipalInsertar {

    public static void main(String[] args) {

        Alumno pedro=
        new Alumno("1","pedro","gomez",30);

        Alumno maria=
        new Alumno("2","maria","perez",22);

        Curso cursoJava=
        new Curso("JAVA2","Introduccion Java",20,300);

        Curso cursoNET=
        new Curso("NET2","Introduccion NET",20,300);

pedro.add(cursoJava);
pedro.add(cursoNET);
maria.add(cursoNET);

        EntityManagerFactory emf =
        Persistence.
        createEntityManagerFactory("UnidadCurso");

        EntityManager em = emf.
        createEntityManager();
        em.getTransaction().begin();

em.persist(pedro);
em.persist(maria);

        em.getTransaction().commit();
        em.close();
        System.out.println("termino");
    }
}
```

Acabamos de cubrir la funcionalidad del método add . Seguidamente abordaremos la funcionalidad de borrado.

Cuando hemos tenido una relación de **@OneToMany** nos hemos apoyado en **Cascade.Remove** para eliminar a todos sus hijos debido a que había una relación de pertenencia. Ahora tenemos que tener más cuidado ya que al tratarse de una relación de n a n, el eliminar un Curso no implica eliminar un Alumno y viceversa: el eliminar un Alumno no implica eliminar un Curso . **Por lo tanto, no deberemos tratar este tipo de atributo en una relación de n a n como ésta.** Ahora bien, aunque no tratemos el atributo sí que podemos generar las reglas de negocio necesarias para eliminar los datos que se almacenan en la tabla intermedia AlumnoCurso. De esa manera si que queremos que un Curso deje de estar relacionado con un Alumno, esta operativa se pueda realizar. Vamos a añadir esta lógica de negocio a la clase Curso.

```
public void remove(Alumno alumno) {  
    alumnos.remove(alumno);  
    alumno.getCurso().remove(this);  
}
```

RESUMEN

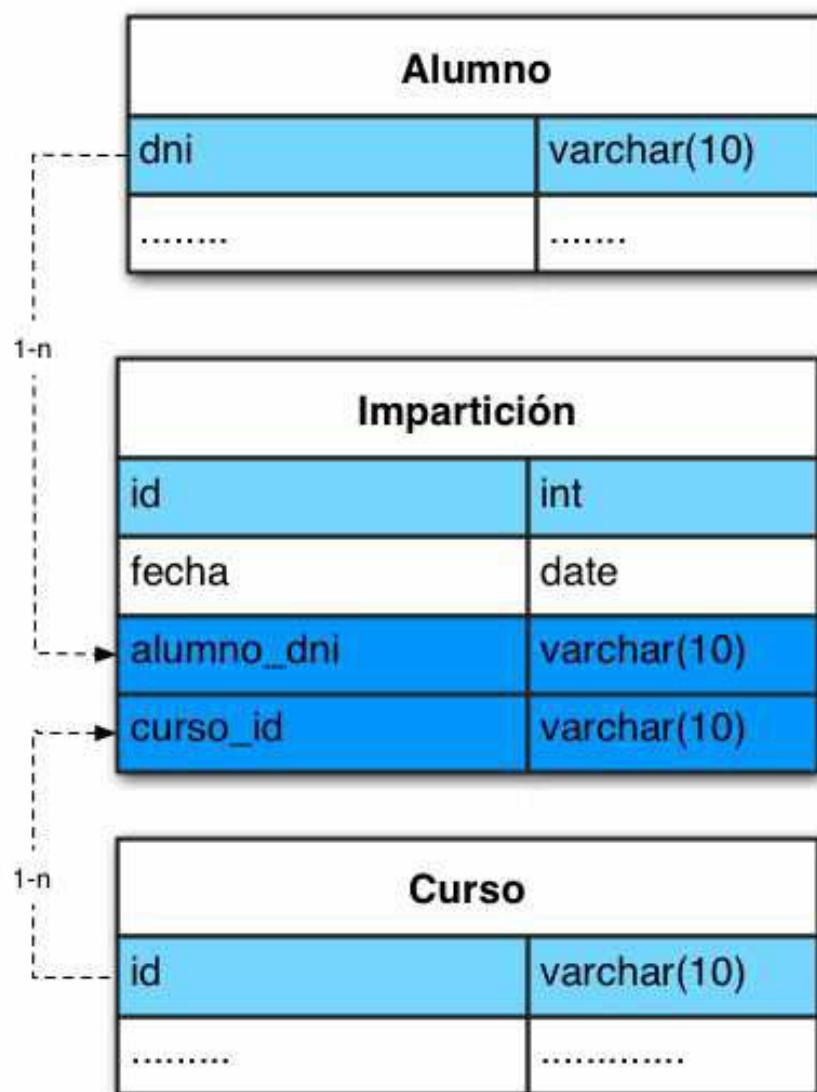
Acabamos de construir una relación sólida entre el concepto de Alumno y Curso clarificando el uso de las relaciones **@ManyToMany** .Es momento de hacer avanzar al modelo hacia relaciones más complejas.

DESCARGAR EJEMPLOS

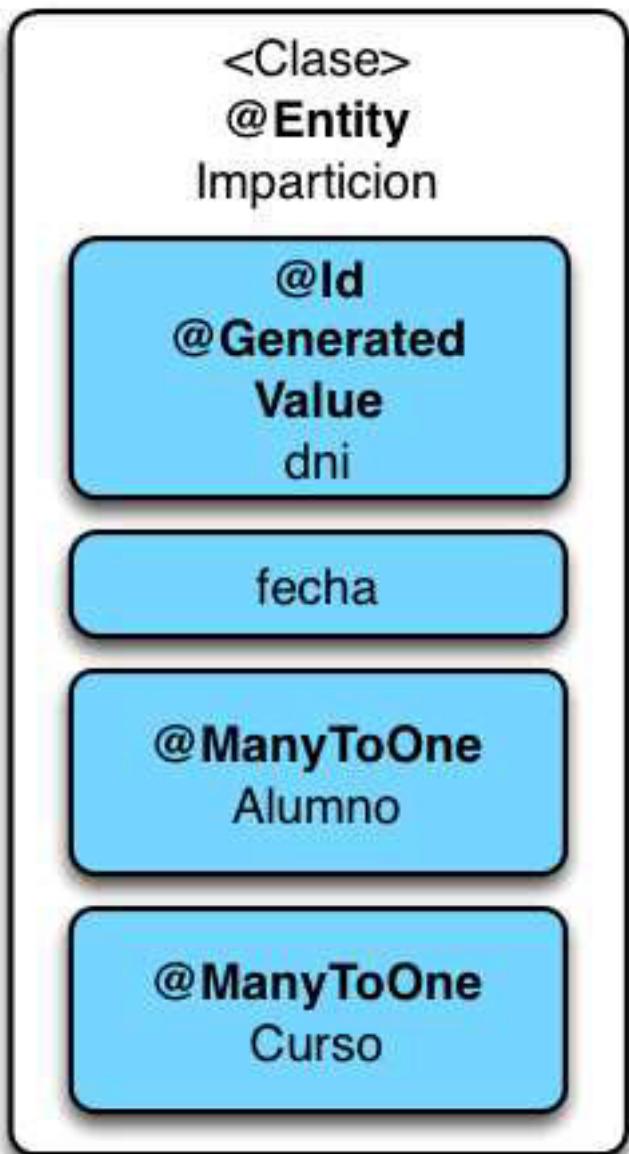
<http://www.arquitecturajava.com/wp-content/uploads/007RelacionesManyToMany.zip>
<http://www.arquitecturajava.com/wp-content/uploads/008JPARElacionesManyToManyReglasNegocio.zip>

Revisando @OneToMany

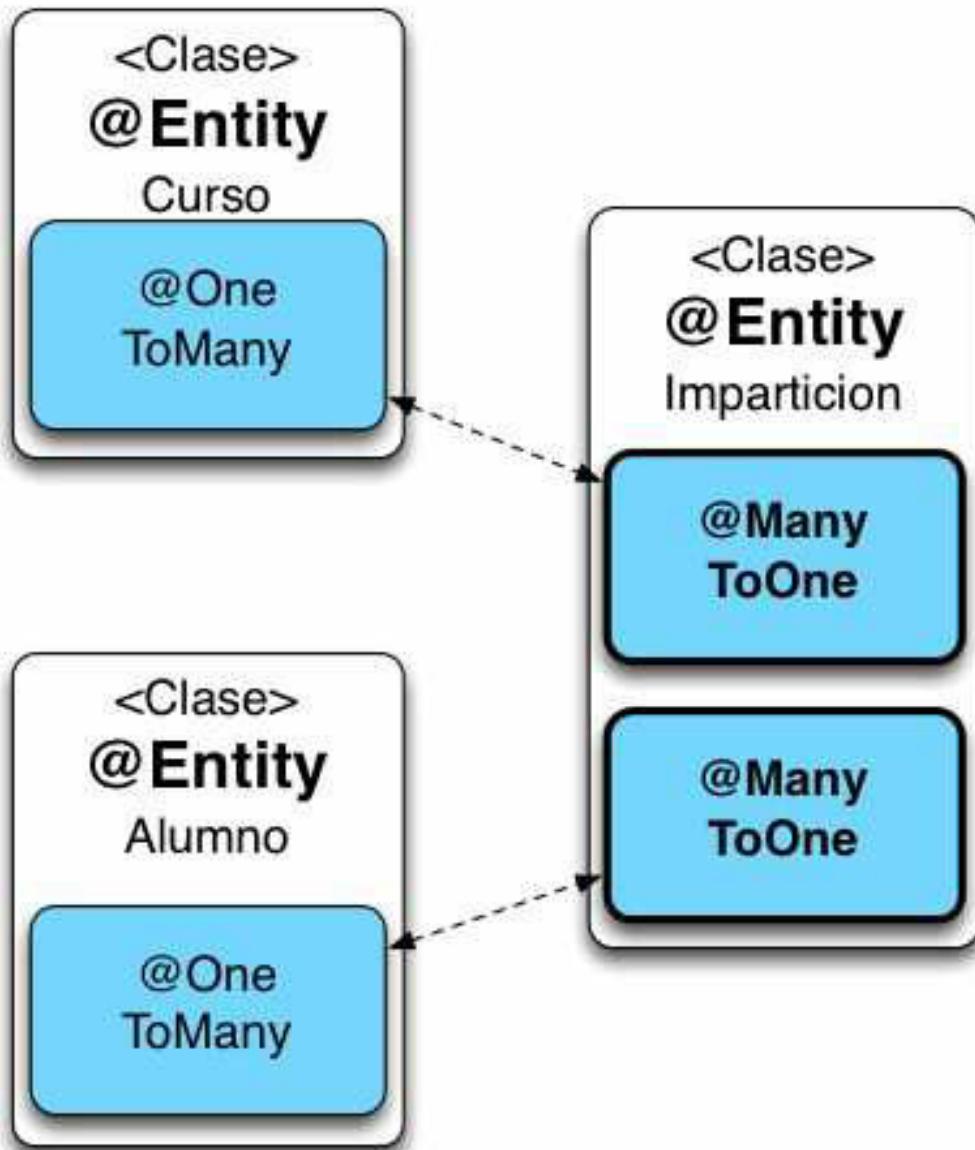
Realizar tareas de modelado es uno de los procesos más complejos de realizar ya que un Analista debe entender los distintos conceptos de negocio que un Cliente determinado tiene . En nuestro caso tenemos definida una relación de n a n entre Alumno y Curso . Sin embargo, la relación no es del todo correcta vamos a desarrollar esta idea.



Existe un concepto que ha quedado oculto y que es **el concepto de Impartición** . Un Alumno no asiste nunca a un Curso . Un Curso es un concepto abstracto que incluye un temario etc . **Un Alumno asiste a una Impartición del Curso la cual tendrá una fecha determinada** .Hemos creado el concepto de Impartición al cual hemos añadido dos campos adicionales .En primer lugar una clave primaria (Id) que será autoincremental y en segundo lugar la fecha en la que esta impartición se realiza. Vamos a ver cómo queda el concepto definido a nivel de clase



Aunque nos parece un cambio pequeño, en el modelo de base de datos, tendrá fuertes implicaciones en el modelo de dominio ya que la relación **@ManyToMany** desaparecerá y será substituida por dos relaciones **@OneToMany**, una en Alumno y otra en Curso.



Así pues, el resultado final es que la clase **Impartición** incluirá dos relaciones una con **Curso** y otra con **Alumno**.

Además incluirá la anotación **@GeneratedValue** anotación se encarga de decirle a JPA que la clave primaria de la tabla asociada a esta clase es de tipo autoincremental. Vamos a ver el código fuente de la clase para despejar dudas:

```

package com.arquitecturajava;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

@Entity
public class Imparticion implements Serializable {

```

```
private static final long serialVersionUID = 1L;

@Id
@GeneratedValue

private int id;

private Date fecha;

@ManyToOne(cascade=CascadeType.PERSIST)
@JoinColumn(name="alumno_dni")

private Alumno alumno;

@ManyToOne(cascade=CascadeType.PERSIST)
@JoinColumn(name="curso_id")

private Curso curso;

public Imparticion() {

    super();
}

public Imparticion(Date fecha, Alumno alumno, Curso curso) {

    super();
    this.fecha = fecha;
    setCurso(curso);
    setAlumno(alumno);
}

public int getId() {

    return id;
}

public void setId(int id) {

    this.id = id;
}

public Date getFecha() {

    return fecha;
}

public void setFecha(Date fecha) {

    this.fecha = fecha;
}

public Alumno getAlumno() {

    return alumno;
}
```

```

public void setAlumno(Alumno alumno) {
    this.alumno = alumno;
    alumno.getImparticiones().add(this);
}

public Curso getCurso() {
    return curso;
}

public void setCurso(Curso curso) {
    this.curso = curso;
    curso.getImparticiones().add(this);
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + id;
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Imparticion other = (Imparticion) obj;
    if (id != other.id)
        return false;
    return true;
}
}

```

Después de haber leído los capítulos anteriores el código es bastante sencillo de entender. Disponemos de dos relaciones `@ManyToOne` bastante habituales. Quizá la única novedad es el campo fecha que es de tipo Date . JPA se encargará de persistirlo de forma correcta contra la base de datos. Ahora bien ¿**Cómo quedan redefinidas las dos relaciones `@ManyToMany` que teníamos en las clases Alumno y Curso?** . Vamos a ver las modificaciones en cuanto a la clase Alumno

```

@OneToOne(mappedBy = "alumno",
cascade = { CascadeType.PERSIST,

```

```

CascadeType.REMOVE},
orphanRemoval = true)
private List<Imparticion> imparticiones
= new ArrayList<Imparticion>();

```

En este caso usaremos una anotación @OneToMany y tanto CascadeType.PERSIST como CascadeType.REMOVE, ya que si borramos el Alumno las Imparticiones a las que asistió deberían ser borradas. Añadiremos también los métodos de negocio clásicos asociados.

```

public void add(Imparticion imparticion) {
    imparticiones.add(imparticion);
    imparticion.setAlumno(this);
}

public void remove(Imparticion imparticion) {
    imparticiones.remove(imparticion);
    imparticion.setAlumno(null);
}

```

Acabamos de ver la relación de Alumno con Impartición. Seguidamente trataremos la relación complementaria entre Curso e Impartición.

```

@OneToOne(mappedBy = "curso",
cascade={CascadeType.PERSIST,
CascadeType.REMOVE},
orphanRemoval=true)
private List<Imparticion> imparticiones
= new ArrayList<Imparticion>();

```

La relación es prácticamente idéntica a la casuística anterior, únicamente que en este caso se trata de Curso e Impartición lo mismo sucederá con los métodos a nivel de negocio .

ALUMNO, IMPARTICIÓN Y CURSO

Es hora de ver cómo los tres conceptos que hemos construido se relacionan a nivel de código. En el siguiente ejemplo crearemos un Curso y un Alumno y construiremos una Impartición nueva a la cual ambos están asociados.

```

package com.arquitecturajava.main;
import java.util.Date;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;

```

```

import javax.persistence.Persistence;

import com.arquitecturajava.Alumno;
import com.arquitecturajava.Curso;
import com.arquitecturajava.Imparticion;

public class PrincipalInsertarTodo {

    public static void main(String[] args) {

        EntityManagerFactory emf =
            Persistence.
                createEntityManagerFactory("UnidadCurso");
        EntityManager em = emf.
            createEntityManager();
        Alumno maria=
            new Alumno("1","maria","perez",25);
        Alumno pedro=
            new Alumno ("2","pedro","gomez",30);
        Curso cursoJava=
            new Curso("JAVA2","Introduccion Java",20,300);
        Curso cursoNET=
            new Curso("NET2","Introduccion NET 2",20,300);
        Curso cursoPHP =
            new Curso("PHP","Introducción a PHP",15,250);
        em.getTransaction().begin();
        Imparticion imparticion=
            new Imparticion(new Date(),pedro,curs oJava);
        Imparticion imparticion2=
            new Imparticion(new Date(),pedro,curs oNET);
        Imparticion imparticion3=
            new Imparticion(new Date(),maria,curs oPHP);

        em.persist(imparticion);
        em.persist(imparticion2);
        em.persist(imparticion3);
        em.getTransaction().commit();
        em.close();
        System.out.println("termino");
    }
}

```

JPA MULTIPLE FECH JOINS

Realizada esta operación, vamos a ver una operación complementaria que nos permita a través de varios **fetch Joins** seleccionar todos los registros con una única consulta.

```
package com.arquitecturajava.main;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;
import com.arquitecturajava.Imparticion;

public class PrincipalConsultaFetch {

    public static void main(String[] args) {

        EntityManagerFactory emf =
            Persistence.
            createEntityManagerFactory("UnidadCurso");
        EntityManager em = emf.createEntityManager();
        TypedQuery<Imparticion> imparticiones =
            em.createQuery("select i from Imparticion i
join fetch i.alumno a join fetch i.curso c",
            Imparticion.class);

        for(Imparticion i :
            imparticiones.getResultList()){
            System.out.println(i.getAlumno()
                .getNombre());
            System.out.println(i.getCurso()
                .getTitulo());
            System.out.println(i.getCurso()
                .getFecha());
        }
    }
}
```

El resultado será el siguiente:

pedro

Introduccion Java

20

pedro

Introduccion NET 2

20

maria

Introducción a PHP

15

RESUMEN

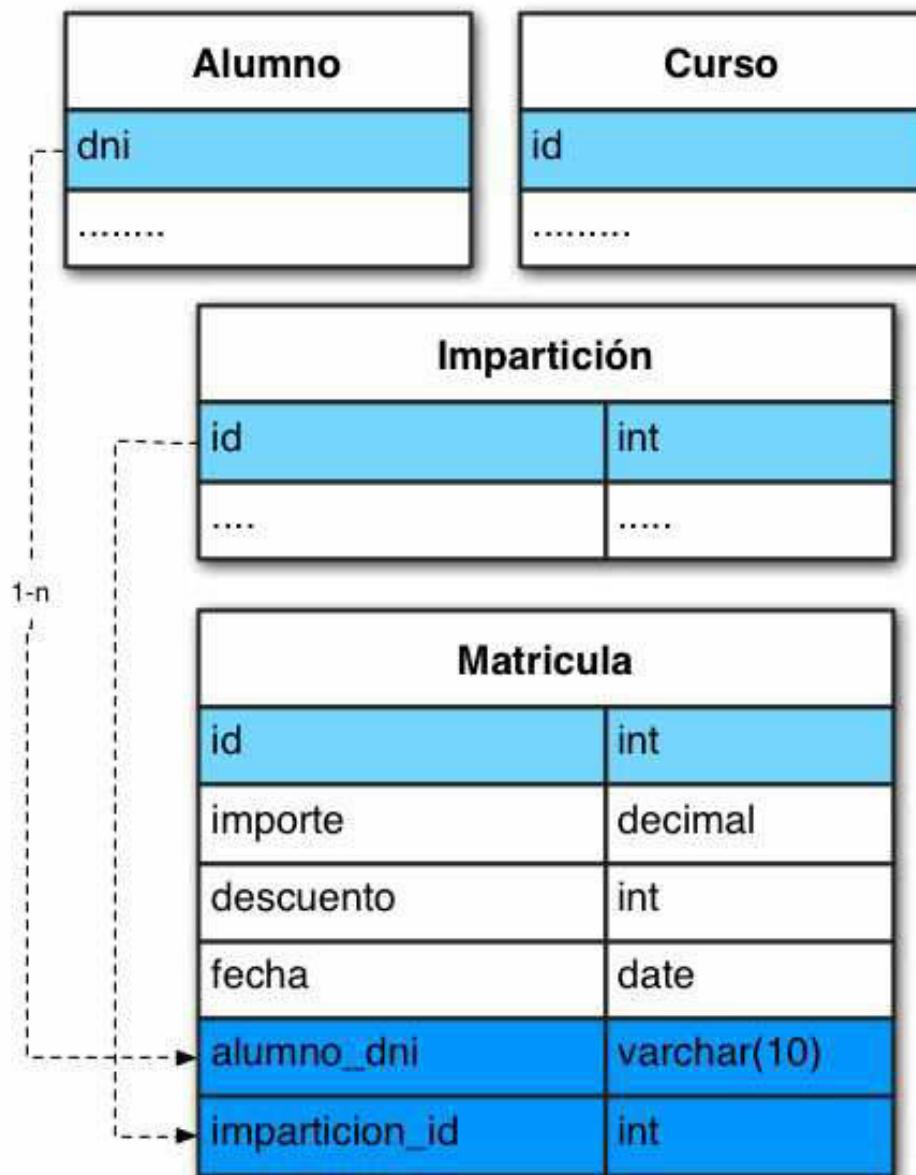
Hemos terminado de ver como se construyen relaciones más complejas. Sin embargo, todavía podemos detectar problemas en el modelo .En estos momentos una Impartición solo puede estar asociada a un Alumno determinado .Parece claro que esto se puede mejorar pues Impartición suele tener varios Alumnos .Vamos a abordarlo en el siguiente capítulo.

DESCARGAR EJEMPLOS

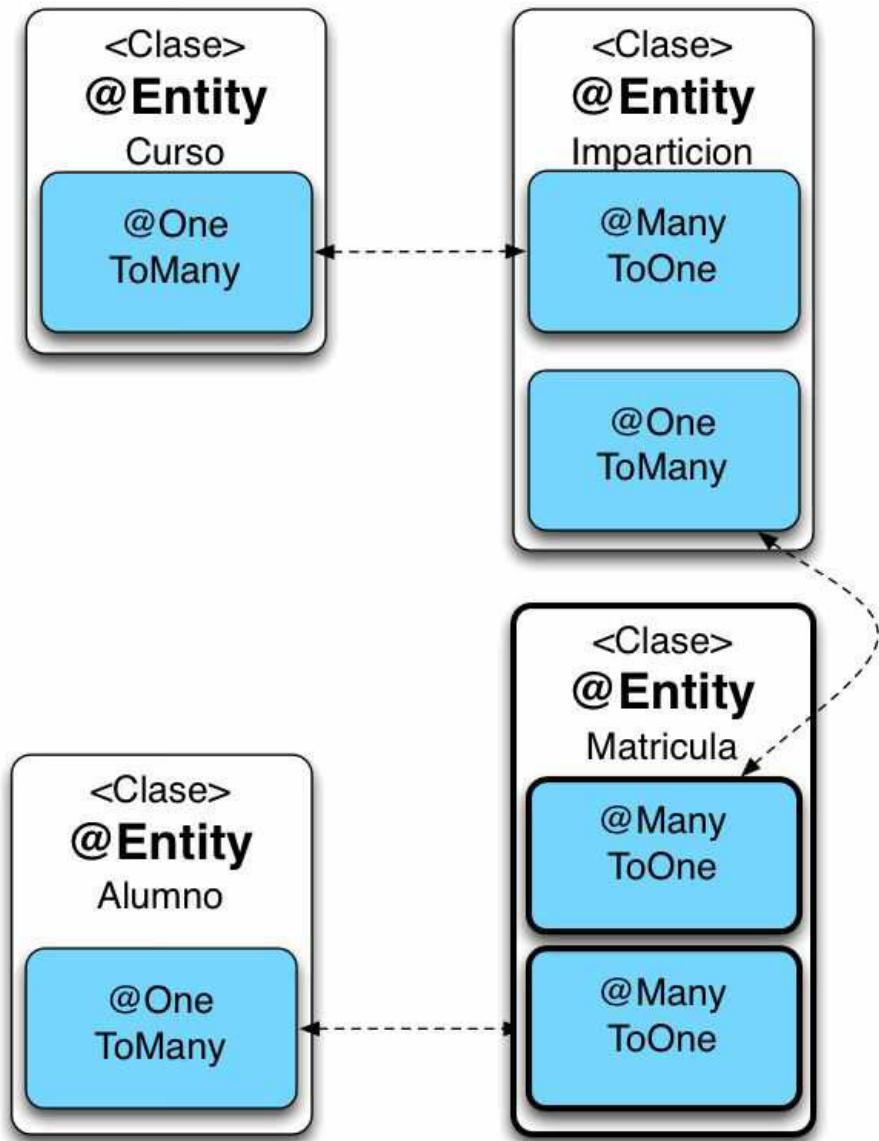
<http://www.arquitecturajava.com/wp-content/uploads/009JPAAAlumnoImparticionGenerateValue.zip>

Relaciones Ternarias

Aunque hemos hecho evolucionar bastante el modelo, todavía nos quedan conceptos que ir aportando a fin de que el modelo mejore y se acerque mas a la realidad . En este capítulo vamos a modificar el modelo para que soporte el concepto de Matrícula . Un Alumno puede matricularse en una Impartición de cada uno de los distintos Cursos que se ofrezcan. Así pues tendremos que modificar el modelo para que encaje con lo siguiente:



Una vez tenemos claro el modelo entidad relación es momento de replicarlo en con el modelo de dominio, creando el concepto de Matrícula a nivel de clases Java y construyendo las relaciones pertinentes.



Vamos a ver el código fuente de la Matricula:

```

package com.arquitecturajava;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

@Entity
public class Matricula implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private int id;
}

```

```
private Date fecha;  
private double importe;  
private int descuento;  
  
@ManyToOne(cascade=CascadeType.PERSIST)  
@JoinColumn(name="imparticion_id")  
private Imparticion imparticion;
```

```
@ManyToOne(cascade=CascadeType.PERSIST)  
@JoinColumn(name="alumno_dni")  
private Alumno alumno;
```

```
public Matricula() {  
    super();  
}
```

```
public Matricula(Date fecha, double importe,  
int descuento,  
Imparticion imparticion, Alumno alumno) {
```

```
    this.fecha = fecha;  
    this.importe = importe;  
    this.descuento = descuento;  
    setAlumno(alumno);  
    setImparticion(imparticion);  
}
```

```
public int getId() {  
    return id;  
}  
public void setId(int id) {  
    this.id = id;  
}  
public Date getFecha() {  
    return fecha;  
}  
public void setFecha(Date fecha) {  
    this.fecha = fecha;  
}  
public double getImporte() {  
    return importe;  
}
```

```
public void setImporte(double importe) {
    this.importe = importe;
}

public int getDescuento() {
    return descuento;
}

public void setDescuento(int descuento) {
    this.descuento = descuento;
}

public Imparticion getImparticion() {
    return imparticion;
}

public void setImparticion(Imparticion imparticion) {
    this.imparticion = imparticion;
    imparticion.getMatriculas().add(this);
}

public Alumno getAlumno() {
    return alumno;
}

public void setAlumno(Alumno alumno) {
    this.alumno = alumno;
    alumno.getMatriculas().add(this);
}

public double getImporteFinal() {

    return importe *(1-descuento/100);
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + id;
    return result;
}

@Override
public boolean equals(Object obj) {
    if(this == obj)
        return true;
    if(obj == null)
        return false;
    if(getClass() != obj.getClass())
        return false;
    return true;
}
```

```

Matricula other = (Matricula) obj;
if(id != other.id)
    return false;
return true;
}
}

```

En el concepto de Matricula podemos ver como éste está relacionado con Alumno y con Impartición a través de dos relaciones `@ManyToOne`

```

@ManyToOne(cascade=CascadeType.PERSIST)
@JoinColumn(name="imparticion_id")
private Imparticion imparticion;

@ManyToOne(cascade=CascadeType.PERSIST)
@JoinColumn(name="alumno_dni")
private Alumno alumno;

```

MATRICULA Y REGLAS DE NEGOCIO

A parte de esta funcionalidad, tenemos el método de negocio `getImporteFinal()` que se encarga de calcularnos el importe final de la matricula a partir del importe menos el descuento.



Éste es un método adicional de negocio que necesitamos para calcular el coste total a pagar por la Matricula ,es algo que evidentemente no pertenece al modelo entidad relación sino al de dominio .

```

public double getImporteFinal() {
    return importe *(1-descuento/100);
}

```

IMPARTICIÓN Y MATRICULA

Es momento de revisar el concepto de Impartición y cómo se relaciona con el resto. Lo primero que tendremos que hacer es **eliminar la relación existente con Alumno**, incluyendo sus métodos set/get/add/remove y las reglas de negocio asociadas. Una vez hecho esto, **añadiremos una nueva relación @OneToMany con Matricula**.

```

@OneToOne(mappedBy="matricula",
cascade={CascadeType.PERSIST,CascadeType.REMOVE},
orphanRemoval=true)
private List<Matricula> matricula;

```

Las Matriculas dependen completamente de las Imparticiones ,de ahí que si eliminamos una Impartición, deberemos borrar también sus Matriculas.Ademas de añadir las capacidades de persistencia de JPA, evidentemente añadiremos las reglas de negocio habituales para añadir y eliminar Matriculas.

```
public void add(Matricula m) {
```

```
    matriculas.add(m);
```

```
    m.setImparticion(this);
```

```
}
```

```
public void remove(Matricula m) {
```

```
    matriculas.remove(m);
```

```
    m.setImparticion(null);
```

```
}
```

Hemos terminado de realizar las modificaciones pertinentes en la clase Impartición. Nos queda ver la clase Alumno detalladamente. Esta clase deberá ahora contener una relación de tipo @oneToMany con Matricula.

```
@OneToOne(mappedBy="alumno"
```

```
,cascade={CascadeType.PERSIST,CascadeType.REMOVE},
```

```
orphanRemoval=true)
```

```
private List<Matricula> matriculas;
```

Igualmente deberá incluir las reglas de negocio para gestionarla:

```
public void add(Matricula matricula) {
```

```
    matriculas.add(matricula);
```

```
    matricula.setAlumno(this);
```

```
}
```

```
public void remove(Matricula matricula) {
```

```
    matriculas.remove(matricula);
```

```
    matricula.setAlumno(null);
```

```
}
```

Vamos a ver cómo podemos gestionar todos los conceptos a nivel de JPA para insertar el modelo de objetos completo.

```
package com.arquitecturajava.main;
```

```
import java.util.Date;
```

```
import javax.persistence.EntityManager;
```

```
import javax.persistence.EntityManagerFactory;
```

```
import javax.persistence.Persistence;
```

```

import com.arquitecturajava.Alumno;
import com.arquitecturajava.Curso;
import com.arquitecturajava.Imparticion;
import com.arquitecturajava.Matricula;

public class PrincipalInsertar {

    public static void main(String[] args) {

        Alumno pedro=
        new Alumno("1","pedro","gomez",30);

        Curso cursoJava=
        new Curso("JAVA2","Introduccion Java",20,300);

        Imparticion imparticion=
        new Imparticion(new Date(),cursoJava);

Matricula matricula=
new Matricula(new Date(),300,0,imparticion ,pedro);

        EntityManagerFactory emf =
        Persistence.
        createEntityManagerFactory("UnidadCurso");

        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();

em.persist(pedro);
em.persist(cursoJava);
em.persist(imparticion);
em.persist(matricula);

        em.getTransaction().commit();
        em.close();
        System.out.println("termino");

    }
}

```

RESUMEN

Hemos terminado de gestionar la clase Matricula y ahora disponemos de un modelo de dominio más sólido en donde encajan con naturalidad los cuatro conceptos Alumno, Impartición,Curso,Matricula.

DESCARGAR EJEMPLOS

<http://www.arquitecturajava.com/wp-content/uploads/010JPAAlumnoMatricula.zip>

Relaciones @OneToOne

Ya disponemos de cuatro conceptos en nuestro modelo : **Alumno**, **Impartición**, **Matricula** y **Curso**. Entre estos elementos hemos cubierto las relaciones fundamentales **@OneToMany**, **@ManyToOne** y **@ManyToMany** gestionado también las reglas de negocio que necesitábamos. Vamos a ocuparnos de un caso un poco mas especial que a veces se da en modelos de dominio , las relaciones **uno a uno** . Para ello, vamos a añadir un nuevo concepto al modelo : **el concepto de Pago** . Una Matricula puede incluir un Pago y un Pago pertenece obligatoriamente a una Matricula. Vamos a ver como se amplía el modelo:

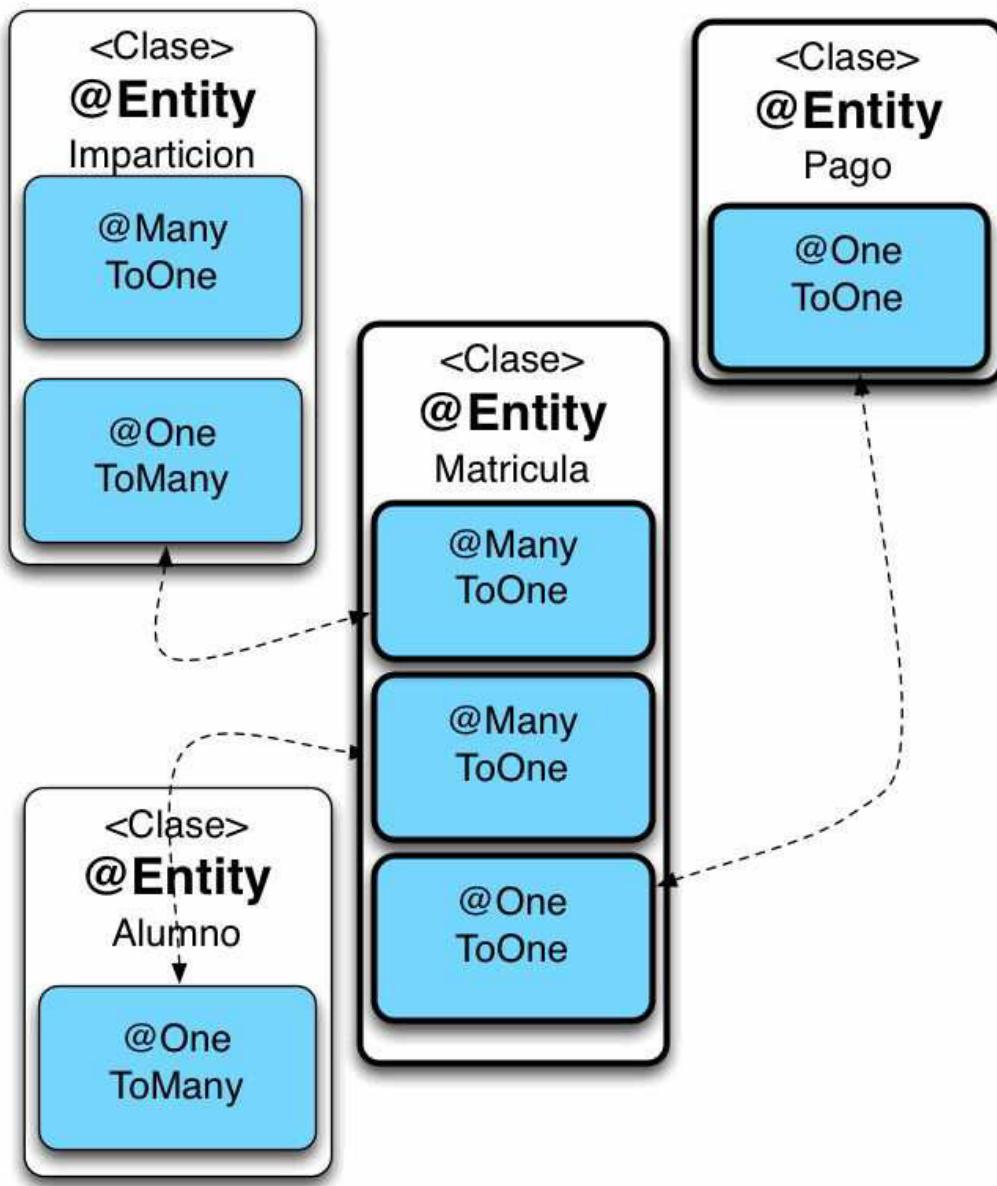
Matricula	
id	int
....
alumno_dni	varchar(10)
imparticion_id	int

Pago	
1 -1	
id	int
fecha	date
cuenta	varchar(20)
tarjeta	varchar(16)
codigoseguridad	varchar(4)
matricula_id	int

Aunque en el modelo entidad relación se trata de una relación normal y corriente de tipo 1 a n , es evidente que el modelo está limitado, ya que un Pago pertenece a una Matricula y una Matricula únicamente soporta un Pago y se trata de una relación 1 a 1. El Pago es un concepto sencillo : incluye la fecha , cuenta corriente o tarjeta de crédito con su código de seguridad .No necesitamos mucho más ya que los importes los almacena la Matricula. Vamos a ver a continuación cómo trata JPA las relaciones 1 a 1.

@ONEToONE

JPA soporta la anotación **@OneToOne** que es la encargada de definir las relaciones 1 a 1 entre dos conceptos del modelo de dominio. Vamos a ver cómo usarla al definir el concepto de Pago . Nuestro modelo de dominio evolucionará de la



siguiente forma:

Como podemos ver la relación de **@OneToOne** es **bidireccional y afecta tanto a la Matricula como al nuevo concepto de Pago**. Vamos a ver el código de la clase Pago:

```

package com.arquitecturajava;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;

@Entity
public class Pago implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id

```

```
private int id;
private Date fecha;
private String numeroCuenta;
private String numeroTarjeta;
private String codigoTarjeta;

@OneToOne (cascade=CascadeType.PERSIST)
@JoinColumn(name="matricula_id")
private Matricula matricula;

public Pago() {
    super();
}

public Pago(int id, Date fecha,
           String numeroCuenta, String numeroTarjeta,
           String codigoTarjeta, Matricula matricula) {
    super();
    this.id = id;
    this.fecha = fecha;
    this.numeroCuenta = numeroCuenta;
    this.numeroTarjeta = numeroTarjeta;
    this.codigoTarjeta = codigoTarjeta;
    setMatricula(matricula);
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public Date getFecha() {
    return fecha;
}

public void setFecha(Date fecha) {
    this.fecha = fecha;
}

public String getNumeroCuenta() {
    return numeroCuenta;
}

public void setNumeroCuenta(String numeroCuenta) {
```

```
        this.numeroCuenta = numeroCuenta;
    }

    public String getNumeroTarjeta() {
        return numeroTarjeta;
    }

    public void setNumeroTarjeta(String numeroTarjeta) {
        this.numeroTarjeta = numeroTarjeta;
    }

}

public String getCodigoTarjeta() {
    return codigoTarjeta;
}

public void setCodigoTarjeta(String codigoTarjeta) {
    this.codigoTarjeta = codigoTarjeta;
}

public Matricula getMatricula() {
    return matricula;
}

public void setMatricula(Matricula matricula) {
    if(!pago.getMatricula().equals(this))
        pago.setMatricula(this);
}

public double getImporte() {
    return matricula.getImporteFinal();
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + id;
    return result;
}

@Override
public boolean equals(Object obj) {
    if(this == obj)
        return true;
    if(obj == null)
        return false;
    if(getClass() != obj.getClass())
        return false;
    return true;
}
```

```

Pago other = (Pago) obj;
if(id != other.id)
    return false;
return true;
}
}

```

RELACIONES INVERSAS

La relación es sencilla de construir y es muy similar a una de la relación @ManyToOne con la que hemos trabajado bastante. Ahora bien vamos a ver la relación inversa entre Matrícula y Pago que es almacenada por la Matrícula.

```

@Entity
public class Matricula implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private int id;

    private Date fecha;

    private double importe;

    private int descuento;

    @ManyToOne(cascade=CascadeType.PERSIST)
    @JoinColumn(name="imparticion_id")
    private Imparticion imparticion;

    @ManyToOne(cascade=CascadeType.PERSIST)
    @JoinColumn(name="alumno_dni")
    private Alumno alumno;

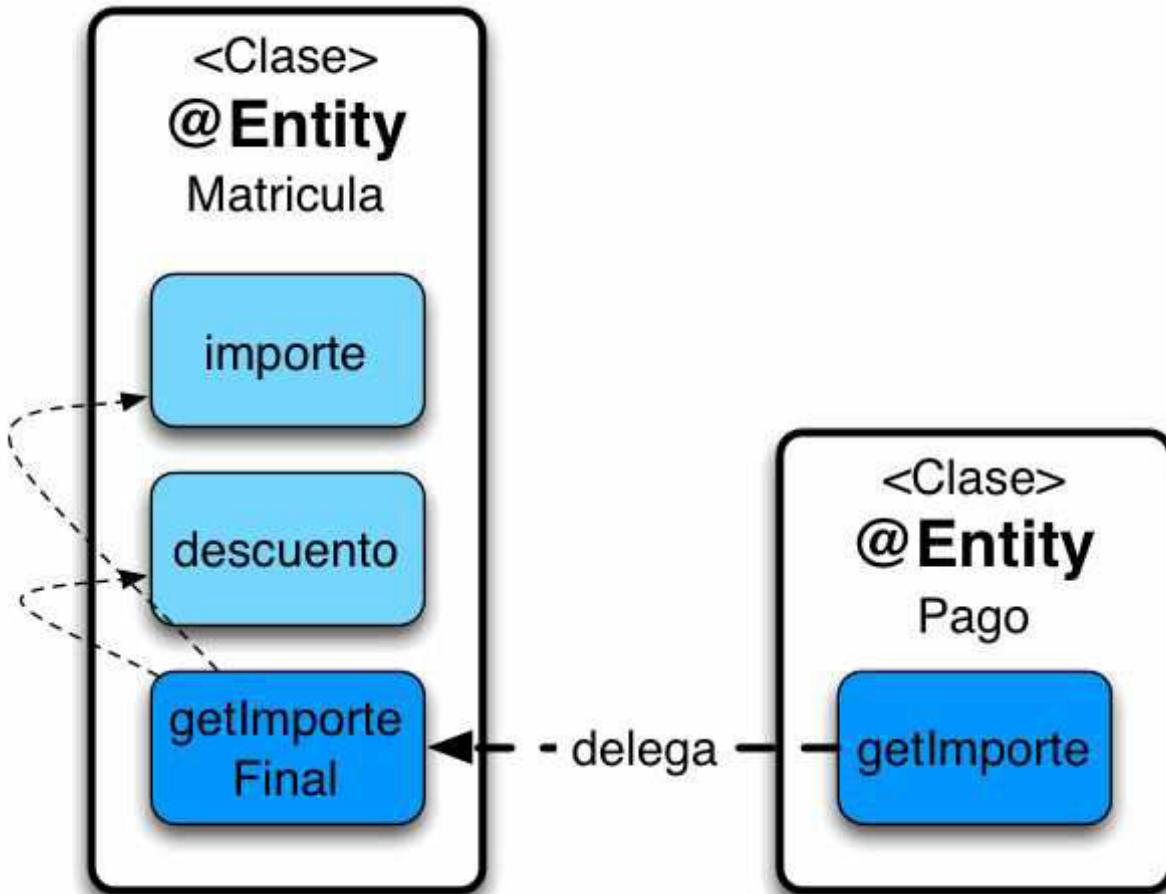
    @OneToOne(mappedBy = "matricula",
    cascade = { CascadeType.PERSIST,
    CascadeType.REMOVE})
    private Pago pago;
}

```

En este caso **la relación varía**, ya que cuando nosotros eliminamos una Matrícula evidentemente **debemos eliminar el Pago (Cascade.Remove en Matricula)**. Sin embargo lo contrario no sucede, por eso el Pago soporta únicamente Cascade.Persist.

MODELOS RICOS (DELEGACIÓN)

Hay situaciones en las que podemos enriquecer el modelo de dominio aplicando conceptos como el de **delegación** . En nuestro caso existe una relación 1 a 1 entre Matricula y Pago . No obstante ,es la Matricula la que almacena el concepto de **ImporteFinal** ,que es calculado con la variable importe aplicándole un descuento.



La clase Pago tiene también un método `getImporte()` pero no hace falta que almacene otra vez el concepto de importe con su descuento etc . Será suficiente con que delegue en la clase Matricula e invoque a su método `getImporteFinal()` .Será la Matricula la que devuelta la información solicitada al Pago ya que es interesante que el Pago disponga de ella de forma directa . A continuación se muestra el código que sirve para clarificar la idea:

```
public double getImporte() {  
    //delegación  
    return matricula.getImporteFinal();  
}
```

De esta forma es como el modelo de dominio se va enriqueciendo y diferenciándose del modelo entidad relación.

RESUMEN

Hemos desarrollado en este capítulo una relación algo mas compleja que las habituales .La relación 1 a 1. Ademas hemos podido revisar el concepto de delegación y como éste afecta a las relaciones. A continuación abordaremos el concepto de herencia dentro del modelo de dominio.

DESCARGAR EJEMPLOS

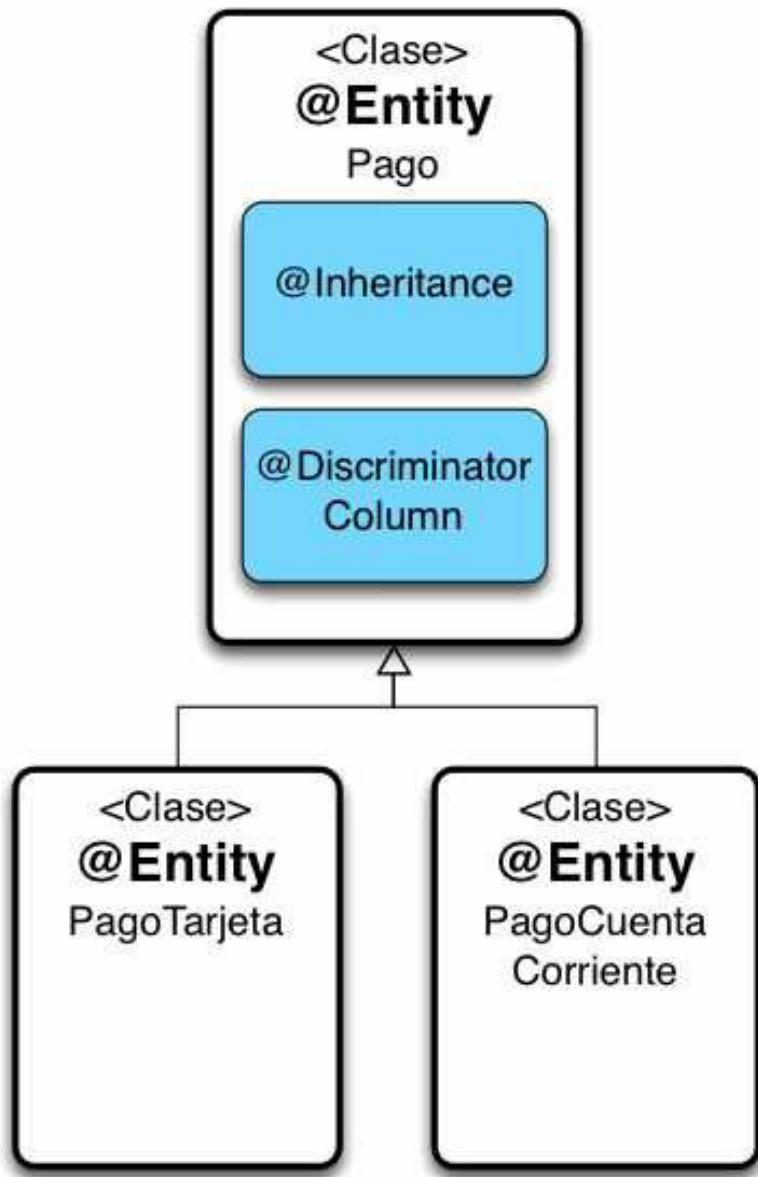
<http://www.arquitecturajava.com/wp-content/uploads/011JPAAAlumnoPago.zip>

Uso de Herencia

Siempre es difícil introducir la herencia dentro de un modelo de dominio . En este caso vamos a optar por modificar el modelo de una forma poco agresiva. Para ello revisaremos el concepto de Pago. Como sabemos existen dos tipos de Pagos que se pueden realizar uno a través de una cuenta corriente y otro a usando una tarjeta de crédito .**Lo que está claro es que son conceptos excluyentes : o pagamos a través de cuenta corriente o realizamos el pago con tarjeta** . Puede ser un buen punto para añadir al modelo una jerarquía de clases que incluya Pago , PagoCuenta y PagoTarjeta y lo haga utilizando la herencia. Vamos a verlo .

Pago	
id	int
fecha	date
cuenta	varchar(20)
numeroTarjeta	varchar(16)
numeroCodigo	varchar(4)
matricula_id	int

En primer lugar no va a ser necesario cambiar el modelo Entidad-Relación de la base de datos. Nos vamos a apoyar en él para trabajar, pero lo que cambiaremos será el modelo de dominio . Vamos a ver cuál es la nueva estructura de los Pagos



@INHERITANCE

Aparecen tres clases : Pago,PagoTarjeta y PagoCuentaCorriente que se encargan de definir esta jerarquía de herencia.
Ahora bien también aparecen dos anotaciones nuevas:

@Inheritance :Anotación que sirve para definir el tipo de herencia soportado por el modelo .Existen varios, cada uno con sus ventajas e inconvenientes

@DiscriminatorColumn: Al tratarse de una relación de herencia sencilla, esta anotación indica qué campo de la tabla de la base de datos identifica registros que son de un tipo (PagoTarjeta) o de otro (PagoCuentaCorriente).Es momento de ver código fuente de cada una de las clases para aclarar ideas.

```

package com.arquitecturajava;

import java.io.Serializable;

import java.util.Date;

import javax.persistence.CascadeType;

import javax.persistence.DiscriminatorColumn;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

```

```
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="tipoPago")
public class Pago implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue
    private int id;
    private Date fecha;

    @OneToOne(cascade=CascadeType.PERSIST)
@JoinColumn(name="matricula_id")
    private Matricula matricula;
    public Pago() {
        super();
    }
    public Pago(Date fecha, Matricula matricula) {
        super();
        this.fecha = fecha;
        setMatricula(matricula);
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public Date getFecha() {
        return fecha;
    }
    public void setFecha(Date fecha) {
        this.fecha = fecha;
    }
    public Matricula getMatricula() {
        return matricula;
    }
}
```

```

public void setMatricula(Matricula matricula) {
    if(!pago.getMatricula().equals(this))
        pago.setMatricula(this);
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + id;
    return result;
}

@Override
public boolean equals(Object obj) {
    if(this == obj)
        return true;
    if(obj == null)
        return false;
    if(getClass() != obj.getClass())
        return false;
    Pago other = (Pago) obj;
    if(id != other.id)
        return false;
    return true;
}
}

```

Si revisamos el código de la clase Pago que es la clase padre dentro de la jerarquía de herencia, nos encontraremos con lo siguiente.

```

@Entity
@Inheritance
(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn
(name="tipoPago")

```

Podemos ver como en la anotación de **@Inheritance** hace uso de una estrategia de una sola tabla ya que es nuestro caso. En segundo lugar se asigna la columna que discrimina uno u otro tipo dentro de la jerarquía. Es momento de ver el código fuente del resto de clases.

```

package com.arquitecturajava;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Entity;

```

```

@Entity
public class PagoCuentaCorriente

```

```

extends Pago implements Serializable {

    private static final long serialVersionUID = 1L;

    private String numeroCuenta;

    public String getNumeroCuenta() {
        return numeroCuenta;
    }

    public PagoCuentaCorriente() {
        super();
    }

    public void setNumeroCuenta(String numeroCuenta) {
        this.numeroCuenta = numeroCuenta;
    }

    public PagoCuentaCorriente(Date fecha,
        int descuento, String numeroCuenta,
        Matricula matricula) {
        super(fecha,matricula);
        this.setNumeroCuenta(numeroCuenta);
    }
}

```

En este caso aparte de extender de la clase Pago poco mas tiene de especial la clase PagoCuentaCorriente. Algo similar pasará con la clase de PagoTarjeta.

```

package com.arquitecturajava;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Entity;

@Entity
public class PagoTarjeta
extends Pago implements Serializable {

    private static final long serialVersionUID = 1L;

    private String tarjeta;
    private String codigoSeguridad;

    public PagoTarjeta() {
        super();
    }
}

```

```

public PagoTarjeta(Date fecha, int descuento, String tarjeta,
                   String codigoSeguridad, Matricula matricula) {
    super(fecha,matricula);
    setTarjeta(tarjeta);
    setCodigoSeguridad(codigoSeguridad);
}

public String getTarjeta() {
    return tarjeta;
}

public void setTarjeta(String tarjeta) {
    this.tarjeta = tarjeta;
}

public String getCodigoSeguridad() {
    return codigoSeguridad;
}

public void setCodigoSeguridad(String codigoSeguridad) {
    this.codigoSeguridad = codigoSeguridad;
}

```

HERENCIA E INSERCIÓNES

Hemos terminado de hacer una revisión a la jerarquía de clases Seguidamente veremos cómo estas clases se pueden utilizar en un programa de forma conjunta para almacenar datos de todo el modelo:

```

//omitimos imports

public class PrincipalInsertar {

    public static void main(String[] args) {
        Alumno pedro= new Alumno("1","pedro","gomez",30);

        Curso cursoJava=
            new Curso("JAVA2","Introduccion Java",20,300);

        Imparticion imparticion=
            new Imparticion(new Date(),cursoJava);

        Matricula matricula= new Matricula(new Date(),
            300,0,imparticion,pedro );

        Pago pago= new PagoCuentaCorriente(new Date(),
        0,"2020-2020-2020-3030303030",
        matricula);

        matricula.setPago(pago);

        Alumno juan= new Alumno("2","juan","perez",25);

        Matricula matricula2=
            new Matricula(new Date(),300,0,imparticion,juan);

        Pago pago2= new PagoTarjeta(new Date(),

```

```
0,"2020-2020-2020-303030",
```

```
“123”,matricula2);
```

```
matricula2.setPago(pago);
```

```
EntityManagerFactory emf =
```

```
Persistence.
```

```
createEntityManagerFactory("UnidadCurso");
```

```
EntityManager em =
```

```
emf.createEntityManager();
```

```
em.getTransaction().begin();
```

```
em.persist(pedro);
```

```
em.persist(juan);
```

```
em.persist(cursoJava);
```

```
em.persist(imparticion);
```

```
em.persist(matricula);
```

```
em.persist(pago);
```

```
em.persist(matricula2);
```

```
em.persist(pago2);
```

```
em.getTransaction().commit();
```

```
em.close();
```

```
System.out.println("termino");
```

```
}
```

```
}
```

```
}
```

De esta forma y utilizando herencia habremos salvado tanto los PagoTarjeta como los PagoCuentaCorriente en la tabla Pago siendo cada uno una clase diferente pero compartiendo una clase padre común.

#	id	fecha	descuento	matricula_id	curso_id	tarjeta	numeroCue	codigoSeguridad	tipoPago
1	14	2014-01-10	0	19	JAVA2	HULL	2020-2020	HULL	PagoCuentaCorriente
2	15	2014-01-10	0	19	JAVA2	2020-2020	HULL	123	PagoTarjeta
*	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL

CONSULTAS POLIMÓRFICAS

A continuación podemos realizar la siguiente consulta de JPA para obtener los datos almacenados de todos los tipos de Pagos. A estas consultas se las denomina polimórficas ya que porque nos pueden traer varios tipos de clase pertenecientes a diferentes niveles de la jerarquía.

```
package com.arquitecturajava.main;  
  
import java.util.List;  
  
import javax.persistence.EntityManager;  
  
import javax.persistence.EntityManagerFactory;  
  
import javax.persistence.Persistence;  
  
import javax.persistence.TypedQuery;
```

```

import com.arquitecturajava.Pago;

public class PrincipalConsulta {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("UnidadCurso");
        EntityManager em = emf.createEntityManager();
        TypedQuery<Pago> consultaPago =
            em.createQuery("select p from Pago p",
            Pago.class);

        List<Pago> pagos = consultaPago.getResultList();
        for(Pago p : pagos) {
            System.out.println(p.getId());
        }
    }
}

```

Esto nos presentará como resultado

1
2

De igual manera podríamos hacer una consulta que solo seleccione los pagos realizados con tarjeta.

```

EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("UnidadCurso");
EntityManager em = emf.createEntityManager();
TypedQuery<PagoTarjeta> consultaPago =
    em.createQuery("select p from PagoTarjeta p",
    PagoTarjeta.class);

```

```

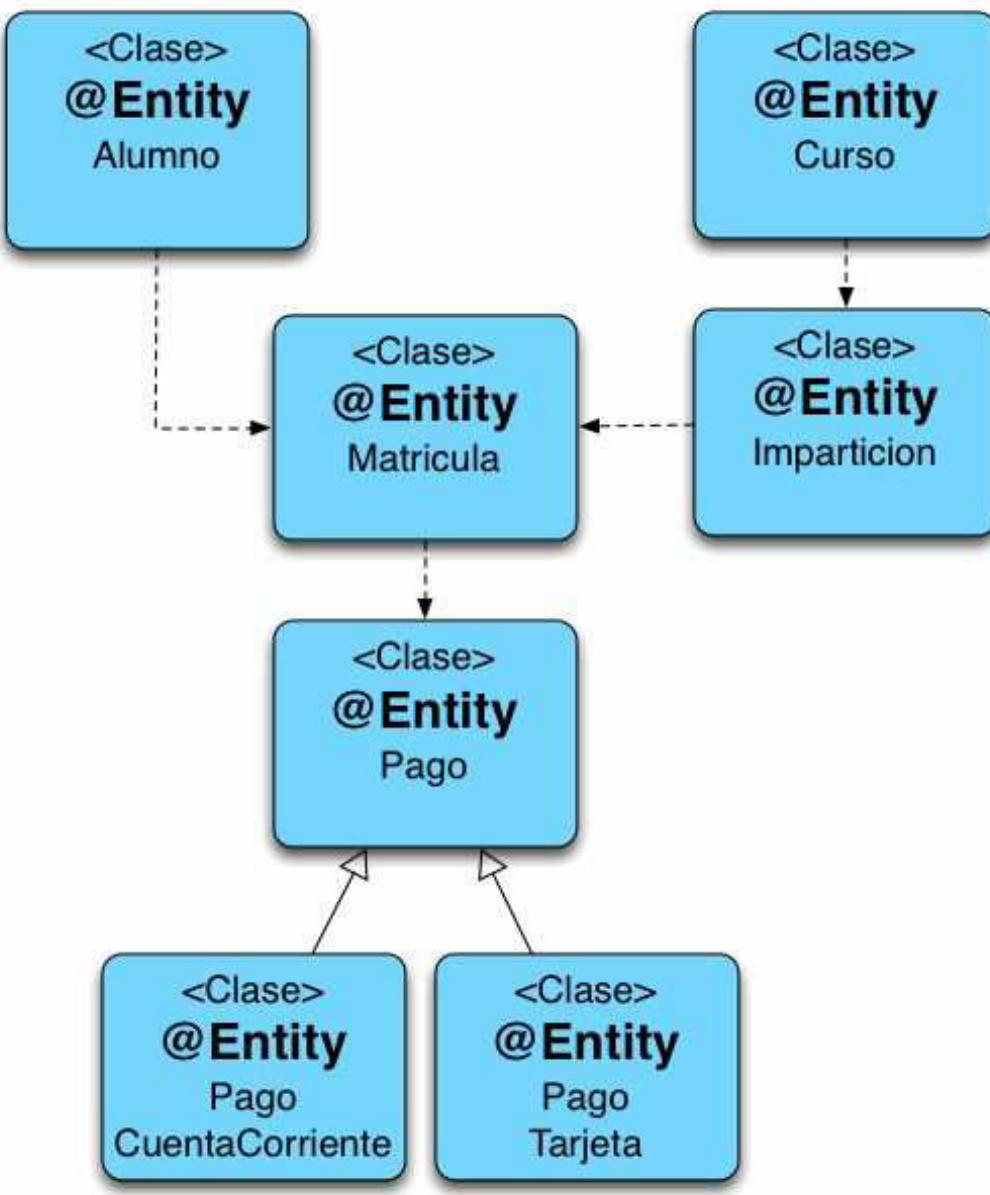
List<PagoTarjeta> pagos = consultaPago.getResultList();
for(PagoTarjeta p : pagos) {
    System.out.println(p.getId());
}

```

El resultado simplemente mostrará el Id del PagoTarjeta

1

Hemos terminado de introducir el concepto de herencia a nivel de JPA y nuestro modelo de dominio ha evolucionado hasta disponer de las siguientes clases:



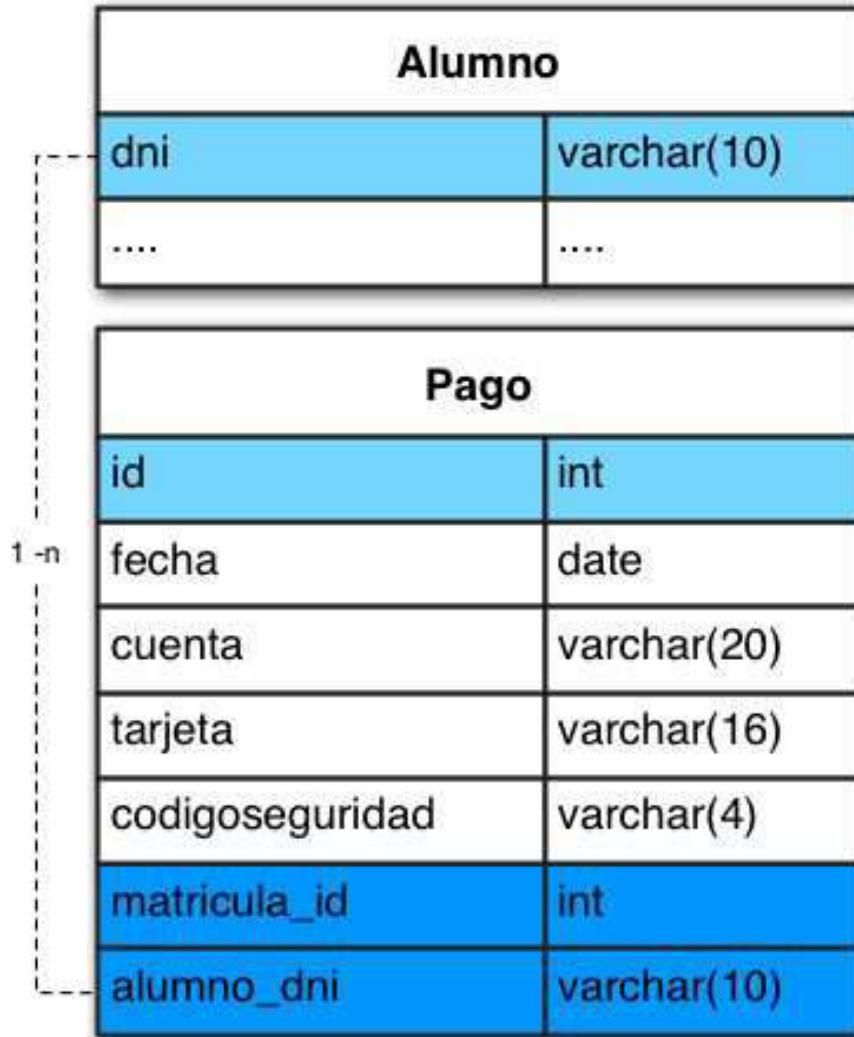
No vamos a añadir más elementos al modelo ,han sido suficientes para aprender los conceptos fundamentales de JPA. En el siguiente capítulo abordaremos algunas cosas cuestiones que sirven de complemento a lo visto hasta aquí.

DESCARGAR EJEMPLOS

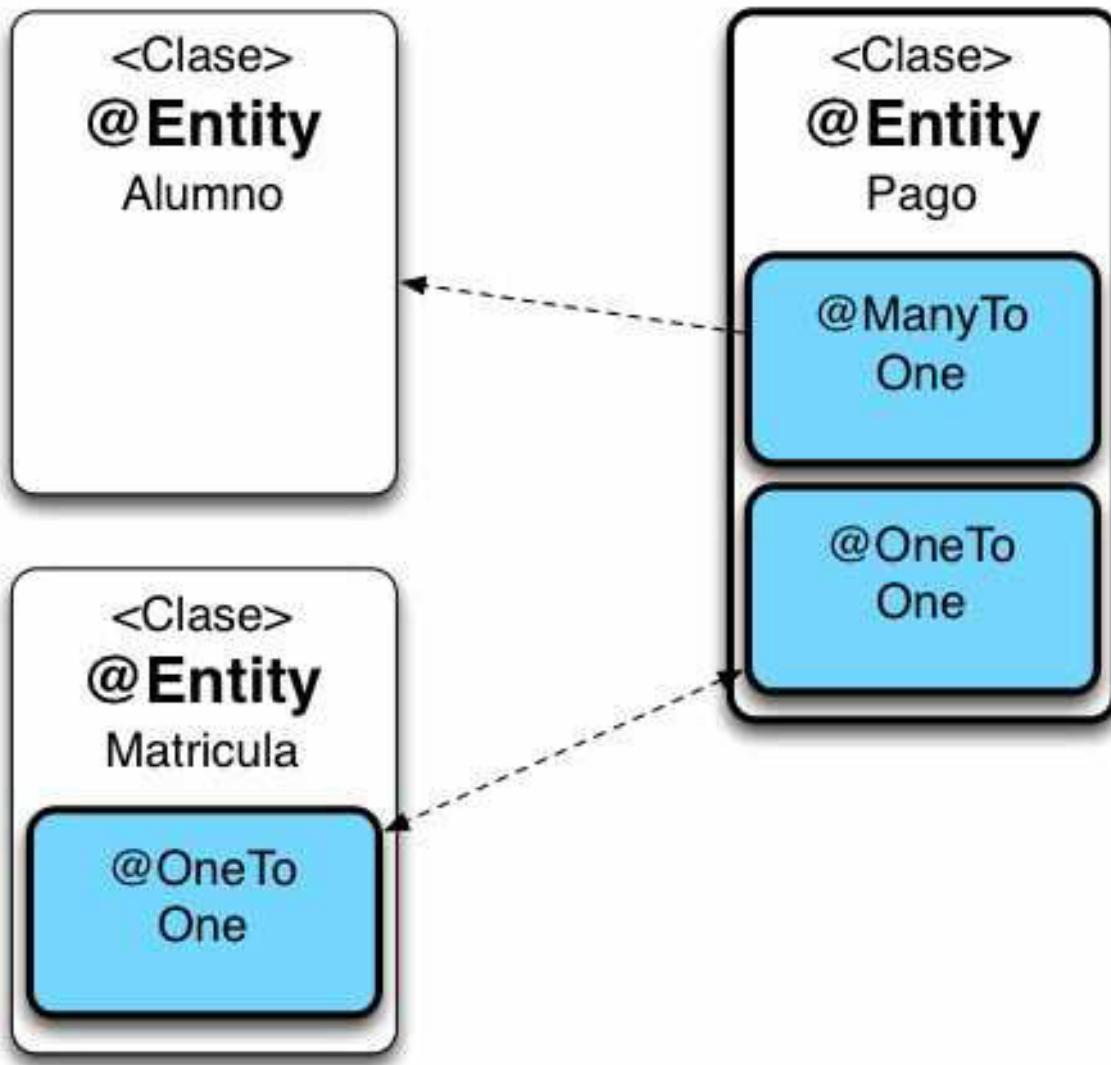
<http://www.arquitecturajava.com/wp-content/uploads/12JPAAAlumnoPagoCursoPolimorfico.zip>

Relaciones UniDireccionales

A veces nuestro modelo no encaja con las necesidades reales y hay que realizar modificaciones . Supongamos el siguiente caso : necesitamos obtener la lista de todos los Pagos del mes y a que Alumno pertenece cada uno . Esto no es directo ya que los Pagos no están asociados a ninguno de los Alumnos de forma directa . La consulta, que un principio parecía sencilla, debe seleccionar primero a las Matriculas y de ahí a los Alumnos . Podríamos dejarlo así ,pero vamos en cambio a desnormalizar el modelo ligando directamente Alumno y Pago



Como podemos ver ,una modificación sencilla del modelo Entidad-Relación es añadir un campo **alumno_dni** a la tabla **Pago** para relacionar ambos conceptos.. Realizada esta operación, podremos modificar el modelo de dominio para que se apoye en ella de tal forma que la relación entre Alumno y Pago exista. Vamos a verlo:



Podemos ver con claridad el nuevo tipo de relación que aparece entre Pago y Alumno. Es momento de ver cómo queda modificado el código de esta clase .Vamos a mostrar únicamente la parte que ha cambiado:

```

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="tipoPago")
public class Pago {

```

```

@Id
@GeneratedValue
private int id;
private Date fecha;
@OneToOne(cascade=CascadeType.PERSIST)
@JoinColumn(name="matricula_id")
private Matricula matricula;

@ManyToOne(cascade=CascadeType.PERSIST)
@JoinColumn(name="alumno_dni")
private Alumno alumno;

```

```

public Pago(Date fecha,
Matricula matricula,
Alumno alumno) {
    super();
    this.fecha = fecha;
    this.matricula = matricula;
    this.alumno = alumno;
}

```

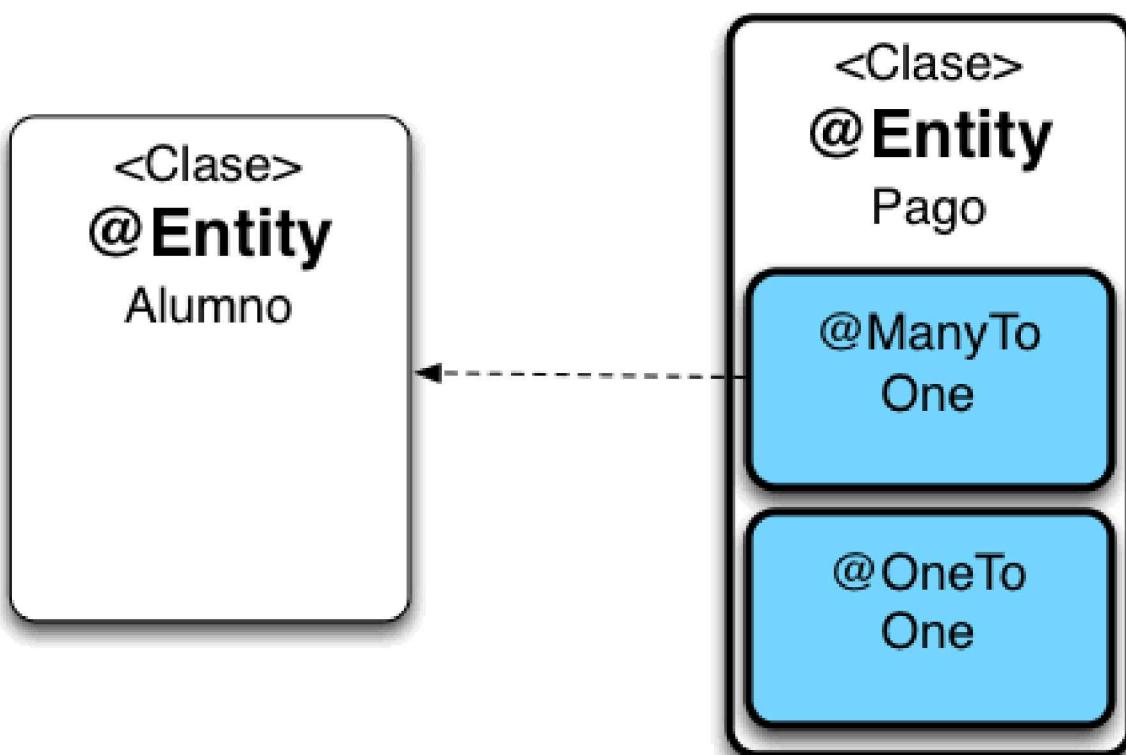
//Resto de código no varía salvo set/get matricula.

No varía sustancialmente ,simplemente se genera una nueva relación @ManyToOne entre los dos conceptos.

ALUMNOS Y PAGOS

Ahora bien **¿que sucede en el sentido contrario?** .Lo lógico sería añadir una relación @OneToMany en la clase Alumno que nos relacione con los Pagos. Sin embargo si reflexionamos un poco,vemos que la relación es problemática **¿Por qué?** . Bueno de repente aparecen dos opciones para seleccionar los Alumnos con sus Matriculas , Imparticiones y Cursos. La opción natural que es a través de Matricula y la nueva opción de Pago.

¿Es recomendable hacer esto? . La realizad es que no .Vamos pues a mantener el modelo sencillo y no modifiquemos la clase Alumno. De esta manera, la relación será Unidireccional ya que solo nos conviene en un sentido.



Es momento de dar por finalizado el modelo .Siempre se podrán hacer más modificaciones ,añadiendo o eliminando conceptos y modificando relaciones . Pero es un buen punto para darlo por terminado . A continuación abordaremos las conclusiones

DESCARGAS EJEMPLOS

<http://www.arquitecturajava.com/wp-content/uploads/13Unidireccional.zip>