

# UT5 - Herramientas de mapeo objeto relacional (ORM)

## Ejemplo\_1: Primer proyecto

1-Add dependencies:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.6.5.Final</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.27</version>
    </dependency>
</dependencies>
```

OJO poner la version 4.13.2 del JUnit ya 5.6.5.Final del org.hibernate

2-Add persistence:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://www.w3.org/2001/XMLSchema-instance">
    <persistence-unit name="PrimerProyecto" transaction-type="RESOURCE_LOCAL">
        <class>es.hectorsanchez.Ejemplo1.Alumno</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3310/ejemplo"/>
            <property name="javax.persistence.jdbc.user" value="root"/>
            <property name="javax.persistence.jdbc.password" value="root"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.hbm2ddl.auto" value="create"/>
        </properties>
    </persistence-unit>
</persistence>
```

3-Clase Alumno:

```
package es.hectorsanchez.Ejemplo1;

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

@Entity
public class Alumno
{
    //Atributos
    @Id
    private int id;

    private String nombre;

    private float nota;

    public int getId()
    {
        return id;
    }
    //Getter_&_Setter...
    public void setId(int id)
    {
        this.id = id;
    }

    public String getNombre()
    {
        return nombre;
    }
}
```

#### 4-Main:

```
package es.hectorsanchez.Ejemplo1;

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

public class App
{
    public static void main( String[] args )
    {
        //Configuramos el EMF a través de la unidad de persistencia
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("PrimerProyecto");

        //Generamos un EntityManager
        EntityManager em = emf.createEntityManager();

        //Iniciamos una transacción
        em.getTransaction().begin();

        // Construimos un objeto Alumno
        Alumno alumno1 = new Alumno();
        alumno1.setId(1);
        alumno1.setNombre("Recu");
        alumno1.setNota(5);

        // Construimos otro objeto Alumno
        Alumno alumno2 = new Alumno();
        alumno2.setId(2);
        alumno2.setNombre("María");
        alumno2.setNota(9);

        //Persistimos los objetos
        em.persist(alumno1);
        em.persist(alumno2);

        //Comiteamos la transacción
        em.getTransaction().commit();

        //Cerramos el EntityManager
        em.close();
    }
}
```

## Ejemplo\_2: Mapeo de entidades

```
package es.hectorsanchez.Ejemplo2;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="alumnos")
//AddNombre a la tabla
public class Alumno
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    //Autoincrementable
    private int id;

    @Column(nullable = false, length = 20)
    //Nombre no puede ser nulo y max 20 char
    private String nombre;

    private float nota;

    //Getter_&_Setter...
    public int getId()
    {
        return id;
    }

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

```

package es.hectorsanchez.Ejemplo2;

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

public class App
{
    public static void main( String[] args )
    {
        //Configuramos el EMF a través de la unidad de persistencia
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("PrimerProyecto");

        //Generamos un EntityManager
        EntityManager em = emf.createEntityManager();

        //Iniciamos una transacción
        em.getTransaction().begin();

        // Construimos un objeto Alumno
        Alumno alumno1 = new Alumno();
        //alumno1.setId(1); No hace falta porque es auto-incrementado
        alumno1.setNombre("Pepe");
        alumno1.setNota(5);

        // Construimos otro objeto Alumno
        Alumno alumno2 = new Alumno();
        //alumno2.setId(2); No hace falta porque es auto-incrementado
        alumno2.setNombre("María");
        alumno2.setNota(9);

        //Persistimos los objetos
        em.persist(alumno1);
        em.persist(alumno2);

        //Commiteamos la transacción
        em.getTransaction().commit();

        //Cerramos el EntityManager
        em.close();
    }
}

```

Existen diferentes estrategias de asignación:

- **GenerationType.AUTO**: se escoge la mejor estrategia en función del dialecto SQL configurado (es decir, dependiendo del RDBMS).
- **GenerationType.SEQUENCE**: espera usar una secuencia SQL para generar los valores.
- **GenerationType.IDENTITY**: se utiliza una columna especial, autonumérica.
- **GenerationType.TABLE**: se usa una tabla extra en nuestra base de datos. Tiene una fila por cada tipo de entidad diferente, y almacena el siguiente valor a utilizar.

## Ejemplo\_3: Tipos embebidos

Únicamente tendremos una única tabla en la base de datos con los campos id, nombre, fechaNacimiento, calle y numero.

Si quisiésemos que la clase Alumno tuviese **dos direcciones**, Hibernate lanzaría una excepción indicando que hay columnas repetidas.

Para solucionarlo, sobreescribiremos los atributos de la clase embebida para que tengan otro nombre usando la anotación **@AttributeOverrides**

```
@Embeddable
public class Direccion {
    @Column
    private String calle;
    @Column
    private int numero;

    //Getter & Setter
    public String getCalle() {
        return calle;
    }
    public void setCalle(String calle) {
        this.calle = calle;
    }
    public int getNumero() {
        return numero;
    }
    public void setNumero(int numero) {
        this.numero = numero;
    }
}

@Entity
@Table(name="alumnos")
public class Alumno
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    @Column(nullable = false, length = 20)
    private String nombre;

    private float nota;

    @Embedded
    private Direccion direccion;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "calle", column = @Column(name="calle_facturacion")),
        @AttributeOverride(name = "numero", column = @Column(name="numero_facturacion"))
    })
    private Direccion direccionFacturacion;
    //Getter & Setter...
```

En el main solo hay que añadir los campos a los objetos Alumno.

```
//Construimos un objeto Alumno-----
Alumno alumno1 = new Alumno();
//alumno1.setId(1); No hace falta porque es auto-incrementado
alumno1.setNombre("Recu");
alumno1.setNota(5);
//Add objeto Direccion al objeto Alumno
Direccion direccion1 = new Direccion();
direccion1.setCalle("PuenteSanMiguel");
direccion1.setNumero(77);
alumno1.setDireccion(direccion1);
//Add objeto Direccion (direccionFacturacion) a Alumno
Direccion direccionFacturacion1 = new Direccion();
direccionFacturacion1.setCalle("Reocin");
direccionFacturacion1.setNumero(1);
alumno1.setDireccionFacturacion(direccionFacturacion1);

//Construimos otro objeto Alumno-----
Alumno alumno2 = new Alumno();
//alumno2.setId(2); No hace falta porque es auto-incrementado
alumno2.setNombre("María");
alumno2.setNota(9);
//Add objeto Direccion al objeto Alumno
Direccion direccion2 = new Direccion();
direccion2.setCalle("Villapresente");
direccion2.setNumero(1);
alumno2.setDireccion(direccion2);
//Add objeto Direccion (direccionFacturacion) a Alumno
Direccion direccionFacturacion2 = new Direccion();
direccionFacturacion2.setCalle("Reocin");
direccionFacturacion2.setNumero(1);
alumno2.setDireccionFacturacion(direccionFacturacion2);
```

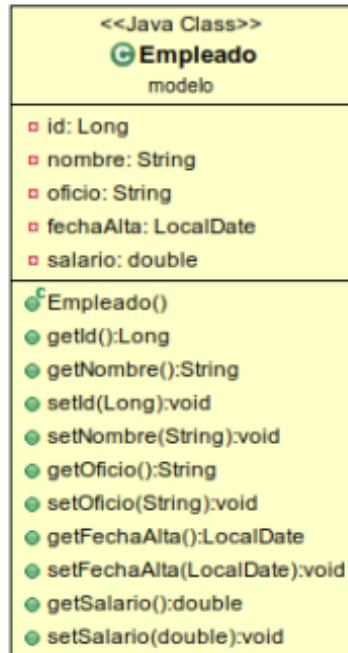
# Hoja\_1

## EJERCICIOS

1.- Crea un proyecto Maven y configura las dependencias JPA y MySQL. Realiza todos los pasos necesarios para convertirlo en un proyecto que use JPA.

Crea una base de datos llamada **empresa** y edita el fichero persistence.xml para configurar la conexión

Crea la entidad **Empleado** dentro del paquete **modelo**:



En la clase **App** crea un EntityManager. Despues crea dos empleados y haz que se guarden en la base de datos.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance"
  <persistence-unit name="PrimerProyecto" transaction-type="RESOURCE_LOCAL">
    <class>es.hectorsanchez.hoja05_orm_01.modelo.Empleado</class>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3310/empresa"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value="root"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
      <property name="hibernate.hbm2ddl.auto" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
=====
```

```
package es.hectorsanchez.hoja05_orm_01;

import java.time.LocalDate;...

public class App {
    public static void main(String[] args) {
        // Configuramos el EMF a través de la unidad de persistencia
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("PrimerProyecto");
        // Generamos un EntityManager
        EntityManager em = emf.createEntityManager();
        // Iniciamos una transacción
        em.getTransaction().begin();
        // Construimos un empleado
        Empleado empleado1 = new Empleado();
        empleado1.setId((long) 1);
        empleado1.setNombre("Recu");
        empleado1.setOficio("Atleta");
        empleado1.setSalario(3000);
        empleado1.setFechaAlta(LocalDate.now());
        // Construimos otro empleado
        Empleado empleado2 = new Empleado();
        empleado2.setId((long) 2);
        empleado2.setNombre("Leras");
        empleado2.setOficio("Piloto");
        empleado2.setSalario(2000);
        empleado2.setFechaAlta(LocalDate.now());
        // Persistimos los objetos
        em.persist(empleado1);
        em.persist(empleado2);

        // Committeamos la transacción
        em.getTransaction().commit();

        // Cerramos el EntityManager
        em.close();
    }
}
```

```
package es.hectorsanchez.hoja05_orm_01.modelo;

import java.time.LocalDate;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Empleado {
    @Id
    private Long id;
    private String nombre;
    private String oficio;
    private LocalDate fechaAlta;
    private double salario;

    public Empleado() {
        super();
    }

    //GETTR & SETTR...
    public Long getId() {
        return id;
    }

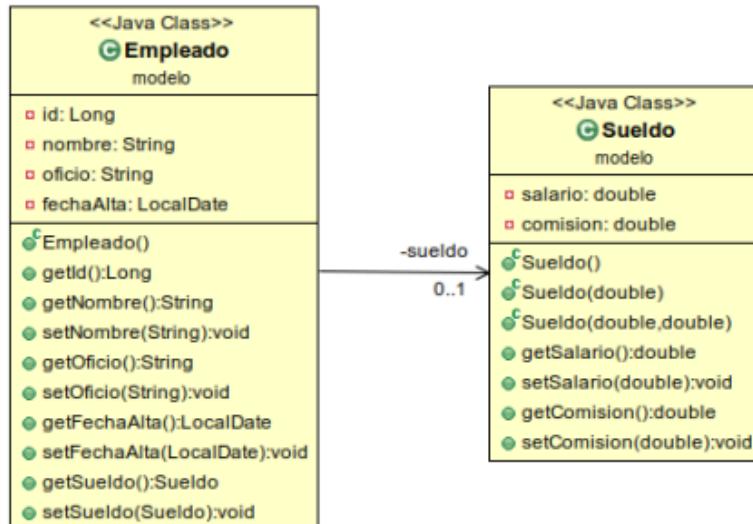
    public void setId(Long id) {
        this.id = id;
    }
}
```

....

## Hoja\_2

1.- Copia el proyecto anterior a otro llamado **Hoja05 ORM\_02** y realiza las siguientes operaciones:

- Haz que la tabla que se genere en la base de datos se llame **empleados**
- Haz que la clave primaria sea autonumérica y que escoja la mejor estrategia. Además, elimina el método **setId** ya que ya no hará falta
- El nombre **NO** puede ser **nulo**
- La fecha de alta debe guardarse en la base de datos como **fecha\_alta**
- La **longitud** del oficio debe ser 50
- Crea un tipo embebido que se llame **Sueldo**. Tendrá los atributos salario y comision (ambos de tipo double). El empleado tendrá un sueldo.
  - El salario **NO** puede ser **nulo**



Modifica el método main de la clase **App** para que se sigan guardando dos empleados.

```
@Entity
@Table(name = "empleados")
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(nullable = false)
    private String nombre;
    @Column(length = 50)
    private String oficio;
    @Column(name = "fecha_alta")
    private LocalDate fechaAlta;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "salario", column = @Column(name = "salario", nullable = false)),
        @AttributeOverride(name = "comision", column = @Column(name = "comision")) })
    private Sueldo sueldo;

    // GETTER_&_SETTER...

    @Embeddable
    public class Sueldo {
        @Column
        private double salario;

        @Column
        private double comision;

        //GETTER_&_SETTER...
    }
}
```

```

public class App {
    public static void main(String[] args) {
        // Configuramos el EMF a través de la unidad de persistencia
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("Hoja_1");

        // Generamos un EntityManager
        EntityManager em = emf.createEntityManager();

        // Iniciamos una transacción
        em.getTransaction().begin();

        // Construimos un objeto Empleado
        Empleado empleado1 = new Empleado();
        empleado1.setNombre("Recu");
        empleado1.setOficio("Atleta");
        empleado1.setFechaAlta(LocalDate.now());
        Sueldo sueldo1 = new Sueldo();
        sueldo1.setSalario(30000);
        sueldo1.setComision(200);
        empleado1.setSueldo(sueldo1);

        // Construimos otro objeto Alumno
        Empleado empleado2 = new Empleado();
        empleado2.setNombre("Leras");
        empleado2.setOficio("Ingeniero");
        empleado2.setFechaAlta(LocalDate.now());
        Sueldo sueldo2 = new Sueldo();
        sueldo2.setSalario(2000);
        sueldo2.setComision(200);
        empleado2.setSueldo(sueldo2);

        // Persistimos los objetos
        em.persist(empleado1);
        em.persist(empleado2);

        // Commiteamos la transacción
        em.getTransaction().commit();

        // Cerramos el EntityManager
        em.close();
    }
}

```

empleados x

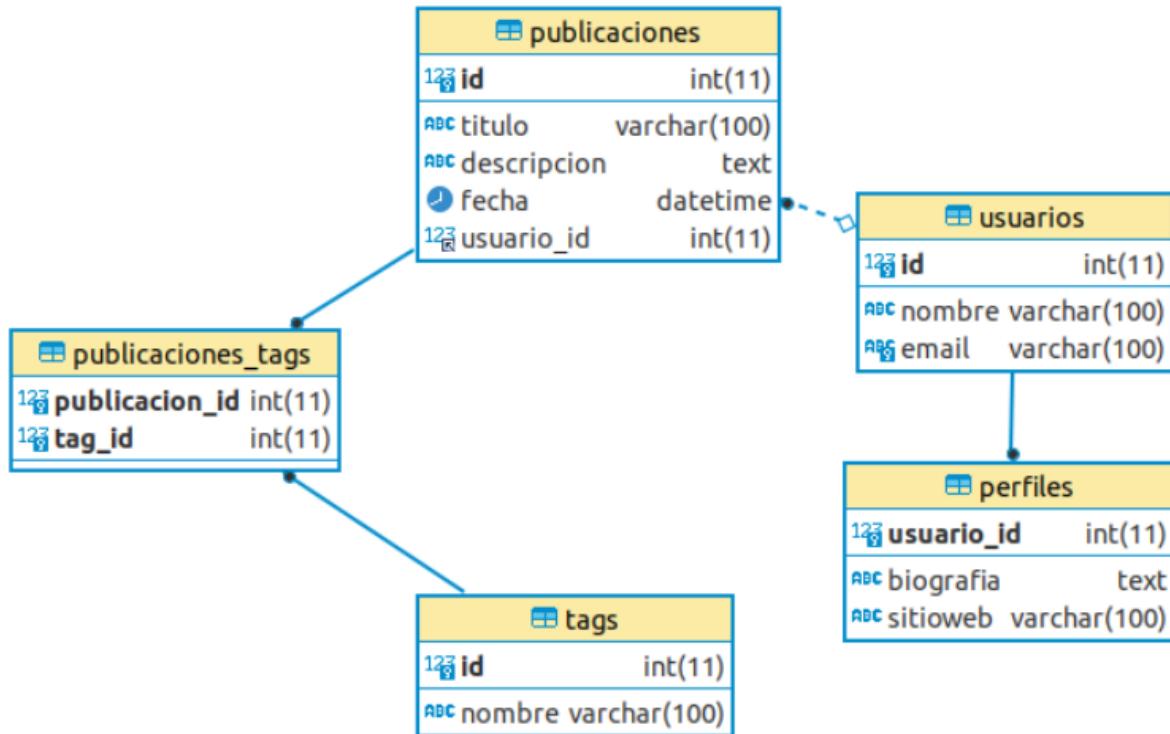
Propiedades Datos Diagrama ER

empleados | Enter a SQL expression to filter results (use Ctrl+Space)

Grilla	123 id ↕	⌚ fecha alta ↕	🔤 nombre ↕	🔤 oficio ↕	123 comision ↕	123 salario ↕
1	1	2022-02-15	Recu	Atleta	200	30.000
2	2	2022-02-15	Leras	Ingeniero	200	2.000

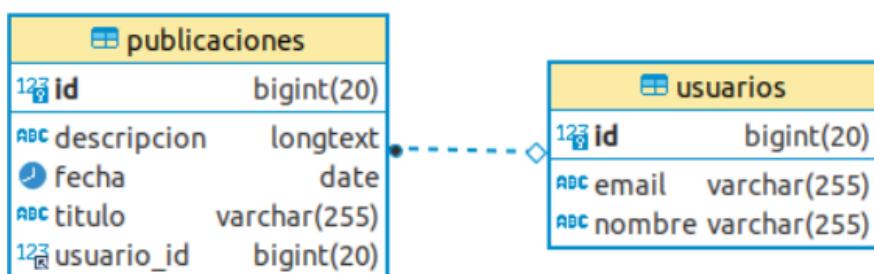
Texto

## Ejemplo\_4: Asociaciones



### Ejemplo\_4.1: Many-to-One Unidireccional

- Sólo se representa la asociación en el lado muchos.
- Se usa la anotación @ManyToOne
- Podemos añadir @JoinColumn para indicar el nombre de la columna que será la clave externa.
- También podemos añadir @ForeignKey para indicar el nombre de la restricción que se creará a nivel de base de datos.



```
package es.hectorsanchez.Ejemplo4;

import javax.persistence.Entity;□

@Entity
@Table(name = "usuarios")
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String email;

    private String nombre;

    //GETTER_&_SETTER...

package es.hectorsanchez.Ejemplo4;

import java.time.LocalDate;

import javax.persistence.Entity;
import javax.persistence.ForeignKey;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.Lob;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name = "publicaciones")
public class Publicacion {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Lob
    private String descripcion;

    private LocalDate fecha;

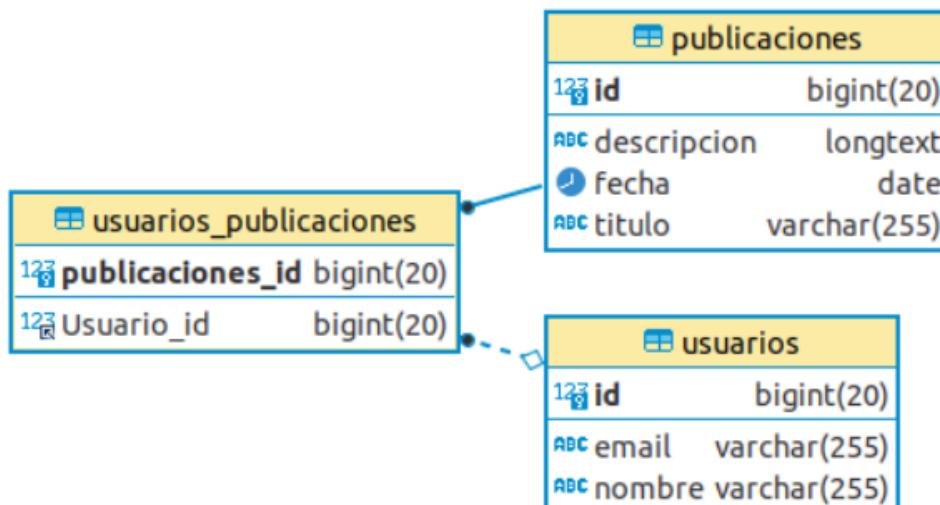
    private String titulo;

    @ManyToOne
    @JoinColumn(name = "usuario_id",foreignKey = @ForeignKey(name = "USUARIO_ID_FK"))
    private Usuario usuario;

    //GETTER_&_SETTER...
```

## Ejemplo\_4.2: One-to-Many Unidireccional

- Representa la asociación en el lado uno.
- Por eso, en la clase debemos colocar una **colección** de elementos.
- Si la asociación `@OneToMany` no tiene la correspondiente asociación `@ManyToOne` será unidireccional. En caso de que sí exista será **bidireccional**.



```
package es.hectorsanchez.Ejemplo4;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "usuarios")
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String email;

    private String nombre;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    // @JoinColumn(name = "usuario_id")
    private List<Publicacion> publicaciones = new ArrayList<>();

    //GETTER & SETTER...
}
```

- `cascade = CascadeType.ALL` propagará (en cascada) todas las operaciones a las entidades relacionadas
- `orphanRemoval = true` indica que la entidad hija será borrada cuando se borre la padre

```

package es.hectorsanchez.Ejemplo4;

import java.time.LocalDate;

import javax.persistence.Entity;
import javax.persistence.ForeignKey;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.Lob;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

```

```

@Entity
@Table(name = "publicaciones")
public class Publicacion {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Lob
    private String descripcion;

    private LocalDate fecha;

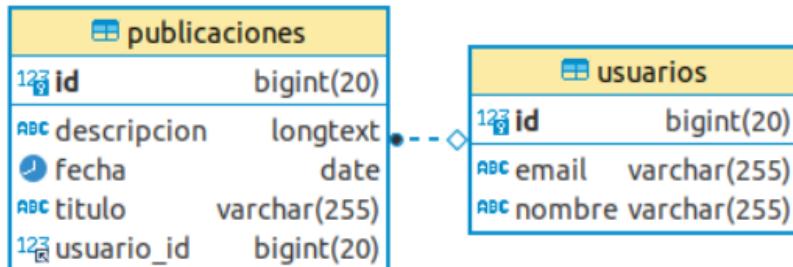
    private String titulo;

```

- Hibernate creará 3 tablas: una tabla para cada entidad, y otra tabla para asociar ambas, añadiendo una clave única al identificador del lado muchos (`publicaciones_id`)
- También podemos añadir la anotación `@JoinColumn` para decirle a JPA que existe una clave foránea `usuario_id` en la tabla `publicaciones` y no crear una tabla para asociar ambas (sólo 2 tablas)

## Ejemplo\_4.3: One-to-Many Bidireccional

- La asociación @OneToMany bidireccional necesita de una asociación @ManyToOne en el lado hijo.



```
package es.hectorsanchez.Ejemplo4;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "usuarios")
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String email;

    private String nombre;

    @OneToMany(mappedBy = "usuario", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Publicacion> publicaciones = new ArrayList<>();
```

```
//GETTER_&_SETTER...

public Long getId() {
    return id;
}

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

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getNombre() {
    return nombre;
}

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

public List<Publicacion> getPublicaciones() {
    return publicaciones;
}

public void setPublicaciones(List<Publicacion> publicaciones) {
    this.publicaciones = publicaciones;
}

public Publicacion addPublicacion(Publicacion publicacion)
{
    getPublicaciones().add(publicacion);
    publicacion.setUsuario(this);

    return publicacion;
}

public Publicacion removePublicacion(Publicacion publicacion)
{
    getPublicaciones().remove(publicacion);
    publicacion.setUsuario(null);

    return publicacion;
}
}
```

- @OneToMany referenciará al otro lado mediante el atributo **mappedBy**.
- Fijarse en los métodos **addPublicacion** y **removePublicacion** ya que además de añadir o borrar una publicación establecerán cuál es su usuario.

```

package es.hectorsanchez.Ejemplo4;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.ForeignKey;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.Lob;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "publicaciones")
public class Publicacion {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Lob
    private String descripcion;

    private LocalDate fecha;

    private String titulo;

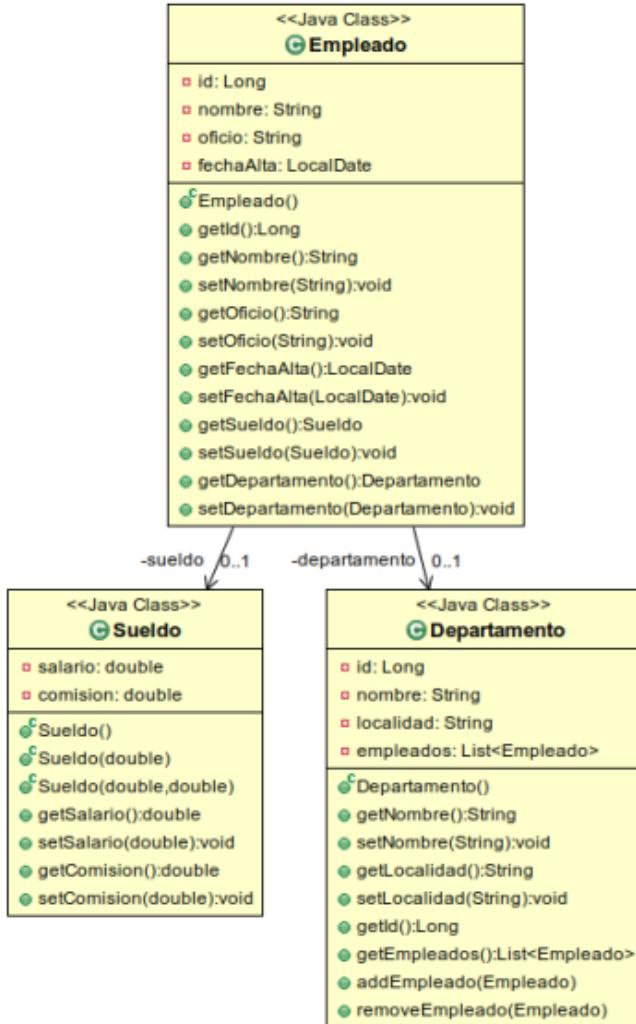
    @ManyToOne
    private Usuario usuario;

    //GETTER & SETTER...
}

```

## Hoja\_3

1.- Copia el proyecto anterior a otro llamado **Hoja05 ORM\_03** y realiza las tareas necesarias para satisfacer el siguiente diagrama de clases:



Hay que tener en cuenta que existe una asociación **one-to-many bidireccional**.

Desde la clase App habrá que crear dos departamentos y cuatro empleados (dos empleados por departamento)

```

@Entity
@Table(name = "empleados")
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(nullable = false)
    private String nombre;
    @Column(length = 50)
    private String oficio;
    @Column(name = "fecha_alta")
    private LocalDate fechaAlta;
    @Embedded
    @AttributeOverrides({ @AttributeOverride(name = "salario", column = @Column(name = "salario", nullable = false)),
        @AttributeOverride(name = "comision", column = @Column(name = "comision")) })
    private Sueldo sueldo;
    @ManyToOne
    private Departamento departamento;

    // GETTER & SETTER...
    public Departamento getDepartamento() {
        return departamento;
    }

    public void setDepartamento(Departamento departamento) {
        this.departamento = departamento;
    }

    //...

```

---

```

@Entity
@Table(name = "departamentos")
public class Departamento {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    private String nombre;
    private String localidad;
    @OneToMany(mappedBy = "departamento", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Empleado> empleados = new ArrayList();

    public long getId() {..}
    public void setId(long id) {..}
    public String getNombre() {..}
    public void setNombre(String nombre) {..}
    public String getLocalidad() {..}
    public void setLocalidad(String localidad) {..}
    public List<Empleado> getEmpleados() {..}
    public void setEmpleados(List<Empleado> empleados) {..}

    public Empleado addEmpleado(Empleado empleado) {
        this.getEmpleados().add(empleado);
        empleado.setDepartamento(this);

        return empleado;
    }
    public Empleado removeEmpleado(Empleado empleado) {
        this.getEmpleados().remove(empleado);
        empleado.setDepartamento(null);

        return empleado;
    }
}

```

```
import javax.persistence.Column;..  
//Preguntar si esta es Embedida o ManyToOne???  
@Embeddable  
public class Sueldo {  
    @Column  
    private double salario;  
  
    @Column  
    private double comision;  
  
    //GETTER_&_SETTER...  
  
    public double getSalario() {  
        return salario;  
    }  
  
    public void setSalario(double salario) {  
        this.salario = salario;  
    }  
  
    public double getComision() {  
        return comision;  
    }  
  
    public void setComision(double comision) {  
        this.comision = comision;  
    }  
}
```

```

public class App {
    public static void main(String[] args) {
        // Configuramos el EMF a través de la unidad de persistencia
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("Hoja_1");

        // Generamos un EntityManager
        EntityManager em = emf.createEntityManager();

        // Iniciamos una transacción
        em.getTransaction().begin();

        // Construimos dos objetos Departamento
        Departamento dep1 = new Departamento();
        dep1.setNombre("RecuEnterprise");
        dep1.setLocalidad("Oreña");

        Departamento dep2 = new Departamento();
        dep2.setNombre("IngenierosPorElMundo");
        dep2.setLocalidad("Santander");

        // Construimos un objeto Empleado
        Empleado empleado1 = new Empleado();
        empleado1.setNombre("Recu");
        empleado1.setOficio("Atleta");
        empleado1.setFechaAlta(LocalDate.now());
        Sueldo sueldo1 = new Sueldo();
        sueldo1.setSalario(30000);
        sueldo1.setComision(200);
        empleado1.setSueldo(sueldo1);
        empleado1.setDepartamento(dep1);

        // Construimos otro objeto Empleado
        Empleado empleado3 = new Empleado();
        empleado3.setNombre("Maria");
        empleado3.setOficio("Enfermera");
        empleado3.setFechaAlta(LocalDate.now());
        Sueldo sueldo3 = new Sueldo();
        sueldo3.setSalario(2500);
        sueldo3.setComision(400);
        empleado3.setSueldo(sueldo3);
        empleado3.setDepartamento(dep1);

        // Construimos otro objeto Empleado
        Empleado empleado4 = new Empleado();
        empleado4.setNombre("BroAlextintor");
        empleado4.setOficio("Ingeniero");
        empleado4.setFechaAlta(LocalDate.now());
        Sueldo sueldo4 = new Sueldo();
        sueldo4.setSalario(2000);
        sueldo4.setComision(400);
        empleado4.setSueldo(sueldo4);
        empleado4.setDepartamento(dep2);

        // Relleno los departamentos
        dep1.addEmpleado(empleado1);
        dep1.addEmpleado(empleado3);
        dep2.addEmpleado(empleado2);
        dep2.addEmpleado(empleado4);

        // Persistimos los objetos
        em.persist(empleado1);
        em.persist(empleado2);
        em.persist(empleado3);
        em.persist(empleado4);
        em.persist(dep1);
        em.persist(dep2);

        // Commiteamos la transacción
        em.getTransaction().commit();

        // Cerramos el EntityManager
        em.close();
    }
}

```

empresa    empleados departamentos

Propiedades Datos Diagrama ER

**empleados** Enter a SQL expression to filter results (use Ctrl+Space)

	123 id	fecha alta	abc nombre	abc oficio	123 comision	123 salario	123 departamento id
1	1	2022-02-17	Recu	Atleta	200	30.000	5
2	2	2022-02-17	Leras	Ingeniero	200	2.000	6
3	3	2022-02-17	Maria	Enfermera	400	2.500	5
4	4	2022-02-17	BroAlexTintor	Ingeniero	400	2.000	6

empresa    empleados departamentos

Propiedades Datos Diagrama ER

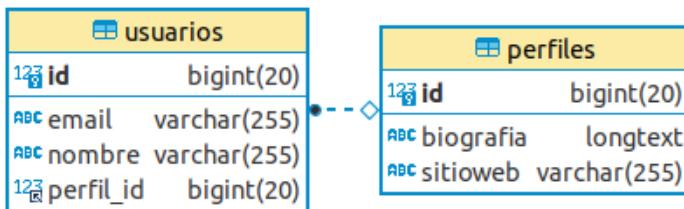
**departamentos** Enter a SQL expression to filter results (use Ctrl+Space)

	123 id	abc localidad	abc nombre
1	5	Oreña	RecuEnterprise
2	6	Santander	IngenierosPorElMundo

# Ejemplo\_5: Asociaciones II

## Ejemplo\_5.1: One-to-One Unidireccional

- Solamente una instancia de una clase se asocia con una instancia de otra.
- En el esquema unidireccional hay que decidir un lado como propietario.



```
@Entity
@Table(name = "usuarios")
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String email;
    private String nombre;

    @OneToOne(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "perfil_id")
    private Perfil perfil;

    @OneToMany(mappedBy = "usuario", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Publicacion> publicaciones = new ArrayList();

    public Long getId() {}
    public void setId(Long id) {}
    public String getEmail() {}
    public void setEmail(String email) {}
    public String getNombre() {}
    public void setNombre(String nombre) {}
    public List<Publicacion> getPublicaciones() {}
    public void setPublicaciones(List<Publicacion> publicaciones) {}
    public Publicacion addPublicacion (Publicacion publicacion) {}
    public Publicacion removePublicacion (Publicacion publicacion) {}
}
```

\* Falta getter y setter del perfil

```

@Entity
@Table(name = "perfiles")
public class Perfil {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Lob
    private String biografia;

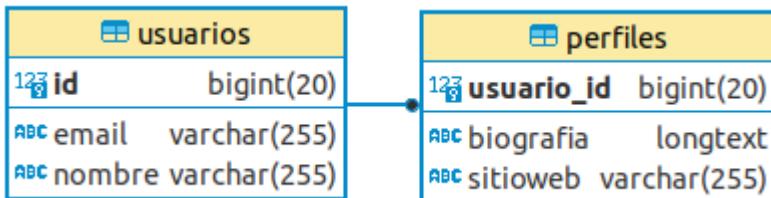
    private String sitioweb;

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getBiografia() {..}
    public void setBiografia(String biografia) {..}
    public String getSitioweb() {..}
    public void setSitioweb(String sitioweb) {..}

}

```

## Ejemplo\_5.2: One-to-One Bidireccional



```

@Entity
@Table(name = "usuarios")
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String email;
    private String nombre;

    @OneToOne(mappedBy = "usuario", cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
    private Perfil perfil;

    @OneToMany(mappedBy = "usuario", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Publicacion> publicaciones = new ArrayList<>();

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getEmail() {..}
    public void setEmail(String email) {..}
    public String getNombre() {..}
    public void setNombre(String nombre) {..}
    public List<Publicacion> getPublicaciones() {..}
    public void setPublicaciones(List<Publicacion> publicaciones) {..}
    public Publicacion addPublicacion (Publicacion publicacion) {..}
    public Publicacion removePublicacion (Publicacion publicacion) {..}

}

```

\* Falta getter y setter del perfil

```

@Entity
@Table(name = "perfiles")
public class Perfil {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Lob
    private String biografia;

    private String sitioweb;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(foreignKey = @ForeignKey(name = "USUARIO_ID_FK"))
    @MapsId
    private Usuario usuario;

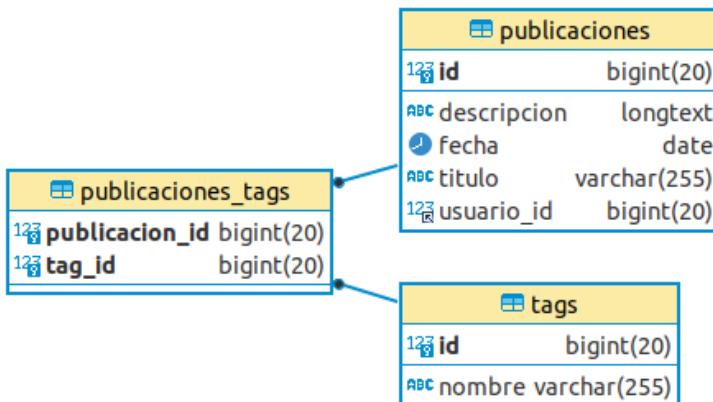
    public Long getId() {..}
    public void setId(Long id) {..}
    public String getBiografia() {..}
    public void setBiografia(String biografia) {..}
    public String getSitioweb() {..}
    public void setSitioweb(String sitioweb) {..}
    public Usuario getUsuario() {..}
    public void setUsuario(Usuario usuario) {..}

}

```

## Ejemplo\_5.3: Many-to-Many Unidireccional

- Una asociación muchos a muchos necesita una tabla que realice de enlace entre dos entidades.
- Tendremos que definir qué lado es el propietario de la asociación. En esa clase, incluimos una colección de elementos de la clase opuesta.
  - Si no queremos que se puedan repetir los elementos en la tabla intermedia utilizaremos una Set
  - Si puede haber repetidos podemos usar una List



```
@Entity
@Table(name = "tags")
public class Tag {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String nombre;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

```
@Entity
@Table(name = "publicaciones")
public class Publicacion {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Lob
    private String descripcion;
    private LocalDate fecha;
    private String titulo;

    @ManyToOne
    private Usuario usuario;

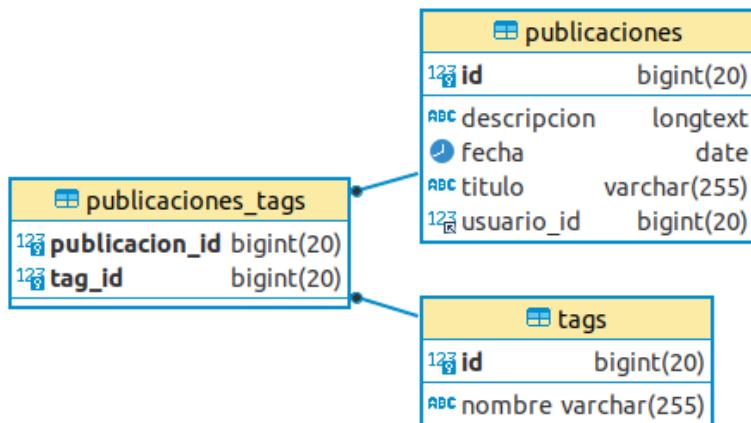
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(name = "publicaciones_tags",
               joinColumns = {@JoinColumn(name = "publicacion_id")},
               inverseJoinColumns = {@JoinColumn(name = "tag_id")})
    private Set<Tag> tags = new LinkedHashSet<>();

    public Long getId() {}
    public void setId(Long id) {}
    public String getDescripcion() {}
    public void setDescripcion(String descripcion) {}
    public LocalDate getFecha() {}
    public void setFecha(LocalDate fecha) {}
    public String getTitulo() {}
    public void setTitulo(String titulo) {}
    public Usuario getUsuario() {}
    public void setUsuario(Usuario usuario) {}
    public Set<Tag> getTags() {}
    public void setTags(Set<Tag> tags) {}

}
```

## Ejemplo\_5.4: Many-to-Many Bidireccional

- Una asociación bidireccional @ManyToMany tiene un lado propietario y un lado mappedBy.



```
@Entity
@Table(name = "tags")
public class Tag {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String nombre;
    @ManyToMany(mappedBy = "tags")
    private Set<Publicacion> publicaciones = new LinkedHashSet();

    public int getId() {}
    public void setId(int id) {}
    public String getNombre() {}
    public void setNombre(String nombre) {}
    public Set<Publicacion> getPublicaciones() {}
    public void setPublicaciones(Set<Publicacion> publicaciones) {}

}
```

```
@Entity
@Table(name = "publicaciones")
public class Publicacion {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Lob
    private String descripcion;
    private LocalDate fecha;
    private String titulo;
    @ManyToOne
    private Usuario usuario;
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(name = "publicaciones_tags",
               joinColumns = {@JoinColumn(name = "publicacion_id")},
               inverseJoinColumns = {@JoinColumn(name = "tag_id")})
    private Set<Tag> tags = new LinkedHashSet<>();

    public Long getId() {}
    public void setId(Long id) {}
    public String getDescripcion() {}
    public void setDescripcion(String descripcion) {}
    public LocalDate getFecha() {}
    public void setFecha(LocalDate fecha) {}
    public String getTitulo() {}
    public void setTitulo(String titulo) {}
    public Usuario getUsuario() {}
    public void setUsuario(Usuario usuario) {}
    public Set<Tag> getTags() {}
    public void setTags(Set<Tag> tags) {}

    public Tag addTag(Tag tag) {
        this.getTags().add(tag);
        tag.getPublicaciones().add(this);

        return tag;
    }
    public Tag removeTag(Tag tag) {
        this.getTags().remove(tag);
        tag.getPublicaciones().remove(this);

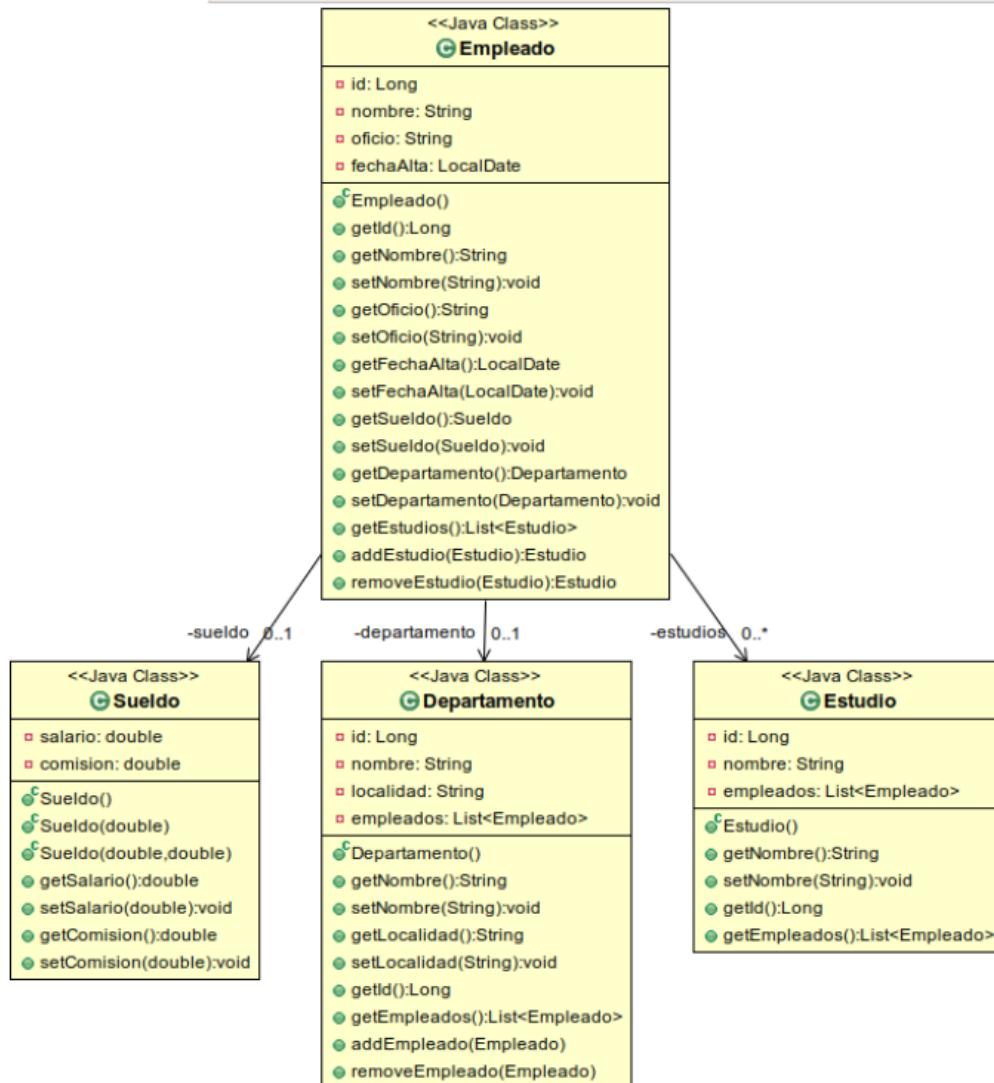
        return tag;
    }
}
```

# Hoja\_4

## Ejercicio\_1

1.- Copia el proyecto anterior a otro llamado **Hoja05 ORM 04**. Ahora vamos a añadir una entidad Estudio. Un empleado podrá tener muchos estudios y un mismo estudio podrá haberlo hecho muchos empleados. Haremos que esta relación sea **bidireccional**.

En la clase App crea algún estudio y haz que algún empleado lo haya cursado.



```

@Entity
@Table(name = "empleados")
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String nombre;

    @Column(length = 50)
    private String oficio;

    @Column(name = "fecha_alta")
    private LocalDate fechaAlta;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "salario", column = @Column(name = "salario", nullable = false)),
        @AttributeOverride(name = "comision", column = @Column(name = "comision"))
    })
    private Sueldo sueldo;

    @ManyToOne
    private Departamento departamento;

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(name = "empleados_estudios",
               joinColumns = {@JoinColumn(name = "empleado_id")},
               inverseJoinColumns = {@JoinColumn(name = "estudio_id")})
    private List<Estudio> estudios = new ArrayList();

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getNombre() {..}
    public void setNombre(String nombre) {..}
    public String getOficio() {..}
    public void setOficio(String oficio) {..}
    public LocalDate getFechaAlta() {..}
    public void setFechaAlta(LocalDate fechaAlta) {..}
    public Sueldo getSueldo() {..}
    public void setSueldo(Sueldo sueldo) {..}
    public Departamento getDepartamento() {..}
    public void setDepartamento(Departamento departamento) {..}
    public List<Estudio> getEstudios() {..}
    public void setEstudios(List<Estudio> estudios) {..}

    public Estudio addEstudio(Estudio estudio) {
        this.getEstudios().add(estudio);
        estudio.getEmpleados().add(this);

        return estudio;
    }
    public Estudio removeEstudio(Estudio estudio) {
        this.getEstudios().remove(estudio);
        estudio.getEmpleados().remove(this);

        return estudio;
    }
}

```

```

@Entity
@Table(name = "estudios")
public class Estudio {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String nombre;

    @ManyToMany(mappedBy = "estudios")
    private List<Empleado> empleados = new ArrayList<>();

    public String getNombre() {..}
    public void setNombre(String nombre) {..}
    public List<Empleado> getEmpleados() {..}
    public void setEmpleados(List<Empleado> empleados) {..}

}

public class App {
    public static void main(String[] args) {
        // Configuramos el EMF a través de la unidad de persistencia
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("Hojal");

        // Generamos un EntityManager
        EntityManager em = emf.createEntityManager();

        // Iniciamos una transacción
        em.getTransaction().begin();

        // Construimos dos departamentos
        Departamento dep1 = new Departamento();
        dep1.setNombre("RecuEnterprise");
        dep1.setLocalidad("Oreña");

        Departamento dep2 = new Departamento();
        dep2.setNombre("IngenierosPorElMundo");
        dep2.setLocalidad("Santander");

        // Construimos dos estudios
        Estudio est1 = new Estudio();
        est1.setNombre("Atletismo");
        Estudio est2 = new Estudio();
        est2.setNombre("Enfermeria");
        Estudio est3 = new Estudio();
        est3.setNombre("Ingenieria");

        // Construimos Empleado
        Empleado empleado1 = new Empleado();
        empleado1.setNombre("Recu");
        empleado1.setOficio("Atleta");
        Sueldo sueldo1 = new Sueldo();
        sueldo1.setSalario(30000);
        sueldo1.setComision(2000);
        empleado1.setSueldo(sueldo1);
        empleado1.setFechaAlta(LocalDate.now());
        empleado1.setDepartamento(dep1);
        empleado1.addEstudio(est1);
    }
}

```

```

// Construimos Empleado
Empleado empleado2 = new Empleado();
empleado2.setNombre("Leras");
empleado2.setOficio("Ingeniero");
Sueldo sueldo2 = new Sueldo();
sueldo2.setSalario(3000);
sueldo2.setComision(200);
empleado2.setSueldo(sueldo2);
empleado2.setFechaAlta(LocalDate.now());
empleado2.setDepartamento(dep2);
empleado2.addEstudio(est3);

// Construimos Empleado
Empleado empleado3 = new Empleado();
empleado3.setNombre("Maria");
empleado3.setOficio("Enfermera");
Sueldo sueldo3 = new Sueldo();
sueldo3.setSalario(5000);
sueldo3.setComision(300);
empleado3.setSueldo(sueldo3);
empleado3.setFechaAlta(LocalDate.now());
empleado3.setDepartamento(dep1);
empleado3.addEstudio(est2);

// Construimos Empleado
Empleado empleado4 = new Empleado();
empleado4.setNombre("Alex");
empleado4.setOficio("Ingeniero");
Sueldo sueldo4 = new Sueldo();
sueldo4.setSalario(3000);
sueldo4.setComision(100);
empleado4.setSueldo(sueldo4);
empleado4.setFechaAlta(LocalDate.now());
empleado4.setDepartamento(dep2);
empleado4.addEstudio(est3);

// Persistimos los objetos
em.persist(empleado1);
em.persist(empleado2);
em.persist(empleado3);
em.persist(empleado4);

// Comitemos la transaccion
em.getTransaction().commit();

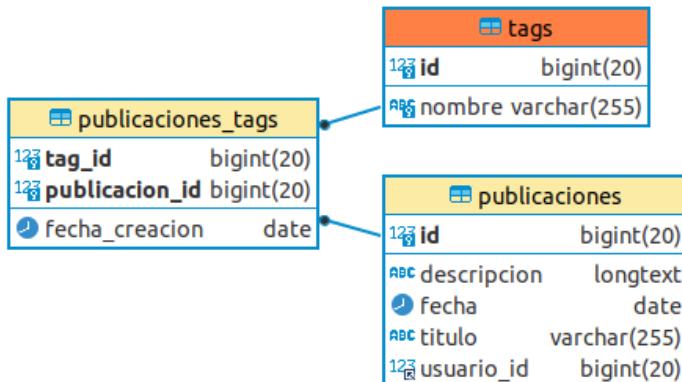
// Cerramos el EntityManager
em.close();
}

```

# Ejemplo\_6: Asociaciones III

## Ejemplo\_6.1: Many-to-Many con atributos extra

- A veces necesitamos añadir algún atributo a la asociación. Por ejemplo, si queremos guardar cuándo se establece un tag para una publicación.



- Crear otra clase **PublicacionTagId** para manejar la clave primaria compuesta
  - @Id sólo está permitida para claves simples.
  - JPA nos obliga a usar **@IdClass** (usaremos esta) o **@EmbeddId**
    - Crearemos una clase pública que implemente **Serializable**
    - Debe implementar **equals** y **hashCode**
    - Debe tener un constructor por defecto y no debe tener clave propia

```
public class PublicacionTagId implements Serializable{
    private static final long serialVersionUID = 1L;

    private Long publicacion;
    private Long tag;

    public Long getPublicacion() {}
    public void setPublicacion(Long publicacion) {}
    public Long getTag() {}
    public void setTag(Long tag) {}
    public static long getSerialversionuid() {}
    @Override
    public int hashCode() {
        return Objects.hash(publicacion, tag);
    }
    public boolean equals(Object obj) {}

}
```

- Crearemos una nueva entidad **PublicacionTag**
  - Tendrá la anotación `@IdClass`
  - Contiene dos `@ManyToOne`

```

@Entity
@IdClass(PublicacionTagId.class)
@Table(name = "publicaciones_tags")
public class PublicacionTag {
    @Id
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "publicacion_id", insertable = false, updatable = false)
    private Publicacion publicacion;

    @Id
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "tag_id", insertable = false, updatable = false)
    private Tag tag;

    @Column(name = "fecha_creacion")
    private LocalDate fechaCreacion;

    public PublicacionTag() {}
    public PublicacionTag(Publicacion publicacion, Tag tag) {}
    public PublicacionTag(Publicacion publicacion, Tag tag, LocalDate fechaCreacion) {}
    public Publicacion getPublicacion() {}
    public void setPublicacion(Publicacion publicacion) {}
    public Tag getTag() {}
    public void setTag(Tag tag) {}
    public LocalDate getFechaCreacion() {}
    public void setFechaCreacion(LocalDate fechaCreacion) {}

    @Override
    public int hashCode() {
        return Objects.hash(publicacion, tag);
    }
    public boolean equals(Object obj) {}

}

```

- Romper la asociación @ManyToMany en las dos asociaciones y añadir dos @OneToMany + @ManyToOne (ya hechas en PublicacionTag)

```

@Entity
@Table(name = "tags")
public class Tag {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @NaturalId
    private String nombre;

    @OneToMany(mappedBy = "tag", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<PublicacionTag> publicaciones = new ArrayList<>();

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getNombre() {..}
    public void setNombre(String nombre) {..}
    public List<PublicacionTag> getPublicaciones() {..}
    public void setPublicaciones(List<PublicacionTag> publicaciones) {..}
    @Override
    public int hashCode() {
        return Objects.hash(nombre);
    }
    public boolean equals(Object obj) {..}

}

```

- Y por último, en la clase Publicación habrá que modificar los métodos addTag y removeTag

```

@Entity
@Table(name = "publicaciones")
public class Publicacion {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Lob
    private String descripcion;
    private LocalDate fecha;
    private String titulo;
    @ManyToOne
    private Usuario usuario;
    @OneToMany(mappedBy = "publicacion", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<PublicacionTag> tags = new ArrayList<>();

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getDescripcion() {..}
    public void setDescripcion(String descripcion) {..}
    public LocalDate getFecha() {..}
    public void setFecha(LocalDate fecha) {..}
    public String getTitulo() {..}
    public void setTitulo(String titulo) {..}
    public Usuario getUsuario() {..}
    public void setUsuario(Usuario usuario) {..}
    public List<PublicacionTag> getTags() {..}
    public void setTags(List<PublicacionTag> tags) {..}

    public Tag addTag(Tag tag) {
        return addTag(tag, LocalDate.now());
    }

    public Tag addTag(Tag tag, LocalDate fechaCreacion) {
        PublicacionTag publicacionTag = new PublicacionTag(this, tag, fechaCreacion);
        this.tags.add(publicacionTag);
        tag.getPublicaciones().add(publicacionTag);

        return tag;
    }

    public Tag removeTag(Tag tag) {
        Iterator<PublicacionTag> it = tags.iterator();
        while (it.hasNext()) {
            PublicacionTag publicacionTag = it.next();

            if (publicacionTag.getPublicacion().equals(this) && publicacionTag.getTag().equals(tag)) {
                it.remove();
                publicacionTag.getTag().getPublicaciones().remove(publicacionTag);
                publicacionTag.setPublicacion(null);
                publicacionTag.setTag(null);
            }
        }
        return tag;
    }

    @Override
    public int hashCode() {
        return Objects.hash(titulo);
    }

    public boolean equals(Object obj) {..}
}

```

# Hoja\_4

## Ejercicio\_2

2.- Copia el proyecto anterior a **Hoja05 ORM\_04-2**. Modificar ahora lo necesario para guardar también fecha de finalización del estudio para un empleado en concreto. Estaremos ante un caso de asociación **many-to-many con atributos extra**

Para ello será necesario crear las clases **EmpleadoEstudioId** y **EmpleadoEstudio**.

Es importante tener en cuenta que en la clase App primeramente hay que persistir los estudios. Luego habrá que guardar los empleados y sus departamentos. Una vez guardadas todas las entidades realizaremos la asociación entre empleados y sus estudios.

```
public class EmpleadoEstudioId implements Serializable{
    private static final long serialVersionUID = 1L;

    private Long empleado;
    private Long estudio;

    public Long getEmpleado() {..}
    public void setEmpleado(Long empleado) {..}
    public Long getEstudio() {..}
    @Override
    public int hashCode() {
        return Objects.hash(empleado, estudio);
    }
    @Override
    public boolean equals(Object obj) {..}

}
```

```

@Entity
@IdClass(EmpleadoEstudioId.class)
@Table(name = "empleados_estudios")
public class EmpleadoEstudio {
    @Id
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "empleado_id", insertable = false, updatable = false)
    private Empleado empleado;

    @Id
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "estudio_id", insertable = false, updatable = false)
    private Estudio estudio;

    @Column(name = "fecha_finalizacion")
    private LocalDate fechaFin;

    public EmpleadoEstudio() {}
    public EmpleadoEstudio(Empleado empleado, Estudio estudio) {}
    public EmpleadoEstudio(Empleado empleado, Estudio estudio, LocalDate fechaFin) {}
    public Empleado getEmpleado() {}
    public void setEmpleado(Empleado empleado) {}
    public Estudio getEstudio() {}
    public void setEstudio(Estudio estudio) {}
    public LocalDate getFechaFin() {}
    public void setFechaFin(LocalDate fechaFin) {}
    @Override
    public int hashCode() {
        return Objects.hash(empleado, estudio);
    }
    public boolean equals(Object obj) {}

}

@Entity
@Table(name = "estudios")
public class Estudio {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @NaturalId
    @Column(nullable = false)
    private String nombre;

    @OneToMany(mappedBy = "estudio", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<EmpleadoEstudio> empleados = new ArrayList<>();

    public String getNombre() {}
    public void setNombre(String nombre) {}
    public List<EmpleadoEstudio> getEmpleados() {}
    public void setEmpleados(List<EmpleadoEstudio> empleados) {}
    @Override
    public int hashCode() {
        return Objects.hash(nombre);
    }
    public boolean equals(Object obj) {}

}

```

```

@Entity
@Table(name = "empleados")
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String nombre;

    @Column(length = 50)
    private String oficio;

    @Column(name = "fecha_alta")
    private LocalDate fechaAlta;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "salario", column = @Column(name = "salario", nullable = false)),
        @AttributeOverride(name = "comision", column = @Column(name = "comision"))
    })
    private Sueldo sueldo;

    @ManyToOne
    private Departamento departamento;

    @OneToMany(mappedBy = "empleado", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<EmpleadoEstudio> estudios = new ArrayList();

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getNombre() {..}
    public void setNombre(String nombre) {..}
    public String getOficio() {..}
    public void setOficio(String oficio) {..}
    public LocalDate getFechaAlta() {..}
    public void setFechaAlta(LocalDate fechaAlta) {..}
    public Sueldo getSueldo() {..}
    public void setSueldo(Sueldo sueldo) {..}
    public Departamento getDepartamento() {..}
    public void setDepartamento(Departamento departamento) {..}
    public List<EmpleadoEstudio> getEstudios() {..}
    public void setEstudios(List<EmpleadoEstudio> estudios) {..}

    public Estudio addEstudio(Estudio estudio) {
        return addEstudio(estudio, LocalDate.now());
    }

    public Estudio addEstudio(Estudio estudio, LocalDate fechaFin) {
        EmpleadoEstudio empleadoEstudio = new EmpleadoEstudio(this, estudio, fechaFin);
        this.estudios.add(empleadoEstudio);
        estudio.getEmpleados().add(empleadoEstudio);

        return estudio;
    }

    public Estudio removeEstudio(Estudio estudio) {
        Iterator<EmpleadoEstudio> it = this.estudios.iterator();
        while(it.hasNext()) {
            EmpleadoEstudio empleadoEstudio = it.next();
            if(empleadoEstudio.getEmpleado().equals(this) && empleadoEstudio.getEstudio().equals(estudio)) {
                it.remove();
                empleadoEstudio.getEstudio().getEmpleados().remove(empleadoEstudio);
                empleadoEstudio.setEmpleado(null);
                empleadoEstudio.setEstudio(null);
            }
        }
        return estudio;
    }
}

```

```
public class App {  
    public static void main(String[] args) {  
        // Configuramos el EMF a través de la unidad de persistencia  
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("Hoja1");  
  
        // Generamos un EntityManager  
        EntityManager em = emf.createEntityManager();  
  
        // Iniciamos una transacción  
        em.getTransaction().begin();  
  
        // Construimos dos departamentos  
        Departamento dep1 = new Departamento();  
        dep1.setNombre("RecuEntreprise");  
        dep1.setLocalidad("Oreña");  
  
        Departamento dep2 = new Departamento();  
        dep2.setNombre("IngenierosPorElMundo");  
        dep2.setLocalidad("Santander");  
  
        // Persistimos los departamentos  
        em.persist(dep1);  
        em.persist(dep2);  
  
        // Construimos dos estudios  
        Estudio est1 = new Estudio();  
        est1.setNombre("Atletismo");  
        Estudio est2 = new Estudio();  
        est2.setNombre("Enfermeria");  
        Estudio est3 = new Estudio();  
        est3.setNombre("Ingenieria");  
  
        // Persistimos los estudios  
        em.persist(est1);  
        em.persist(est2);  
        em.persist(est3);
```

```

// Construimos Empleado
Empleado empleado1 = new Empleado();
empleado1.setNombre("Recu");
empleado1.setOficio("Atleta");
Sueldo sueldo1 = new Sueldo();
sueldo1.setSalario(30000);
sueldo1.setComision(2000);
empleado1.setSueldo(sueldo1);
empleado1.setFechaAlta(LocalDate.now());
empleado1.setDepartamento(dep1);
empleado1.addEstudio(est1);

// Construimos Empleado
Empleado empleado2 = new Empleado();
empleado2.setNombre("Leras");
empleado2.setOficio("Ingeniero");
Sueldo sueldo2 = new Sueldo();
sueldo2.setSalario(3000);
sueldo2.setComision(200);
empleado2.setSueldo(sueldo2);
empleado2.setFechaAlta(LocalDate.now());
empleado2.setDepartamento(dep2);
empleado2.addEstudio(est3);

// Construimos Empleado
Empleado empleado3 = new Empleado();
empleado3.setNombre("Maria");
empleado3.setOficio("Enfermera");
Sueldo sueldo3 = new Sueldo();
sueldo3.setSalario(5000);
sueldo3.setComision(300);
empleado3.setSueldo(sueldo3);
empleado3.setFechaAlta(LocalDate.now());
empleado3.setDepartamento(dep1);
empleado3.addEstudio(est2);

// Construimos Empleado
Empleado empleado4 = new Empleado();
empleado4.setNombre("Alex");
empleado4.setOficio("Ingeniero");
Sueldo sueldo4 = new Sueldo();
sueldo4.setSalario(3000);
sueldo4.setComision(100);
empleado4.setSueldo(sueldo4);
empleado4.setFechaAlta(LocalDate.now());
empleado4.setDepartamento(dep2);
empleado4.addEstudio(est3);

// Persistimos los objetos
em.persist(empleado1);
em.persist(empleado2);
em.persist(empleado3);
em.persist(empleado4);

// Comiteamos la transaccion
em.getTransaction().commit();

// Cerramos el EntityManager
em.close();
}
}

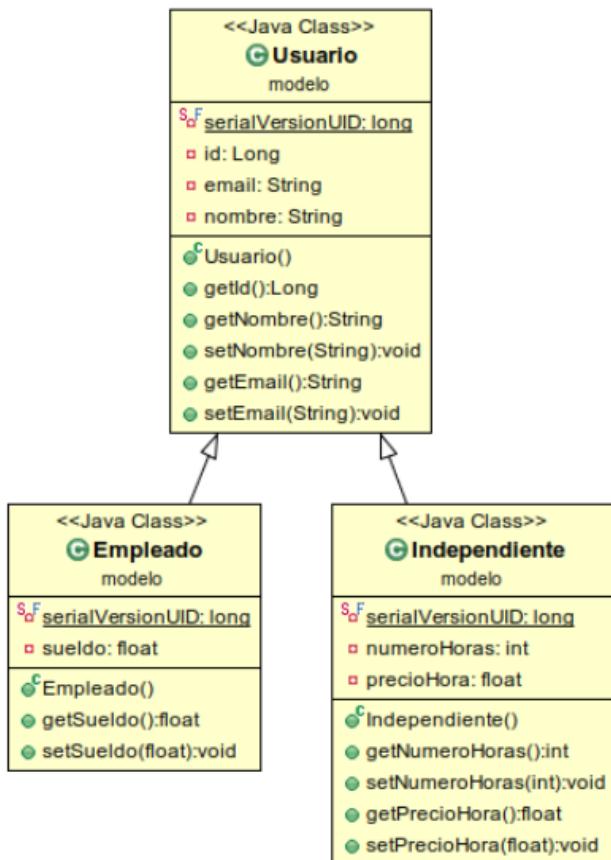
```

# Ejemplo\_7: Herencia

- La herencia es un mecanismo definido por la programación orientada a objetos, mediante el cual podemos definir una relación jerárquica entre varias clases
- JPA permite elegir entre 4 estrategias para trasladar la herencia a nuestra base de datos:
  - @MappedSuperclass
  - Table per Class
  - Single Table
  - Joined

## Ejemplo\_7.1: @MappedSuperclass

- Es el enfoque más simple para mapear una estructura de herencia a las tablas de la base de datos.
- Asigna cada clase concreta a su propia tabla, pero **no traslada la clase padre** (que no será una entidad) a la base de datos.
- Eso significa que no puede usar consultas **polimórficas** que seleccionen todas las entidades y habría que seleccionar ambas entidades por separado.



```
@MappedSuperclass
public class Usuario implements Serializable{
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String email;
    private String nombre;

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getEmail() {..}
    public void setEmail(String email) {..}
    public String getNombre() {..}
    public void setNombre(String nombre) {..}

}

@Entity
@Table(name = "empleados")
public class Empleado extends Usuario{
    private static final long serialVersionUID = 1L;

    private float sueldo;

    public Empleado() {
        super();
    }

    public float getSueldo() {..}
    public void setSueldo(float sueldo) {..}

}
```

```

@Entity
@Table(name = "independientes")
public class Independiente extends Usuario{
    private static final long serialVersionUID = 1L;

    private int numeroHoras;
    private float precioHora;

    public Independiente()
    {
        super();
    }

    public int getNumeroHoras()...
    public void setNumeroHoras(int numeroHoras)...
    public float getPrecioHora()...
    public void setPrecioHora(float precioHora)...
}

```

- Generará las siguientes tablas:

empleados	
123	<b>id</b> bigint(20)
ABC	<b>email</b> varchar(255)
ABC	<b>nombre</b> varchar(255)
123	<b>sueldo</b> float

independientes	
123	<b>id</b> bigint(20)
ABC	<b>email</b> varchar(255)
ABC	<b>nombre</b> varchar(255)
123	<b>numeroHoras</b> int(11)
123	<b>precioHora</b> float

- Podemos ver que cada una de las entidades hijas recibirá todos los atributos de la padre

## Ejemplo\_7.2: Table per Class

- Similar a MappedSuperclass pero con la diferencia que ahora la superclase es también una entidad.
- En la clase padre se deberá añadir la anotación @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)
- Se pueden utilizar consultas polimórficas, pero agrega mucha complejidad. Se hace un UNION entre los resultados de las consultas a las tablas.

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@Table(name = "usuarios")
public class Usuario implements Serializable{
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String email;
    private String nombre;

    public Long getId() {}
    public void setId(Long id) {}
    public String getEmail() {}
    public void setEmail(String email) {}
    public String getNombre() {}
    public void setNombre(String nombre) {}

}

@Entity
@Table(name = "empleados")
public class Empleado extends Usuario{
    private static final long serialVersionUID = 1L;

    private float sueldo;

    public Empleado() {
        super();
    }

    public float getSueldo() {}
    public void setSueldo(float sueldo) {}

}
```

```

@Entity
@Table(name = "independientes")
public class Independiente extends Usuario{
    private static final long serialVersionUID = 1L;

    private int numeroHoras;
    private float precioHora;

    public Independiente()
    {
        super();
    }

    public int getNumeroHoras(){}
    public void setNumeroHoras(int numeroHoras){}
    public float getPrecioHora(){}
    public void setPrecioHora(float precioHora){}
}

```

- Generará las siguientes tablas:

usuarios	empleados	independientes
123 <b>id</b> bigint(20)	123 <b>id</b> bigint(20)	123 <b>id</b> bigint(20)
abc email varchar(255)	abc email varchar(255)	abc email varchar(255)
abc nombre varchar(255)	abc nombre varchar(255)	abc nombre varchar(255)
	123 sueldo float	123 numeroHoras int(11)
		123 precioHora float

- Cada entidad hija tendrá los atributos de la clase base y los suyos propios (como en @MappedSuperclass), pero también existirá una tabla para la clase base.

## Ejemplo\_7.3: Single Table

- Asigna todas las entidades de la estructura de herencia a la misma tabla de base de datos.
- Las consultas polimórficas serán muy eficientes.
- Pero cada tupla usa solo un subconjunto de las columnas disponibles y establece el resto como nulos. Por lo tanto, no podemos definir ninguna columna como NOT NULL
- Esta estrategia es adecuada si el número de atributos que añaden las clases extendidas no es muy grande.
- En la tabla de la base de datos se añadirá un campo **DTYPE** que actúa de discriminante y almacena el nombre de la clase entidad.
- Este valor de discriminante se puede modificar mediante la anotación `@DiscriminatorValue`

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@Table(name = "usuarios")
public class Usuario implements Serializable{
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String email;
    private String nombre;

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getEmail() {..}
    public void setEmail(String email) {..}
    public String getNombre() {..}
    public void setNombre(String nombre) {..}

}
```

```

@Entity
@DiscriminatorValue("EMP")
@Table(name = "empleados")
public class Empleado extends Usuario{
    private static final long serialVersionUID = 1L;

    private float sueldo;

    public Empleado() {
        super();
    }

    public float getSueldo() {..}
    public void setSueldo(float sueldo) {..}

}

@Entity
@DiscriminatorValue("IND")
@Table(name = "independientes")
public class Independiente extends Usuario{
    private static final long serialVersionUID = 1L;

    private int numeroHoras;
    private float precioHora;

    public Independiente()
    {
        super();
    }

    public int getNumeroHoras(){..}
    public void setNumeroHoras(int numeroHoras){..}
    public float getPrecioHora(){..}
    public void setPrecioHora(float precioHora){..}
}

```

- Generará únicamente una tabla

usuários	
123	<b>Id</b> bigint(20)
ABC	<b>DTYPE</b> varchar(31)
ABC	<b>email</b> varchar(255)
ABC	<b>nombre</b> varchar(255)
123	<b>numeroHoras</b> int(11)
123	<b>precioHora</b> float
123	<b>sueldo</b> float

## Ejemplo\_7.4: Joined

- Este esquema de trabajo, también conocido como **table per subclass**, genera las siguientes tablas:
  - Una tabla para la entidad base de la jerarquía. Tendrá todos los atributos de la clase base.
  - Una tabla para cada entidad extendida de la jerarquía. Tendrá una referencia a la entidad base, y los atributos propios.
- Cada consulta de una subclase requiere una combinación de las 2 tablas (JOIN) para seleccionar las columnas de todos los atributos de la entidad.
- Eso aumenta la complejidad de cada consulta, pero también le permite utilizar restricciones no nulas en los atributos de subclase y garantizar la integridad de los datos.
- Si queremos dar un nombre diferente a las claves externas, podemos usar la anotación `@PrimaryKeyJoinColumn`

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name = "usuarios")
public class Usuario implements Serializable{
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String email;
    private String nombre;

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getEmail() {..}
    public void setEmail(String email) {..}
    public String getNombre() {..}
    public void setNombre(String nombre) {..}

}

@Entity
@PrimaryKeyJoinColumn(name = "usuario_id")
@Table(name = "empleados")
public class Empleado extends Usuario{
    private static final long serialVersionUID = 1L;

    private float sueldo;

    public Empleado() {
        super();
    }
    public float getSueldo() {..}
    public void setSueldo(float sueldo) {..}

}
```

```

@Entity
@PrimaryKeyJoinColumn(name = "usuario_id")
@Table(name = "independientes")
public class Independiente extends Usuario{
    private static final long serialVersionUID = 1L;

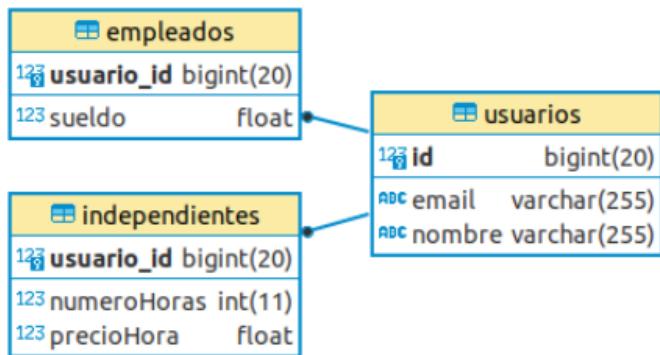
    private int numeroHoras;
    private float precioHora;

    public Independiente()
    {
        super();
    }

    public int getNumeroHoras()...
    public void setNumeroHoras(int numeroHoras)...
    public float getPrecioHora()...
    public void setPrecioHora(float precioHora)...
}

```

- Generará las siguientes tablas:



# Ejemplo 8: Valores generados

## Ejemplo\_8.1: @CreationTimestamp

- La anotación `@CreationTimestamp` nos permite indicar a Hibernate que en el atributo anotado debe almacenarse la actual fecha y hora de la JVM cuando la entidad sea almacenada.

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name = "usuarios")
public class Usuario implements Serializable{
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @CreationTimestamp
    private LocalDateTime fechaCreacion;
    private String email;
    private String nombre;

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getEmail() {..}
    public void setEmail(String email) {..}
    public String getNombre() {..}
    public void setNombre(String nombre) {..}
    public LocalDateTime getFechaCreacion() {..}
    public void setFechaCreacion(LocalDateTime fechaCreacion) {..}

}
```

## Ejemplo\_8.2: @UpdateTimestamp

- La anotación `@UpdateTimestamp` nos permite indicar a Hibernate que en el atributo anotado debe almacenarse la actual fecha y hora de la JVM cuando la entidad sea actualizada.

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name = "usuarios")
public class Usuario implements Serializable{
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @CreationTimestamp
    private LocalDateTime fechaCreacion;

    @UpdateTimestamp
    private LocalDateTime fechaActualizacion;
    private String email;
    private String nombre;

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getEmail() {..}
    public void setEmail(String email) {..}
    public String getNombre() {..}
    public void setNombre(String nombre) {..}
    public LocalDateTime getFechaCreacion() {..}
    public void setFechaCreacion(LocalDateTime fechaCreacion) {..}

}
```

## Ejemplo\_8.3: @ColumnTransformer

- Podemos personalizar el código SQL que se utiliza para leer o almacenar los valores de algunas columnas.
- Por ejemplo, podemos usar alguna función de encriptación para almacenar una columna (por ejemplo, una contraseña).

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name = "usuarios")
public class Usuario implements Serializable{
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @CreationTimestamp
    private LocalDateTime fechaCreacion;

    @UpdateTimestamp
    private LocalDateTime fechaActualizacion;

    @ColumnTransformer(write = " MD5(?) ")
    private String password;

    private String email;
    private String nombre;

    public Long getId() {..}
    public void setId(Long id) {..}
    public String getEmail() {..}
    public void setEmail(String email) {..}
    public String getNombre() {..}
    public void setNombre(String nombre) {..}
    public LocalDateTime getFechaCreacion() {..}
    public void setFechaCreacion(LocalDateTime fechaCreacion) {..}
    public LocalDateTime getFechaActualizacion() {..}
    public void setFechaActualizacion(LocalDateTime fechaActualizacion) {..}
    public String getPassword() {..}
    public void setPassword(String password) {..}

}
```

## Ejemplo\_9: Generación de esquemas

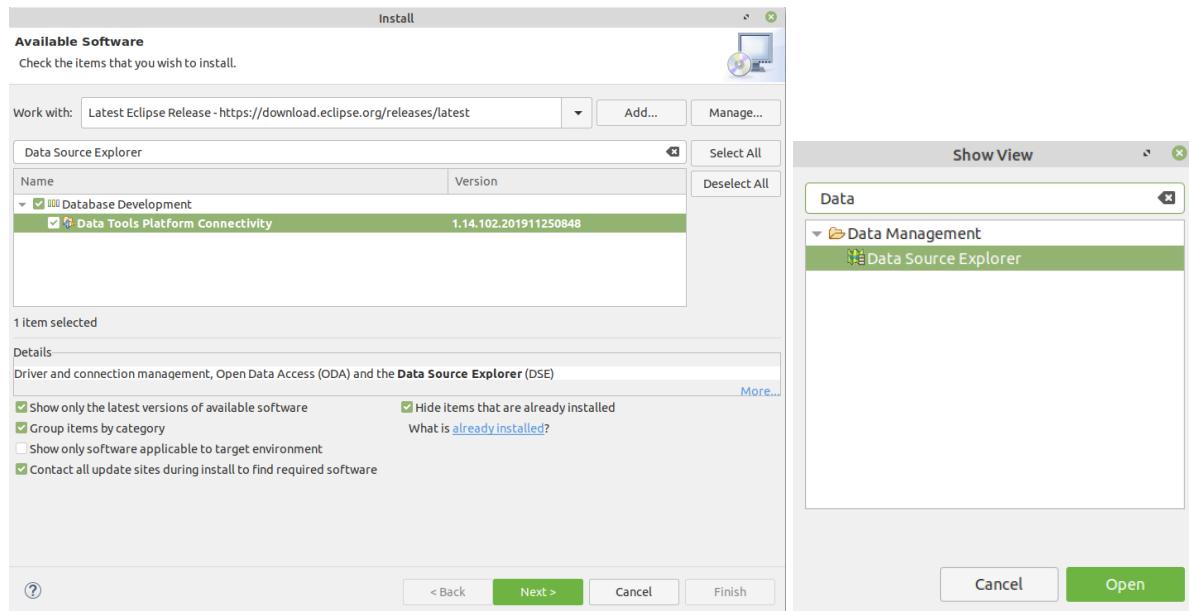
- Hemos visto que con Hibernate podemos generar el esquema de la base de datos a partir del mapeo que hayamos realizado con anotaciones.
- En un entorno de producción no se suele trabajar así.
- Podemos modificar la propiedad `hibernate.hbm2ddl.auto` que nos permite generar el DDL
- Las opciones son las siguientes:
  - **none**: valor por defecto. No realiza ninguna acción
  - **create-only**: solamente realiza el proceso de creación de la base de datos
  - **drop**: realiza sólo el borrado de la base de datos
  - **create**: realiza un borrado de la base de datos, y posteriormente su creación
  - **create-drop**: elimina el esquema y crea al crear el contexto de persistencia. Luego, se borra cuando se cierra el contexto de persistencia.
  - **validate**: valida el esquema de la base de datos
  - **update**: actualiza la base de datos, con los cambios necesarios.

### Ejemplo\_9.1: Generación de entidades de una Base de Datos

- A partir de una base de datos podemos generar las entidades necesarias para manejarla.
- Eclipse y Spring Tool Suite nos ofrece esta funcionalidad: *New > JPA Entities from Tables*

## Ejemplo\_9.2: Instalación y uso

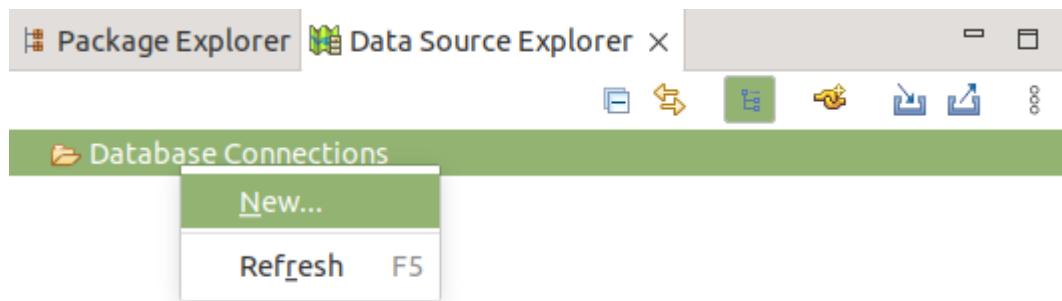
- En primer lugar vamos a instalar en Eclipse "Data Source Explorer" y visualizaremos la vista desde *Window > Show views > Data Source Explorer*.



Primero: Help -> Install New Software

Segundo: Windows -> Show Views -> Others

- Crearemos una conexión a la base de datos que queramos mapear y nos conectaremos a ella:



**New Driver Definition**

**Specify a Driver Template and Definition Name**

**Driver files not specified in driver definition.**

Name	System Vendor	System Version
Database		
Generic JDBC Driver		

Driver name: Generic JDBC Driver

Driver type: Generic JDBC Driver

**New Driver Definition**

**Specify a Driver Template and Definition Name**

**A driver already exists with that name. Please provide a unique driver name.**

Name/Type	JAR List	Properties
Driver files:		
<code>/home/usuario/.m2/repository/org/postgresql/postgresql/42.3.1/postgre</code>		
<input type="button" value="Add JAR/Zip..."/> <input type="button" value="Edit JAR/Zip..."/> <input type="button" value="Remove JAR/Zip"/> <input type="button" value="Clear All"/>		

**New Driver Definition**

**Specify a Driver Template and Definition Name**

**A driver already exists with that name. Please provide a unique driver name.**

Name/Type	JAR List	Properties												
Properties:														
<table border="1"> <thead> <tr> <th>Property</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>General</td> <td></td> </tr> <tr> <td>Connection URL</td> <td>jdbc:datoscoces</td> </tr> <tr> <td>Database Name</td> <td></td> </tr> <tr> <td>Driver Class</td> <td></td> </tr> <tr> <td>User ID</td> <td></td> </tr> </tbody> </table>			Property	Value	General		Connection URL	jdbc:datoscoces	Database Name		Driver Class		User ID	
Property	Value													
General														
Connection URL	jdbc:datoscoces													
Database Name														
Driver Class														
User ID														

**Available Classes from Jar List**

Provide the name of the driver class or select a class from the available jars.

Type class name

Browse for class

`org.postgresql.Driver`

**New Driver Definition**

**Specify a Driver Template and Definition Name**

**A driver already exists with that name. Please provide a unique driver name.**

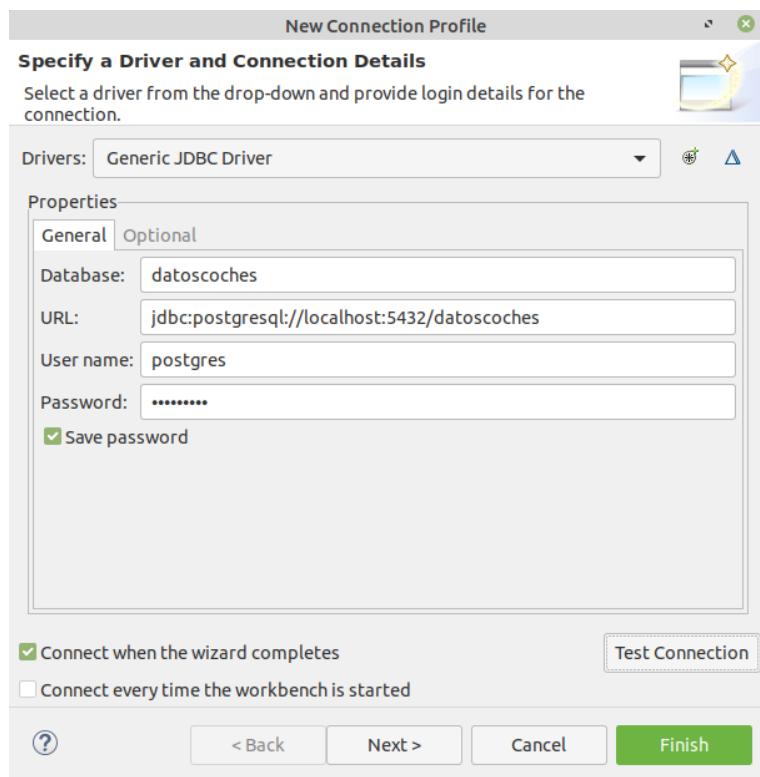
Name/Type	JAR List	Properties												
Properties:														
<table border="1"> <thead> <tr> <th>Property</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>General</td> <td></td> </tr> <tr> <td>Connection URL</td> <td>jdbc:datoscoces</td> </tr> <tr> <td>Database Name</td> <td></td> </tr> <tr> <td>Driver Class</td> <td></td> </tr> <tr> <td>User ID</td> <td></td> </tr> </tbody> </table>			Property	Value	General		Connection URL	jdbc:datoscoces	Database Name		Driver Class		User ID	
Property	Value													
General														
Connection URL	jdbc:datoscoces													
Database Name														
Driver Class														
User ID														

**New Driver Definition**

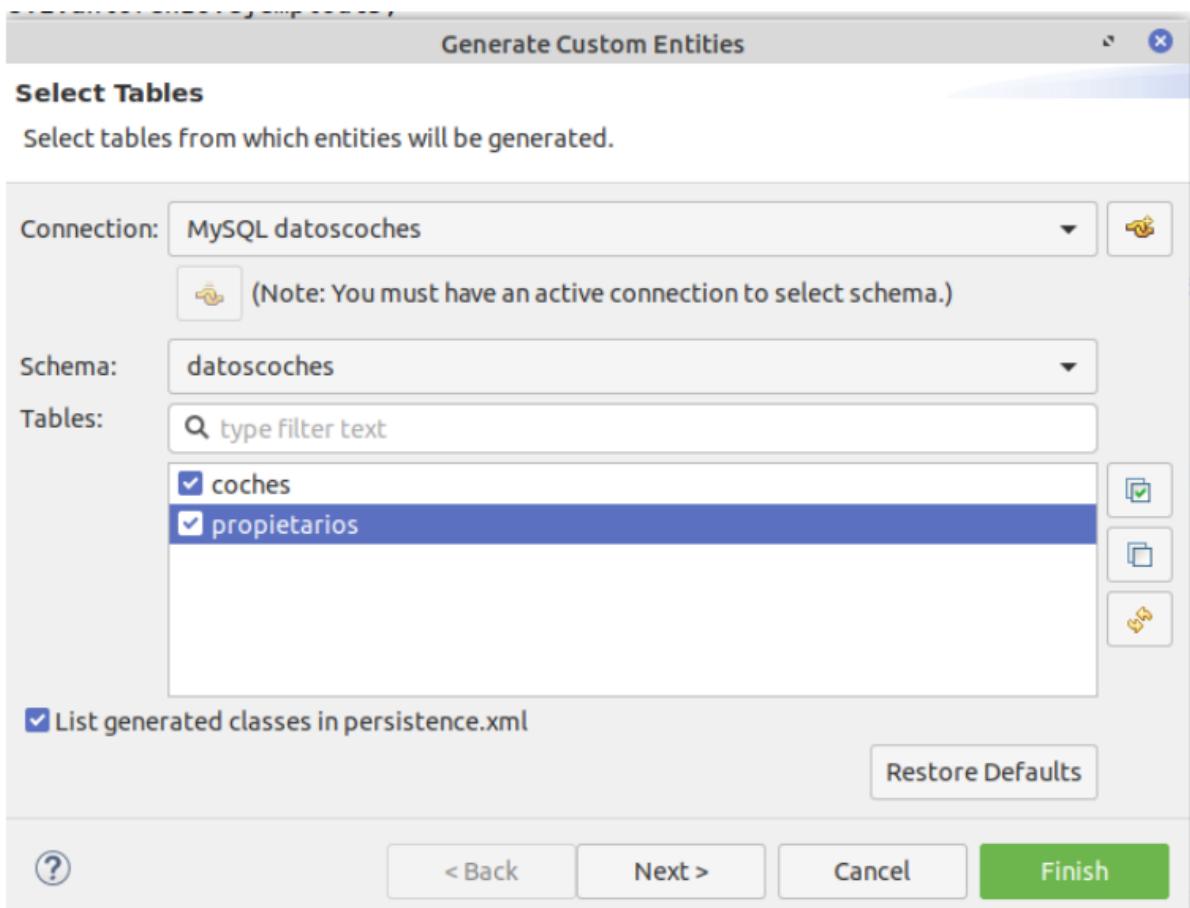
**Specify a Driver Template and Definition Name**

**A driver already exists with that name. Please provide a unique driver name.**

Name/Type	JAR List	Properties												
Properties:														
<table border="1"> <thead> <tr> <th>Property</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>General</td> <td></td> </tr> <tr> <td>Connection URL</td> <td>jdbc:datoscoces</td> </tr> <tr> <td>Database Name</td> <td></td> </tr> <tr> <td>Driver Class</td> <td></td> </tr> <tr> <td>User ID</td> <td></td> </tr> </tbody> </table>			Property	Value	General		Connection URL	jdbc:datoscoces	Database Name		Driver Class		User ID	
Property	Value													
General														
Connection URL	jdbc:datoscoces													
Database Name														
Driver Class														
User ID														



- Seleccionamos los tablas cuyas entidades vamos a generar



- Se muestran las tablas y las asociaciones con su cardinalidad

Generate Custom Entities

**Table Associations**

Edit a table association by selecting it and modifying the controls in the editing panel.

Table associations

 coches \*  propietarios 1

Each propietarios has many coches.

+ ×

?

< Back

Next >

Cancel

Finish

- En el siguiente paso, seleccionamos algunos elementos importantes:
  - El generador de identificadores
  - El tipo de acceso: a través de los atributos (field) o de los métodos get (property)
  - El tipo de fetch de las asociaciones: por defecto, ágil (eager) o perezoso (lazy)
  - Más opciones como el paquete, si heredan de alguna clase padre y las interfaces que implementan
- En el último paso podemos personalizar las entidades que se van a generar sus atributos

**Generate Custom Entities**

### Customize Defaults

Optionally customize aspects of entities that will be generated by default from database tables. A Java package should be specified.

**Mapping defaults**

Key generator:

Sequence name:

You can use the patterns \$table and/or \$pk in the sequence name. These patterns will be replaced by the table name and the primary key column name when a table mapping is generated.

Entity access:  Field  Property

Associations fetch:  Default  Eager  Lazy

Collection properties type:  java.util.Set  java.util.List

Always generate optional JPA annotations and DDL parameters

**Domain java class**

Source Folder:

Package:

Superclass:

Interfaces:  java.io.Serializable

**?** < Back Next > Cancel **Finish**

**Generate Custom Entities**

### Customize Individual Entities

**Tables and columns**

- coches
  - matricula
  - marca
  - precio
- proprietarios
  - DNI
  - edad
  - nombre

**Mapping defaults**

Class name:

Key generator:

Sequence name:

You can use the patterns \$table and/or \$pk in the sequence name. These patterns will be replaced by the table name and the primary key column name when a table mapping is generated.

Entity access:  Field  Property

**Domain java class**

Superclass:

Interfaces:  java.io.Serializable

**?** < Back Next > Cancel **Finish**

- Cuando finalicemos el asistente, generará las siguientes clases y las añadirá a nuestra unidad de persistencia:

```

@Entity
@Table(name="propietarios")
@NamedQuery(name="Propietario.findAll", query="SELECT p FROM Propietario p")
public class Propietario implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="DNI")
    private String dni;

    private int edad;

    private String nombre;

    //bi-directional many-to-one association to Coche
    @OneToMany(mappedBy="propietario")
    private List<Coche> coches;

    public Propietario() {...}
    public String getDni() {...}
    public void setDni(String dni) {...}
    public int getEdad() {...}
    public void setEdad(int edad) {...}
    public String getNombre() {...}
    public void setNombre(String nombre) {...}
    public List<Coche> getCoches() {...}
    public void setCoches(List<Coche> coches) {...}
    public Coche addCoche(Coche coche) {
        getCoches().add(coche);
        coche.setPropietario(this);

        return coche;
    }
    public Coche removeCoche(Coche coche) {
        getCoches().remove(coche);
        coche.setPropietario(null);

        return coche;
    }
}

```

```
@Entity
@Table(name = "coches")
@NamedQuery(name = "Coche.findAll", query = "SELECT c FROM Coche c")
public class Coche implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private String matricula;

    private String marca;

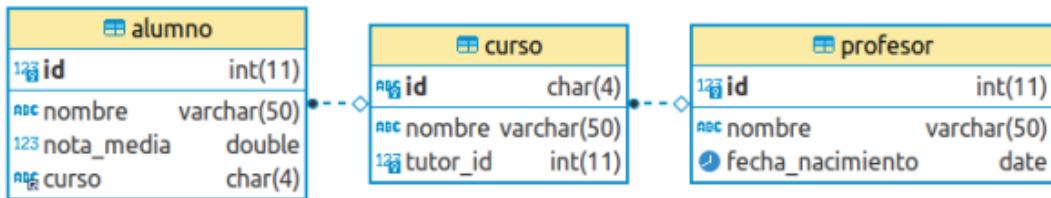
    private float precio;

    // bi-directional many-to-one association to Propietario
    @ManyToOne
    @JoinColumn(name = "DNI")
    private Propietario propietario;

    public Coche() {}
    public String getMatricula() {}
    public void setMatricula(String matricula) {}
    public String getMarca() {}
    public void setMarca(String marca) {}
    public float getPrecio() {}
    public void setPrecio(float precio) {}
    public Propietario getPropietario() {}
    public void setPropietario(Propietario propietario) {}
}
```

# Hoja\_6

1.- Crea la base de datos **cursos** en **MariaDB** e importa el script **cursos.sql**



2.- Crea un proyecto llamado **Hoja05\_ORM\_06** y genera las entidades a partir de la base de datos cursos. Las entidades se crearán en el paquete **modelo**

Recuerda poner la propiedad **hibernate.hbm2ddl.auto** a **none** para que no se modifique el DDL

```
<property name="hibernate.hbm2ddl.auto" value="none"/>
```

# Ejemplo\_10

## Ejemplo\_10.1: JPQL Básico

- Hasta ahora hemos hecho alguna consulta simple con el método `find`.
- Pero este método es muy limitado ya que sólo podemos obtener un elemento de una entidad a partir de su clave primaria.

```
Propietario propietario = em.find(Propietario.class, "1A");
System.out.println(propietario.getNombre());
```

- JPQL nos permite hacer consultas en base a **muchos criterios**.
- También así como obtener **más de un valor** por consulta
- La consulta JPQL más básica tiene la siguiente estructura:

```
SELECT p FROM Propietario p
```

- En JPA tenemos dos interfaces para crear consultas. Son `Query` y `TypedQuery` del paquete `javax.persistence`
- Podemos crear una consulta con el método `createQuery` del `EntityManager`.

```
Query query = em.createQuery(
    "SELECT p FROM Propietario p");
List<Propietario> propietarios= (List<Propietario>) query.getResultList();
propietarios.forEach(p -> System.out.println(p.getNombre()));
```

- Como se puede ver, para las consultas que devuelven **más de un resultado** usaremos el método `getResultList()`
- El método devuelve una `List` y debemos hacerle un cast al tipo de objeto adecuado. Como el compilador no puede checkear que el cast sea correcto nos muestra un warning. Si queremos quitarlo podemos usar la anotación `@SuppressWarnings("unchecked")`
- Si no queremos tener un warning podemos usar `TypedQuery`

```
TypedQuery<Propietario> query = em.createQuery(
    "SELECT p FROM Propietario p",
    Propietario.class);
List<Propietario> propietarios= query.getResultList();
propietarios.forEach(p -> System.out.println(p.getNombre()));
```

- Si la consulta devuelve un solo resultado, tendremos entonces que llamar a `getSingleResult()`

```
TypedQuery<Long> query = em.createQuery(
    "SELECT COUNT(c) FROM Coche c",
    Long.class);
long coches = query.getSingleResult();
System.out.println("Hay " + coches + " coches en la base de datos");
```

## Ejemplo\_10.2: SELECT en JPQL

- Las palabras SELECT y FROM tienen el mismo significado que en el lenguaje SQL.
- Además, la p será un alias

```
SELECT p.nombre, p.edad FROM Propietario p
```

### Parámetros

- Si queremos incluir parámetros en una consulta podemos hacerlo en base a un índice

```
SELECT p FROM Propietario p WHERE p.edad BETWEEN ?1 AND ?2
```

- También podemos usar un nombre

```
SELECT c FROM Coche c WHERE c.marca = :marca
```

- Si nuestra consulta tiene parámetros podemos usar el método setParameter
- Como hemos visto, acepta tanto un índice como un nombre de parámetro

```
TypedQuery<Propietario> query = em.createQuery(  
        "SELECT p FROM Propietario p WHERE p.edad BETWEEN ?1 AND ?2 ORDER BY p.edad DESC",  
        Propietario.class);  
query.setParameter(1, 30);  
query.setParameter(2, 40);  
  
List<Propietario> propietarios= query.getResultList();  
propietarios.forEach(p -> System.out.println(p.getNombre()));  
  
TypedQuery<Coche> query = em.createQuery(  
        "SELECT c FROM Coche c WHERE c.marca = :marca",  
        Coche.class);  
query.setParameter("marca", "Opel");  
  
List<Coche> coches= query.getResultList();  
coches.forEach(c -> System.out.println(c.getMatricula()));
```

### Establecer orden

- Podemos establecer un orden con la cláusula ORDER BY

```
SELECT c FROM Coche c ORDER BY c.precio DESC
```

### Múltiples valores devueltos

- Si la consulta devuelve varios campos de una tabla (pero no todos), obtendremos un Object[]

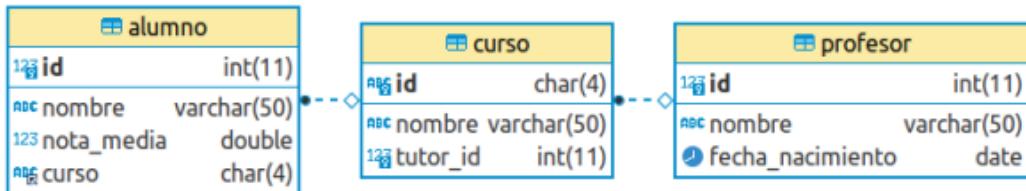
```
TypedQuery<Object[]> query = em.createQuery(  
        "SELECT p.nombre, p.edad FROM Propietario p",  
        Object[].class);  
  
List<Object[]> propietarios = query.getResultList();  
propietarios.forEach(p -> System.out.println((String)p[0] + ":" + p[1]));
```

### Group by

- También tenemos la cláusula group by que permite la agregación de valores según un conjunto de propiedades.

```
TypedQuery<Object[]> query = em.createQuery(  
        "SELECT c.marca, count(*) FROM Coche c GROUP BY c.marca", Object[].class);  
List<Object[]> coches = query.getResultList();  
coches.forEach(c -> System.out.println(c[0] + " -> " + c[1]));
```

1.- Utilizaremos la base de datos **curso**s de **MariaDB**



En el proyecto **Hoja05\_ORM\_06** en el que ya tenemos las entidades crearemos un menú con las siguientes opciones:

- 1.- Datos de alumno
- 2.- Datos de curso
- 3.- Listado de cursos
- 4.- Listado de alumnos de curso
- 0.- Salir

Debes desarrollar el programa para implementar la funcionalidad de cada uno de los métodos:

- 1.- En **datosAlumno** se ha de mostrar el nombre del alumno cuyo id se introduzca por teclado y el código (id) del curso al que pertenece.
- 2.- En **datosCurso** se mostrará el título del curso cuyo id se hay dado por teclado y el nombre del profesor tutor del curso.
- 3.- En **listadoCursos** se muestra un listado de todos los cursos existentes.
- 4.- En **listadoAlumnosCurso** se pide el código (id) de un curso y, si existe, se muestran los nombres de todos sus alumnos.

## Ejemplo\_10.3: JOINs con JPQL

- JOIN nos sirve para realizar una unión entre dos tablas a partir de una clave externa.
- JPQL permite realizar los mismos JOIN que en SQL (JOIN, LEFT | RIGHT JOIN)

### Join implícito

- El desarrollador no especifica ninguna unión
- Cuando hay asociaciones JPA crea automáticamente un join implícito

```
TypedQuery<Propietario> query = em.createQuery("SELECT c.propietario FROM Coche c",
                                                Propietario.class);
List<Propietario> propietarios = query.getResultList();
propietarios.forEach(p -> System.out.println(p.getNombre()));
```

### Join explícito

- Aquí especificamos el JOIN
- Las consultas SQL resultantes serán muy similares

```
TypedQuery<Propietario> query = em.createQuery(
    "SELECT p FROM Coche c JOIN c.propietario p",
    Propietario.class);
List<Propietario> propietarios = query.getResultList();
propietarios.forEach(p -> System.out.println(p.getNombre()));
```

### Join Fetch

- Observa el siguiente código:

```
TypedQuery<Propietario> query = em.createQuery(
    "SELECT p FROM Propietario p", Propietario.class);
List<Propietario> propietarios = query.getResultList();
for (Propietario propietario : propietarios)
{
    propietario.getCoches().forEach(c -> System.out.println(c.getMatricula()));
}
```

- Se realizan **n+1 queries**. En primer lugar se hace una para buscar todos los propietarios. Y luego para cada propietario se buscan sus coches
- Cuando haya 100 propietarios se harán 101 consultas.
- Para solucionar esto podemos usar JOIN FETCH

```
TypedQuery<Propietario> query = em.createQuery(
    "SELECT p FROM Propietario p JOIN FETCH p.coches", Propietario.class);
List<Propietario> propietarios = query.getResultList();
for (Propietario propietario : propietarios)
{
    System.out.println(propietario.getNombre());
    propietario.getCoches().forEach(c -> System.out.println("\t"+c.getMatricula()));
}
```

- Únicamente genera una única consulta
- Se incluyen los coches en la consulta a través de un JOIN
- Los frameworks de persistencia aportan muchas ventajas, pero un mal uso puede generar resultados mucho peores.

## Ejemplo\_10.4: Consultas de actualización

- JPQL también nos permite lanzar consultas de actualización de datos

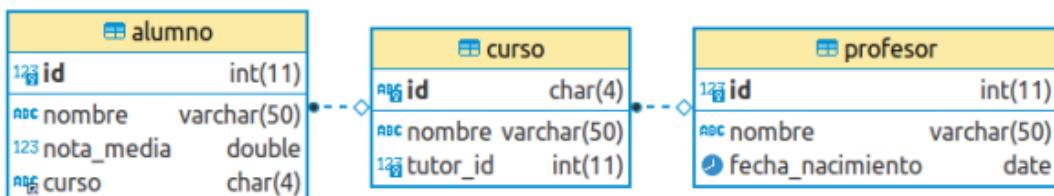
Update

```
int numeroCochesActualizados = em.createQuery(  
        "UPDATE Coche c SET c.precio = c.precio * 1.1")  
        .executeUpdate();  
System.out.println("Número de registros afectados: " + numeroCochesActualizados);
```

Delete

```
int numeroCochesBorrados = em.createQuery(  
        "DELETE Coche c WHERE c.precio > 10000")  
        .executeUpdate();  
System.out.println("Número de registros afectados: " + numeroCochesBorrados);
```

1.- Utilizaremos la base de datos **cursos de MariaDB**



En el proyecto **Hoja05\_ORM\_06** añadiremos más opciones a nuestro menú:

- 5.- Modificar título de curso
- 6.- Añadir curso
- 7.- Añadir o modificar curso
- 8.- Modificar tutor de curso
- 9.- Eliminar alumno

Debes desarrollar el programa para implementar la funcionalidad de cada uno de los métodos:

5.- En **modificarTituloCurso** se pide el código (id) de un curso y, si existe, se muestra su título, se pide un nuevo título y se modifica el nombre.

6.- En **addCurso** se pide por teclado el código (id) de un curso y su nombre y se añade a la tabla cursos. Se ha de controlar la excepción de tipo *PersistenceException* al guardar (provocada posiblemente por que el curso ya existe).

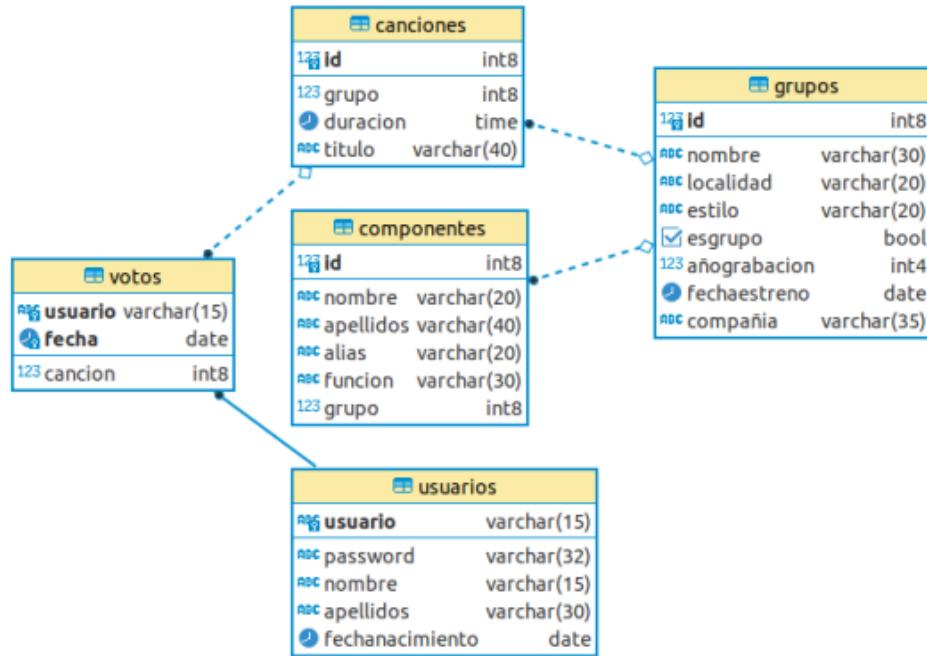
7.- En **addModificarCurso** se pide por teclado el código (id) de un curso y su nombre y, si no existe el curso, se añade a la tabla cursos. Si ya existe el curso, se ha de modificar con los nuevos datos.

8.- En **modificarTutorCurso** se pide el código (id) de un curso y, si existe, se muestra su nombre y el nombre del tutor. A continuación, se pide el id de un profesor que va a ser tutor del curso y se modifica el curso con el nuevo tutor.

Se debe controlar que, si el profesor es ya tutor de otro curso, no pueda serlo del nuevo.

- 9.- En **eliminarAlumno** se pide un id de alumno y se elimina de la tabla alumnos.

1.- Utilizaremos la base de datos **concursumusicacompleto** de PostgreSQL



Crea el proyecto **Hoja05 ORM\_09** y genera las entidades a partir de la base de datos **concursumusicacompleto**. Las entidades se crearán en el paquete **modelo**

Recuerda poner la propiedad **hibernate.hbm2ddl.auto** a **none** para que no se modifique el DDL.

```
<property name="hibernate.hbm2ddl.auto" value="none"/>
```

Realiza las modificaciones que necesites dentro de las clases.

El programa debe usar **JPQL** para realizar las operaciones contenidas en el menú:

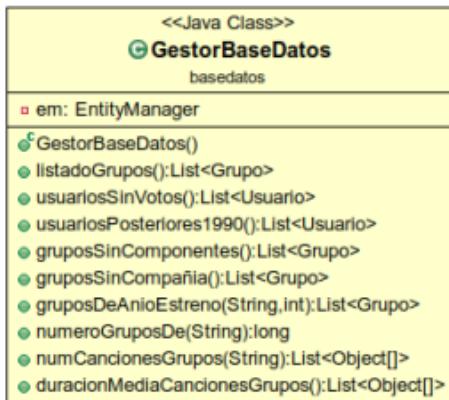
- 1.- Listado de grupos
- 2.- Listado de usuarios que no han votado ninguna vez
- 3.- Listado de usuarios nacidos a partir de 1990
- 4.- Grupos sin componentes cargados
- 5.- Grupos sin compañía cargada
- 6.- Grupos de Barcelona con primer disco en año antes de 2010
- 7.- Número de grupos de Madrid

**8.- Número de canciones de cada grupo de Madrid**

**9.- Duración media de las canciones de cada grupo**

**0.- Salir**

Debes desarrollar el programa para implementar la funcionalidad de cada una las opciones. Se realizará una clase **GestorBaseDatos** en el paquete **basedatos** que contenga los siguientes métodos:



## OPCIÓN 1

Debe mostrarse un listado de todos los grupos existentes en la tabla **grupos** ordenado por nombre de grupo y conteniendo el nombre del grupo, la localidad, la compañía (si es que la tiene) y la fecha de su estreno.

## OPCIÓN 2

Debe mostrarse un listado de todos los usuarios que no han emitido ningún voto. El listado debe estar ordenado alfabéticamente por apellidos, nombre. De cada usuario se debe mostrar nombre, apellidos y fecha de nacimiento.

## OPCIÓN 3

Debe mostrarse un listado de todos los usuarios que han nacido a partir de 1990, es decir, en ese año o en años posteriores. El listado debe estar ordenado por fecha de nacimiento desde el usuario con mayor edad. De cada usuario se debe mostrar nombre, apellidos y fecha de nacimiento.

## **OPCIÓN 4**

Debe mostrarse un listado de todos los grupos que no tienen componentes registrados en la tabla componentes. Sólo hay que mostrar los nombres de los grupos. La consulta sólo hay que realizarla sobre la tabla grupos o, lo que es lo mismo, sobre la clase Grupo (no se necesita usar la tabla componentes).

## **OPCIÓN 5**

Debe mostrarse un listado de todos los grupos que no tienen registrada compañía a la que pertenecen.

## **OPCIÓN 6**

Debe mostrarse un listado de todos los grupos cuya localidad de procedencia sea Barcelona y que hayan grabado su primer disco antes de 2010.

## **OPCIÓN 7**

Debe mostrarse solamente el número de grupos que son de la localidad de Madrid.

## **OPCIÓN 8**

Debe mostrarse un listado de todos los grupos que sean de Madrid y cuántas canciones tiene cada uno en la tabla canciones.

## **OPCIÓN 9**

Debe mostrarse un listado con los nombres de los grupos que tengan canciones cargadas y la duración media de esas canciones con el formato **mm:ss** ordenados por duración descendente.

## Ejemplo\_10.5: NamedQueries

- Para evitar consultas repetidas en distintas partes del código podemos usar **NamedQueries**
- Son predefinidas usando la notación `@NamedQuery` o `@NamedQueries` en la definición de la entidad
- Este comportamiento estático las hace más eficientes y por lo tanto ofrecen un mejor rendimiento
- El alcance es el contexto de persistencia actual, por lo tanto el nombre de cada query será único bajo el mismo contexto de persistencia

```
@Entity
@Table(name="coches")
@NamedQuery(name="Coche.findAll", query="SELECT c FROM Coche c")
public class Coche implements Serializable {
    //Resto del código
}
```

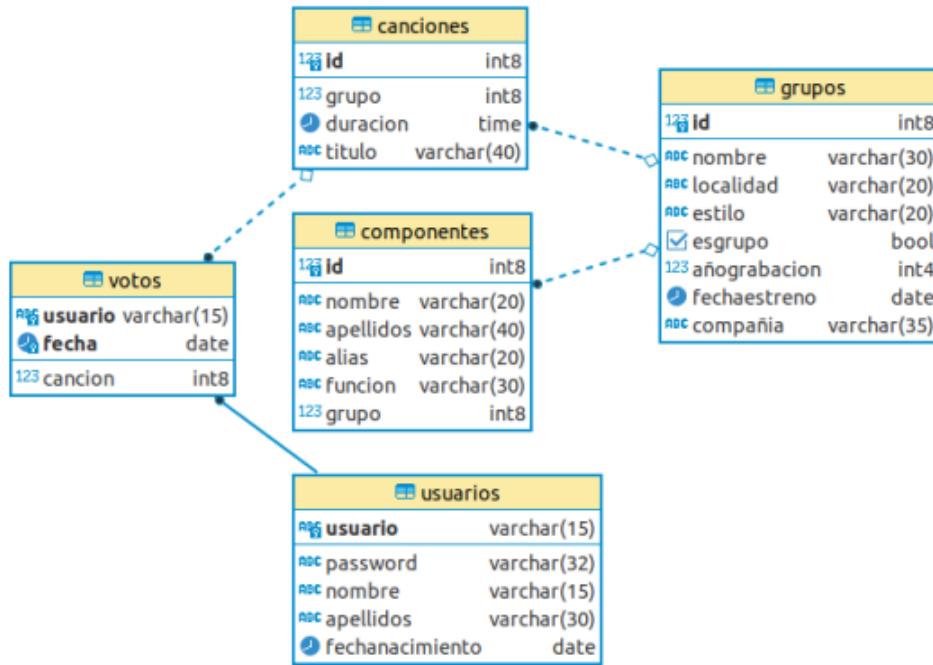
- Se necesita el nombre de la consulta y la consulta en sí
- Si queremos añadir más de una consulta usaremos `@NamedQueries`

```
@Entity
@Table(name="coches")
@NamedQueries({
    @NamedQuery(name="Coche.findAll", query="SELECT c FROM Coche c"),
    @NamedQuery(name="Coche.porPrecio", query="SELECT c FROM Coche c WHERE c.precio < :precio")
})
public class Coche implements Serializable {
    //Resto del código
}
```

- Luego creamos la consulta utilizando el método `createNamedQuery` del EntityManager.

```
TypedQuery<Coche> query = em.createNamedQuery("Coche.porPrecio", Coche.class)
    .setParameter("precio", 10000f);
List<Coche> coches = query.getResultList();
for (Coche coche : coches)
    System.out.println(coche.getMarca());
```

1.- Utilizaremos la base de datos **concursemusicacompleto** de PostgreSQL

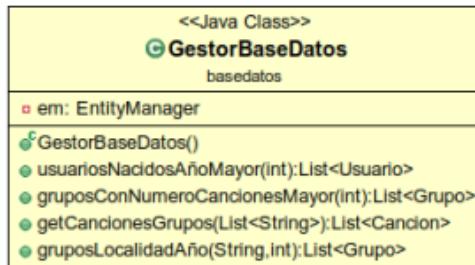


Se puede copiar el proyecto **Hoja05 ORM\_09** a **Hoja05 ORM\_10** y utilizar las entidades ya existentes en paquete **modelo**

El programa debe usar **JPQL** para realizar las operaciones contenidas en el menú:

- 1.- Listado de usuarios nacidos a partir de año ...
- 2.- Grupos que tienen mas de N canciones
- 3.- Canciones de grupos
- 4.- Grupos de localidad con primer disco en año antes de año
- 0.- Salir

Debes desarrollar el programa para implementar la funcionalidad de cada una las opciones. Se realizará una clase **GestorBaseDatos** en el paquete **basedatos** que contenga los siguientes métodos:



## OPCIÓN 1

Se pide por teclado un año y se obtiene un listado con el nombre y apellidos de los usuarios nacidos a partir de ese año.

## OPCIÓN 2

Se pide un número por teclado y se obtienen los nombres de grupos que tienen más canciones que ese número recogido por teclado. **Usa una NamedQuery para ello**

## OPCIÓN 3

Se piden por teclado los nombres de varios grupos y se realiza un listado con todas las canciones de esos grupos y el nombre del grupo al que pertenece cada canción.

## OPCIÓN 4

Se pide el nombre de una localidad y un año por teclado y se muestra un listado de todos los grupos de la localidad que han grabado su primer disco antes del año recogido. **Usa una NamedQuery para ello**

## Ejemplo\_10.6: Consultas con SQL nativo

- JPA permite la ejecución de SQL nativo.
- Útil cuando nuestro sistema tiene funcionalidades específicas
- Se usa el método `createNativeQuery` del EntityManager
- Podemos hacer **consultas escalares** que nos devuelvan un array de Object (Object[])
- También podremos **consultar entidades**

```
List<Coche> coches =  
    em.createNativeQuery("SELECT * FROM coches ORDER BY precio DESC", Coche.class)  
        .getResultList();  
    for (Coche coche : coches)  
        System.out.printf("%-10s %5.0f\n", coche.getMarca(), coche.getPrecio());
```

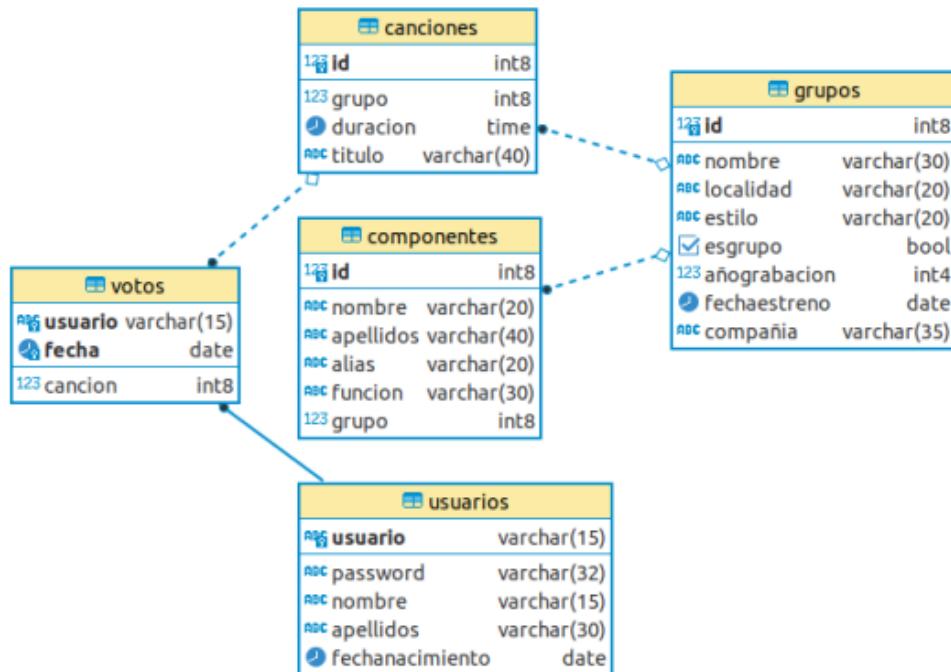
- Por último, podemos definir **consultas con nombre**, como ocurría con JPQL. Estas se definen a través de la anotación `@NamedNativeQuery`

```
@Entity  
@Table(name="coches")  
@NamedNativeQuery(name = "Coche.precioDes", query="SELECT * FROM Coche ORDER BY precio DESC",  
resultClass = Coche.class)  
public class Coche implements Serializable {  
    //...  
}
```

- Luego llamaríamos a la consulta con el método `createNamedQuery` del EntityManager

```
TypedQuery<Coche> query = em.createNamedQuery("Coche.precioDes", Coche.class);  
List<Coche> coches = query.getResultList();  
for (Coche coche : coches)  
    System.out.printf("%-10s %5.0f\n", coche.getMarca(), coche.getPrecio());
```

1.- Utilizaremos la base de datos **concursomusicacompleto** de **PostgreSQL**



Se puede copiar el proyecto **Hoja05 ORM\_10** a **Hoja05 ORM\_11** y utilizar las entidades ya existentes en paquete **modelo**

El programa debe usar **JPQL** para realizar las operaciones contenidas en el menú:

- 1.- Cambiar componente de grupo
- 2.- Borrar votos de usuario para un grupo
- 3.- Insertar votos a grupo
- 0.- Salir

Debes desarrollar el programa para implementar la funcionalidad de cada una las opciones. Se realizará una clase **GestorBaseDatos** en el paquete **basedatos** que contenga los siguientes métodos:



## OPCIÓN 1

Se pide por teclado el nombre y apellidos del componente del grupo a cambiar. También se solicita el nombre del nuevo grupo. El método tiene que modificar el grupo del componente. **Utilizar una consulta nativa con nombre**

## OPCIÓN 2

Se pide el usuario (clave primaria) y el nombre de un grupo. Se deben borrar todos los votos que ha emitido ese usuario asociados a ese grupo.

*En PostgreSQL no funciona bien el DELETE cuando hay que unir varias tablas ya que Hibernate genera “cross join” en vez de “using”. Por tanto, hay que seleccionar todos los votos e ir borrándolos uno a uno*

## OPCIÓN 3

Se debe crear un bot que inserte votos a un grupo. Se pide por teclado el nombre del grupo. La opción debe seleccionar aleatoriamente un usuario de la base de datos. Con ese usuario, se debe insertar un voto a cada canción del grupo pasado por teclado con la fecha actual.

//  
New conexion  
Seleccionar base de datos  
Editar el que hay, seleccionar los drivers  
Drivers: usuario -> .m2 -> Repository -> org -> Postgresql -> ultima version -> .jar  
postgresql - Pass!WD10 (+-)  
Probar  
Renombrar  
=====

New JPA Entity from tables.  
Seleccionar tablas.  
Comprobar relaciones.  
    Si no estan todas hay que generarlas a mano.  
    Si no estan correctas hay que corregirlas.  
Claves auto.  
Corregir si no estan bien las claves primarias.  
Finalizar.

---

A raíz de esto nos genera una las tablas en la base de datos.