

# **Formación Migración Cobol – Java**



# MÓDULO 4:

# Persistencia

- ▶ En lenguaje de programación orientado a objetos los datos se representan mediante objetos.
- ▶ Su estado es accesible a runtime y están en memoria volátil.
- ▶ Al finalizar la ejecución, su estado desaparece.
- ▶ Algunos objetos deben ser persistentes
  - ▶ deben residir en un almacén de datos permanente para que su estado se pueda reutilizar

# Persistencia

- ▶ Los datos persistentes constituyen información que puede sobrevivir al programa que la crea.
- ▶ La mayoría de los programas complejos utilizan datos persistentes.

# Alternativas para Persistencia

- ▶ Serialization
- ▶ Java Database Connectivity (JDBC)
- ▶ Object-relational mapping (ORM) propietarios
- ▶ Bases de Datos Orientadas a Objetos
- ▶ Enterprise Java Beans (EJBs)
- ▶ Java Data Object (JDO)

# Serialization

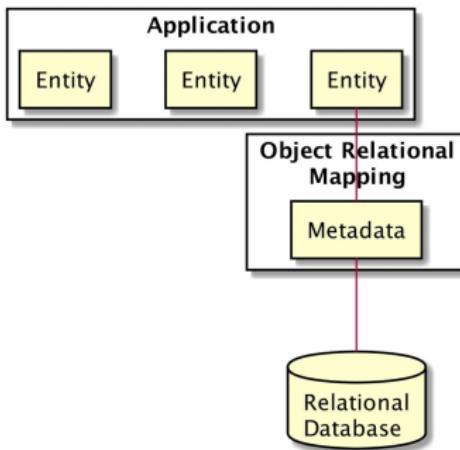
- ▶ Serialización: mecanismo incorporado en Java para transformar un grafo de objetos en una serie de bytes, que luego se pueden enviar a través de la red o almacenar en un archivo
- ▶ (+) es fácil de usar.
- ▶ (-) problemas y limitaciones.
  - ▶ INSEGURIDAD
  - ▶ Debe almacenar y recuperar todo el grafo del objeto.
  - ▶ Problemas de acceso concurrente.
  - ▶ No proporciona capacidades de consulta.

# Java Database Connectivity (JDBC)

- ▶ API para manipular datos persistentes en bases de datos relacionales.
- ▶ Tiene mecanismos para garantizar la integridad de los datos,
- ▶ admite el acceso simultáneo a la información
- ▶ proporciona un lenguaje de consulta (SQL).
- ▶ JDBC no fue diseñado para almacenar objetos y, por lo tanto, obliga a abandonar la programación orientada a objetos para las partes de código que tratan con datos persistentes
  - ▶ el programador debe asignar conceptos orientados a objetos (como herencia a bases de datos relacionales).

# Mapeo relacional de objetos (ORM)

- ▶ El principio del Mapeo Objeto-Relacional (ORM) es unir el mundo de las bases de datos relacionales y los objetos.
- ▶ Los ORM son herramientas externas que brindan una vista orientada a objetos a datos relativacionales y viceversa.



- ▶ *Lightweight persistence*: el almacenamiento y la recuperación de datos persistentes con poco o ningún trabajo por parte del desarrollador.

## (ORM) propietarios

- ▶ Productos que realizan el mapeo entre objetos y tablas de bases de datos relacionales.
- ▶ (+) Permiten concentrarse en el modelo de objetos y no preocuparse por la falta de coincidencia entre los paradigmas relacionales y orientados a objetos.
- ▶ (-) cada producto tiene su propia API.
- ▶ Dependencias de proveedor

## Base de datos objetos

- ▶ Forma de base de datos diseñada específicamente para almacenar objetos.
- ▶ (+) Facilidad de uso que el software respecto mapeo relacional de objetos (no hay que cambiar paradigma)
- ▶ Se definió una API estándar, pero pocos proveedores cumplen con las recomendaciones de la ODMG.
- ▶ En consecuencia hay una vinculación fuerte con el proveedor.
- ▶ Resistencia a cambiar la tecnología relacional
  - ▶ tecnología de base de datos de objetos relativamente desconocida.
  - ▶ Hay menos herramientas de análisis de datos disponibles
  - ▶ grandes cantidades de datos ya almacenados en bases de datos relacionales

# Enterprise Java Beans (EJB)

- ▶ La plataforma Java Enterprise Edition introdujo el concepto de Enterprise Java Beans (EJB).
- ▶ Las entidades EJB son componentes que representan información persistente en un almacén de datos.
- ▶ proporcionan una vista orientada a objetos de los datos persistentes (como los ORM)
- ▶ las entidades EJB no se limitan a las bases de datos relacionales;
  - ▶ la información persistente que representan puede provenir de un sistema de información empresarial (EIS) u otro dispositivo de almacenamiento.
- ▶ utilizan un estándar estricto, lo que las hace portátiles entre proveedores.
- ▶ limitaciones en los conceptos orientados a objetos que puede representar (herencia, polimorfismo y relaciones complejas)

# Java Data Object (JDO)

- ▶ La especificación JDO utiliza una API que es similar a JPA.
- ▶ JDO, admite bases de datos no relacionales

# Serialización

- ▶ Un framework para codificar objetos como streams de bytes (serializar) y reconstruir objetos a partir de sus codificaciones (deserializar).
- ▶ La codificación puede enviarse de una VM a otra o almacenarse en disco para su posterior deserialización.
- ▶ Una instancia de cualquier clase que implemente la interface Serializable se puede guardar y restaurar desde una secuencia de bytes.
- ▶ Serializable es una Marker Interface/tagger:
  - ▶ no tiene métodos (ni campos).
  - ▶ las clases que la implementan no tienen que implementar ningún método.
  - ▶ Indicación para el compilador y/o VM (similar a las anotaciones que son posteriores).

# Ejemplo

```
1 public class Persona implements Serializable {  
2     private String nombre;  
3     private String apellido;  
4     private LocalDate fechaNacimiento;  
5     //constructor  
6     //getters y setters  
7 }
```

# Serialización

- ▶ Utilizamos el método `writeObject` de la clase `ObjectOutputStream` (subclase de `Stream`), que serializa al objeto en un `OutputStream`.

```
1 String filepath = "resources/persona1.serial";
2 Persona persona = new Persona("Juana", "Perez", LocalDate.
    of(1989, 10, 3));
3 FileOutputStream archivo = new FileOutputStream(filepath);
4 ObjectOutputStream output = new ObjectOutputStream(archivo)
    ;
5 output.writeObject(persona);
```

- ▶ El archivo `resources/persona1.serial` contiene una serialización de persona

# Deserialización

- ▶ Utilizamos el método `readObject` de la clase `ObjectInputStream` (subclase de `Stream`) para deserializar un objeto desde un `InputStream`.

```
1 String filepath = "resources/persona1.serial";
2 FileInputStream archivo = new FileInputStream(filepath);
3 ObjectInputStream input = new ObjectInputStream(archivo);
4 Persona p = (Persona) input.readObject();
5 assertEquals("Perez",p.getApellido());
6 assertEquals("Juana",p.getNombre());
7 assertEquals(LocalDate.of(1989, 10, 3), p.
    getFechaNacimiento());
```

# Serialización y modificación de clases

- ▶ Suponer que redefinimos `toString` en `Persona`.

```
1 public class Persona implements Serializable {  
2     ....  
3     @Override  
4     public String toString() {  
5         return "Persona [nombre=" + nombre + ", apellido=" +  
6             apellido + ","  
7             + " fechaNacimiento=" + fechaNacimiento + "]";  
8     }  
}
```

- ▶ Si intentamos deserializar el archivo generado anteriormente, obtenemos

```
java.io.InvalidClassException:.... local class  
incompatible: stream classdesc serialVersionUID = ...
```

## Serialización y modificación de clases

- ▶ Este mecanismo es alterado si se define el atributo de clase  
**static final long serialVersionUID = ....**
- ▶ Si la clase define serialVersionUID se usa su valor en lugar de calcular el hash en la clase.
- ▶ Este valor se escribirá con las versiones serializadas y se usará para comparar al deserializar.
- ▶ De esta manera el programador controla qué versiones de la clase son compatibles con qué representaciones serializadas.
- ▶ Podemos crear nuestra clase serializable desde el principio con SUID = 1L y solo incrementarlo si hay cambios verdaderamente incompatibles.
- ▶ Sino se planeo, se puede obtener el valor sobre la version original (Eclipse, usar el agregar SUID generado sobre la clase original).

## Serialización y modificación de clases

- ▶ Este mecanismo es alterado si se define el atributo de clase  
**static final long serialVersionUID = ....**
- ▶ Si la clase define serialVersionUID se usa su valor en lugar de calcular el hash en la clase.
- ▶ Este valor se escribirá con las versiones serializadas y se usará para comparar al deserializar.
- ▶ De esta manera el programador controla qué versiones de la clase son compatibles con qué representaciones serializadas.
- ▶ Podemos crear nuestra clase serializable desde el principio con SUID = 1L y solo incrementarlo si hay cambios verdaderamente incompatibles.
- ▶ Sino se planeo, se puede obtener el valor sobre la version original (Eclipse, usar el agregar SUID generado sobre la clase original).

# Serialización y modificación de clases

- ▶ Las versiones pueden ser compatibles aún si se eliminan o agregan atributos (se buscan por nombres).
  - ▶ Los atributos sobrantes se ignoran al deserializar;
  - ▶ Los atributos faltantes se asignan valores por default (`0`, `false`, `null`)
- ▶ Sin embargo, no cualquier cambio está permitido (por ejemplo, cambiar el tipo de un atributo).

# Atributos transientes

- ▶ Atributos que no se serializan.
- ▶ Se declaran con el modificador **transient**

```
1 public class Persona implements Serializable {  
2     ...  
3     private transient int edad;  
4  
5     public Persona(String nombre, String apellido, LocalDate  
6                     fechaNacimiento) {  
7         this.nombre = nombre;  
8         this.apellido = apellido;  
9         this.fechaNacimiento = fechaNacimiento;  
10        edad = Period.between(fechaNacimiento, LocalDate.now())  
11                      .getYears();  
12    }  
13    ...  
14    public int getEdad() {  
15        return edad;  
16    }  
17}
```

- ▶ Los atributos transientes no se serializan.

## Metodo `readObject`

- ▶ A menudo, la simple deserialización por sí sola no es suficiente para reconstruir el estado completo de un objeto.
- ▶ Por ejemplo
  - ▶ el objeto tiene campos **transient**.
  - ▶ La clase fue extendida con nuevos atributos.
- ▶ Los objetos pueden hacer su propia configuración después de la deserialización implementando el método especial `readObject()`.
- ▶ Para ser reconocido y utilizado, debe ser privado y tener la firma **private void** `readObject(ObjectInputStream s)`

## Método readObject

```
1  public class Persona implements Serializable {  
2      ...  
3      private transient int edad;  
4  
5      public Persona(String nombre, String apellido, LocalDate  
6                      fechaNacimiento) {  
7          this.nombre = nombre;  
8          this.apellido = apellido;  
9          this.fechaNacimiento = fechaNacimiento;  
10         calcularEdad();  
11     }  
12     ...  
13     private void calcularEdad() {  
14         edad = Period.between(fechaNacimiento, LocalDate.now()).  
15             getYears();  
16     }  
17  
18     private void readObject(ObjectInputStream s)  
19         throws IOException, ClassNotFoundException {  
20         s.defaultReadObject();  
21         calcularEdad();  
22     }  
23 }
```

# Objetos compuestos

```
1 public class Persona implements Serializable {  
2     ...  
3     private Direccion direccion;  
4     ...  
5     //getters y setters para direccion;  
6 }  
7  
8 public class Direccion{  
9     private String calle;  
10    private String numero;  
11    private String ciudad;  
12    //Constructor, getters y setters  
13 }
```

- ▶ Al intentar serializar una instancia de Persona ocurre `java.io.InvalidClassException:`  
`....Direccion; class invalid for deserialization`
- ▶ Todos los objetos contenidos (no **transient**) deben implementar serializable.
- ▶ Hay problemas al deserializar si las clases contenidas cambian.

# Riesgos de la serialización

- ▶ Alto potencial de ataques:
  - ▶ `readObject` de `ObjectInputStream` es un constructor mágico que puede:
    - ▶ crear instancias de cualquier tipo que implemente `Serializable`.
    - ▶ ejecutar código de cualquiera de estos tipos durante la deserialización.
  - ▶ Esto da un gran abanico para posibles ataques.
- ▶ Ataques notorios (San Francisco Metropolitan Transit Agency Municipal Railway, 2 días fuera '16), bombas de deserialización (DoS sólo usando `HashSets`)

# Buenas prácticas

- ▶ En sistemas nuevos:
  - ▶ No usar serialización Java
  - ▶ Adoptar mecanismos de representación de objetos como datos estructurados (tipo, JSON y protobuf).
- ▶ En sistemas existentes:
  - ▶ Nunca deserializar datos de origen no confiable
  - ▶ Si es inevitable, usar `java.io.ObjectInputFilter` que permite controlar el proceso de deserialización.
  - ▶ Diseñar cuidadosamente la clase a serializar (y los métodos de `readObject` y `writeObject`).

# JDBC: Java DataBase Connectivity

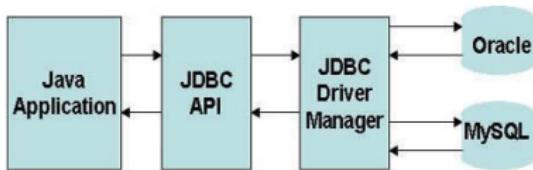
- ▶ Proporciona acceso a los datos desde Java.
- ▶ Una base común sobre la cual se construyen herramientas y otras interfaces.
- ▶ Se compone de dos paquetes:
  - ▶ java.sql
  - ▶ javax.sql
- ▶ apareció en 1997 y está incluida desde JSE 8
- ▶ Es muy robusta pero de bajo nivel (requiere varias líneas de código para vincular un objeto a una consulta SQL).

# Vamos a necesitar una BD

- ▶ Crearse una cuenta en una base online (▶ [freesqldatabase](#))

# JDBC: Java DataBase Connectivity

- ▶ JDBC (Java Database Connectivity) que es una API para conectar y ejecutar consultas en una base de datos.
- ▶ Para utilizar la API con una base de datos en particular se necesita un controlador que media entre JDBC y la base de datos.



# JDBC: Java DataBase Connectivity

- ▶ Vamos a usar un conector mysql
- ▶ Lo seteamos con Maven

```
1 <dependencies>
2   <dependency>
3     <groupId>mysql</groupId>
4     <artifactId>mysql-connector-java</artifactId>
5     <version>8.0.28</version>
6   </dependency>
7 </dependencies>
```

# Pasos para interactuar con una BD

1. Establecer una conexión.
2. Crear una sentencia.
3. Ejecutar la sentencia.
4. Procesar el resultado.
5. Cerrar la conexión

## Establecer una conexión

- ▶ Hay dos modos. A través de
  - ▶ **DriverManager**: Conecta una aplicación a una fuente de datos especificada mediante una URL de base de datos.
  - ▶ **DataSource**: Administrador de conexiones

## Con java.sql.DriverManager

```
1 public static Connection getConnection(String url, String user,  
String password) throws SQLException
```

- ▶ Una URL de JDBC identifica a la base de datos para que el controlador apropiado la reconozca y establezca una conexión con ella.
- ▶ La sintaxis estándar para las URL de JDBC tiene tres partes, que están separadas por dos puntos.

`jdbc:<subprotocolo>:<subnombre>`

- ▶ jdbc es el protocolo (siempre JDBC ).
- ▶ <subprotocolo> el nombre del controlador o mecanismo de conectividad a la base de datos.
- ▶ <subnombre> una forma de identificar a la fuente de datos.
  - ▶ su forma puede variar según el subprotocolo (su sintaxis depende del controlador).

- ▶ `//host:port/database[?prop1][=val1][&...]`

- ▶ En nuestro caso

`//sql10.freysqlatabase.com:3306/sql10486001` donde  
sql10486001 es el nombre de la base de datos que creamos

## Creamos una conexión

```
1 public class UsaJDBC {  
2     private final static String URL = "jdbc:mysql://sql10.  
3         freesqldatabase.com:3306/sql10486001";  
4     private final static String USR = "sql10486001";  
5     private final static String PWD = ".....";  
6  
6     private static void establecerConexion() throws SQLException {  
7         conexion = DriverManager.getConnection(URL, USR, PWD);  
8     }  
9 }
```

# Statement

- ▶ La interface Statement se utiliza para ejecutar sentencias SQL en una base de datos relacional.
- ▶ Una Statement se obtiene a partir de una Connection

```
1 Connection con = DriverManager.getConnection(url, usr, pwd)
      ;
2 Statement stmt = con.createStatement();
```

- ▶ Se puede usar para ejecutar una query SQL;

```
1 String sql = "CREATE TABLE Persona("
2     + "ID BIGINT, "
3     + "NOMBRE VARCHAR(20), "
4     + "Apellido VARCHAR(20),"
5     + "PRIMARY KEY (ID))";
6 try(Statement stmt = conexion.createStatement()){
7     stmt.execute(sqlCreate);
8 }
```

Si la tabla existe, hay excepción

# Agregar Registros

- ▶ Vamos a usar Statement como introducción, pero no se debería (problemas de SQL-injection, ineficiencia)

```
1  sql = "INSERT INTO Persona VALUES (1,'Pepa','Pepe')";  
2  stmt.execute(sql);
```

## Ver algo de logging...

- ▶ Vamos a usar las propiedad `profileSQL` con el valor `true`.

```
1 String URL = "jdbc:mysql://sql10.freesqldatabase.com:3306/  
    sql10486001?fileSQL=true";
```



# Eliminar y update

## ► Ejemplo Update

```
1  sql = "UPDATE Persona SET Nombre = 'Pedro', Apellido = '  
Perez' WHERE Id = 1";
```

## ► Con una instancia se pueden ejecutar consultas a la base de datos o actualizaciones.

```
1  String sql = "DELETE FROM Persona WHERE Id = 1";
```

# Consultar

```
1 ResultSet resultado = stmt.executeQuery("SELECT * FROM Persona")  
;
```

- ▶ El resultado es un objeto de tipo ResultSet

# ResultSet

- ▶ Representa una tabla de datos que corresponde al resultado de la ejecución de una sentencia SQL.
- ▶ Se puede iterar sobre los registros.
- ▶ Forma básica: `next()` que avanza y devuelve verdadero hasta llegar al final, cuando devuelve falso.
- ▶ Se pueden acceder a las columnas de un registro utilizando `get_TIPO(identificación columna)`
- ▶ La identificación de la columna puede ser el número o el nombre.

```
1 while(resultado.next()) {  
2     System.out.println(  
3         resultado.getInt("Id")+": " +  
4         resultado.getString(1)+" "+  
5         resultado.getString(2));  
6 }
```

# PreparedStatement

- ▶ Un subtipo de Statement con algunas funciones adicionales
- ▶ Permite :
  - ▶ Definir parámetros en un sentencia SQL e insertarlos
  - ▶ Reutilizar la definición con nuevos valores de parámetros.
  - ▶ mas eficientes
  - ▶ Evita problemas de SQL injection.

```
1  sql = "UPDATE Persona SET Nombre = ?, Apellido = ? WHERE  
2      Id = ?";  
3  PreparedStatement pstmt = conexion.prepareStatement(sql)  
4      ;  
5  pstmt.setString(1, "Juana");  
6  pstmt.setString(2, "Diaz");  
7  pstmt.setInt(3, 2);  
  
7  pstmt.execute();
```

# Ejercicio

Definir una clase que provea operaciones para :

- ▶ Crear una Tabla Artista, con los siguientes atributos:
  - ▶ Id: BIGINT (Clave)
  - ▶ Bio : VARCHAR(2000)
  - ▶ FechaNacimiento: DATE
  - ▶ Apellido: VARCHAR(25)
  - ▶ Nombre: VARCHAR(25)
  - ▶ Email: VARCHAR(100)
- ▶ eliminar la tabla artista.
- ▶ agregar un artista a la base de datos los datos necesarios.
- ▶ mostrar por pantalla a un artista dado un Id.
- ▶ modificar un artista dado los datos necesarios.
- ▶ eliminar a un artista dado un Id.

Usar PreparedStatement y lanzar excepciones oportunas si la operación no es posible.

## Considerar que la clave es Auto-Increment

- ▶ Al crear la tabla usar Id BIGINT auto\_increment
- ▶ Usar la version

```
prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)
```
- ▶ Al insertar, usar id **null** (o directamente no pasar el valor)
- ▶ Obtener la clave generada luego de generar la sentencia, con  
`stmt.getGeneratedKeys()`.

# Conexiones a través de DataSource

```
1 //conector mysql
2 import com.mysql.cj.jdbc.MysqlDataSource;
3 ...
4 MysqlDataSource mysqlDataSource = new MysqlDataSource();
5 mysqlDataSource.setUser(USR);
6 mysqlDataSource.setPassword(PWD);
7 mysqlDataSource.setUrl(URL);
8 ...
9 conexion = mysqlDataSource.getConnection();
```

# Ejercicio

- ▶ Instalar base de datos en memoria ( [H2](#) )
- ▶ Dependencias

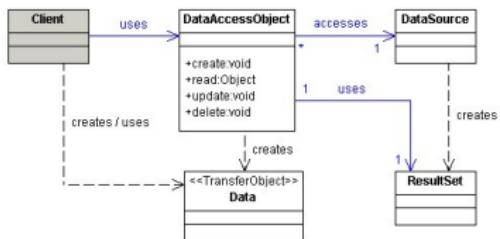
```
1 <dependency>
2   <groupId>com.h2database</groupId>
3   <artifactId>h2</artifactId>
4     <version>2.1.212</version>
5 </dependency>
```

```
1 String URL = "jdbc:h2:tcp://localhost/~/test";
2 String USR = "sa";
3 String PWD = "";
4
5 connection = DriverManager.getConnection(URL, USR, PWD);
```

```
1 import org.h2.jdbcx.JdbcDataSource;
2
3 ...
4 JdbcDataSource myh2DataSource = new JdbcDataSource();
5
6 }
```

# Pattern Data Access Object

- ▶ Se desea encapsular el acceso a datos (en un medio persistente)
- ▶ Desacoplar la implementación del almacenamiento persistente del resto de la aplicación.
- ▶ Proporcionar una API de acceso a datos uniforme para un mecanismo persistente.
- ▶ Solución: utilizar un objeto de acceso a datos para abstraer y encapsular todo el acceso al almacén persistente.
- ▶ El objeto de acceso a datos gestiona la conexión con la fuente de datos para obtener y almacenar datos



# Ejercicio

Considerar la base de datos Artista. Implementar el pattern DAO.

- ▶ Definir una clase Artista (role **Data**).
- ▶ Definir la clase ArtistaDAO (role **DataAccessObject**)
- ▶ ArtistaDAO debe implementar a la interface DAO<Artista>, donde

```
1 public interface DAO<T> {  
2     boolean save(T objeto);  
3     Optional<T> read(Long id);  
4     boolean update(T t);  
5     boolean borrar(T t);  
6     Collection<T> readAll();  
7 }
```

## Ejercicio

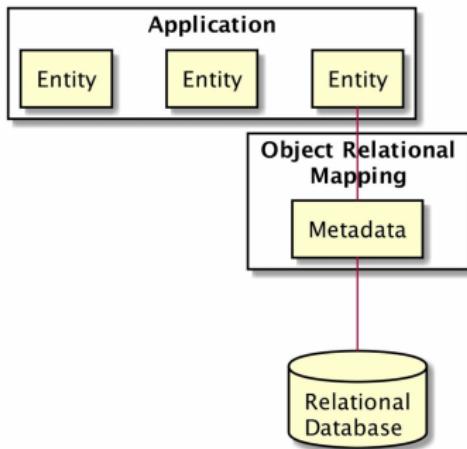
Analizar las dependencias de la clase ArtistaDAO con la fuente de datos. De ser necesario, invertir las dependencias.

# Fabrica de conexiones

Crear una fábrica para poder utilizar distintas conexiones.

# Mapeo relacional de objetos (ORM)

- ▶ El principio del Mapeo Objeto-Relacional (ORM) es unir el mundo de las bases de datos relacionales y los objetos.
- ▶ Los ORM son herramientas externas que brindan una vista orientada a objetos a datos relativacionales y viceversa.



# Java Persistence API (JPA) ahora Jakarta Persistence

- ▶ JPA es una especificación de una API Java que permite acceder, persistir y administrar datos como objetos Java en una base de datos relacional.
- ▶ Define conceptos que son implementados por distintos frameworks o herramientas.
- ▶ Tiene distintas implementaciones (EclipseLink, Hibernate, OpenJPA, DataNucleus)
- ▶ Hibernate es una implementación estándar de la especificación JPA, con algunas características adicionales que son específicas de Hibernate.

▶ Especificación

# Entidad

- ▶ Los *objetos* son instancias que viven en la memoria.
  - ▶ POJO: *Plain Old Java Object*.
- ▶ Las *entidades* son objetos que viven brevemente en memoria, pero de forma persistente en una base de datos relacional.
- ▶ Se usa el término **Entidad** en lugar de objeto cuando se habla de **un objeto que se debe mapear a una base de datos relacional** (objetos persistentes).
- ▶ Una *entidad* en JPA es un POJO que representa datos que deben persistirse en una base de datos.
- ▶ Por ser un POJO, una entidad se declara, se instancia y se puede usar como cualquier otra clase de Java.

# POJO Persona

- ▶ La clase (POJO) Persona que representa la información de una persona que queremos mantener en una tabla.

```
1 public class Persona {  
2     Long codigo;  
3     String nombre;  
4     String apellido;  
5 }
```

Notar: Diseño cuestionable por la visibilidad de atributos (ya lo mejoraremos)

# Mapeo de entidades

- ▶ ORM delega a herramientas o frameworks externos la tarea de establecer una correspondencia entre objetos y tablas.
- ▶ Las clases, objetos y atributos se traducen en conceptos de bases de datos relacionales: tablas que contienen filas y columnas.
- ▶ El mapeo brinda una vista orientada a objetos de los datos en las bases de datos: se pueden usar entidades de manera transparente en lugar de tablas.

# Metadatos

- ▶ El mapeo en JPA entre objetos y bases de datos se realiza utilizando *metadatos*.
  - ▶ Se asocian metadatos a las entidades para describir el mapeo.
  - ▶ El proveedor ORM usa los metadatos para reconocer entidades y mapearlas.
- ▶ Los metadatos se proporcionan en forma de:
  - ▶ Anotaciones: se usan anotaciones @ de Java en el código de las entidades
  - ▶ Descriptores XML: un archivo externo XML proporciona información sobre cómo mapear entidades.

Nos enfocamos primero en anotaciones

# Definiendo la Entidad Persona

- ▶ Para establecer que una clase Java (POJO) es una entidad se usa la anotación @Entidad.
- ▶ Definida en
  - ▶ javax.persistence.Entity (JPA2) o
  - ▶ jakarta.persistence.Entity (JAKARTA).
- ▶ Significado de @Entidad:
  - ▶ Clase que debe ser persistida

```
1 import javax.persistence.Entity;  
2  
3 @Entity  
4 public class Persona {  
5     Long codigo;  
6     String nombre;  
7     String apellido;  
8 }
```

# Clave Primaria e Identidad

- ▶ Toda Entidad debe tener una clave primaria
  - ▶ simple: un único atributo de la clase de entidad, o
  - ▶ compuesta (lo veremos más adelante)
- ▶ Usamos la anotación @Id para identificar al atributo que es clave primaria

```
1 import javax.persistence.Entity;
2 import javax.persistence.Id;
3
4 @Entity
5 public class Persona {
6     @Id
7     Long id;
8     String nombre;
9     String apellido;
10 }
```

codigo es la clave primaria (simple) de la entidad persona.

# Administración de entidades

- ▶ Una vez definido el mapeo, las entidades pueden ser administradas por JPA
  - ▶ Se pueden persistir, modificar, eliminar y consultar entidades a través de la interface JPA.
- ▶ De esta manera se manipulan entidades y se “esconde”, el acceso a la base de datos.

# Primer Ejemplo

- ▶ Dependencias: vamos a usar Maven, pom.xml
- ▶ Descriptor de deployment para JPA.
- ▶ Entidad Persona
- ▶ Uso de la entidad: UsaPersona (main)

# Dependencias Maven

- ▶ Dos dependencias:
  - ▶ Una implementación de JPA (Hibernate, EclipseLink, etc.).
  - ▶ El driver de JDBC (dependiendo de la base de datos a utilizar)

# Ejemplo: Hibernate (JPA 2.2) + H2

```
1 <dependencies>
2   <dependency>
3     <groupId>org.hibernate</groupId>
4     <artifactId>hibernate-core</artifactId>
5     <version>5.6.5.Final</version>
6   </dependency>
7   <dependency>
8     <groupId>com.h2database</groupId>
9     <artifactId>h2</artifactId>
10    <version>2.1.210</version>
11  </dependency>
12 </dependencies>
```

Notar: la dependencia `javax.persistence-api-2.2`

# Ejemplo: Hibernate (Jakarta) + H2

```
1 <dependencies>
2   <dependency>
3     <groupId>org.hibernate</groupId>
4     <artifactId>hibernate-core</artifactId>
5     <version>6.0.0.Final</version>
6   </dependency>
7   <dependency>
8     <groupId>com.h2database</groupId>
9     <artifactId>h2</artifactId>
10    <version>2.1.210</version>
11  </dependency>
12 </dependencies>
```

Notar: la dependencia `jakarta.persistence-api-3.0.0`

# Ejemplo: EclipseLink (Jakarta) + MySQL

```
1 <dependencies>
2   <dependency>
3     <groupId>org.eclipse.persistence</groupId>
4     <artifactId>org.eclipse.persistence.jpa</artifactId>
5     <version>3.1.0-M1</version>
6   </dependency>
7   <dependency>
8     <groupId>mysql</groupId>
9     <artifactId>mysql-connector-java</artifactId>
10    <version>8.0.28</version>
11  </dependency>
12 </dependencies>
```

Notar: la dependencia `jakarta.persistence-api-3.0.0`

# Descriptor de deployment

- ▶ Vincula el proveedor JPA (ej., Hibernate) a la base de datos (ej., H2)
- ▶ Proporciona información como
  - ▶ las clases de entidad que JPA debe mapear en la base de datos.
  - ▶ especifica la conexión con la base de datos: el driver SQL, la URL de la base, el usuario y la contraseña de acceso.
  - ▶ el proveedor de persistencia (Hibernate, EclipseLink, ...)
- ▶ Archivo `persistence.xml`
  - ▶ Debe colocarse en el directorio META-INF dentro del `classpath` de la aplicación.
  - ▶ En nuestro caso, `src/main/resources`
- ▶ Define la `persistence-unit`

# Descriptor de deployment

## persistence.xml

```
1<persistence
2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
4    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
5    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
6    version="2.2">
7
8    <!-- Definición del deployment --> ...
9
10</persistence>
```

Define el espacio de nombres y la dirección del schema xml (la gramática que debe seguir el xml)

# Descriptor de deployment

- ▶ La descripción está estructurada en unidades de persistencia (persistence unit)
- ▶ Una unidad de persistencia define un conjunto de clases entidad que están en un único almacén de datos.

```
1 <persistence...>
2   <persistence-unit name="rrhhPU"
3     transaction-type="RESOURCE_LOCAL">
4     <provider>org.hibernate.jpa.HibernatePersistenceProvider</
5       provider>
6     <class>cursobbva.modulo4.intro.Persona</class>
7     <properties> ...
8       </properties>
9     </persistence-unit>
 /></persistence>
```

# Descriptor de deployment

- ▶ La descripción está estructurada en unidades de persistencia (persistence unit)
- ▶ Una unidad de persistencia define un conjunto de clases entidad que residen en un único almacén de datos.
- ▶ <persistence-unit></persistence-unit>

```
1 <persistence...>
2   <persistence-unit ...>
3     ...
4   </persistence-unit>
5 </persistence>
```

# Descriptor de deployment

```
1 <persistence...>
2   <persistence-unit name="rrhhPU"
3     transaction-type="RESOURCE_LOCAL">
4     ...
5   </persistence-unit>
6 </persistence>
```

- ▶ Se le dá un nombre lógico "**rrhhPU**" a la unidad
- ▶ Se elige la forma de manejar las transacciones:
  - ▶ "**RESOURCE\_LOCAL  - ▶ "**JTA****

## "RESOURCE\_LOCAL"

- ▶ Las transacciones contra la base de datos están controladas por la aplicación
- ▶ El programador debe encargarse de manejar las transacciones para administrar a las entidades

Ahora nos vamos a focalizar en "RESOURCE\_LOCAL"

# Descriptor de deployment

```
1 <persistence...>
2   <persistence-unit ...>
3     <provider>org.hibernate.jpa.HibernatePersistenceProvider</
        provider>
4     ...
5   </persistence-unit>
6 </persistence>
```

- ▶ El tag <provider> establece la implementación de JPA a utilizar para ejecutar el código
- ▶ En el ejemplo, la implementación hibernate.
- ▶ alternativas
  - ▶ org.eclipse.persistence.jpa.PersistenceProvider (EclipseLink)
  - ▶ ...

# Descriptor de deployment

```
1 <persistence...>
2   <persistence-unit ...>
3     <provider>...</provider>
4     <class>cursobbva.modulo4.intro.Persona</class>
5     ...
6   </persistence-unit>
7 </persistence>
```

- ▶ Se listan las clases que componen la unidad persistente
- ▶ Notar que es necesario usar el nombre cualificado (con el paquete)
- ▶ a veces se puede omitir (volveremos)

# Descriptor de deployment

```
1 <persistence...>
2   <persistence-unit ...>
3     ...
4     <properties>
5       <property name="javax.persistence.jdbc.driver"
6         value="org.h2.Driver" />
7       <property name="javax.persistence.jdbc.url"
8         value="jdbc:h2:tcp://localhost/~test" />
9       <property name="javax.persistence.jdbc.user" value="sa" />
10      <property name="javax.persistence.jdbc.password" value="" />
11      ...
12      ...
13    </properties>
14  </persistence-unit>
</persistence>
```

- ▶ Propiedades que describen la conexión a la base de datos
  - ▶ Driver JDBC (aquí h2)
  - ▶ Url de la base de datos, más credenciales (usuario y password)

# Proveedor de persistencia y DB schema

- ▶ El proveedor de persistencia se puede configurar para manipular las tablas de la base de datos:
  - ▶ crear y/o eliminar tablas y cargar datos.
- ▶ Se configura usando propiedades estándar del descriptor de deployment.
- ▶ Para crear o eliminar tablas se usa  
`javax.persistence.schema-generation.database.action` con el valor oportuno (**"drop-and-create"** para borrar y crear)

```
1 <persistence...>
2   <persistence-unit ...> ...
3     <properties> ...
4       <property
5         name="javax.persistence.schema-generation.database.
6           action"
7           value="drop-and-create" />
8       </properties>
9     </persistence-unit>
10    </persistence>
```

# Descriptor de deployment

```
1 <persistence...>
2   <persistence-unit name="rrhhPU"
3     transaction-type="RESOURCE_LOCAL">
4     <provider>org.hibernate.jpa.HibernatePersistenceProvider</
5       provider>
6     <class>cursoBbva.modulo4.intro.Persona</class>
7     <properties>
8       <property name="javax.persistence.jdbc.driver"
9         value="org.h2.Driver" />
10      <property name="javax.persistence.jdbc.url"
11        value="jdbc:h2:tcp://localhost/~test" />
12      <property name="javax.persistence.jdbc.user" value="sa" />
13      <property name="javax.persistence.jdbc.password" value="" />
14      <property
15        name="javax.persistence.schema-generation.database.
16          action"
17        value="drop-and-create" />
18    </properties>
19  </persistence-unit>
20</persistence>
```

# Estado Inicial Base de Datos

The screenshot shows a web-based interface for the H2 database. At the top, there's a toolbar with standard browser controls (back, forward, search, etc.) and a connection status bar showing 'localhost'. Below the toolbar is a navigation menu with 'Run', 'Run Selected', 'Auto complete', and 'Clear' buttons, followed by a 'SQL statement:' input field. The left sidebar lists the available databases: 'INFORMATION\_SCHEMA' and 'Users'. A note at the bottom of the sidebar indicates the version: 'H2 2.1.210 (2022-01-17)'. The main content area contains two sections: 'Important Commands' and 'Sample SQL Script'.

**Important Commands**

	Displays this Help Page
	Shows the Command History
	Ctrl+Enter Executes the current SQL statement
	Shift+Enter Executes the SQL statement defined by the text selection
	Ctrl+Space Auto complete
	Disconnects from the database

**Sample SQL Script**

Delete the table if it exists	DROP TABLE IF EXISTS TEST;
Create a new table with ID and NAME columns	CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));
Add a new row	INSERT INTO TEST VALUES(1, 'Hello');
Add another row	INSERT INTO TEST VALUES(2, 'World');
Query the table	SELECT * FROM TEST ORDER BY ID;
Change data in a row	UPDATE TEST SET NAME='Hi' WHERE ID=1;
Remove a row	DELETE FROM TEST WHERE ID>2;
Help	HELP ...

**Adding Database Drivers**

Additional database drivers can be registered by adding the Jar file location of the driver to the environment variables H2DRIVERS or CLASSPATH. Example (Windows): to add the database driver library C:/Programs/hsqldb/lib/hsqldb.jar, set the environment variable H2DRIVERS to C:/Programs/hsqldb/lib/hsqldb.jar.

No hay tablas.

# Creación de instancias

```
1
2 public class UsaPersona {
3
4     public static void main(String[] args) {
5         Persona unaPersona = new Persona();
6         unaPersona.codigo = 1L;
7         unaPersona.nombre = "Pepa";
8         unaPersona.apellido = "Pepe";
9     }
10 }
```

- ▶ Notar que la instancia no es persistente. (Si ejecutamos UsaPersona no cambia el estado de la base de datos)

# Manejo manual de la persistencia

- ▶ `javax.persistence.EntityManager`: Interface que define las operaciones para crear, modificar, consultar y eliminar instancias de entidades persistentes.
- ▶ Una instancia de `EntityManager` está asociada con un *contexto de persistencia*.
  - ▶ un conjunto de instancias de entidad
  - ▶ para cualquier identidad de entidad persistente existe una instancia de entidad única.
  - ▶ se gestionan las instancias de entidad y su ciclo de vida.
- ▶ una instancia de `EntityManager` gestiona sólo entidades de una unidad de persistencia (`persistence-unit`)

# Administración Entidades

- ▶ JPA nos permite mapear entidades a una tabla y también consultarlas usando diferentes criterios.
- ▶ Se pueden consultar entidades y sus relaciones de una manera orientada a objetos (sin pensar en claves externas o columnas de la base de datos)
- ▶ La pieza central de la API responsable  
`javax.persistence.EntityManager`
  - ▶ administrar entidades (leer y escribir en una base de datos), permitir operaciones CRUD simples y consultas Java Persistence Query Language (JPQL)
  - ▶ Es una interface, cuya implementación la realiza el proveedor de persistencia (por ejemplo, EclipseLink o Hibernate).
  - ▶ El EntityManager delega todas las llamadas de bajo nivel a JDBC.

## Abstracciones para manejar persistencia (cont)

- ▶ Clase de entrada: `javax.persistence.Persistence` que permite obtener un `EntityManagerFactory` para una unidad de persistencia.
- ▶ `javax.persistence.EntityManagerFactory`
  - ▶ Permite crear al `EntityManager` que puede administrar entidades (relacionadas lógicamente y definidas en un "persistence-unit")

## Abstracciones para manejar persistencia (cont.)

```
1 import javax.persistence.Persistence;
2 import javax.persistence.EntityManagerFactory;
3 import javax.persistence.EntityManager;
4
5 public class UsaPersona {
6     public static void main(String[] args) {
7         EntityManagerFactory emf = Persistence.
8             createEntityManagerFactory("rrhhPU");
9         EntityManager em = emf.createEntityManager();
10    ...
11 }
```

Notar:

- ▶ nombre "**"rrhhPU"**" de la persistence-unit definida en el descriptor de deployment `persistence.xml`.
- ▶ Ver efectos de ejecutar `UsaPersona`.

# Estado Base de Datos luego de ejecutar UsaPersona

The screenshot shows a database management interface with the following details:

- Connection:** jdbc:h2:~/test
- Schemas:** PERSONA, INFORMATION\_SCHEMA, Users
- Version:** H2 2.1.210 (2022-01-17)
- Toolbar Buttons:** Auto commit, Max rows: 1000, Auto complete (Off), Auto select (On), SQL statement input field.
- Query Result:** SHOW COLUMNS FROM PERSONA; (3 rows, 80 ms)

FIELD	TYPE	NULL	KEY	DEFAULT
CODIGO	BIGINT	NO	PRI	NULL
APELLIDO	CHARACTER VARYING(255)	YES		NULL
NOMBRE	CHARACTER VARYING(255)	YES		NULL

- ▶ Se creo la tabla PERSONA, con los atributos CODIGO, NOMBRE y APELLIDO, con clave primaria CODIGO.

# Estado Base de Datos luego de ejecutar UsaPersona

The screenshot shows a browser window with a toolbar at the top. The address bar says "localhost". Below the toolbar is a menu bar with various icons. The main area has a sidebar on the left listing database objects: "PERSONA", "INFORMATION\_SCHEMA", and "Users". The main content area contains a SQL statement "SHOW COLUMNS FROM PERSONA" and its results.

```
SHOW COLUMNS FROM PERSONA;
```

FIELD	TYPE	NULL	KEY	DEFAULT
CODIGO	BIGINT	NO	PRI	NULL
APELLIDO	CHARACTER VARYING(255)	YES		NULL
NOMBRE	CHARACTER VARYING(255)	YES		NULL

(3 rows, 80 ms)

- ▶ No existen registros (no hay instancia persistente)

# Creación de instancias

```
1 public class UsaPersona {  
2  
3     public static void main(String[] args) {  
4         EntityManagerFactory emf = Persistence.  
5             createEntityManagerFactory("rrhhPU");  
6         EntityManager em = emf.createEntityManager();  
7  
8         Persona unaPersona = new Persona();  
9         unaPersona.codigo = 1L;  
10        unaPersona.nombre = "Pepa";  
11        unaPersona.apellido = "Pepe";  
12  
13        em.close();  
14        emf.close();  
15    }  
}
```

- ▶ Notar que la instancia no es persistente

## Persistir una entidad

- ▶ Se deben invocar la operación

```
void persist(Object entity)
```

de la interface EntityManager.

- ▶ Se debe invocar dentro del contexto de transacción.
- ▶ Para contextos con transacciones "RESOURCE\_LOCAL", se usa

```
EntityTransaction getTransaction()
```

de la interface EntityManager

- ▶ EntityTransaction: Interface que provee operaciones para controlar transacciones en modo resource-local
  - ▶ por ejemplo, begin y commit

# Ejemplo de uso

```
1 import javax.persistence.Persistence;
2 import javax.persistence.EntityManagerFactory;
3 import javax.persistence.EntityManager;
4 import javax.persistence.EntityTransaction;
5
6 public class UsaPersona {
7
8     public static void main(String[] args) {
9         EntityManagerFactory emf = Persistence.
10             createEntityManagerFactory("rrhhPU");
11         EntityManager em = emf.createEntityManager();
12         EntityTransaction tx = em.getTransaction();
13         ...
14     }
15 }
```

## Ejemplo de uso

```
1 ...
2 public class UsaPersona {
3     public static void main(String[] args) {
4         ...
5         EntityTransaction tx = em.getTransaction();
6
7         Persona unaPersona = new Persona();
8         ...
9
10        tx.begin();
11        em.persist(unaPersona);
12        tx.commit();
13
14        em.close();
15        emf.close();
16    }
17 }
18 }
```

# Estado Base de Datos luego de ejecutar UsaPersona

The screenshot shows a JDBC client interface connected to a local host. The left sidebar displays the database schema with tables PERSONA, INFORMATION\_SCHEMA, and Users. The main area contains a SQL statement window with the query "SELECT \* FROM PERSONA" and its results.

SQL statement:

```
SELECT * FROM PERSONA
```

Results:

CODIGO	APELLIDO	NOMBRE
1	Pepe	Pepa

(1 row, 1 ms)

Edit

- ▶ Se agregó un registro

# Posibles excepciones

- ▶ Referenciar a una persistence-unit que no se encuentra definida en persistence.xml

```
1 EntityManagerFactory emf = Persistence.  
    createEntityManagerFactory("Unidad");  
2 }
```

```
javax.persistence.PersistenceException: No Persistence  
provider for EntityManager named Unidad
```

## Posibles excepciones

- ▶ Invocar `persist` fuera de una transacción (sin hacer antes `begin`)

```
java.lang.IllegalStateException: Transaction not  
    successfully started  
}
```

- ▶ Persistir un objeto que no es una entidad

```
1     em.persist(Integer.valueOf(10));
```

```
java.lang.IllegalArgumentException: Unknown entity: java.  
    lang.Integer
```

- ▶ También aplica a entidades que no están en la `persistence-unit` (más adelante)

## Persistir entidades con clave duplicada

```
Persona unaPersona = new Persona();
unaPersona.codigo = 1L;
unaPersona.nombre = "Juan";
unaPersona.apellido = "Pepe";
Persona otraPersona = new Persona();
otraPersona.codigo = 1L;
otraPersona.nombre = "Juan";
otraPersona.apellido = "Pepe";

tx.begin();
em.persist(unaPersona);
em.persist(otraPersona);
tx.commit();
```

```
javax.persistence.EntityExistsException: A different object
with the same identifier value was already associated with
the session : [cursobbva.modulo4.intro.Persona#1]
```

Igual si es en distintas transacciones

# Persistir entidades con clave existente BD

- ▶ Ahora no dropeamos la BD: comentar en persistence

```
<property name="javax.persistence.schema-generation.  
database.action" value="drop-and-create" />
```

- ▶ Ejecutamos nuevamente UsaPersona

```
javax.persistence.RollbackException: Error while committing  
the transaction at ...  
Caused by: org.hibernate.exception.  
    ConstraintViolationException: could not execute  
    statement  
...  
Caused by: org.h2.jdbc.  
    JdbcSQLException: Unique  
    index or primary key violation: "PRIMARY KEY ON PUBLIC.  
    PERSONA(CODIGO) ( /* key:1 */ CAST(1 AS BIGINT), 'Pepe'  
    , 'Juan')"; ...
```

# Persistir entidades sin clave

```
1 Persona unaPersona = new Persona();
2 //unaPersona.codigo = 1L;
3 unaPersona.nombre = "Pepa";
4 unaPersona.apellido = "Pepe";
5
6 tx.begin();
7 em.persist(unaPersona);
8 tx.commit();
```

```
javax.persistence.PersistenceException: org.hibernate.id.
IdentifierGenerationException: ids for this class must be
manually assigned before calling save(): cursobbva.modulo4.
intro.Persona
```

## Sin embargo, otros atributos pueden no inicializarse

```
1 Persona unaPersona = new Persona();
2 unaPersona.codigo = 1L;
3 //unaPersona.nombre = "Pepa";
4 unaPersona.apellido = "Pepe";
5 tx.begin();
6 em.persist(unaPersona);
7
```

The screenshot shows a Java IDE interface with a code editor and a database query tool.

**Code Editor:**

```
1 Persona unaPersona = new Persona();
2 unaPersona.codigo = 1L;
3 //unaPersona.nombre = "Pepa";
4 unaPersona.apellido = "Pepe";
5 tx.begin();
6 em.persist(unaPersona);
7
```

**Database Query Tool:**

- Toolbar: Auto commit, Max rows: 1000, Auto complete, Off, Auto select On.
- Connection: jdbc:h2:~/test
- Schemas: PERSONA, INFORMATION\_SCHEMA, Users.
- Version: H2 2.1.210 (2022-01-17)
- SQL Statement: SELECT \* FROM PERSONA
- Result:

CODIGO	APELLIDO	NOMBRE
1	Pepe	null

(1 row, 1 ms)

Buttons: Run, Run Selected, Auto complete, Clear, SQL statement, Edit.

# Generación automática de ids

- ▶ @GeneratedValue se puede aplicar a junto con la anotación @Id para indicar que es generada automáticamente

```
1  @Entity  
2  public class Persona {  
3      @Id  
4      @GeneratedValue  
5      Long codigo;  
6      String nombre;  
7      String apellido;  
8  }
```

```
1  Persona unaPersona = new Persona();  
2  //    unaPersona.codigo = 1L;  
3  unaPersona.nombre = "Juan";  
4  unaPersona.apellido = "Pepe";  
5  tx.begin();  
6  em.persist(unaPersona);  
7  tx.commit();  
8  System.out.println(unaPersona.codigo);
```

# Estado Base de Datos luego de ejecutar UsaPersona

The screenshot shows the DBeaver interface with the following details:

- Toolbar:** Includes standard window controls (red, yellow, green circles), back/forward arrows, and a search bar labeled "localhost".
- Top Bar:** Shows connection status (Auto commit checked), max rows (1000), and various tool icons. It also displays "Auto complete Off" and "Auto select On".
- Left Sidebar (Database Tree):** Lists the database structure:
  - jdbc:h2:~/test
  - PERSONA
  - INFORMATION\_SCHEMA
  - Sequences
    - HIBERNATE\_SEQUENCE
      - Current value: 2
  - Users
  - H2 2.1.210 (2022-01-17)
- Central Panel:** A large empty area for running SQL statements.
- Bottom Panel (Important Commands):** A table listing keyboard shortcuts:

Icon	Description
?	Displays this Help Page
!	Shows the Command History
Ctrl+Enter	Executes the current SQL statement
Shift+Enter	Executes the SQL statement defined by the text selection

- ▶ Notar la creación de la secuencia en la BD

# Estrategia para la generación automática de Ids

- ▶ @GeneratedValue especifica la forma en que se generan los valores de las claves primarias.
- ▶ Se puede utilizar el atributo strategy para indicar el mecanismo de la BD a usar para la generación

```
public enum GenerationType {TABLE, SEQUENCE, IDENTITY, AUTO  
}
```

- ▶ TABLE: utiliza una tabla de base de datos
- ▶ SEQUENCE: una secuencia
- ▶ IDENTITY: una columna de identidad
- ▶ AUTO: el proveedor de persistencia elige

AUTO es el valor por default (mas detalles en el futuro)

## Ejemplo: strategy= TABLE

```
import static javax.persistence.GenerationType.TABLE;  
....  
@GeneratedValue(strategy= TABLE)
```

The screenshot shows a web-based interface for the H2 database. The top navigation bar includes icons for back, forward, and search, followed by the URL 'localhost'. Below the URL are various toolbar buttons and dropdowns for database configuration.

The left sidebar displays the database schema:

- jdbc:h2:~/test
- HIBERNATE\_SEQUENCES
- PERSONA
- INFORMATION\_SCHEMA
- Sequences
- Users
- H2 2.1.210 (2022-01-17)

The main content area contains a SQL statement and its results:

```
SELECT * FROM HIBERNATE_SEQUENCES
```

SEQUENCE_NAME	NEXT_VAL
default	1

(1 row, 1 ms)

An 'Edit' button is located at the bottom of the result table.

- ▶ Notar la creación de la tabla

## No inicializar Id si es @GeneratedValue

```
1 @Entity  
2 public class Persona {  
3     @Id @GeneratedValue  
4     Long codigo; ...  
5 }
```

Si se inicializa antes de persistir:

```
1 Persona unaPersona = new Persona();  
2 unaPersona.codigo = 1L;  
3 unaPersona.nombre = "Juan";  
4 unaPersona.apellido = "Pepe";  
5 tx.begin();  
6 em.persist(unaPersona);  
7 tx.commit();
```

Arroja:

```
javax.persistence.PersistenceException: org.hibernate.  
    PersistentObjectException: detached entity passed to  
    persist: cursobbva.modulo4.intro.Persona
```

## Sobre las entidades

- ▶ Una clase entidad debe tener un constructor sin argumentos (aunque puede tener otros). Puede ser el default.
- ▶ El constructor sin argumentos debe ser **public** o **protected**.
- ▶ La clase de entidad debe ser una clase top-level (no anidada).
  - ▶ Enumeraciones e interfaces no pueden ser entidades.
- ▶ La clase no puede ser **final** ni sus métodos o variables de instancia persistente pueden ser **final**.
- ▶ Las entidades admiten herencia.
- ▶ Tanto clases abstractas como las concretas pueden ser entidades.
- ▶ y algunas cosas mas ...

## Configuration-by-exception

- ▶ Sin metadatos, una clase es tratada como un POJO (no deber persistido)
- ▶ Regla general: si no se proporcionan metadatos, se aplica el tratamiento por default del proveedor de persistencia.
- ▶ Por default una clase no tiene representación de la base de datos.
- ▶ Si se debe cambiar, se debe proveer el metadato (anotación la clase con @Entity).
- ▶ De manera análoga para @Id, un atributo no es clave principal, a menos que se especifique @Id.
- ▶ Por default, el valor no lo genera el proveedor, si se necesita lo especifica el usuario @GeneratedValue.

# Ejercicio

- ▶ Crear un proyecto maven para persistir la entidad Artista, donde la clave sea generada automáticamente.
- ▶ Artista debe pertenecer a la unidad de persistencia ColeccionPU.
- ▶ Usar como proveedor EclipseLink y como connector H2.
- ▶ Definir un programa principal que persista dos entidades distintas.

## Representación

- ▶ El estado persistente de una entidad está representado por variables de instancia.
- ▶ Una variable de instancia debe ser accedida directamente solo desde dentro de los métodos de la entidad.
- ▶ Los clientes de la entidad no deberían acceder a las variables de instancia.
- ▶ El estado de la entidad está disponible para los clientes solo a través de los métodos de la entidad, es decir, métodos de acceso (métodos getter/setter) u otros métodos.

# Atributos persistentes

- ▶ En tiempo de ejecución, el proveedor de persistencia accede al estado persistente de una entidad a través de:
  - ▶ variables de instancia (**acceso de campo (field)**), o
  - ▶ propiedades (**acceso de propiedad** à la JavaBeans)
- ▶ Para acceso de propiedad, se deben seguir las convenciones para las propiedades de lectura / escritura de JavaBeans:

```
1 T getProperty()  
2 void setProperty(T t)
```

- ▶ Para propiedades booleanas se puede usar `isProperty()` en lugar del getter.

# Tipos de atributos persistentes

- ▶ Los atributos pueden tomar la mayoría de los tipos (incluso definidos por el usuario, con algunas restricciones).
- ▶ Los campos o propiedades persistentes de una entidad pueden ser de los siguientes tipos.
  - ▶ tipos primitivos de Java,
  - ▶ java.lang.String,
  - ▶ Tipos serializables de Java (incluidos los wrappers de tipos primitivos, `java.math.BigInteger`,  
`java.math.BigDecimal`, `java.util.Date`,  
`java.util.Calendar`, `java.sql.Date`, `java.sql.Time`,  
`java.sql.Timestamp`, **byte[]**, **Byte[]**, **char[]**,  
`Character[]`, `java.time.LocalDate`,  
`java.time.LocalTime`, `java.time.LocalDateTime`,  
`java.time.OffsetTime`, `java.time.OffsetDateTime` y  
tipos definidos por el usuario que implementan la interface  
`Serializable`); enumeraciones; tipos de entidades;  
colecciones de tipos de entidad; clases *embeddable*;  
colecciones de tipos básicos e *embeddable*.

## Campos y propiedades persistentes

- ▶ el proveedor de persistencia accede al estado persistente de una entidad en tiempo de ejecución mediante:
- ▶ accesos de propiedad (*property access*) al estilo JavaBeans;
- ▶ "acceso de campo" mediante variables de instancia.
- ▶ Cuando se usa acceso de propiedad, los métodos de acceso deben ser públicos o protegidos.

## Acceso por propiedad

- ▶ para cada propiedad persistente propiedad de tipo T hay un método getter `getPropiedad` y un método setter `setPropiedad`.
- ▶ Para propiedades booleanas, se puede usar como un nombre alternativo para el método getter `isProperty`
- ▶ Para las propiedades persistentes de un solo valor, estas firmas de métodos son:

```
1 T getPropiedad()  
2 void setPropiedad(T t)
```

## Ejemplo - Acceso por Campo

```
1 @Entity
2 public class Persona {
3     @Id @GeneratedValue
4     private Long codigo;
5     private String nombre;
6     private String apellido;
7     public Long getCodigo() {
8         return codigo;
9     }
10    public void setCodigo(Long codigo) {
11        this.codigo = codigo;
12    }
13    public String getNombre() {
14        return nombre;
15    }
16    public void setNombre(String nombre) {
17        this.nombre = nombre;
18    }
19    public String getApellido() {
20        return apellido;
21    }
22    public void setApellido(String apellido) {
23        this.apellido = apellido;
24    }
```

## Ejemplo - Acceso por Campo

```
1 public class UsaPersona {  
2     public static void main(String[] args) {  
3         EntityManagerFactory emf = Persistence.  
4             createEntityManagerFactory("rrhhPU");  
5         EntityManager em = emf.createEntityManager();  
6         EntityTransaction tx = em.getTransaction();  
7  
7         Persona unaPersona = new Persona();  
8         unaPersona.setNombre("Juan");  
9         unaPersona.setApellido("Pepe");  
10  
11         tx.begin();  
12         em.persist(unaPersona);  
13         tx.commit();  
14  
15         em.close();  
16         emf.close();  
17     }  
18 }
```

## Ejemplo - Acceso por Propiedad

```
1 @Entity
2 public class Persona {
3     private Long codigo;
4     private String nombre;
5     private String apellido;
6     @Id @GeneratedValue
7     public Long getCodigo() {
8         return codigo;
9     }
10    public void setCodigo(Long codigo) {
11        this.codigo = codigo;
12    }
13    public String getNombre() {
14        return nombre;
15    }
16    public void setNombre(String nombre) {
17        this.nombre = nombre;
18    }
19    public String getApellido() {
20        return apellido;
21    }
22    public void setApellido(String apellido) {
23        this.apellido = apellido;
24    }
```

## Notar que

- ▶ En este caso no hay diferencias significativas porque
  - ▶ el mapping entre campos y propiedades es uno a uno (y tienen el mismo nombre y tipo)
  - ▶ no hay lógica en los getters y setters

## Ejemplo: agregar propiedad apellidoNombre

```
1 @Entity
2 public class Persona {
3     private Long codigo;
4     private String nombre;
5     private String apellido;
6
7     @Id
8     @GeneratedValue
9     private Long getCodigo() {
10         return codigo;
11     }
12     ...
13     public String getApellidoNombre() {
14         return apellido+" "+nombre;
15     }
16     public void setApellidoNombre(String apellido) {
17     }
18 }
```

# Notar cambios en BD

The screenshot shows a MySQL Workbench interface. The top bar displays the connection information "localhost" and various tool icons. The left sidebar lists the database schema with tables like PERSONA, INFORMATION\_SCHEMA, Sequences, and Users. The main area contains a SQL editor with the query "SELECT \* FROM PERSONA;" and its results.

SQL statement:

```
SELECT * FROM PERSONA;
```

Results:

CODIGO	APELLIDO	APELLIDOYNOMBRE	NOMBRE
1	Pepe	Pepe Juan	Juan

(1 row, 1 ms)

Edit

## Accesos a una propiedad

- ▶ Además de devolver y establecer el estado persistente de la instancia, los métodos de acceso a una propiedad pueden contener lógica, como por ejemplo, realizar la validación.
- ▶ El proveedor de persistencia ejecuta esta lógica cuando se utiliza el acceso basado en propiedades.
- ▶ Se debe tener cuidado al agregar lógica a los métodos de acceso cuando se utiliza el acceso por propiedad.
  - ▶ No está definido el orden en el que el proveedor de persistencia llama a estos métodos al cargar o almacenar el estado persistente.
  - ▶ Por eso
    - ▶ la lógica no debe basarse en un orden de invocación específico

# Modificando un getter

```
1 @Entity
2 public class Persona {
3     private Long codigo;
4     private String nombre;
5     private String apellido;
6
7     @Id
8     @GeneratedValue
9     private Long getCodigo() {
10         return codigo;
11     }
12     ...
13     public String getApellido() {
14         return apellido+"!!!!";
15     }
16 }
```

# Modificando un getter

The screenshot shows a Java IDE interface with a database connection to `jdbc:h2:~/test`. The left sidebar displays the database schema with tables `PERSONA`, `INFORMATION_SCHEMA`, `Sequences`, and `Users`, along with the version `H2 2.1.210 (2022-01-17)`.

In the main area, the SQL statement `SELECT * FROM PERSONA` is entered in the SQL editor. Below the editor, the results of the query are displayed in a table:

CODIGO	APELLIDO	APELLIDOYNOMBRE	NOMBRE
1	Pepe!!!	Pepe Juan	Juan

(1 row, 0 ms)

An `Edit` button is located at the bottom of the result table.

# Mapeo por default

- ▶ Las reglas de mapeo por default siguen el principio de **configuración por excepción** (también **programación por excepción o convención sobre configuración**).  
*A menos que se especifique lo contrario, el proveedor debe aplicar las reglas por default*
- ▶ Proporcionar una configuración es la excepción a la regla.
- ▶ Esto permite escribir la cantidad mínima de código para ejecutar una aplicación, dependiendo de la configuración por default del proveedor.
- ▶ Si se necesita aplicar otras reglas, se puede personalizar utilizando metadatos.

## Mapeo por default

- ▶ El nombre de la entidad se asigna a un nombre de tabla relacional
  - ▶ entidad Persona → tabla PERSONA
- ▶ Cada nombre de atributo se asigna a un nombre de columna
  - ▶ campo o propiedad id → columna ID
- ▶ Se usan las reglas JDBC se aplican para mapear tipos primitivos de Java a tipos de datos relacionales.
  - ▶ String → VARCHAR(255) (VARCHAR de tamaño 255)
  - ▶ **long** → BIGINT
  - ▶ **boolean** → SMALLINT
- ▶ las reglas de asignación por default son diferentes de una base de datos a otra.
  - ▶ una cadena se asigna a un VARCHAR en H2 y a un VARCHAR2 en Oracle
  - ▶ Un entero se asigna a un INTEGER en H2 y un NUMBER en Oracle

# Tablas

- ▶ El mapeo por default establece que la entidad y el nombre de la tabla son iguales
- ▶ Esto podría ser adecuado en la mayoría de los casos, pero se puede desear
  - ▶ asignar sus datos a una tabla con nombre diferente
  - ▶ o incluso asignar una entidad a varias tablas.

## @Table

- ▶ La anotación `@javax.persistence.Table` permite cambiar el mapeo predeterminado para una tabla.
- ▶ Por ejemplo,
  - ▶ el nombre de la tabla, el catálogo y el esquema de la base de datos.
  - ▶ restricciones de unicidad: usar `@UniqueConstraint` junto con `@Table`

```
1 @Entity
2 @Table(name = "T_PERSONA")
3 public class Persona {
4     private Long codigo;
5     private String nombre;
6     private String apellido;
7     ...
8 }
```

# Convenciones sobre nombres

- ▶ Las anotaciones y los elementos de las anotaciones contienen (o asumen) nombres de objetos de la base de datos
  - ▶ tablas, columnas y otros elementos de la base de datos.
- ▶ por default se tratan como identificadores no delimitados
  - ▶ sin comillas, deben comenzar con una letra; pueden contener dígitos; guión bajo “\_” o pesos “\$”
- ▶ es posible usar identificadores delimitados
  - ▶ por nombre
    - ▶ en anotaciones `@Table(name = "\customer")`

## @SecondaryTable

- ▶ Hasta ahora asumimos que una entidad se mapea a una sola tabla (tabla principal)
- ▶ Cuando se tiene un modelo de datos existente, es posible que se necesite distribuir los datos en varias tablas (tablas secundarias)
- ▶ Con la anotación @SecondaryTable se asocia una tabla secundaria a una entidad.
- ▶ Se pueden distribuir los datos de una entidad entre columnas de una tabla principal y de las tablas secundarias
- ▶ y especificando para cada atributo a qué tabla pertenece (con la anotación @Column)

# Ejemplo

```
1 @Entity
2 @SecondaryTable(name = "ciudad")
3 @SecondaryTable(name = "pais")
4 public class Direccion {
5     @Id
6     @GeneratedValue
7     private Long id;
8     private String calle1;
9     private String calle2;
10    @Column(table = "ciudad")
11    private String ciudad;
12    @Column(table = "ciudad")
13    private String provincia;
14    @Column(table = "ciudad")
15    private String codigoPostal;
16    @Column(table = "pais")
17    //Constructores, getters y setters
18 }
```

# Mapeo a varias tablas

The screenshot shows a MySQL Workbench interface. The left sidebar displays the database schema with the following structure:

- jdbc:h2:~/test
- CIUDAD
  - CIUDAD
  - CODIGOPOSTAL
  - PROVINCIA
  - ID
  - Indexes
- DIRECCION
  - ID
  - CALLE1
  - CALLE2
  - Indexes
- PAIS
  - PAIS
  - ID
  - Indexes
- INFORMATION\_SCHEMA
- Sequences
- Users

The main pane shows the "Important Commands" table:

Icon	Description
?	Displays this Help Page
History icon	Shows the Command History
Ctrl+Enter icon	Executes the current SQL statement
Shift+Enter icon	Executes the SQL statement defined by the text selection

- ▶ Por default, los atributos se asignan a la tabla principal (**DIRECCION**).
- ▶ los restantes según anotación (@Column(table="CIUDAD") o @Columna(table="PAIS"))
- ▶ Las tablas tienen la misma clave principal (para hacer el join).

## Ejercicio

Considerar nuevamente a la entidad Artista. Definir el mapeo de manera tal que

- ▶ Nombre, apellido y fecha de nacimiento sean mapeados a una tabla "**ARTISTAS\_PRINCIPAL**"
- ▶ El email a "**CONTACTOS\_ARTISTA**"
- ▶ La bio a "**DETALLES\_ARTISTA**"

## Claves primarias

- ▶ En BD, una clave principal identifica de forma única cada fila de una tabla.
- ▶ Comprende una o un conjunto de columnas.
- ▶ Las claves primarias deben ser únicas, ya que identifican a una sola fila (no se permite un valor nulo).
- ▶ El valor de la clave principal no se puede actualizar una vez que se ha asignado.

# Ejemplo

```
1 tx.begin();
2 em.persist(unaPersona);
3 tx.commit();
4 unaPersona.setCodigo(3L);
5 tx.begin();
6 em.persist(unaPersona);
7 tx.commit();
```

```
1 javax.persistence.RollbackException: Error while committing the
   transaction
2 ...
3 Caused by: javax.persistence.PersistenceException: org.hibernate
   .HibernateException: identifier of an instance of cursobbva
   .modulo4.intro.Persona was altered from 1 to 3
```

## @Id and @GeneratedValue

- ▶ Una clave primaria simple corresponde a un único atributo de la entidad.
- ▶ @Id se utiliza para denotar una clave principal simple.
- ▶ Puede ser de uno de los siguientes tipos:
  - ▶ Tipo primitivo: **byte**, **int**, **short**, **long**, **char**, etc.
  - ▶ Wrapper de tipos primitivos: Byte, Integer, Short, Long, Character, etc.
  - ▶ `java.lang.String`; `java.util.Date`; `java.sql.Date`; `java.math.BigDecimal`; `java.math.BigInteger`, `java.time.LocalDate`, `java.time.LocalDateTime`, `java.time.OffsetTime`, `java.time.OffsetDateTime`.
  - ▶ Si el proveedor permite otras, no se garantiza la portabilidad.

## Claves primarias compuestas

- ▶ Hay casos en los que se requiere una clave principal compuesta
  - ▶ mapear a una BD existente
  - ▶ cuando las claves principales deben seguir ciertas reglas de negocio (una fecha y un valor, o un código de país y un timestamp)
- ▶ Se define una clase de clave principal para representar a la clave compuesta.
- ▶ Se usa
  - ▶ `@EmbeddedId`
  - ▶ `@IdClass`
- ▶ Se generarán los mismos esquemas, pero cambia la manera de consultar por la entidad.

## Claves primarias compuestas

- ▶ Las clases de clave principal deben definir `equals()` y `hashCode()` para poder administrar consultas y colecciones internas.
- ▶ La igualdad de estos métodos debe ser consistente con la igualdad de la base de datos
- ▶ Sus atributos deben tener como tipo los válidos para una clave primaria;
- ▶ Ser **public**
- ▶ Deben implementar `Serializable`
- ▶ Deben tener un constructor sin argumentos.

## Ejemplo

- ▶ Suponer que tenemos que mapear a la entidad noticia que tiene un contenido, un título y, dado que puede estar escrita en varios idiomas, un código de idioma (ES, para castellano, EN para inglés, etc.).
- ▶ La clave principal de la noticia podría ser el título y el idioma, porque un artículo se puede traducir a varios idiomas pero conservando su título original.
- ▶ Entonces, la clase de clave principal IdNoticia se compone de dos atributos de String **titulo** e **idioma**

## Ejemplo @Embeddable

```
1  @Embeddable
2  public class IdNoticia implements Serializable{
3      private String titulo;
4      private String idioma;
5
6      public String getTitulo() {
7          return titulo;
8      }
9      public void setTitulo(String titulo) {
10         this.titulo = titulo;
11     }
12     public String getIdioma() {
13         return idioma;
14     }
15     public void setIdioma(String idioma) {
16         this.idioma = idioma;
17     }
18 }
```

## Ejemplo @Embeddable

```
1  @Entity
2  public class Noticia {
3      @EmbeddedId
4      private IdNoticia id;
5      private String contenido;
6      protected Noticia() {}
7      public Noticia(IdNoticia id, String contenido) {
8          this.id = id;
9          this.contenido = contenido;
10     }
11     public IdNoticia getId() {
12         return id;
13     }
14     public void setId(IdNoticia id) {
15         this.id = id;
16     }
17     public String getContenido() {
18         return contenido;
19     }
20     public void setContenido(String content) {
21         this.contenido = contenido;
22     }
23 }
```

## Ejemplo @Embeddable

```
1 IdNoticia id = new IdNoticia();
2 id.setTitulo("Goal de ...");
3 id.setIdioma("ES");
4
5 tx.begin();
6 em.persist(new Noticia(id,"en el partido"));
7 tx.commit();
```

## @Embeddable

- ▶ Define un objeto embebido.
- ▶ No tiene identidad (no tiene una clave principal propia)-
- ▶ Sus atributos serán mapeados como columnas en la tabla de la entidad que lo contiene.
- ▶ Usar convenciones JavaBean (debe tener un constructor sin argumentos, getter, setter, equals() y hashCode()).
- ▶ La clase en sí no tiene una identidad propia (sin anotación @Id).

## @Embeddable

```
1 @Embeddable
2 public class IdNoticia implements Serializable{
3
4     private String titulo;
5     private String idioma;
6
7     //getter y setters
8 }
```

## @Embeddable

```
1 @Entity
2 public class Noticia {
3     @EmbeddedId
4     private IdNoticia id;
5     private String contenido;
6
7     protected Noticia() {}
8
9     public Noticia(IdNoticia id, String contenido) {
10         this.id = id;
11         this.contenido = contenido;
12     }
13     //setters y getters
14 }
```

## @Embeddable

```
1  IdNoticia id = new IdNoticia();
2  id.setTitulo("Goal de ...");
3  id.setIdioma("ES");
4
5  tx.begin();
6  em.persist(new Noticia(id,"en el partido"));
7  tx.commit();
```

## @Embeddable

The screenshot shows a MySQL Workbench interface running on a Mac OS X system. The title bar indicates the connection is to 'localhost'. The left sidebar shows the database structure:

- jdbc:h2:~/test
- NOTICIA (selected)

  - IDIOMA
  - TITULO
  - CONTENIDO
  - Indexes

- INFORMATION\_SCHEMA
- Users
- H2 2.1.210 (2022-01-17)

The main pane displays the SQL command 'SHOW COLUMNS|FROM NOTICIA' entered in the SQL statement field. Below the command, the results are shown in a table:

FIELD	TYPE	NULL	KEY	DEFAULT
IDIOMA	CHARACTER VARYING(255)	NO	PRI	NULL
TITULO	CHARACTER VARYING(255)	NO	PRI	NULL
CONTENIDO	CHARACTER VARYING(255)	YES		NULL

(3 rows, 13 ms)

## @Embeddable

- ▶ @Id en @Embeddable

```
1 org.hibernate.AnnotationException: cursobbva.modulo4.  
    introjpa.IdNoticia must not have @Id properties when  
    used as an @EmbeddedId:
```

- ▶ Si la clase no implementa Serializable

```
1 javax.persistence.PersistenceException: [PersistenceUnit:  
    rrhhPU] Unable to build Hibernate SessionFactory  
2 at ....  
3 Caused by: org.hibernate.MappingException: Composite-id  
    class must implement Serializable:
```

## @Embeddable

- ▶ Si no se inicializa la clave:

```
1 javax.persistence.PersistenceException: org.hibernate.id.  
IdentifierGenerationException: null id generated
```

- ▶ si no se inicializa un campo en la clase embebida:

```
1 javax.persistence.RollbackException: ...  
2 ...  
3 Caused by: org.h2.jdbc.  
    JdbcSQLIntegrityConstraintViolationException: NULL not  
    allowed for column "TITULO";
```

## @Embeddable

- ▶ Hibernate no require constructor por default:

1 No **default** (no-argument) constructor **for class**: cursobbva.  
modulo4.introjpa.IdNoticia (**class** must be instantiated  
by Interceptor)

- ▶ No requiere redefinir equals ni hashCode

1 WARN: HHH000038: Composite-id **class** does not override  
equals(): ...  
2 WARN: HHH000039: Composite-id **class** does not override  
hashCode(): ...

Importante porque algunas operaciones de EntityManager  
dependen de esto (**Más en el futuro**)

# Atención

- ▶ Un objeto (POJO) serializable puede ser clave

```
1 public class IdNoticia implements Serializable{  
2     private String titulo;  
3     private String idioma;  
4  
5     //constructors, getter, setter  
6 }
```

# Atención

The screenshot shows a Java application window with a title bar "localhost". The main area contains a database browser and a SQL editor.

**Database Browser:**

- Connection: jdbc:h2:~/test
- Schemas:
  - NOTICIA
    - ID (BINARY VARYING(255))
    - CONTENIDO
  - INFORMATION\_SCHEMA
  - Users
- H2 2.1.210 (2022-01-17)

**SQL Editor:**

Max rows: 1000 | Auto complete: Off | Auto select: On

SQL statement:

```
SELECT * FROM NOTICIA;
```

Result:

```
SELECT * FROM NOTICIA;
ID
aced000573720024637572736f626276612e6d6f64756c6f342e696e74726f6a70612e49644e6f746963696128a12395a7
(1 row, 11 ms)
```

**Bottom Left:**

Edit

## Ejemplo @IdClass

- ▶ Cada atributo de clave principal debe declararse en la clase de entidad y anotarse con @Id.
- ▶ Se declara una clase que contiene los atributos de la clave primaria clave primaria compuesta.
- ▶ La clase de la clave primaria es un POJO que no requiere ninguna anotación

## Ejemplo @IdClass

```
1 public class IdNoticia implements Serializable{  
2     private String titulo;  
3     private String idioma;  
4  
5     public void setTitulo(String titulo) {  
6         this.titulo = titulo;  
7     }  
8     public String getIdioma() {  
9         return idioma;  
10    }  
11    public void setIdioma(String idioma) {  
12        this.idioma = idioma;  
13    }  
14    public String getTitulo() {  
15        return titulo;  
16    }  
17    ...  
18 }
```

## Ejemplo @IdClass

```
1  @Entity
2  @IdClass(IdNoticia.class)
3  public class Noticia {
4      @Id
5      private String titulo;
6      @Id
7      private String idioma;
8      private String contenido;
9      // Constructores, getters y setters
10 }
```

## Ejemplo @IdClass

```
1 Noticia n = new Noticia();
2 n.setTitulo("Goal de ...");
3 n.setIdioma("ES");
4
5 tx.begin();
6 em.persist(n);
7 tx.commit();
```

## @IdClass

The screenshot shows a MySQL Workbench interface connected to a local host. The left sidebar displays the database schema with a tree view of tables (NOTICIA, INFORMATION\_SCHEMA, Users) and their respective columns (IDIOMA, TITULO, CONTENIDO). The main pane contains the SQL statement 'SHOW COLUMNS FROM NOTICIA' and its execution results.

SQL statement:

```
SHOW COLUMNS FROM NOTICIA;
```

Execution results:

FIELD	TYPE	NULL	KEY	DEFAULT
IDIOMA	CHARACTER VARYING(255)	NO	PRI	NULL
TITULO	CHARACTER VARYING(255)	NO	PRI	NULL
CONTENIDO	CHARACTER VARYING(255)	YES		NULL

(3 rows, 3 ms)

Notar que es el mismo esquema. Hay diferencias en la forma de consultar (lo veremos)

## @IdClass

- ▶ La entidad Noticia define la clave principal mediante la anotación @IdClass.
- ▶ Anotar cada componente de la clave con @Id.
- ▶ Para fijar la clave de la entidad deberá establecer un valor para `titulo` e `idioma`.

## @IdClass

- ▶ los campos en la clase @IdClass deben aparecer como atributos de la entidad.

1 org.hibernate.AnnotationException: Property of @IdClass  
not found in entity ...

- ▶ Si los tipos no son compatibles:

1 javax.persistence.PersistenceException: org.hibernate.  
PropertyAccessException: Could not set field value ...

## Comparación

- ▶ `@IdClass` es más propenso a errores porque necesita definir cada atributo de clave principal tanto en la `IdClass` como en la entidad.
- ▶ Ventaja: no necesita cambiar el código de la clase de clave principal (no se necesita ninguna anotación).
- ▶ Se diferencian en la forma en que hace referencia a la entidad en JPQL.

## Ejercicio

- ▶ Se debe implementar la entidad Cuenta que debe mapearse a una tabla definida de la siguiente manera:

```
CREATE TABLE CUENTA (
    NUMERO BIGINT,
    SUCURSAL INTEGER,
    DESCRIPCION CHAR(255),
    FECHACREACION DATE,
    PRIMARY KEY (NUMERO, SUCURSAL));
```

# Generación de schema

- ▶ Facilidad que permite generar tablas y otros artefactos de la base de datos.

## Propiedades en persistence.xml

### `jakarta.persistence.schema-generation.database.action`

- ▶ Especifica la acción que debe realizar el proveedor de persistencia para la generación del schema.
- ▶ Los valores son `none`, `create`, `drop-and-create`, `drop`.
- ▶ Si no se especifica esta propiedad, se supone que la generación de esquemas no es necesaria

### `jakarta.persistence.schema-generation.scripts.action`

- ▶ Especifica los scripts que debe generar el proveedor de persistencia.
- ▶ Los valores para esta propiedad son `none`, `create`, `drop-and-create`, `drop`.
- ▶ Solo se generará un script si se especifica el destino del script.
- ▶ Si no se especifica esta propiedad, no se generarán los scripts.

## Propiedades en persistence.xml

### `jakarta.persistence.schema-generation.create-source`

- ▶ Especifica si la creación de artefactos debe ocurrir sobre la base de los metadatos de mapeo relacional/objeto, el script DDL o una combinación de ambos.
- ▶ Los valores son `metadata`, `script`, `metadata-then-script`, `script-then-metadata`.

### `jakarta.persistence.schema-generation.drop-source`

- ▶ Similar al anterior, pero para los drops.

## Propiedades en persistence.xml

`jakarta.persistence.schema-generation.scripts.create-target`,  
`jakarta.persistence.schema-generation.scripts.drop-target`

- ▶ Especifican las ubicaciones de los destinos para la escritura de los scripts

`jakarta.persistence.schema-generation.create-script-source`,  
`jakarta.persistence.schema-generation.drop-script-source`

- ▶ Especifican las ubicaciones de los los scripts a utilizar

## Atributos

- ▶ Una entidad debe tener una clave primaria (simple o compuesta).
- ▶ Los restantes atributos que conforman su estado deben asignarse a la tabla.
  - ▶ Tipos primitivos de Java (int, double, float, etc.) y wrappers Integer, Double, Float, etc.)
  - ▶ Arrays de bytes y chars (byte[], Byte[], char[], Char[]);
  - ▶ String, numéricos (java.math.BigInteger, java.math.BigDecimal), temporales (java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.time.LocalDate, java.time.LocalDateTime, java.time.OffsetTime, java.time.OffsetDateTime)
  - ▶ Enumerados y definidos por el usuario que implementan Serializable;
  - ▶ Colecciones de tipos básicos e embeddables.
- ▶ Otra entidad (o colección de entidades) : Relaciones.

## Configuración Atributos

- ▶ Configuración por excepción, todos los atributos se asignan mediante reglas de asignación predeterminadas.
- ▶ Se pueden personalizar esa asignación usando anotaciones (o su equivalente XML).

# Atributos no persistentes

- ▶ `@Transient` para anotar una propiedad o un campo que no es persistente.

```
1 @Target({METHOD, FIELD})  
2     @Retention(RUNTIME)  
3     public @interface Transient {}
```

- ▶ Ejemplo

```
1 @Entity  
2 public class Persona {  
3     @Id  
4     @GeneratedValue  
5     private Long codigo;  
6     private String nombre;  
7     private String apellido;  
8     @Transient  
9     private int edad;  
10    ...  
11 }
```

## @javax.persistence.Basic

- ▶ La anotación básica para propiedades del mapeo de un atributo a una columna de la base de datos.

```
1 @Target({METHOD, FIELD})  
2 @Retention(RUNTIME)  
3 public @interface Basic {  
4     FetchType fetch() default EAGER;  
5     boolean optional() default true;  
6 }
```

- ▶ Los parámetros son hints:
  - ▶ optional sugiere si el valor del atributo puede ser nulo.
    - ▶ No se tiene en cuenta para los tipos primitivos.
  - ▶ fetch sugiere el modo en que el proveedor de persistencia debe obtener los datos:
    - ▶ EAGER: al carga inicialmente la entidad.
    - ▶ LAZY: cuando la aplicación accede a la propiedad.

## Sugerencias (hints)

- ▶ Una sugerencia especifica una preferencia por parte de la aplicación.
- ▶ Debe ser observada por el proveedor si es posible,
- ▶ Puede no ser observada.
- ▶ No se debe depender de que la sugerencia se cumpla.

## Sugerencias (hints)

- ▶ Suponer que la Entidad Noticia, puede tener audio asociado
- ▶ El audio es un archivo WAV, esto es BLOB (objeto binario grande) de algunos megabytes.
- ▶ Cuando se carga una Noticia, podría no querer cargarse el archivo, se lo puede anotar como  
`@Basic(fetch = FetchType.LAZY)`

## Propiedad optional

- ▶ Si optional = **false** y se trata de persistir con un valor nulo, entonces

1 org.hibernate.PropertyValueException: not-**null** property  
references a **null** or **transient** value

- ▶ Las columnas que pueden almacenar grandes objetos requieren llamadas JDBC especiales
- ▶ Para informar al proveedor, se debe agregar la anotación @Lob
- ▶ @Lob indica que el valor es un objeto grande (LOB).
- ▶ el atributo audio de tipo **byte[]** se anota además con @Lob para almacenar el valor como un objeto grande (LOB).

```
1 @Entity
2 @IdClass(IdNoticia.class)
3 public class Noticia {
4     @Id
5     private String idioma;
6     private String titulo;
7     protected double fecha;
8     @Lob
9     @Basic(fetch = FetchType.LAZY)
10    private byte[] audio;
11    ...
12 }
```

The screenshot shows a MySQL Workbench interface with the following details:

- Toolbar:** Includes standard Mac OS X window controls (red, yellow, green buttons), a title bar with "localhost", and various icons for connection, refresh, and help.
- Session Bar:** Shows "Auto commit" checked, "Max rows: 1000", and "Auto complete" set to "Off". It also has "Auto select" set to "On".
- Left Panel (Object Browser):** Displays the database structure:
  - Root: jdbc:h2:~/test
  - NOTICIA schema:
    - IDIOMA
    - TITULO
    - AUDIO
    - BINARY LARGE OBJECT (selected)
    - Indexes
  - INFORMATION\_SCHEMA
  - Users
- Center Panel (SQL Statement):** Shows the SQL command "SHOW COLUMNS FROM NOTICIA;" entered in the text area.
- Bottom Panel (Results):** Displays the results of the query in a table format:

FIELD	TYPE	NULL	KEY	DEFAULT
IDIOMA	CHARACTER VARYING(255)	NO	PRI	NULL
TITULO	CHARACTER VARYING(255)	NO	PRI	NULL
AUDIO	BINARY LARGE OBJECT	YES		NULL

(3 rows, 2 ms)

## @javax.persistence.Column

- ▶ Define las propiedades de una columna: nombre, tamaño; autorizar (o no) que la columna tenga un valor nulo, sea única o permitir que su valor sea actualizable o insertable

```
1 @Target({METHOD, FIELD})
2 @Retention(RUNTIME)
3 public @interface Column {
4     String name()
5     boolean unique()
6     boolean nullable()
7     boolean insertable() default true;
8     boolean updatable() default true;
9     String columnDefinition() default "";
10    String table()      default "";
11    int length()       default 255;
12    int precision()    default 0; // decimal precision
13    int scale()        default 0; // decimal scale
14 }
```

- ▶ La mayoría de los elementos de @Column tienen influencia en el DDL generado por el proveedor.
- ▶ Si cambia la longitud del atributo (con length) y se genera el DDL, la longitud de la columna se cambia
- ▶ Si no se genera el DDL, la anotación no tiene efecto.
  - ▶ al mapear a una tabla existente, la longitud será la establecida por la base de datos.
- ▶ No todas las combinaciones de atributos con los tipos de datos son válidas (por ejemplo, la longitud solo se aplica a valores de cadena).

- ▶ `updatable` e `insertable` tienen influencia durante el tiempo de ejecución:
  - ▶ Indica si el atributo se debe incluir en la sentencia SQL al insertar / actualizar la entidad.
- ▶ Se fija en **false** cuando se desea que el proveedor de persistencia se asegure de que no insertará ni actualizará los datos en la tabla en respuesta a cambios en la entidad.
- ▶ Esto no implica que el atributo de la entidad no cambie en la memoria.
  - ▶ Se puede cambiar el valor en la memoria, pero no se sincronizará con la base de datos.
  - ▶ la sentencia SQL generada no incluye estas columnas.

## Propiedad @Temporal

- ▶ Se pueden usar las clases `java.time` para almacenar fechas en diferentes formatos, como una fecha, una hora o milisegundos.

<b>Tipo Java</b>	<b>Tipo JDBC</b>
<code>java.time.Duration</code>	<code>BIGINT</code>
<code>java.time.Instant</code>	<code>TIMESTAMP</code>
<code>java.time.LocalDateTime</code>	<code>TIMESTAMP</code>
<code>java.time.LocalDate</code>	<code>DATE</code>
<code>java.time.LocalTime</code>	<code>TIME</code>
<code>java.time.OffsetDateTime</code>	<code>TIMESTAMP</code>
<code>java.time.OffsetTime</code>	<code>TIME</code>
<code>java.time.ZonedDateTime</code>	<code>TIMESTAMP</code>

- ▶ Para código que usa `java.util.Date` y `java.util.Calendar`, se usa `@javax.persistence.Temporal` para especificar el mapeo.
- ▶ Tiene tres valores posibles: DATE, TIME, or TIMESTAMP (solo fecha, solo la hora, o ambas).

## Mapping default de Date

```
1 @Entity
2 @IdClass(IdNoticia.class)
3 public class Noticia {
4     @Id
5     private String idioma;
6     @Column(length = 400)
7     private String titulo;
8     @Lob
9     @Basic(fetch = FetchType.LAZY)
10    private byte[] audio;
11    //por default: TIMESTAMP
12    private Date fecha;
13    ...
14 }
```

# Mapping default de Date

The screenshot shows a web-based database interface with the following details:

- Toolbar:** Includes standard browser controls (back, forward, search, etc.), a connection to "localhost", and various tool icons.
- Session Information:** Shows "Auto commit" checked, "Max rows: 1000", and other settings like "Auto complete" (Off) and "Auto select" (On).
- Left Sidebar:** Displays the database connection path "jdbc:h2:~/test" and a tree view of the database schema:
  - NOTICIA
  - INFORMATION\_SCHEMA
  - UsersA note indicates the version is H2 2.1.210 (2022-01-17).
- SQL Statement:** The SQL command "SHOW COLUMNS FROM NOTICIA;" is entered in the text input field.
- Result Preview:** Below the input field, the results of the query are shown with the heading "SHOW COLUMNS FROM NOTICIA;". The results are presented in a table:

FIELD	TYPE	NULL	KEY	DEFAULT
IDIOMA	CHARACTER VARYING(255)	NO	PRI	NULL
TITULO	CHARACTER VARYING(400)	NO	PRI	NULL
AUDIO	BINARY LARGE OBJECT	YES		NULL
FECHA	TIMESTAMP	YES		NULL

(4 rows, 135 ms)

## @Temporal

```
1 @Entity
2 @IdClass(IdNoticia.class)
3 public class Noticia {
4     @Id
5     private String idioma;
6     @Column(length = 400)
7     private String titulo;
8     @Lob
9     @Basic(fetch = FetchType.LAZY)
10    private byte[] audio;
11    @Temporal(TemporalType.DATE)
12    private Date fecha;
13    ...
14 }
```

Si se anota un tipo distinto de Date y Calendar

```
1 org.hibernate.AnnotationException: @Temporal should only be set
2       on a java.util.Date or java.util.Calendar property....
3 }
```

# Notar cambios en BD

The screenshot shows a web-based database management interface. The top navigation bar includes standard browser controls (back, forward, search, etc.) and a title bar showing "localhost". Below the title bar are several configuration buttons: "Auto commit" (checked), "Max rows: 1000", "Auto complete" (Off), "Auto select" (On), and a help button. On the left side, there is a sidebar with a tree view of database objects:

- jdbc:h2:~/test
- NOTICIA
- INFORMATION\_SCHEMA
- Users
- H2 2.1.210 (2022-01-17)

The main area contains a SQL command input field with the text "SHOW COLUMNS FROM NOTICIA" and several execution buttons: "Run", "Run Selected", "Auto complete", "Clear", and "SQL statement:". Below this, the results of the query are displayed in a table:

SHOW COLUMNS FROM NOTICIA;

FIELD	TYPE	NULL	KEY	DEFAULT
IDIOMA	CHARACTER VARYING(255)	NO	PRI	NULL
TITULO	CHARACTER VARYING(400)	NO	PRI	NULL
AUDIO	BINARY LARGE OBJECT	YES		NULL
FECHA	DATE	YES		NULL

(4 rows, 7 ms)

## @Enumerate

- ▶ Los valores de una enumeración son constantes.
- ▶ Tienen una asignación ordinal implícita que está determinada por el orden en que se declaran en la enumeración.
- ▶ Se usa el ordinal para almacenar el valor del tipo enumerado en la base de datos.

# Ejemplo

```
1 public enum Ambito {  
2     NACIONAL,  
3     REGIONAL,  
4     INTERNACIONAL  
5 }
```

# Ejemplo

```
1 @Entity
2 @IdClass(IdNoticia.class)
3 public class Noticia {
4     @Id
5     private String idioma;
6     @Column(length = 400)
7     private String titulo;
8     @Lob
9     @Basic(fetch = FetchType.LAZY)
10    private byte[] audio;
11    @Temporal(TemporalType.DATE)
12    private Date fecha;
13    @Transient
14    private int antiguedad;
15    private Ambito ambito;
16 }
```

# Ejemplo

```
1 Noticia n1 = new Noticia();
2 n1.setTitulo("Goal de Boca ...");
3 n1.setIdioma("ES");
4 n1.setAmbito(Ambito.NACIONAL);
5
6 Noticia n2 = new Noticia();
7 n2.setTitulo("Pierde el PSG ....");
8 n2.setIdioma("ES");
9 n2.setAmbito(Ambito.INTERNACIONAL);
10
11 tx.begin();
12 em.persist(n1);
13 em.persist(n2);
14 tx.commit();
15 }
```

# Mapping default del enumerado

The screenshot shows a Java IDE interface with a database browser and a SQL editor.

**Database Browser:**

- Connected to: jdbc:h2:~/test
- Schemas listed: NOTICIA, INFORMATION\_SCHEMA, Users
- NOTICIA schema details:
  - Tables: IDIOMA, TITULO, AMBITO
  - Column types: INTEGER, AUDIO, FECHA
  - Indexes
- INFORMATION\_SCHEMA
- Users
- H2 2.1.210 (2022-01-17)

**SQL Editor:**

Auto commit: On | Max rows: 1000 | Auto complete: Off | Auto select: On

SQL statement:

```
SELECT * FROM NOTICIA;
```

**Result:**

```
SELECT * FROM NOTICIA;
```

IDIOMA	TITULO	AMBITO	AUDIO	FECHA
ES	Goal de Boca ...	0	null	null
ES	Pierde el PSG ....	2	null	null

(2 rows, 2 ms)

**Buttons:**

- Run, Run Selected, Auto complete, Clear, SQL statement
- Edit

## @Enumerated

- ▶ Si se cambia la definición del enumerado

```
1 public enum Ambito {LOCAL, NACIONAL, REGIONAL,  
INTERNACIONAL}
```

Cambia la interpretación de los elementos en la tabla.

- ▶ Alternativa: almacenar el nombre del valor como una String (en lugar del ordinal).
- ▶ Anotación @Enumerated
  - ▶ ORDINAL : es el valor predeterminado
  - ▶ STRING

```
1 @Entity  
2 @IdClass(IdNoticia.class)  
3 public class Noticia {  
4     ...  
5     @Enumerated(EnumType.STRING)  
6     private Ambito ambito;  
7 }
```

# Mapping de enumerado como string

The screenshot shows a Java application window with a database interface. The title bar says "localhost". The toolbar includes standard icons like back, forward, and search, along with "Auto commit" checked, "Max rows: 1000", "Auto complete Off", "Auto select On", and a help icon.

The left sidebar shows the database structure:

- jdbc:h2:~/test
- NOTICIA
  - IDIOMA
  - TITULO
  - AMBITO
  - CHARACTER VARYING(2)
  - AUDIO
  - FECHA
  - Indexes
- INFORMATION\_SCHEMA
- Users
- H2 2.1.210 (2022-01-17)

The main panel contains a SQL editor and a results table.

SQL Editor:

```
SELECT * FROM NOTICIA
```

Results Table:

IDIOMA	TITULO	AMBITO	AUDIO	FECHA
ES	Goal de Boca ...	NACIONAL	null	null
ES	Pierde el PSG ....	INTERNACIONAL	null	null

(2 rows, 4 ms)

## Conversiones definidas por el usuario

- ▶ Es posible convertir un tipo de datos de un formato a cualquier otro formato.
- ▶ Es necesario crear un convertidor que transforma los tipos del atributo de la entidad a los de la columna de base de datos y viceversa.
- ▶ Debemos dar una implementación para la interface `AttributeConverter`.
- ▶ Para hacerlo en JPA, tenemos que anotar un clase con `@Converter`.

## Interfaz pública AttributeConverter<X,Y>

- ▶ X: tipo del atributo de la entidad
- ▶ Y: tipo de la columna de la base de datos
- ▶ Provee operaciones para convertir el valor de un atributo de la entidad en una representación de la columna de la base de datos y viceversa.
  - ▶ Y convertToDatabaseColumn(X attribute)
  - ▶ X convertToEntityAttribute(Y dbData):

## Anotación @Converter

- ▶ Especifica que la clase anotada es un convertidor.

# Ejemplo

```
1 @Converter
2 public class AmbitoConverter implements AttributeConverter<
3     Ambito, Character> {
4     @Override
5     public Character convertToDatabaseColumn(Ambito ambito) {
6         switch (ambito) {
7             case INTERNACIONAL:
8                 return 'I';
9             case NACIONAL:
10                return 'N';
11            case REGIONAL:
12                return 'R';
13            default:
14                throw new IllegalArgumentException(ambito + "Desconocido")
15                ;
16        } }
17    ...
18 }
```

# Ejemplo

```
1 @Converter
2 public class AmbitoConverter implements AttributeConverter<
3     Ambito, Character> {
4     ...
5     @Override
6     public Ambito convertToEntityAttribute(Character letra) {
7         switch (letra) {
8             case 'I':
9                 return INTERNACIONAL;
10            case 'N':
11                return NACIONAL;
12            case 'R':
13                return REGIONAL;
14            default:
15                throw new IllegalArgumentException(letra + "Desconocida");
16        }
17    }
```

# Conversiones definidas por el usuario

```
1 @Entity  
2 @IdClass(IdNoticia.class)  
3 public class Noticia {  
4     ...  
5     @Convert(converter = AmbitoConverter.class)  
6     private Ambito etiqueta;  
7 }
```

```
1     Noticia n1 = new Noticia();  
2     ...  
3     n1.setEtiqueta(NACIONAL);
```

# Notar cambios en BD

The screenshot shows a MySQL Workbench window. On the left, the database schema is displayed under the connection 'jdbc:h2:~/test'. The 'NOTICIA' table is expanded, showing columns: IDIOMA, TITULO, AUDIO, and ETIQUETA. The 'ETIQUETA' column is defined as 'CHARACTER(255)'. Below the schema, the 'INFORMATION\_SCHEMA' and 'Users' sections are visible, along with the version information 'H2 2.1.210 (2022-01-17)'.

In the main area, a SQL statement 'SELECT \* FROM NOTICIA' is entered in the 'SQL statement:' field. Below it, the results of the query are shown in a table:

IDIOMA	TITULO	AUDIO	ETIQUETA
ES	Goal de Boca ...	null	N

The message '(1 row, 0 ms)' indicates the query executed quickly.

Ver el tamaño default de la columna. Se puede modificar usando  
@Column(length = 1)

## Colección de tipos básicos

- ▶ Usamos la anotación @ElementCollection para indicar que un atributo de tipo java.util.Collection contiene una colección de instancias de tipos básicos (no entidades) o embeddable
- ▶ El atributo puede ser de los siguientes tipos:
  - ▶ java.util.Collection
  - ▶ java.util.Set
  - ▶ java.util.List

## @ElementCollection

```
1 @Entity
2 @IdClass(IdNoticia.class)
3 public class Noticia {
4     @Id
5     private String idioma;
6     @Column(length = 400)
7     private String titulo;
8     @Lob
9     @Basic(fetch = FetchType.LAZY)
10    private byte[] audio;
11    @Temporal(TemporalType.DATE)
12    private Date fecha;
13    @Transient
14    private int antiguedad;
15    @ElementCollection
16    private Set<String> etiquetas = new HashSet<>();
17
18    public void agregarEtiqueta(String e) {
19        etiquetas.add(e);
20    }
21 }
```

## @ElementCollection

```
1 Noticia n1 = new Noticia();
2 n1.setTitulo("Goal de Boca ...");
3 n1.setIdioma("ES");
4 n1.agregarEtiqueta("Nacional");
5 n1.agregarEtiqueta("Deporte");
6 n1.agregarEtiqueta("Futbol");
```

## @ElementCollection

The screenshot shows a MySQL Workbench interface connected to a database named 'test'. The left sidebar displays the database schema with tables 'NOTICIA' and 'NOTICIA\_ETIQUETAS'. A query window is open with the SQL statement 'SELECT \* FROM NOTICIA'. The results show one row:

IDIOMA	TITULO	AUDIO	FECHA
ES	Goal de Boca ...	null	null

(1 row, 0 ms)

## @ElementCollection

The screenshot shows a MySQL Workbench interface with the following details:

- Toolbar:** Includes standard Mac OS X window controls, a connection icon, and a search bar labeled "localhost".
- Connections:** A dropdown menu shows "Auto commit" is selected.
- Session Information:** Max rows: 1000, Auto complete: Off, Auto select: On.
- Left Panel (Schema Browser):**
  - Selected connection: jdbc:h2:~/test
  - Database structure:
    - NOTICIA (with sub-tables: IDIOMA, TITULO, AUDIO, FECHA)
    - NOTICIA\_ETIQUETAS (with sub-tables: NOTICIA\_IDIOMA, NOTICIA\_TITULO, ETIQUETAS)
    - Indexes
  - INFORMATION\_SCHEMA
  - Users
  - H2 2.1.210 (2022-01-17)
- Central Panel (Query Editor):**
  - SQL statement: `SELECT * FROM NOTICIA_ETIQUETAS;`
  - Result set:

NOTICIA_IDIOMA	NOTICIA_TITULO	ETIQUETAS
ES	Goal de Boca ...	Nacional
ES	Goal de Boca ...	Futbol
ES	Goal de Boca ...	Deporte

(3 rows, 15 ms)

# Tipo de acceso por default

- ▶ @ElementCollection indica que el atributo **etiquetas** es una colección.
- ▶ Crea una tabla cuyos registros corresponden a un elemento de la colección.
  - ▶ El nombre de la tabla por defecto es una concatenación del nombre de la entidad contenedora y el nombre del atributo de tipo colección **NOTICIA\_ETIQUETAS**.
  - ▶ Notar que la clave de la contendora son columnas de **NOTICIA\_ETIQUETAS**.
  - ▶ La table **NOTICIA\_ETIQUETAS** tiene una columna con el nombre del atributo. **ETIQUETAS**
  - ▶ Se puede cambiar.

## @ElementCollection

```
1 @CollectionTable(name = "Etiquetas")
2 @Column(name = "valor")
3 private Set<String> etiquetas = new HashSet<>();
```

## @ElementCollection

The screenshot shows a MySQL Workbench window with the following details:

- Toolbar:** Includes standard Mac OS X window controls, a title bar with "localhost", and various tool icons.
- Connection:** "Auto commit" is checked. Other connection parameters like "Max rows: 1000" and "Auto complete" are also visible.
- Left Panel (Schema Browser):** Shows the database structure:
  - Test schema: contains ETIQUETAS (NOTICIA\_IDIOMA, NOTICIA\_TITULO, VALOR), NOTICIA (IDIOMA, TITULO, AUDIO, FECHA), and two indexes.
  - INFORMATION\_SCHEMA: contains USERS.
  - System information: H2 2.1.210 (2022-01-17).
- Central Panel:** A large text area for running SQL statements. Buttons above it include "Run", "Run Selected", "Auto complete", "Clear", and "SQL statement:".
- Bottom Panel (Important Commands):** A table listing keyboard shortcuts:

Action	Key Shortcut
Displays this Help Page	?
Shows the Command History	Up Arrow
Executes the current SQL statement	Ctrl+Enter
Executes the SQL statement defined by the text selection	Shift+Enter

## @ElementCollection

```
1 @Target({METHOD, FIELD})  
2 @Retention(RUNTIME)  
3 public @interface ElementCollection {  
4     Class targetClass() default void.class;  
5     FetchType fetch() default LAZY;  
6 }
```

- ▶ targetClass:(Opcional) La clase del tipo que es el elemento de la colección.
- ▶ Es opcional sólo si la propiedad o el campo de colección se define mediante genéricos de Java.

# Ejemplo

- ▶ Por ejemplo para una colección de Strings sin uso de generics

```
1 @ElementCollection(targetClass = String.class, fetch =  
2     FetchType.LAZY)  
private Set etiquetas = new HashSet();
```

- ▶ Si no se especifica targetClass alcanza excepción

```
1 @ElementCollection(fetch = FetchType.LAZY)  
2 private Set etiquetas = new HashSet();
```

```
1 org.hibernate.AnnotationException: Collection has neither  
generic type or OneToMany.targetEntity() defined ...
```

## Mapeo de colecciones tipo Map

```
1 @Entity  
2 @IdClass(IdNoticia.class)  
3 public class Noticia {  
4     ...  
5     @ElementCollection  
6     private Map<Integer, String> etiquetas = new HashMap<>();  
7     ...  
8 }
```

## @ElementCollection

The screenshot shows a MySQL Workbench interface running on a Mac OS X system. The title bar says "localhost". The left sidebar displays the database schema:

- jdbc:h2:~/test
- NOTICIA
- NOTICIA\_ETIQUETAS
  - NOTICIA\_IDIOMA
  - NOTICIA\_TITULO
  - ETIQUETAS
  - ETIQUETAS\_KEY
- Indexes
- INFORMATION\_SCHEMA
- Users
- H2 2.1.210 (2022-01-17)

The main pane shows the following SQL statement and its results:

```
SELECT * FROM NOTICIA_ETIQUETAS;
```

NOTICIA_IDIOMA	NOTICIA_TITULO	ETIQUETAS	ETIQUETAS_KEY
ES	Goal de Boca ...	Nacional	0
ES	Goal de Boca ...	Deporte	1
ES	Goal de Boca ...	Futbol	2

(3 rows, 2 ms)

Buttons at the bottom include "Edit".

## @MapKeyColumn

- ▶ Esta anotación se utiliza para especificar el mapeo para la columna que almacena la clave del diccionario.
- ▶ Si no se especifica, la columna se nombra con la concatenación del nombre del atributo de relación de referencia y \_KEY.

# Ejemplo

```
1 @ElementCollection  
2 @CollectionTable(name = "Etiquetas")  
3 @MapKeyColumn(name = "position")  
4 @Column(name = "valor")  
5 private Map<Integer, String> etiquetas = new HashMap<>();
```

## @ElementCollection

The screenshot shows a Java application window with a database interface. The title bar says "localhost". The toolbar includes "Auto commit" (checked), "Max rows: 1000", "Auto complete" (Off), "Auto select" (On), and other standard icons.

The left sidebar shows the database schema:

- jdbc:h2:~/test
- ETIQUETAS
  - NOTICIA\_IDIOMA
  - NOTICIA\_TITULO
  - VALOR
    - CHARACTER VARYING(2)
  - POSITION
    - INTEGER
  - Indexes
- NOTICIA
- INFORMATION\_SCHEMA
- Users
- H2 2.1.210 (2022-01-17)

The main area contains a SQL editor and a results table.

SQL Editor:

```
SELECT * FROM ETIQUETAS
```

Results:

```
SELECT * FROM ETIQUETAS;
```

NOTICIA_IDIOMA	NOTICIA_TITULO	VALOR	POSITION
ES	Goal de Boca ...	Nacional	0
ES	Goal de Boca ...	Deporte	1
ES	Goal de Boca ...	Futbol	2

(3 rows, 6 ms)

## Ejercicio

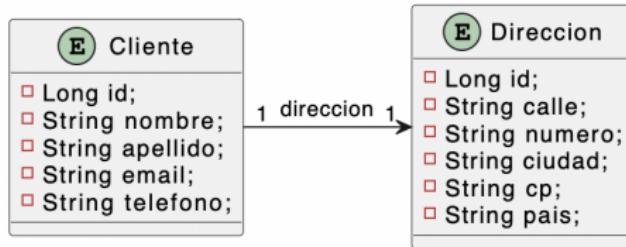
- ▶ Considerar a la clase Empleado y los scripts dados.
- ▶ Definir el mapping necesario para persistir a la clase Empleado sobre las tablas.
- ▶ La clase Empleado sólo puede ser modificada agregando anotaciones.

# Relaciones en OO

- ▶ Asociaciones son relaciones estructurales: vinculan objetos de un tipo con objetos de otro.
- ▶ Posibilitan que un objeto realice una acción sobre otro.
- ▶ Una asociación tiene una dirección:
  - ▶ puede ser unidireccional (un objeto puede navegar hacia otro) o
  - ▶ bidireccional ( un objeto puede navegar hacia otro y viceversa).
- ▶ En Java, se utiliza la sintaxis *punto* (.) para navegar a través de los objetos.

# Relaciones en OO

- ▶ En UML (Unified Modelling Language),
  - ▶ Se representa a una asociación unidireccional con una flecha
- ejemplo**



# Relaciones en BD

- ▶ Una base de datos relacional es una colección de relaciones (tablas),
- ▶ Cualquier cosa que se modele es una tabla (relación).
- ▶ No hay listas, conjuntos o diccionarios; sólo tablas.
- ▶ Una asociación entre dos entidades se puede modelar en la base de datos de dos maneras diferentes:
  - ▶ usando una clave externa (también conocida como columna *join*) o
  - ▶ usando una tabla *join*.
- ▶ una columna que hace referencia a una clave de otra tabla es una columna de clave externa (*foreign key column*).

## Relaciones entre entidades

- ▶ La mayoría de las entidades necesitan referenciar o tener relaciones con otras entidades.
- ▶ JPA permite mapear asociaciones para que una entidad pueda vincularse con otra en un modelo relacional.
- ▶ JPA usa la configuración por excepción para las asociaciones.
- ▶ tiene una forma predeterminada de definir una relación pero existen anotaciones para personalizar el mapeo.

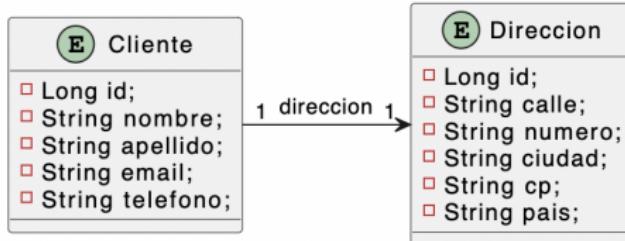
## Relaciones entre entidades

- ▶ La cardinalidad de la relación puede ser de uno a uno, de uno a muchos, de muchos a uno o de muchos a muchos.
- ▶ Se usan @OneToOne, @OneToMany, @ManyToOne o @ManyToMany
- ▶ Cada anotación se puede utilizar de forma unidireccional o bidireccional.

## unidireccional o bidireccional

- ▶ En una asociación unidireccional, el objeto A apunta al objeto B y no al revés;
- ▶ en una asociación bidireccional, ambos objetos se refieren entre sí.
- ▶ Se necesita algo de trabajo al asignar una relación bidireccional a una base de datos relacional.

## Ejemplo: unidireccional



- ▶ Un cliente tiene una dirección.
- ▶ La relación es unidireccional.
- ▶ Se navega de Cliente a Direccion.
- ▶ Se dice que el Cliente es el dueño de la relación.
- ▶ Implementación: Cliente tiene un atributo de tipo Direccion.
- ▶ En términos de la base de datos, la tabla CLIENTE tendrá una clave externa (columna join) que refiere a DIRECCIÓN.

## Ejemplo: unidireccional

```
1 @Entity
2 public class Cliente {
3     @Id
4     @GeneratedValue
5     private Long id;
6     private String nombre;
7     private String apellido;
8     private String email;
9     private String telefono;
10    //Implementa la relación direccion
11    private Direccion direccion;
12
13    // Constructores, getters y setters
14 }
15 }
```

## Ejemplo: unidireccional

```
1 @Entity
2 public class Direccion {
3     @Id
4     @GeneratedValue
5     private Long id;
6     private String calle;
7     private String numero;
8     private String ciudad;
9     private String cp;
10    private String pais;
11
12    // Constructores, getters y setters
13 }
14 }
```

## Ejemplo: unidireccional

- ▶ Las dos entidades tienen las anotaciones mínimas requeridas: `@Entity`, `@Id` y para la clave principal.
- ▶ Un mapeo uno a uno se establece porque la entidad `Cliente` incluye a la entidad `Dirección` como un atributo.
- ▶ Usamos `@OneToOne` para declarar la relación uno-a-uno.
- ▶ El proveedor de persistencia asignará estas dos entidades a dos tablas y una clave externa para la relación (desde el cliente que apunta a la dirección).

## Ejemplo: Anotación @OneToOne

```
1  @Entity
2  public class Cliente {
3      @Id
4      @GeneratedValue
5      private Long id;
6      private String nombre;
7      private String apellido;
8      private String email;
9      private String telefono;
10     @OneToOne
11     private Direccion direccion;
12     // Constructores, getters y setters
13 }
14 }
```

# Ejemplo: unidireccional

The screenshot shows a Java-based application window with the following details:

- Toolbar:** Includes standard Mac OS X window controls (red, yellow, green buttons), a back/forward button, a search field containing "localhost", and various tool icons.
- MenuBar:** Contains "File", "Edit", "View", "Tools", "Help".
- Toolbars:** "Auto commit" (checkbox checked), "Max rows: 1000", "Run Selected", "Auto complete" (checkbox off), "Auto select" (checkbox on).
- Left Panel (Database Browser):** Shows the database structure:
  - Root: jdbc:h2:~/test
  - CLIENTE table:
    - ID
    - APELLIDO
    - EMAIL
    - NOMBRE
    - TELEFONO
    - DIRECCION\_ID
    - Indexes (indicated by a downward arrow)
  - DIRECCION table:
    - ID
    - CALLE
    - CIUDAD
    - CP
    - NUMERO
    - PAIS
    - Indexes (indicated by a downward arrow)
  - INFORMATION\_SCHEMA table:
    - Sequences
- Central Panel:** A large empty area labeled "SQL statement:" with buttons for "Run", "Run Selected", "Auto complete", and "Clear".
- Bottom Panel (Help):** Titled "Important Commands", it lists keyboard shortcuts:

	Displays this Help Page
	Shows the Command History
	Ctrl+Enter Executes the current SQL statement
	Shift+Enter Executes the SQL statement defined by the text selection

Notar que CLIENTE contiene una columna DIRECCION\_ID

# Ejemplo: unidireccional

The screenshot shows a MySQL Workbench window with the following details:

- Toolbar:** Includes standard Mac OS X window controls (red, yellow, green buttons), a back/forward button, a search icon, and a connection status bar showing "localhost".
- Connection:** "jdbc:h2:~/test" is selected.
- Schemas:** The left sidebar lists the database structure:
  - CLIENTE:** Contains columns ID, APELLIDO, EMAIL, NOMBRE, TELEFONO, and DIRECCION\_ID.
  - DIRECCION:** Contains columns ID, CALLE, CIUDAD, CP, NUMERO, and PAIS.
  - INFORMATION\_SCHEMA** and **Sequences** are also listed.
- SQL Editor:** Shows the SQL statement "SELECT \* FROM DIRECCION" in the "SQL statement:" field. Buttons include Run, Run Selected, Auto complete, Clear, and SQL statement:.
- Results:** Below the editor, the results of the query are displayed:

SELECT \* FROM DIRECCION;

ID	CALLE	CIUDAD	CP	NUMERO	PAIS
1	Cabildo	CABA	1424	5030	Argentina

(1 row, 11 ms)

An "Edit" button is located below the results table.

# Ejemplo: unidireccional

The screenshot shows a MySQL Workbench interface with the following details:

- Database:** jdbc:h2:~/test
- Schemas:** CLIENTE, DIRECCION, INFORMATION\_SCHEMA, Sequences, Users
- Tables:** CLIENTE (with columns ID, APELLIDO, EMAIL, NOMBRE, TELEFONO, DIRECCION\_ID), DIRECCION (with columns ID, CALLE, CIUDAD, CP, NUMERO, PAIS)
- SQL Statement:** SELECT \* FROM CLIENTE
- Result:**

ID	APELLIDO	EMAIL	NOMBRE	TELEFONO	DIRECCION_ID
1	Perez	jp@jp.ar	Juan	null	1

(1 row, 1 ms)

**Edit**

## Ejemplo: unidireccional

- ▶ Por defecto, el nombre de la columna de clave externa es `DIRECCION_ID`, que es el concatenación del nombre del atributo de relación (`direccion`), el símbolo `_` y el nombre de la columna de clave primaria de la tabla de destino (`ID` de la columna de la tabla `DIRECCION`)
- ▶ la columna `DIRECCION_ID` puede ser **null** (así que puede tomar el valor 0).
- ▶ Notar que sucede si persiste el propietario sin el destino de la relación

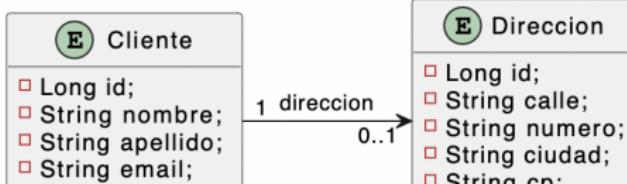
# Ejemplo: unidireccional

The screenshot shows a browser window with the address bar set to 'localhost'. The left sidebar lists database objects: 'jdbc:h2:~/test', 'CLIENTE', 'DIRECCION', 'INFORMATION\_SCHEMA', 'Sequences', and 'Users'. A note at the bottom indicates the version: 'H2 2.1.210 (2022-01-17)'. The main content area contains the SQL command 'SHOW COLUMNS FROM CLIENTE' and its results.

FIELD	TYPE	NULL	KEY	DEFAULT
ID	BIGINT	NO	PRI	NULL
APELLIDO	CHARACTER VARYING(255)	YES		NULL
EMAIL	CHARACTER VARYING(255)	YES		NULL
NOMBRE	CHARACTER VARYING(255)	YES		NULL
TELEFONO	CHARACTER VARYING(255)	YES		NULL
DIRECCION_ID	BIGINT	YES		NULL

(6 rows, 105 ms)

En realidad modela relaciones 1 a {0, 1}



## @OneToOne

```
1 @Target({METHOD, FIELD})
2 @Retention(RUNTIME)
3 public @interface OneToOne {
4     Class targetEntity() default void.class;
5     CascadeType[] cascade() default {};
6     FetchType fetch() default EAGER;
7     boolean optional() default true;
8     String mappedBy() default "";
9     boolean orphanRemoval() default false;
10 }
```

## Ejemplo: Anotación @OneToOne (not null)

```
1  @Entity
2  public class Cliente {
3      @Id
4      @GeneratedValue
5      private Long id;
6      private String nombre;
7      private String apellido;
8      private String email;
9      private String telefono;
10     @OneToOne(optional = false)
11     private Direccion direccion;
12     // Constructores, getters y setters
13 }
14 }
```

## Ejemplo: Anotación @OneToOne (not null)

The screenshot shows a MySQL Workbench interface with the following details:

- Toolbar:** Includes standard icons for file operations, search, and database navigation.
- Connection:** Connected to "localhost" via "jdbc:h2:~/test".
- Status Bar:** Shows "Auto commit" is checked, "Max rows: 1000", and various status indicators.
- Left Panel (Schemas):** Lists available databases and schemas: "CLIENTE", "DIRECCION", "INFORMATION\_SCHEMA", "Sequences", and "Users". It also displays the version "H2 2.1.210 (2022-01-17)".
- SQL Editor:** Contains the SQL command "SHOW COLUMNS FROM CLIENTE".
- Results Grid:** Displays the columns of the "CLIENTE" table.

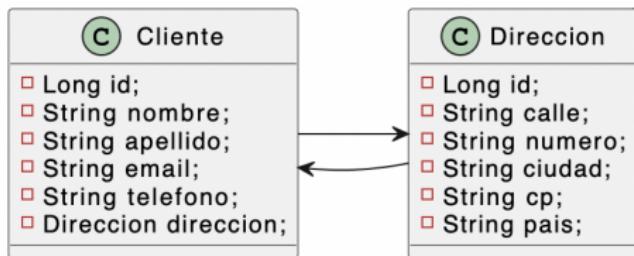
FIELD	TYPE	NULL	KEY	DEFAULT
ID	BIGINT	NO	PRI	NULL
APELLIDO	CHARACTER VARYING(255)	YES		NULL
EMAIL	CHARACTER VARYING(255)	YES		NULL
NOMBRE	CHARACTER VARYING(255)	YES		NULL
TELEFONO	CHARACTER VARYING(255)	YES		NULL
DIRECCION_ID	BIGINT	NO	UNI	NULL
- Message:** Below the grid, it says "(6 rows, 14 ms)" indicating the execution time of the query.

# Relación bidireccional



- ▶ Las relaciones pueden ser bidireccionales.
- ▶ Para poder navegar entre Dirección y Cliente, debe transformar una relación unidireccional en una bidireccional
- ▶ Agregar un atributo Cliente a la entidad Dirección
- ▶ Es similar a tener dos relaciones uno-a-uno separadas, uno en cada dirección.
- ▶ Se puede pensar en una relación bidireccional como un par de relaciones unidireccionales, en ambos sentidos.

## Ejemplo: One-To-One bidireccional



## Ejemplo: One-To-One bidireccional

```
1 public class Cliente {  
2     private Long id;  
3     private String nombre;  
4     private String apellido;  
5     private String email;  
6     private String telefono;  
7     private Direccion direccion;  
8     ...  
9 }
```

```
1 public class Direccion {  
2     private Long id;  
3     private String calle;  
4     private String numero;  
5     private String ciudad;  
6     private String cp;  
7     private String pais;  
8     //Navegación inversa  
9     private Cliente cliente;  
10 }
```

## Mapeo bidireccionales

- ▶ Si simplemente se anota cada entidad con @OneToOne, entonces tenemos dos relaciones unidireccionales independientes.

## Ejemplo: One-To-One bidireccional

```
1 @Entity
2 public class Cliente {
3     @Id
4     @GeneratedValue
5     private Long id;
6     ...
7     @OneToOne
8     private Direccion direccion;
9     ...
10 }
```

```
1 @Entity
2 public class Direccion {
3     @Id
4     @GeneratedValue
5     private Long id;
6     ...
7     @OneToOne
8     private Cliente cliente;
9 }
```

# Ejemplo: One-To-One bidireccional (mal)

The screenshot shows a Java application window with a toolbar at the top. The toolbar includes icons for file operations, a connection icon labeled 'localhost', volume control, and other standard icons. Below the toolbar is a menu bar with 'File', 'Edit', 'View', 'Tools', 'Help', and a 'Database' menu. The main area has a left sidebar showing a database schema:

- jdbc:h2:~/test
- CLIENTE
  - ID
  - APELLIDO
  - EMAIL
  - NOMBRE
  - TELEFONO
  - DIRECCION\_ID
  - Indexes
- DIRECCION
  - ID
  - CALLE
  - CIUDAD
  - CP
  - NUMERO
  - PAIS
  - CLIENTE\_ID
  - Indexes
- INFORMATION\_SCHEMA
- Sequences

To the right of the schema is a SQL editor pane with the following content:

```
SQL statement:  
DROP ALL OBJECTS
```

Below the SQL editor is a section titled "Important Commands" containing the following table:

	Displays this Help Page
	Shows the Command History
	Ctrl+Enter Executes the current SQL statement
	Shift+Enter Executes the SQL statement defined by the text selection
	Ctrl+Space Auto complete

Las dos tablas tienen claves externas

# Ejemplo: One-To-One bidireccional (mal)

The screenshot shows a MySQL Workbench interface. On the left, the database tree shows a connection named 'jdbc:h2:~/test' and several schema nodes: 'CLIENTE', 'DIRECCION', 'INFORMATION\_SCHEMA', 'Sequences', and 'Users'. A status message at the bottom indicates 'H2 2.1.210 (2022-01-17)'. The main pane displays the results of a SQL query:

```
SELECT * FROM CLIENTE
```

Below the results, the query is repeated:

```
SELECT * FROM CLIENTE;
```

ID	APELLIDO	EMAIL	NOMBRE	TELEFONO	DIRECCION_ID
2	Perez	jp@jp.ar	Juan	null	1

(1 row, 1 ms)

An 'Edit' button is visible below the table.

Según **CLIENTE**, el cliente ID 2 está en relación con la dirección 1

# Ejemplo: One-To-One bidireccional (mal)

The screenshot shows a MySQL Workbench interface. On the left, the database tree shows a connection to 'jdbc:h2:~/test' containing tables 'CLIENTE' and 'DIRECCION', and schemas 'INFORMATION\_SCHEMA' and 'Sequences'. The right pane displays the results of a SQL query:

```
SELECT * FROM DIRECCION
```

The results table shows the following data:

ID	CALLE	CIUDAD	CP	NUMERO	PAIS	CLIENTE_ID
1	Cabildo	CABA	1424	5030	Argentina	null
3	Monroe	CABA	1424	1200	Argentina	2

(2 rows, 1 ms)

**Edit**

Según DIRECCION, el cliente ID 2 está en relación con la dirección 3

## Relaciones bidireccionales

- ▶ Se debe indicar que las dos declaraciones son extremos de una relación bidireccional.
- ▶ Las relaciones bidireccionales tienen un lado propietario y otro inverso, que deben especificarse explícitamente
- ▶ El lado inverso de una relación bidireccional debe hacer referencia a su lado propietario mediante el uso del elemento @mappedBy de la anotación @OneToOne.
- ▶ El elemento @mappedBy designa el atributo en la entidad que es propietaria de la relación.
- ▶ Para relaciones bidireccionales uno-a-uno, el lado propietario corresponde al lado que contiene la clave foránea correspondiente.

## Ejemplo: Cliente como owner

```
1 @Entity  
2 public class Cliente {  
3     @Id @GeneratedValue  
4     private Long id;  
5     ...  
6     @OneToOne  
7     private Direccion direccion;  
8     ...  
9 }
```

```
1 @Entity  
2 public class Direccion {  
3     @Id @GeneratedValue  
4     private Long id;  
5     ...  
6     @OneToOne (mappedBy = "direccion")  
7     private Cliente cliente;  
8 }
```

## Ejemplo: Ejemplo: Cliente como owner

localhost

Auto commit | Max rows: 1000 | Auto complete | Off | Auto select On | ?

jdbc:h2:~/test

CLIENTE

- ID
- APELLIDO
- EMAIL
- NOMBRE
- TELEFONO
- DIRECCION\_ID
- Indexes

DIRECCION

- ID
- CALLE
- CIUDAD
- CP
- NUMERO
- PAIS
- Indexes

INFORMATION\_SCHEMA

Sequences

Users

Run Run Selected Auto complete Clear SQL statement:

SELECT \* FROM CLIENTE;

ID	APELLIDO	EMAIL	NOMBRE	TELEFONO	DIRECCION_ID
2	Perez	jp@jp.ar	Juan	null	1

(1 row, 0 ms)

Edit

Notar la clave externa DIRECCION\_ID en CLIENTE (no en DIRECCION)

## Ejemplo: Direccion como owner

```
1 @Entity  
2 public class Cliente {  
3     @Id @GeneratedValue  
4     private Long id;  
5     ...  
6     @OneToOne(mappedBy = "cliente")  
7     private Direccion direccion;  
8     ...  
9 }
```

```
1 @Entity  
2 public class Direccion {  
3     @Id @GeneratedValue  
4     private Long id;  
5     ...  
6     @OneToOne  
7     private Cliente cliente;  
8 }
```

## Ejemplo: Direccion como owner

The screenshot shows a MySQL Workbench interface connected to a database named 'test'. The left sidebar displays the schema structure:

- CLIENTE** table:
  - ID
  - APELLIDO
  - EMAIL
  - NOMBRE
  - TELEFONO
  - Indexes
- DIRECCION** table:
  - ID
  - CALLE
  - CIUDAD
  - CP
  - NUMERO
  - PAIS
  - CLIENTE\_ID
  - Indexes
- INFORMATION\_SCHEMA**
- Sequences
- Users

The main pane shows the results of the SQL query `SELECT * FROM CLIENTE;`:

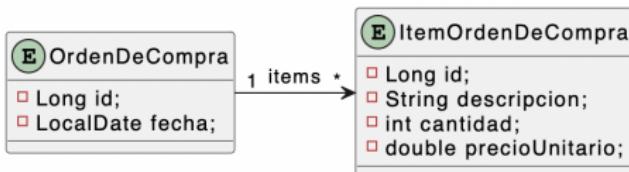
```
SELECT * FROM CLIENTE;
+----+-----+-----+-----+-----+
| ID | APELLIDO | EMAIL | NOMBRE | TELEFONO |
+----+-----+-----+-----+-----+
| 2  | Perez    | jp@jp.ar | Juan   | null     |
+----+-----+-----+-----+-----+
```

(1 row, 0 ms)

**Edit**

Notar la clave externa CLIENTE\_ID enDIRECCION (no en CLIENTE )

## Ejemplo: One-To-Many



- ▶ Una orden de compra tiene muchos artículos (item).
- ▶ La relación es unidireccional.
- ▶ Se navega de `OrdenDeCompra` a `ItemOrdenDeCompra`.
- ▶ Se dice que el `OrdenDeCompra` es el dueño de la relación.
- ▶ Implementación: `OrdenDeCompra` tiene un atributo que es una colección de `ItemOrdenDeCompra`.

## Ejemplo: One-To-Many

```
1 @Entity
2 public class ItemOrdenDeCompra {
3     @Id
4     @GeneratedValue
5     private long id;
6     private String descripcion;
7     private int cantidad;
8     private double precioUnitario;
9
10    // Constructores, getters y setters
11 }
```

## Ejemplo: One-To-Many

```
1 @Entity
2 public class OrdenDeCompra {
3     @Id
4     @GeneratedValue
5     private long id;
6     private LocalDate fecha;
7     //Implementación de la relación
8     private List<ItemOrdenDeCompra> items = new ArrayList<>();
9     // Constructores, getters y setters
10 }
```

## Ejemplo: One-To-Many

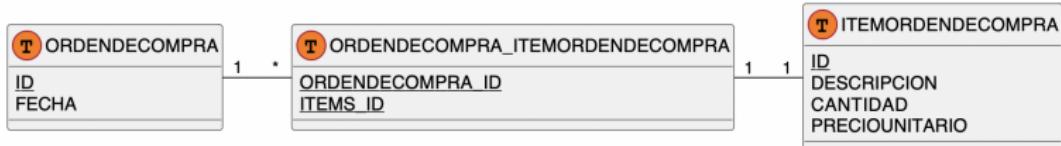
- ▶ Las dos entidades tienen las anotaciones mínimas requeridas: `@Entity`, `@Id` y `@GeneratedValue` para la clave principal.
- ▶ Un mapeo uno a muchos se establece porque la entidad `OrdenDeCompra` incluye una colección de entidades `ItemOrdenDeCompra`.
- ▶ Usamos `@OneToMany` para declarar la relación uno-a-muchos.
- ▶ El proveedor de persistencia asignará estas dos entidades a dos tablas y una tercera tabla para la relación (con pares de claves de las dos entidades).

## Ejemplo: Anotación @OneToMany

```
1  @Entity
2  public class OrdenDeCompra {
3      @Id
4      @GeneratedValue
5      private long id;
6      private LocalDate fecha;
7      @OneToMany
8      private List<ItemOrdenDeCompra> items = new ArrayList<>();
9      // Constructores, getters y setters
10 }
```

# Mapping default de @One-To-Many unidireccional

- ▶ Usa una tabla *join* para mantener la información de la relación, con dos columnas de clave externa.
  - ▶ Una columna para la clave de la tabla ORDENDECOMPRA
  - ▶ Otra columna para la clave de la tabla ITEMORDENDECOMPRA
- ▶ El nombre de la tabla join es el nombre de ambas entidades, separadas por \_: ORDENDECOMPRA\_ITEMORDENDECOMPRA.



# Ejemplo: One-To-Many

The screenshot shows a Java Swing application window titled "localhost". The window has a toolbar at the top with various icons for file operations, search, and database management. Below the toolbar is a menu bar with "File", "Edit", "View", "Tools", "Help", and "About". The main area contains a database browser tree on the left and a command editor on the right.

**Database Browser Tree:**

- jdbc:h2:~/test
  - ITEMORDENDECOMPRA
    - ID
    - CANTIDAD
    - DESCRIPCION
    - PRECIOUNITARIO
    - Indexes
  - ORDENDECOMPRA
    - ID
    - FECHA
    - Indexes
  - ORDENDECOMPRA\_ITEMORDENDECOMPRA
  - INFORMATION\_SCHEMA
    - Sequences
    - Users
  - H2 2.1.210 (2022-01-17)

**Command Editor:**

SQL statement:  
Run Run Selected Auto complete Clear SQL statement:

**Important Commands**

	Displays this Help Page
	Shows the Command History
	Ctrl+Enter Executes the current SQL statement
	Shift+Enter Executes the SQL statement defined by the text selection
	Ctrl+Space Auto complete

# Ejemplo: One-To-Many

The screenshot shows a MySQL Workbench interface with the following details:

- Left Panel (Object Browser):** Shows the database structure:
  - ITEMDENDECOMPRA:** Contains fields: ID, CANTIDAD, DESCRIPCION, PRECIOUNITARIO.
  - ORDENDECOMPRA:** Contains fields: ID, FECHA.
  - ORDENDECOMPRA\_ITEMORDENDECOMPRA:** A junction table.
  - INFORMATION\_SCHEMA:** Sequences and Users.
  - H2 2.1.210 (2022-01-17)** indicates the version of the database engine.
- Top Bar:** localhost, various icons, and dropdowns for Auto commit, Max rows: 1000, Auto complete (Off), Auto select (On), and SQL statement.
- Right Panel (Results):** Two queries are displayed:
  - SHOW COLUMNS FROM ORDENDECOMPRA\_ITEMORDENDECOMPRA;** Returns two columns: ORDENDECOMPRA\_ID and ITEMS\_ID, both of type BIGINT.
  - SHOW COLUMNS FROM ORDENDECOMPRA\_ITEMORDENDECOMPRA;** Returns two columns: FIELD, TYPE, NULL, KEY, and DEFAULT. The data is:

FIELD	TYPE	NULL	KEY	DEFAULT
ORDENDECOMPRA_ID	BIGINT	NO		NULL
ITEMS_ID	BIGINT	NO	UNI	NULL

(2 rows, 20 ms)

# Ejemplo: One-To-Many

The screenshot shows the MySQL Workbench interface with the following details:

- Left Panel (Object Browser):** Shows the database structure:
  - ITEMORDENDECOMPRA:** Contains columns ID, CANTIDAD, DESCRIPCION, PRECIOUNITARIO, and Indexes.
  - ORDENDECOMPRA:** Contains columns ID, FECHA, and Indexes.
  - ORDENDECOMPRA\_ITEMORDENDECOMPRA:** A junction table.
  - INFORMATION\_SCHEMA:** Contains Sequences and Users.
  - H2 2.1.210 (2022-01-17):** Version information.
- Top Bar:** localhost, various icons, and dropdown menus for Auto commit, Max rows: 1000, Auto complete (Off), Auto select (On), and SQL statement.
- SQL Editor:** Contains the following SQL queries and results:
  - Top Query:** SELECT \* FROM ORDENDECOMPRA;  
Result: (empty)
  - Bottom Query:** SELECT \* FROM ORDENDECOMPRA;  
Result:

ID	FECHA
3	2022-03-16

(1 row, 1 ms)  
Edit button

# Ejemplo: One-To-Many

The screenshot shows a MySQL Workbench interface with the following details:

- Toolbar:** Includes standard icons for file operations, navigation, and database management.
- Address Bar:** Shows "localhost".
- Tool Buttons:** Auto commit (checked), Max rows: 1000, Auto complete (Off), Auto select (On).
- Left Panel (Object Browser):** Lists database objects:
  - ITEMORDENDECOMPRA:** Contains columns ID, CANTIDAD, DESCRIPCION, PRECIOUNITARIO, and Indexes.
  - ORDENDECOMPRA:** Contains columns ID, FECHA, and Indexes.
  - ORDENDECOMPRA\_ITEMORDENDECOMPRA:** A junction table.
  - INFORMATION\_SCHEMA:** Sequences and Users.
  - H2 2.1.210 (2022-01-17)** (version information).
- Top Right Panel (Query Editor):** Buttons: Run, Run Selected, Auto complete, Clear, SQL statement: "SELECT \* FROM ITEMORDENDECOMPRA".
- Bottom Right Panel (Results):** Results of the query "SELECT \* FROM ITEMORDENDECOMPRA;":

ID	CANTIDAD	DESCRIPCION	PRECIOUNITARIO
1	1	yerba	405.1
2	1	café	957.35

(2 rows, 1 ms)

**Edit** button.

# Ejemplo: One-To-Many

The screenshot shows a MySQL Workbench interface with the following details:

- Left Panel (Object Browser):** Shows the database structure:
  - ITEMORDENDECOMPRA:** Contains columns ID, CANTIDAD, DESCRIPCION, PRECIOUNITARIO, and Indexes.
  - ORDENDECOMPRA:** Contains columns ID, FECHA, and Indexes.
  - ORDENDECOMPRA\_ITEMORDENDECOMPRA:** A junction table.
  - INFORMATION\_SCHEMA:** Sequences and Users.
  - H2 2.1.210 (2022-01-17)** indicates the database engine and version.
- Top Bar:** localhost, various icons, and dropdown menus for Auto commit, Max rows: 1000, Auto complete, Auto select (On), and SQL statement.
- Central Area:**
  - SQL Editor:

```
SELECT * FROM ORDENDECOMPRA_ITEMORDENDECOMPRA;
```
  - Results Grid:

ORDENDECOMPRA_ID	ITEMS_ID
3	1
3	2
  - Message: (2 rows, 6 ms)
  - Buttons: Run, Run Selected, Auto complete, Clear, SQL statement.

## Personalización de mapping default de @OneToMany

- ▶ `@JoinColumn` para cambiar las columnas de clave externa.
- ▶ `@JoinTable` para el mapeo de la tabla de unión.
- ▶ `@OneToMany` similar a `@OneToOne`.

## @JoinTable

```
1 @Target({METHOD, FIELD})
2 @Retention(RUNTIME)
3 public @interface JoinTable {
4     String name() default "";
5     String catalog() default "";
6     String schema() default "";
7     JoinColumn[] joinColumns() default {};
8     JoinColumn[] inverseJoinColumns() default {};
9     ForeignKey foreignKey() default @ForeignKey(PROPVIDER_DEFAULT);
10    ForeignKey inverseForeignKey() default
11        @ForeignKey(PROPVIDER_DEFAULT);
12    UniqueConstraint[] uniqueConstraints() default {};
13    Index[] indexes() default {};
14 }
```

## @JoinTable

- ▶ @JoinTable tiene los atributos :joinColumns e InverseJoinColumns de tipo JoinColumn[].
- ▶ Están asociados al lado propietario y el lado inverso de la relación.
- ▶ joinColumns describe al lado propietario de la relación (ORDENDECOMPRA)
- ▶ InverseJoinColumns especifica el lado inverso (el destino) de la relación ( ITEMORDENDECOMPRA)

## Ejemplo @JoinTable

```
1  @Entity
2  public class OrdenDeCompra {
3      ...
4      @OneToMany
5      @JoinTable(name = "ITEMS_EN_ORDEN",
6          joinColumns = @JoinColumn(name = "CLAVE_ORDEN"),
7          inverseJoinColumns = @JoinColumn(name = "CLAVE_ITEM")
8      )
9      private List<ItemOrdenDeCompra> items = new ArrayList<>();
10     ...
11 }
```

# Ejemplo @JoinTable

The screenshot shows a Java application window with a title bar "localhost". The main area contains a database browser and a help panel.

**Database Browser:**

- Left pane: Database connections and schema tree.
  - Connections: jdbc:h2:~/test
  - Schemas:
    - ITEMORDENDECOMPRA
    - ITEMS\_EN\_ORDEN
      - CLAVE\_ORDEN
      - CLAVE\_ITEM
      - Indexes
    - ORDENDECOMPRA
    - INFORMATION\_SCHEMA
    - Sequences
    - Users
- Right pane: SQL statement input field with buttons: Run, Run Selected, Auto complete, Clear, and a placeholder "SQL statement:".

**Help Panel:**

### Important Commands

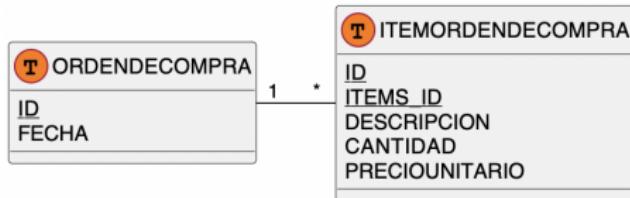
	Displays this Help Page
	Shows the Command History
	Ctrl+Enter Executes the current SQL statement
	Shift+Enter Executes the SQL statement defined by the text selection
	Ctrl+Space Auto complete

# Mapeo con clave externas

- ▶ Por default, una relación unidireccional uno-a-muchos usa una tabla *join*
- ▶ se puede cambiar para usar de claves externas usando la anotación @JoinColumn en lugar de @JoinTable

```
1 @Entity
2 public class OrdenDeCompra {
3     ....
4     @OneToMany
5     @JoinColumn
6     private List<ItemOrdenDeCompra> items = new ArrayList<>();
7     ....
8 }
```

# Mapeo con clave externas



Screenshot of the H2 Database Browser interface:

- Toolbar:** Auto commit (checked), Max rows: 1000, Auto complete (Off), Auto select (On).
- Left Panel (Schema Tree):**
  - ITEMORDENDECOMPRA**: ID, CANTIDAD, DESCRIPCION, PRECIOUNITARIO, ITEMS\_ID, Indexes.
  - ORDENDECOMPRA**: ID, FECHA, Indexes.
- SQL Statement Area:** SELECT \* FROM ITEMORDENDECOMPRA
- Result Area:** SELECT \* FROM ITEMORDENDECOMPRA;

ID	CANTIDAD	DESCRIPCION	PRECIOUNITARIO	ITEMS_ID
1	1	yerba	405.1	3
2	1	café	957.35	3

(2 rows, 24 ms)
- Bottom Buttons:** Edit, a small logo.

## @JoinColumn

- ▶ Es muy similar a @Column (No se puede usar @Column con @OneToOne)
- ▶ Se utiliza para personalizar la columna join (clave foránea del lado propietario)

```
1 @Target({METHOD, FIELD})
2 @Retention(RUNTIME)
3 @Repeatable(JoinColumns.class)
4 public @interface JoinColumn {
5     String name() default "";
6     String referencedColumnName() default "";
7     boolean unique() default false;
8     boolean nullable() default true;
9     boolean insertable() default true;
10    boolean updatable() default true;
11    String columnDefinition() default "";
12    String table() default "";
13    ForeignKey foreignKey() default @ForeignKey(
14        PROVIDER_DEFAULT);}
```

## Ejemplo: @JoinColumn

```
1  @Entity
2  public class OrdenDeCompra {
3      ....
4      @OneToMany
5      @JoinColumn (name ="orden")
6      private Set<ItemOrdenDeCompra> items = new HashSet<>();
7      ....
8  }
```

# Ejemplo: Mapeo personalizado con clave externas

The screenshot shows a MySQL Workbench window with the following details:

- Connection:** jdbc:h2:~/test
- Schemas:** ITEMORDENDECOMPRA, ORDENDECOMPRA, INFORMATION\_SCHEMA, Sequences, Users.
- Current Schema:** ITEMORDENDECOMPRA
- Toolbar Buttons:** Auto commit (checked), Max rows: 1000, Auto complete (Off), Auto select (On).
- SQL Statement:** SELECT \* FROM ITEMORDENDECOMPRA
- Result Area:** Displays the query results in a table format.

ID	CANTIDAD	DESCRIPCION	PRECIOUNITARIO	ORDEN
1	1	yerba	405.1	3
2	1	café	957.35	3

(2 rows, 1 ms)

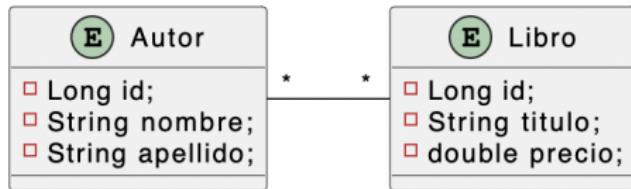
**Edit**

## Relaciones uno-a-muchos bidireccionales

- ▶ @ManyToOne define una asociación de un solo valor a otra clase de entidad que tiene una multiplicidad de muchos a uno.
- ▶ Normalmente, no es necesario especificar la entidad de destino explícitamente, ya que normalmente se puede inferir del tipo de objeto al que se hace referencia.

# Relaciones muchos-a-muchos

- ▶ Una relación bidireccional de muchos a muchos existe cuando un objeto origen se refiere a muchos destinos y cuando un destino se refiere a muchos orígenes.
- ▶ Varios Autores escriben un libro y varios libros pueden tener un mismo autor.
- ▶ Como objetos, cada entidad tendrá una colección de entidades del otro tipo.
- ▶ En el mundo relacional, una relación de muchos a muchos se mapea en una tabla de join



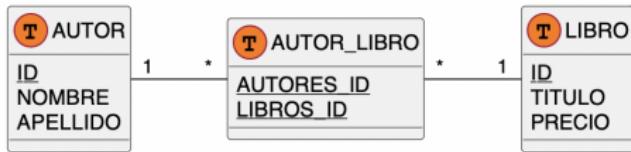
# Relaciones muchos-a-muchos

```
1 @Entity  
2 public class Autor {  
3     @Id @GeneratedValue  
4     private int id;  
5     ...  
6     @ManyToMany  
7     private Set<Libro> libros = new HashSet<>();  
8     ....  
9 }
```

```
1 @Entity  
2 public class Libro {  
3     @Id @GeneratedValue  
4     private int id;  
5     ...  
6     @ManyToMany(mappedBy = "libros")  
7     private Set<Autor> autores = new HashSet<>();  
8     ....  
9 }
```

- ▶ En principio se puede utilizar cualquier colección.
- ▶ Algunos proveedores (ej, Hibernate) generan código ineficiente con Listas (borran todas las asociaciones del

# Mapeo con clave externas



Screenshot of a database management tool interface showing the results of a query against the AUTOR\_LIBRO table.

The interface includes:

- Toolbar with standard window controls, a connection to "localhost", and various configuration buttons like "Auto commit", "Max rows: 1000", "Auto complete Off", "Auto select On", etc.
- Left sidebar showing the database schema structure:

  - jdbc:h2:~/test**: Root connection.
  - AUTOR**: Contains columns ID, APELLIDO, NOMBRE, and an Indexes section.
  - AUTOR\_LIBRO**: Contains columns AUTORES\_ID, LIBROS\_ID, and an Indexes section.
  - LIBRO**: Contains columns ID, PRECIO, TITULO, and an Indexes section.
  - INFORMATION\_SCHEMA**: Contains Sequences and Users sections.
  - H2 2.1.210 (2022-01-17)**: Connection details.

- SQL statement input field: `SELECT * FROM AUTOR_LIBRO;`
- Result grid showing the data from the query:

AUTORES_ID	LIBROS_ID
1	3
1	4
2	3
2	4

- Text below the grid: `(4 rows, 2 ms)`

## mappedBy

- ▶ una relación bidireccional de muchos a muchos y de uno a uno, cualquier lado puede designarse como el propietario.
- ▶ No importa qué lado se designe como propietario, el otro lado debe incluir el elemento mappedBy.
- ▶ Si no, el proveedor tratará como dos relaciones unidireccionales separadas.
- ▶ No es válido tener mappedBy en ambos lados de la relación.

## Relaciones bidireccionales

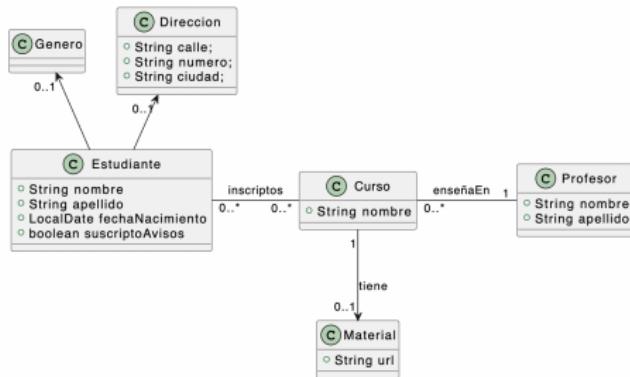
- ▶ Las siguientes reglas se aplican a las relaciones bidireccionales:
  - ▶ El lado inverso de una relación bidireccional debe hacer referencia a su lado propietario mediante el uso del elemento `mappedBy` de la anotación `@OneToOne`, `@OneToMany` o `@ManyToMany`
  - ▶ El elemento `mappedBy` designa la propiedad o campo en la entidad que es propietaria de la relación.
  - ▶ El lado **muchos** de las relaciones bidireccionales uno-a-muchos/muchos-a-uno debe ser el lado propietario
  - ▶ por lo tanto, el elemento `@mappedBy` no se puede especificar en la anotación `@ManyToOne`.
  - ▶ Para relaciones bidireccionales uno-a-uno, el lado propietario corresponde al lado que contiene la clave foránea correspondiente.
  - ▶ Para relaciones bidireccionales de muchos-a-muchos, cualquiera de los lados puede ser el lado propietario.

## Obtener relaciones de la BD

- ▶ Las anotaciones para relaciones @OneToOne, @OneToMany, @ManyToOne y @ManyToMany tienen el atributo `fetching` que puede ser `eager` o `Lazy`
- ▶ indica cómo se cargarán los objetos asociados.
- ▶ Tiene impacto en la performance:
  - ▶ `lazy`: cuando se accede a la relación
  - ▶ `eager`: cuando la entidad se lee inicialmente.

# Ejercicio

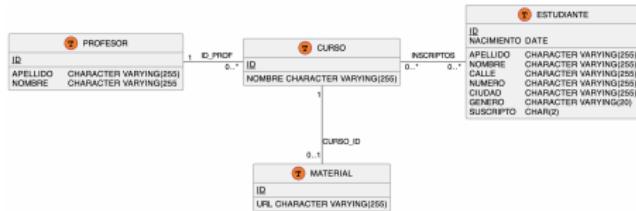
Considerar el siguiente diagrama UML



Ver código

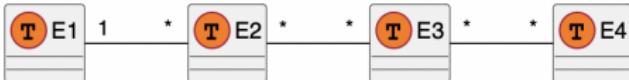
# Ejercicio

Se desea mapear al siguiente esquema relacional



Definir el mapping correspondiente.

# Obtener relaciones de la BD



- ▶ Todas EAGER: tan pronto como cargue E1 todos los objetos dependientes se cargan automáticamente en la memoria.
- ▶ Todas LAZY: Si carga una entidad E1, sólo se cargan sus atributos.
- ▶ Cuando se accede explícitamente a E1 (`e1.getE2()`) se cargan los asociados de tipo E2 (y así sucesivamente).
- ▶ Si se desea manipular todo el gráfico de objetos, se cargan sucesivamente.
- ▶ EAGER traerá todos los datos en la memoria usando una pequeña cantidad de acceso a la base de datos (joins)
- ▶ LAZY evita cargar en memoria datos, pero hay que acceder a la base cada vez que se necesite.
- ▶ La elección depende de la aplicación (si siempre que se acceda a una entidad se necesita acceder a las relacionadas, entonces EAGER)

## Valores por default de fetch

@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

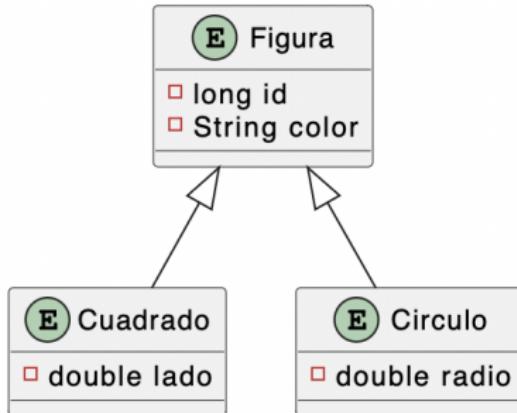
## Mapeo de herencia

- ▶ La herencia es un concepto completamente desconocido y no implementado de forma nativa en un mundo relacional.
- ▶ El concepto de herencia presenta variantes cuando se guardan objetos en una base de datos relacional.
- ▶ Varias alternativas para achatar la jerarquía.

## Mapeo de herencia: Estrategias

- ▶ *tabla única por jerarquía*: la suma de todos los atributos (propios y heredados) se mapean a una clase (esta es la estrategia por default).
- ▶ *subclases unidas*: cada entidad en la jerarquía, concreta o abstracta, se asigna a su propia tabla dedicada.
- ▶ *tabla por clase concreta*: cada entidad concreta en la jerarquía se mapea a su propia tabla.

# Ejemplo



- ▶ La entidad **Figura** es la entidad raíz y tiene una id y un color.
- ▶ Las entidades **Cuadrado** y **Circulo**
  - ▶ heredan esto atributos de **Figura**.
  - ▶ agregan atributos adicionales.

## Estrategia: tabla única por jerarquía

- ▶ Es la estrategia de mapeo por default.
- ▶ todas las entidades en la jerarquía se mapean a una misma tabla.
- ▶ Como es la estrategia por defecto,
  - ▶ basta con anotar a las clases como entidades,
  - ▶ y notar que se hereda la identidad (atributo con anotación @Id).

# Ejemplo

```
1 @Entity  
2 public class Figura {  
3     @Id @GeneratedValue  
4     private long id;  
5     private String color;  
6     ...  
7 }
```

```
1 @Entity  
2 public class Cuadrado extends Figura {  
3     private long lado;  
4     ...  
5 }
```

```
1 @Entity  
2 public class Circulo extends Figura{  
3     private int radio;  
4     ...  
5 }
```

## Ejemplo: tabla única por jerarquía

- ▶ Sin herencia, las tres entidades se mapearían en sus propias tablas separadas (pero todas necesitarían la anotación @Id).
- ▶ Con la estrategia de una sola tabla por jerarquía:
  - ▶ todas terminan en la misma tabla de la base de datos.
  - ▶ el nombre por defecto es el nombre de la clase en la raíz de la jerarquía: FIGURA.

# Ejemplo

The screenshot shows a MySQL Workbench interface running on a Mac OS X system. The title bar says "localhost". The toolbar includes standard icons for file operations and a "Run" button. The connection is set to "Auto commit" and "Max rows: 1000". The "Auto complete" and "Auto select" options are also present.

The left sidebar shows the database structure:

- test (selected)
- FIGURA (table)
  - DTYPE
  - ID
  - COLOR
  - RADIO
  - LADO
  - Indexes
- INFORMATION\_SCHEMA
- Sequences
- Users
- H2 2.1.210 (2022-01-17)

The main pane displays the results of the SQL query "SELECT \* FROM FIGURA".

```
SELECT * FROM FIGURA;
```

DTYPE	ID	COLOR	RADIO	LADO
Figura	1	rojo	null	null
Cuadrado	2	azul	null	10
Circulo	3	amarillo	15	null

(3 rows, 6 ms)

An "Edit" button is located at the bottom of the results pane.

## Ejemplo: tabla única por jerarquía

- ▶ La tabla FIGURA tiene todos los atributos de las entidades Figura, Cuadrado, Circulo.
- ▶ la columna adicional DTYPE:
  - ▶ no se relaciona con ninguno de los atributos de las entidades.
  - ▶ es la columna discriminadora.
  - ▶ le permite al proveedor de persistencia necesita saber qué entidad representa cada fila.
  - ▶ el proveedor instanciará el tipo de objeto apropiado (Figura, Cuadrado o Circulo) cuando lea la tabla.

## Ejemplo: tabla única por jerarquía

- ▶ La tabla FIGURA tiene todos los atributos de las entidades Figura, Cuadrado, Circulo.
- ▶ la columna adicional DTYPPE:
  - ▶ no se relaciona con ninguno de los atributos de las entidades.
  - ▶ es la columna discriminadora.
  - ▶ le permite al proveedor de persistencia necesitar saber qué entidad representa cada fila.
  - ▶ el proveedor instanciará el tipo de objeto apropiado (Figura, Cuadrado o Circulo) cuando lea la tabla.
- ▶ Desventaja: no todas las columnas son significativas para cada entidad.

# Discriminador

- ▶ Por default, la columna discriminador:
  - ▶ se denomina DTTYPE,
  - ▶ es de tipo String (mapeada a VARCHAR(31)),
  - ▶ el valor es el nombre de la entidad.
- ▶ el nombre y tipo por default se pueden modificar con @DiscriminatorColumn
- ▶ el valor de esta columna se puede modificar con @DiscriminatorValue anotación.

## @DiscriminatorColumn

```
1 @Target({TYPE})
2 @Retention(RUNTIME)
3 public @interface DiscriminatorColumn {
4     String name() default "DTYPE";
5     DiscriminatorType discriminatorType() default STRING;
6     String columnDefinition() default "";
7     int length() default 31;
8 }
9
10 public enum DiscriminatorType { STRING, CHAR, INTEGER };
```

# Ejemplo

```
1 @Entity
2 @DiscriminatorColumn(name = "disc", discriminatorType =
3     DiscriminatorType.INTEGER)
4 public class Figura {
5     @Id @GeneratedValue
6     private long id;
7     private String color;
8 }
```

# Ejemplo

The screenshot shows the H2 Console interface running on localhost. The left sidebar displays the database schema:

- jdbc:h2:~/test
- FIGURA
  - DISC
  - INTEGER
  - ID
  - COLOR
  - RADIO
  - LADO
  - Indexes
- INFORMATION\_SCHEMA
- Sequences
- Users

The right pane shows the results of the following SQL query:

```
SELECT * FROM FIGURA;
```

DISC	ID	COLOR	RADIO	LADO
2104194816	1	rojo	null	null
140796079	2	azul	null	10
-1847355359	3	amarillo	15	null

(3 rows, 1 ms)

(poco significativos)

Los valores al discriminador son asignados automáticamente

## @DiscriminatorValue

- ▶ Se usa para especificar el valor de la columna discriminadora para entidades del tipo dado.
- ▶ Solo se puede especificar en una clase de entidad concreta.
- ▶ Si no se especifica la anotación DiscriminatorValue se utilizará una función específica del proveedor para generar un valor que represente el tipo de entidad.
- ▶ El valor del discriminador, si no está predeterminado, debe especificarse para cada clase de entidad en la jerarquía.
- ▶ El valor debe ser consistente en tipo con el tipo de discriminador (Si es INTEGER debe poder convertirse a un valor entero)

```
1 @Target({TYPE})  
2 @Retention(RUNTIME)  
3 public @interface DiscriminatorValue {  
4     String value();  
5 }
```

# Ejemplo

```
1 @Entity  
2 @DiscriminatorColumn(name = "disc", discriminatorType =  
3 DiscriminatorType.INTEGER)  
4 @DiscriminatorValue(value = "1")  
5 public class Figura {  
6     ...  
}
```

```
1 @Entity  
2 @DiscriminatorValue(value = "2")  
3 public class Cuadrado extends Figura {  
4     ...  
5 }
```

```
1 @Entity  
2 @DiscriminatorValue(value = "3")  
3 public class Circulo extends Figura{  
4     ...  
5 }
```

# Ejemplo

The screenshot shows the H2 Console application running in a web browser on a Mac OS X system. The title bar indicates the connection is to 'localhost'. The main interface has a sidebar on the left listing database objects: 'jdbc:h2:~/test', 'FIGURA', 'INFORMATION\_SCHEMA', 'Sequences', 'Users', and a note about the version 'H2 2.1.210 (2022-01-17)'. The central area contains a toolbar with buttons for 'Run', 'Run Selected', 'Auto complete', 'Clear', and 'SQL statement'. Below the toolbar is a text input field containing the SQL query 'SELECT \* FROM FIGURA'. At the bottom of the main window, the results of the query are displayed in a table:

DISC	ID	COLOR	RADIO	LADO
1	1	rojo	null	null
2	2	azul	null	10
3	3	amarillo	15	null

The status bar at the bottom right shows the text 'epidata displaying 3 rows'.

```
localhost
H2 Console
DiscriminatorColumn (javax.persistence-api 2.2 API)
Auto commit Max rows: 1000 Auto complete Off Auto select On
jdbc:h2:~/test
FIGURA
INFORMATION_SCHEMA
Sequences
Users
H2 2.1.210 (2022-01-17)

Run Run Selected Auto complete Clear SQL statement:
SELECT * FROM FIGURA

SELECT * FROM FIGURA;
+---+---+---+---+---+
| DISC | ID | COLOR | RADIO | LADO |
+---+---+---+---+---+
| 1 | 1 | rojo | null | null |
| 2 | 2 | azul | null | 10 |
| 3 | 3 | amarillo | 15 | null |
+---+---+---+---+---+
(3 rows, 3 ms)
```

## @DiscriminatorValue

- ▶ Es importante que los valores sean distintos para entidades distintas en la jerarquía.
- ▶ No es obligatorio especificarlo en todas.
- ▶ Si se usa CHAR, Hibernate lanza una excepción si no se especifica @DiscriminatorValue.

```
1 org.hibernate.AnnotationException: Using default
   @DiscriminatorValue for a discriminator of type CHAR is not
   safe
```

## Estrategia de tabla única

- ▶ La estrategia de tabla única, es la más fácil de entender
- ▶ Funciona bien cuando la jerarquía es relativamente simple y estable.
- ▶ Agregar nuevas entidades a la jerarquía o agregar atributos a entidades existentes implica agregar nuevas columnas a la tabla, migrar datos y cambiar índices.
- ▶ Esta estrategia también requiere que las columnas de las entidades secundarias puedan ser nula.

# Ejemplo

```
1 @Entity  
2 public class Circulo extends Figura{  
3     @Column(nullable = false)  
4     private int radio;
```

# Ejemplo

The screenshot shows the H2 Console interface running on localhost. The left sidebar lists database connections and schemas: jdbc:h2:~/test, FIGURA, INFORMATION\_SCHEMA, Sequences, Users, and version information (H2 2.1.210 (2022-01-17)). The main area contains a toolbar with various icons and dropdowns for connection settings. Below the toolbar is a SQL statement input field containing "SHOW COLUMNS FROM FIGURA". A button bar above the results includes "Run", "Run Selected", "Auto complete", "Clear", and "SQL statement". The results section displays a table of column definitions:

FIELD	TYPE	NULL	KEY	DEFAULT
DTYPE	CHARACTER VARYING(31)	NO		NULL
ID	BIGINT	NO	PRI	NULL
COLOR	CHARACTER VARYING(255)	YES		NULL
RADIO	INTEGER	NO		NULL
LADO	BIGINT	YES		NULL

(5 rows, 6 ms)

- ▶ Notar que RADIO no puede ser nulo.
- ▶ Excepción al persistir una instancia de Cuadrado

## Estrategia de unión para subclases

- ▶ Cada entidad de la jerarquía se mapea a su propia tabla.
- ▶ La entidad raíz se asigna a una tabla que
  - ▶ define la clave principal que utilizarán todas las tablas de la jerarquía,
- ▶ Cada subclase está representada por una tabla separada que contiene:
  - ▶ sus propios atributos (no los heredados de la clase raíz).
  - ▶ y una clave principal que se refiere a la clave primaria de la tabla raíz.
- ▶ Las tablas no raíz no contienen una columna discriminadora.
- ▶ Usa `@Inheritance(strategy = InheritanceType.JOINED)` en la raíz de la jerarquía.

# Ejemplo

```
1 @Entity  
2 @Inheritance(strategy = InheritanceType.JOINED)  
3 public class Figura {  
4     @Id @GeneratedValue  
5     private long id;  
6     private String color;  
7     ...  
8 }
```



# Ejemplo

The screenshot shows the H2 Console application running in a web browser on localhost. The interface includes a toolbar with various icons, a top navigation bar with tabs for 'localhost' and 'DiscriminatorColumn (javax.persistence-api 2.2 API)', and a main area divided into two panes.

**Left Pane:** A tree view of the database schema:

- jdbc:h2:~/test
  - CIRCULO
    - RADIO
    - ID
    - Indexes
  - CUADRADO
    - LADO
    - ID
    - Indexes
  - FIGURA
    - ID
    - COLOR
    - Indexes
- INFORMATION\_SCHEMA
- Sequences
- Users

H2 2.1.210 (2022-01-17)

**Right Pane:** A large text input field labeled 'SQL statement:' with several buttons above it: Run, Run Selected, Auto complete, Clear, and SQL statement. Below the input field is a section titled 'Important Commands' with a table:

Icon	Description
?	Displays this Help Page
!	Shows the Command History
Ctrl+Enter	Executes the current SQL statement
Shift+Enter	Executes the SQL statement defined by the text selection

# Ejemplo

The screenshot shows the H2 Console interface running on localhost. The left sidebar displays database schema objects:

- CIRCULO**: Contains **RADIO**, **ID**, and **Indexes**.
- CUADRADO**: Contains **LADO**, **ID**, and **Indexes**.
- FIGURA**: Contains **ID**, **COLOR**, and **Indexes**.
- INFORMATION\_SCHEMA**: Contains **Sequences** and **Users**.
- H2 2.1.210 (2022-01-17)**

The main area shows the results of two SQL queries:

```
SELECT * FROM FIGURA;
```

ID	COLOR
1	rojo
2	azul
3	amarillo

(3 rows, 0 ms)

# Ejemplo

The screenshot shows the H2 Console interface running on localhost. The left sidebar displays the database schema with tables CIRCULO, CUADRADO, and FIGURA, along with the INFORMATION\_SCHEMA. The right panel shows the results of a SQL query.

**Toolbar:** Auto commit checked, Max rows: 1000, Auto complete Off, Auto select On.

**SQL Statement:** SELECT \* FROM CIRCULO;

**Result:**

RADIO	ID
15	3

(1 row, 1 ms)

**Buttons:** Run, Run Selected, Auto complete, Clear, SQL statement, Edit.

**Information:** H2 2.1.210 (2022-01-17)

# Ejemplo

The screenshot shows the H2 Console interface running on localhost. The left sidebar displays the database schema:

- jdbc:h2:~/test
- CIRCULO
- CUADRADO
- FIGURA
- INFORMATION\_SCHEMA
- Sequences
- Users

The right pane shows the results of two SQL queries:

1. The first query is `SELECT * FROM CUADRADO`. The results are displayed in a table:

LADO	ID
10	2

(1 row, 9 ms)

2. The second query is `SELECT * FROM CUADRADO;` which returns the same result as the first query.

# Personalización

- ▶ Se pueden usar las anotaciones @DiscriminatorColumn y @DiscriminatorValue para personalizar la columna y los valores del discriminador.

## Estrategia de unión

- ▶ Intuitiva y se acerca al mecanismo de herencia de objetos.
- ▶ Las consultas pueden tener un impacto en el rendimiento.
- ▶ Para cargar una instancia instancia de una subclase se necesita unir con la tabla de la clase raíz.
- ▶ Cuanto más profunda sea la jerarquía, más uniones se necesitarán para ensamblar una entidad.
- ▶ Problemas de performance para jerarquías extensas.
- ▶ Proporciona un buen soporte para las relaciones polimórficas.
- ▶ Pero requiere que se realicen una o más operaciones de combinación al crear instancias de subclases de entidad.

## Estrategia: Tabla por clase

- ▶ Cada entidad se mapea a su propia tabla.
- ▶ Todos los atributos heredados se mapearán como columnas de la tabla de entidades.
- ▶ Esta estrategia desnормaliza el modelo y hace que todos los atributos de la entidad raíz se redefinan en las tablas de todas las entidades que heredan de esta
- ▶ No hay tabla compartida, columnas compartidas ni columna discriminadora.

# Ejemplo

```
1 @Entity
2 @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
3 public class Figura {
4     @Id @GeneratedValue
5     private long id;
6     private String color;
7     ...
8 }
```

# Ejemplo

The screenshot shows the H2 Console application running in a web browser on localhost. The interface includes a toolbar with various icons, a top navigation bar with tabs and settings, and a main area displaying a database schema and command history.

**Toolbar:** Includes standard window controls (red, yellow, green buttons), back/forward arrows, a refresh icon, and a search bar with placeholder "localhost".

**Top Bar:** Shows the title "H2 Console" and a tab labeled "DiscriminatorColumn (javax.persistence-api 2.2 API)". It also displays a count of 1 and the status "DiscriminatorColumn (javax.persistence-api 2.2 API)".

**Database Structure:** On the left, a tree view shows the database structure:

- Root: jdbc:h2:~/test
  - CIRCULO
    - ID
    - COLOR
    - RADIO
    - Indexes
  - CUADRADO
    - ID
    - COLOR
    - LADO
    - Indexes
  - FIGURA
    - ID
    - COLOR
    - Indexes
  - INFORMATION\_SCHEMA
    - Sequences
    - Users

**Command Area:** The central area contains a large text input field for SQL statements. Above it are several buttons: Run, Run Selected, Auto complete, Clear, and SQL statement:.

**Important Commands:** A table below lists key commands:

Icon	Description
?	Displays this Help Page
!	Shows the Command History
Ctrl+Enter	Executes the current SQL statement
Shift+Enter	Executes the SQL statement defined by the text selection

# Ejemplo

The screenshot shows the H2 Console interface running on localhost. The left sidebar displays the database schema:

- jdbc:h2:~/test
- CIRCULO
  - ID
  - COLOR
  - RADIO
  - Indexes
- CUADRADO
  - ID
  - COLOR
  - LADO
  - Indexes
- FIGURA
  - ID
  - COLOR
  - Indexes
- INFORMATION\_SCHEMA
- Sequences
- Users

The main area shows the results of two SQL queries:

```
SELECT * FROM FIGURA;
```

ID	COLOR
1	rojo

(1 row, 12 ms)

Below the table is an "Edit" button.

# Ejemplo

The screenshot shows the H2 Console interface running on localhost. The left sidebar displays the database schema:

- CIRCULO**: ID, COLOR, RADIO, Indexes
- CUADRADO**: ID, COLOR, LADO, Indexes
- FIGURA**: ID, COLOR, LADO, Indexes
- INFORMATION\_SCHEMA**
- Sequences**
- Users**

The main area contains the following content:

SQL statement: `SELECT * FROM CUADRADO`

Result of the query:

```
SELECT * FROM CUADRADO;
+----+-----+-----+
| ID | COLOR | LADO |
+----+-----+-----+
| 2  | azul  | 10   |
+----+-----+-----+
(1 row, 4 ms)
```

Buttons: Run, Run Selected, Auto complete, Clear, SQL statement.

# Ejemplo

The screenshot shows the H2 Console interface running on localhost. The left sidebar displays the database schema:

- CIRCULO**: ID, COLOR, RADIO, Indexes
- CUADRADO**: ID, COLOR, LADO, Indexes
- FIGURA**: ID, COLOR, Radio, Indexes
- INFORMATION\_SCHEMA**
- Sequences**
- Users**

The main area shows the results of two SQL queries:

```
SELECT * FROM CIRCULO;
```

ID	COLOR	RADIO
3	amarillo	15

(1 row, 1 ms)

**Edit**

## Estrategia: tabla por clases concreta

- ▶ Funciona bien cuando se consultan instancias de una entidad, ya que la consulta se limita a una sola tabla.
- ▶ La desventaja es que realizar consultas polimórficas (por ejemplo, todos las figuras) es más costoso;
  - ▶ Debe consultar todas las tablas de subclase usando la operación UNION operación, que es costosa cuando se trata de una gran cantidad de datos.
  - ▶ El soporte para la estrategia de tabla por clase concreta es opcional en JPA 2.2.

# Tipos de clases y Herencia

- ▶ Hasta ahora consideramos jerarquías con clases concretas donde todos son entidades.
- ▶ En general, una jerarquía de clases puede mezclar todo tipo de clases diferentes: entidades y también no entidades (o clases transitorias), entidades abstractas y superclases mapeadas.
- ▶ Heredar de diferentes tipos de clases se refleja en el mapeo.

## Entidad abstracta

- ▶ Anotamos con @Entity a una clase abstracta
- ▶ Una entidad abstracta se diferencia de una entidad concreta solo en que no se puede instanciar directamente con la palabra clave **new**.
- ▶ Proporciona un estructura de datos común para sus entidades hijas.
- ▶ para el proveedor de persistencia, las entidades abstractas son como entidades concretas pero no se pueden instanciar.
- ▶ se pueden aplicar las distintas estrategias de mapeo.
- ▶ Las entidades abstractas se pueden consultar al igual que las entidades concretas.

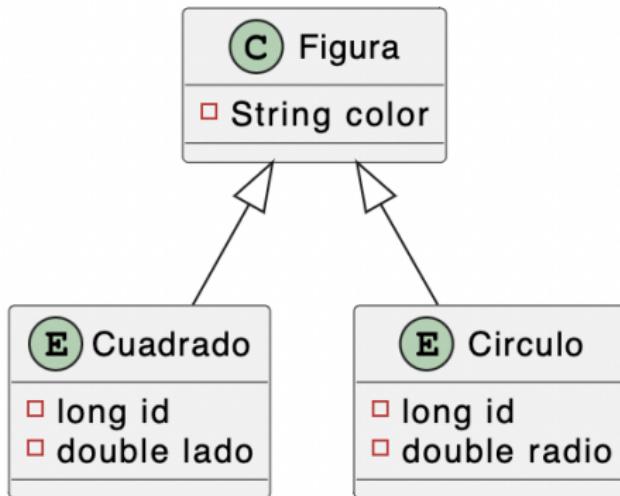
# Estrategia de mapeo y entidades abstractas

- ▶ SINGLE\_TABLE y JOINED: las tablas que se generan son idénticas al caso de las entidades concretas.
- ▶ TABLE\_PER\_CLASS: No se genera una tabla si la entidad es una superclase abstracta (para el resto, funciona igual). Una hoja abstracta genera tabla.

## Clases transient (No entidades) o POJOs

- ▶ Una clase que no es entidad es una clase sin anotación `@Entity`
- ▶ Una entidad puede heredar de una clase no entidad.
- ▶ El estado de una superclase que no es una entidad no es persistente porque no es administrado por el proveedor de persistencia.

# Ejemplo



los atributos en **Figura** son transitorios (transient)

# Ejemplo

```
1 public class Figura {  
2     private String color;  
3     ...  
4 }
```

```
1 @Entity  
2 public class Circulo extends Figura{  
3     @Id @GeneratedValue  
4     private long id;  
5     private int radio;  
6     ...  
7 }
```

```
1 @Entity  
2 public class Cuadrado extends Figura {  
3     @Id @GeneratedValue  
4     private long id;  
5     private long lado;  
6     ...  
7 }
```

# Ejemplo



los atributos en Figura son transitorios (transient)  
No hay diferencia si Figura es abstracta o concreta

## Superclases mapeadas

- ▶ Son clases que no son entidades
- ▶ pero permiten mapear información en las entidades que heredan.
- ▶ No son administrados por el proveedor de persistencia (sin tabla).
- ▶ Proporcionan propiedades persistentes a cualquier entidad que se extienda desde ellas.
- ▶ Se usa la anotación @MappedSuperclass.
- ▶ Pueden ser clases abstractas o concretas.

# Ejemplo

```
1 @MappedSuperclass  
2 public class Figura {  
3     @Id @GeneratedValue  
4     private long id;  
5     private String color;  
6     ...  
7 }
```

```
1 @Entity  
2 public class Cuadrado extends Figura {  
3     private long lado;  
4     ...  
5 }
```

```
1 @Entity  
2 public class Circulo extends Figura{  
3     private int radio;  
4     ...  
5 }
```

# Ejemplo

The screenshot shows the H2 Database browser interface. The left sidebar displays the database schema:

- jdbc:h2:~/test
- CIRCULO (with columns ID, COLOR, RADIO, and Indexes)
- CUADRADO (with columns ID, COLOR, LADO, and Indexes)
- INFORMATION\_SCHEMA
- Sequences
- Users
- H2 2.1.210 (2022-01-17)

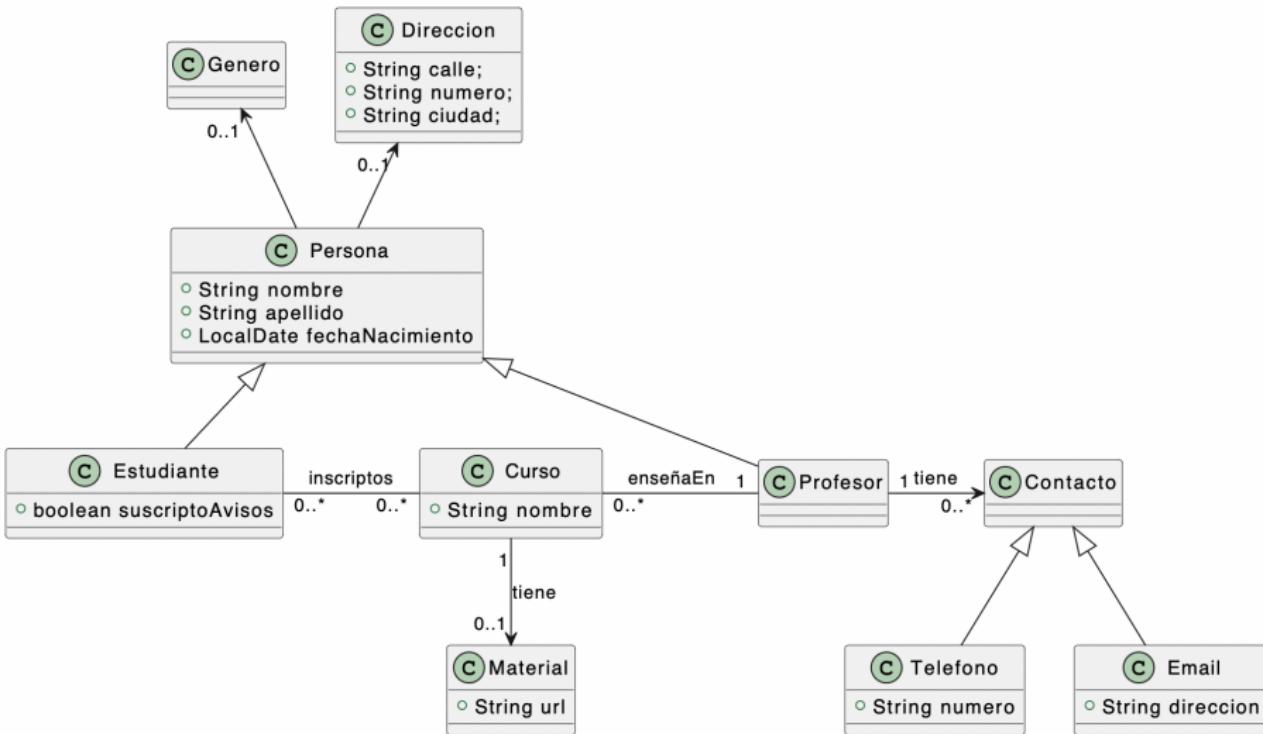
The main panel shows the CIRCULO and CUADRADO tables. Below the tables, a "Important Commands" section provides keyboard shortcuts:

Command	Description
?	Displays this Help Page
Ctrl+Enter	Shows the Command History
Ctrl+Enter	Executes the current SQL statement
Shift+Enter	Executes the SQL statement defined by the text selection
Ctrl+Space	Auto complete

No se genera tabla para Figura, cualquiera sea la estrategia

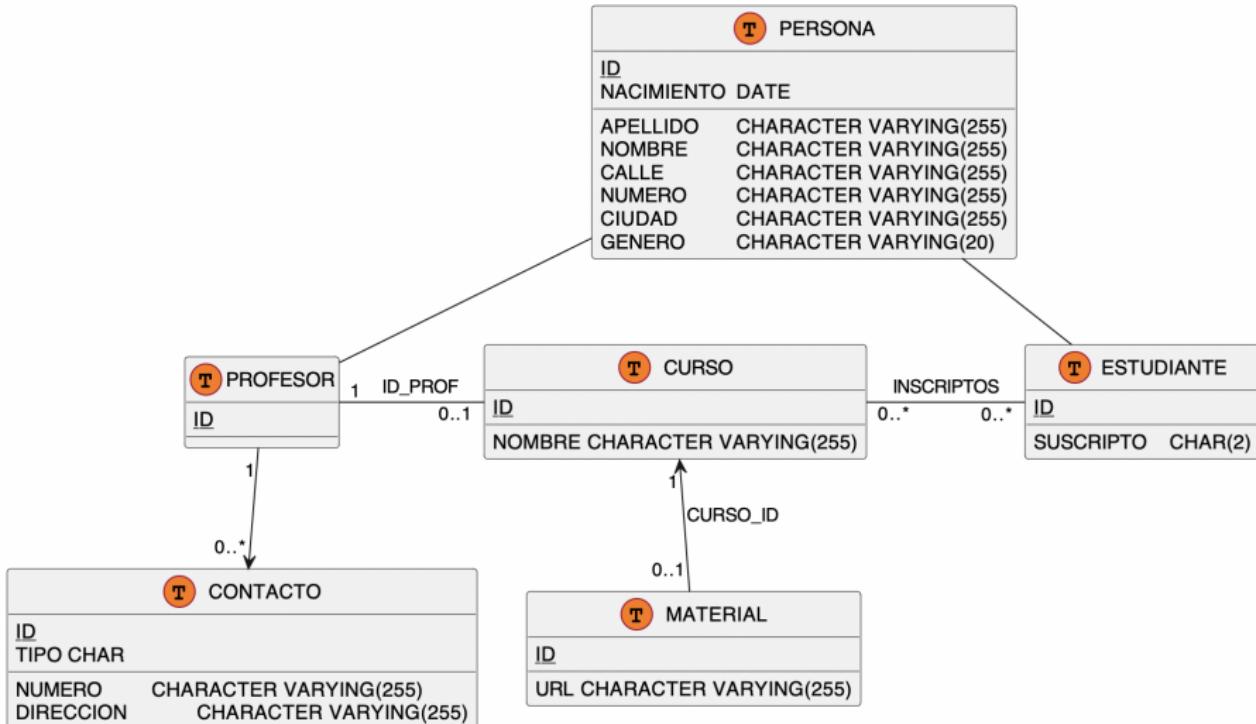
# Ejercicio

Considerar el siguiente modelo



# Ejercicio

Se desea mapear a



# Manejo de Entidades

- ▶ JPA tiene dos patas:
  - ▶ la capacidad de asignar objetos a una base de datos relacional.
  - ▶ Consultar estos objetos mapeados.
- ▶ El servicio para manipular instancias de entidades está centralizado en el administrador de entidades (`EntityManager`):
  - ▶ Proporciona operaciones para crear, buscar, eliminar y sincronizar objetos con la base de datos.
  - ▶ Permite ejecutar queries usando JPQL (Java Persistence Query Language).
  - ▶ Provee mecanismos de bloqueo.

# API de administración de entidades JPA

- ▶ las principales interfaces que se encuentran en tres paquetes diferentes.
  - ▶ el paquete `javax.persistence` (raiz) se encuentran operaciones para obtener un EntityManager, transacciones y consultas.
  - ▶ El subpaquete `javax.persistence.criteria` alberga las interfaces para conconsultas con criterios,
  - ▶ subpaquete `javax.persistence.metamodel` está relacionado con el metamodelo.

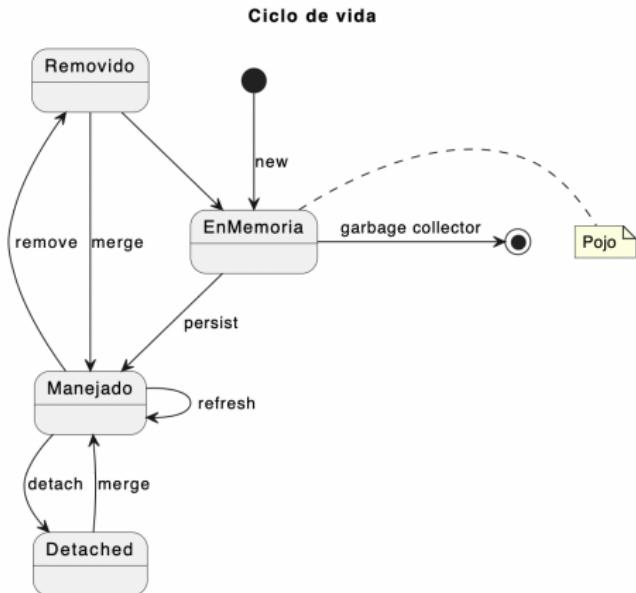
- ▶ Administra
  - ▶ el estado y el ciclo de vida de las entidades
    - ▶ es responsable de crear y eliminar instancias de entidades persistentes
    - ▶ encontrar entidades por su clave principal.
    - ▶ bloquear entidades para protegerlas contra el acceso concurrente.
  - ▶ la consulta de entidades dentro de un contexto de persistencia
    - ▶ usar consultas JPQL para recuperar entidades siguiendo ciertos criterios

- ▶ Cuando el administrador de una entidad obtiene una referencia a una entidad, se dice que la entidad está "administrada" (*managed*).
- ▶ Antes, la entidad se ve como un POJO normal (*detached*).
- ▶ La fortaleza de JPA es que las entidades pueden ser utilizadas como objetos regulares por diferentes capas de una aplicación y ser administradas por el administrador de entidades cuando necesita cargar o insertar datos en la base de datos.
- ▶ Cuando se administra una entidad, puede realizar operaciones de persistencia y el administrador de la entidad sincronizará automáticamente el estado de la entidad con la base de datos.
- ▶ Cuando la entidad se desconecta (es decir, no se administra), vuelve a un POJO, y puede usarse sin sincronizar su estado con la base de datos.

## EntityManager

- ▶ `javax.persistence.EntityManager` proporciona la API para manipular entidades.
- ▶ Es una interfaz implementada por un proveedor de persistencia que generará y ejecutará sentencias SQL.
- ▶ Al manipular entidades individuales, EntityManager puede verse como un objeto de acceso a datos (DAO) genérico, que permite operaciones CRUD en cualquier entidad.

# Ciclo de vida



## Persistir una entidad

- ▶ Persistir una entidad significa insertar datos en la base de datos cuando estos aún no existen.
- ▶ Para hacerlo, es necesario
  - ▶ crear una nueva instancia de entidad utilizando el operador **new**.
  - ▶ establecer los valores de los atributos.
  - ▶ enlazar una entidad con otra cuando hay asociaciones.
  - ▶ y finalmente llamar al `EntityManager.persist()`.

# Ejemplo

```
1 Direccion d = new Direccion("Cabildo", "5030", "CABA", "1424", "Argentina");
2 Cliente c = new Cliente("Juan", "Perez", "jp@jp.ar", null, d);
3
4 tx.begin();
5 em.persist(d);
6 em.persist(c);
7 tx.commit();
```

- ▶ En línea 4, c y d son solo dos objetos que residen en la memoria de JVM.
- ▶ En líneas 5 y 6, los objetos c y d
  - ▶ se vuelven entidades administradas (managed) (`em.persist()`).
  - ▶ son elegibles para ser insertados en la base de datos
- ▶ Con el commit de la transacción (`tx.commit()`), los datos se escriben (*flushed*) en la base de datos:
  - ▶ Se genera una fila en cada una de las tablas.

## Persistir una entidad

- ▶ El orden de los métodos `persist()` en una transacción es irrelevante
  - ▶ .. aunque no siempre (dependiendo del proveedor y de las constraints en la base de datos)
- ▶ El administrador de entidades es una caché de primer nivel.
- ▶ Hasta el commit de la transacción, los datos permanecen en la memoria y no se persisten en la base de datos.
- ▶ El `EntityManager` almacena en caché los datos y, al momento de commit, almacena los datos en el orden en que lo espera la base de datos (respetando las restricciones de integridad).

## Buscar una entidad por identidad

- ▶ Existen dos métodos:
  - ▶ `EntityManager.find()`, y
  - ▶ `EntityManager.getReference()`, y

## EntityManager.find()

- ▶ Tiene dos parámetros:
  - ▶ la clase de entidad, y
  - ▶ la id.
- ▶ Si se encuentra, devuelve la entidad correspondiente; si no, devuelve **null**.

```
1 Cliente cli = em.find(Cliente.class, id);
2 if (cli != null) {
3     // procesar objeto
4 }
```

## EntityManager.getReference()

- ▶ tiene los mismos parámetros:
  - ▶ la clase de entidad, y
  - ▶ la id.
- ▶ Devuelve un proxy, no la entidad en sí.
- ▶ Recupera una referencia, pero no los datos (que se obtienen de manera *lazy*).
- ▶ Si la entidad no se encuentra cuando se necesita, lanza `EntityNotFoundException`.
- ▶ Para cuando se necesita una instancia, pero no se accede a ningún dato.

```
1 try {  
2     Cliente cli = em.getReference(Cliente.class, id);  
3     // Procesar  
4 } catch (  
5     EntityNotFoundException ex) {  
6     // Entidad no encontrada  
7 }
```

# Actualizar una entidad

- ▶ Si una entidad está administrada:
  - ▶ El administrador de entidades almacena en caché cualquier acción de entidad.
  - ▶ se reflejarán en la base de datos automáticamente (al momento del commit de la transacción).

```
1 Direccion dir = new Direccion(...);  
2 Cliente cli = new Cliente("Juan", ..., dir);  
3  
4 tx.begin();  
5 em.persist(cli);  
6 em.persist(dir);  
7 tx.commit();  
8  
9 tx.begin();  
10 cli.setNombre("Maria");  
11 tx.commit();
```

En los proveedores los cambios pueden estar fuera de la tx.  
Se acumula y se persisten al commitear una transaccion.

## Eliminar una entidad

- ▶ Se puede eliminar con `EntityManager.remove()`.
  - ▶ Se elimina de la base de datos,
  - ▶ Se desconecta del administrador de entidades (`detached`)
  - ▶ Ya no se puede sincronizar con la base de datos.
- ▶ En términos de objetos de Java, la entidad sigue siendo accesible hasta que la recoje el garbage collector.

## Eliminación de huérfanos

- ▶ Considerar el ejemplo entre Cliente y Dirección.
- ▶ Asumir que se persiste un Cliente y su Dirección y luego se remueve al cliente.
- ▶ Notar que la dirección queda huérfana (sin cliente asociado).
- ▶ Los huérfanos son no deseables – van contra la calidad (coherencia - limpieza):
  - ▶ Quedan filas a las que no hace referencia ninguna otra tabla (no se puede acceder).
- ▶ Se puede configurar al proveedor para que elimine automáticamente los huérfanos.
- ▶ Las asociaciones uno-a-uno o uno-a-muchos tiene la opción de eliminación de huérfanos.
  - ▶ opción `orphanRemoval=true`.

## orphanRemoval

```
1 @Entity  
2 public class Cliente {  
3     @Id @GeneratedValue  
4     private Long id;  
5     ....  
6     @OneToOne(orphanRemoval = true)  
7     private Direccion direccion;  
8     ...  
9 }
```

Notar que al eliminar un cliente se elimina también la dirección asociada.

## orphanRemoval

- ▶ La entidad Dirección también se elimina si la relación se rompe (se asigna null u otra dirección).
- ▶ La operación de eliminación se aplica en el momento de hacer el flush (commit de transacción).

```
1 Direccion dir = new Direccion("Cabildo", ...);  
2 Cliente cli = new Cliente("Juan", ..., dir);  
3  
4 tx.begin();  
5 em.persist(dir);  
6 em.persist(cli);  
7 tx.commit();  
8  
9 Direccion nuevadir = new Direccion("Lavalle", "100", "Cordoba",  
10      "2000", "Argentina");  
11 cli.setDireccion(nuevadir);  
12  
13 tx.begin();  
14 em.persist(nuevadir);  
15 tx.commit();
```

Notar que se elimina dir.

## Sincronización con la Base de Datos

- ▶ En los ejemplos anteriores, la sincronización con la base de datos ocurre al momento del commit de la transacción.
- ▶ El `EntityManager` es una caché de primer nivel, y espera hasta el commit para descargar los datos en la base de datos.

# Sincronización con la Base de Datos

```
1 tx.begin();  
2 em.persist(cliente);  
3 em.persist(direccion);  
4 tx.commit();
```

- ▶ Los cambios requieren sentencias SQL.
- ▶ En este caso, dos `insert` que se realizan al momento de `commit`.
- ▶ También se puede manejar la sincronización explícitamente:
  - ▶ descargar datos con (`flush`),
  - ▶ actualizar entidades con los datos de la base de datos (`refresh`).
- ▶ Si se hace un (`flush`) y más adelante se hace `rollback` de la transacción, los datos se eliminan de la base de datos.

# Flush

```
1 tx.begin();
2 em.persist(cli);
3 em.flush();
4 em.persist(dir);
5 tx.commit();
```

- ▶ `em.flush()` obliga al proveedor a escribir los datos en la base (sin esperar el `commit`)
- ▶ Se generará y ejecutará un `INSERT` para `cli`
- ▶ Habrá una excepción `IllegalStateException` porque `cli` referencia a `dir`, que no fue persistida.
- ▶ Notar que sin el flush funciona porque el `EntityManager` almacena en caché todos los cambios, los ordena y ejecuta de forma coherente para la base de datos.

## Refresh

- ▶ Se utiliza para sincronizar los datos en el dirección opuesta al flush.
- ▶ Sobrescribe el estado actual de una entidad gestionada con los datos presentes en la base de datos.
- ▶ Uso típico: deshacer los cambios que se han realizado en memoria.

# Ejemplo

```
1 Direccion dir = new Direccion(...);
2 Cliente cli = new Cliente("Juan", ..., dir);
3
4 tx.begin();
5 em.persist(cli);
6 em.persist(dir);
7 tx.commit();
8
9 cli.setDireccion(null);
10 System.out.println(cli.getDireccion()); //null
11 em.refresh(cli);
12
13 System.out.println(cli.getDireccion()); //una direccion
```

## Contenido del Contexto de persistencia

- ▶ El contexto de persistencia contiene a las entidades gestionadas.
- ▶ La interfaz EntityManager, provee operaciones para verificar si una entidad está manejada, para separarla (detach) o limpiar (clear) el contexto de persistencia.

## contains

- ▶ EntityManager.contains() permite verificar si una instancia de entidad está actualmente administrada por el EntityManager dentro del contexto de persistencia actual.
- ▶ devuelve un valor booleano.

```
1 tx.begin();
2 em.persist(cli);
3 em.persist(dir);
4 tx.commit();
5
6 System.out.println(em.contains(cli)); //true
7
8 tx.begin();
9 em.remove(cli);
10 tx.commit();
11
12 System.out.println(em.contains(cli)); //false
```

## detach y clear

- ▶ clear() vacía el contexto de persistencia.
- ▶ todas las entidades gestionadas pasan al estado detached.
- ▶ detach() elimina a la entidad dada como parámetro del contexto de persistencia.
- ▶ Si las entidades fueron modificadas, los cambios no se sincronizarán con la base de datos

## Merge

- ▶ Una entidad detached no está mas manejada.
- ▶ Para volver a manejar a una entidad detached se usa `merge` (reattach).

```
1 Direccion dir = new Direccion(...);
2 Cliente cli = new Cliente("Juan", ..., dir);
3
4 tx.begin();
5 em.persist(cli);
6 em.persist(dir);
7 tx.commit();
8
9 em.clear();
10 cli.setNombre("Maria");
11
12 tx.begin();
13 em.merge(cli);
14 tx.commit();
```

## Eventos en cascada

- ▶ Por default, cada operación del EntityManager se aplica solo a la entidad pasada como argumento.
- ▶ Muchas veces, cuando se realiza una operación en una entidad, se desea propagar la operación en las asociaciones de la entidad.
- ▶ Esto se conoce como eventos en cascada.

## Ejemplo

```
1 Direccion dir = new Direccion(...);  
2 Cliente cli = new Cliente("Juan", ..., dir);  
3  
4 tx.begin();  
5 em.persist(cli);  
6 em.persist(dir);  
7 tx.commit();
```

- ▶ Debido a que existe una relación entre el Cliente y la Dirección, se puede configurar en cascada la acción persistente del cliente a la dirección.
- ▶ Eso significaría que `em.persist(customer)` persiste (en cascada) a la entidad Dirección
- ▶ Se puede eliminar la sentencia `em.persist(address)`

## Modificación en cascada

- ▶ Los eventos en cascada se configuran cuando se da el mapeo de la relación.
- ▶ Las anotaciones @OneToOne, @OneToMany, @ManyToOne y @ManyToMany tienen el atributo cascade de tipo CascadeType[]
- ▶ Donde CascadeType indica el tipo de operación que debe propagarse en cascada al destino de la asociación.
- ▶ Puede tomar los siguientes valores: PERSIST, REMOVE, MERGE, REFRESH, DETACH, ALL
- ▶ Son riesgosos, especialmente en relaciones @ManyToMany (desaconsejado totalmente el REMOVE).

# Java Persistence Query Language

- ▶ En la práctica puede ser necesario recuperar
  - ▶ a una entidad por criterios distintos a su Id (por ejemplo, por nombre, dirección, etc.)
  - ▶ un conjunto de entidades según diferentes criterios (p. ej., todos los clientes de CABA)
- ▶ Esta posibilidad es inherente a las bases de datos relacionales,
- ▶ Java/Jakarta Persistence Query Language (JPQL) es el lenguaje de JPA que permite esta interacción.

# Java Persistence Query Language

- ▶ JPQL se utiliza para definir búsquedas de entidades persistentes independientes de la base de datos subyacente.
- ▶ JPQL es un lenguaje de consulta que tiene sus raíces en la sintaxis de SQL,
- ▶ La principal diferencia es que los resultados en SQL son tablas y en JPQL son una entidad o una colección de entidades.
- ▶ La sintaxis de JPQL está orientada a objetos (usa notación objetos)
- ▶ Permite concentrarse en el modelo de dominio y no en la tabla
- ▶ JPQL utiliza el mecanismo de mapeo para transformar una consulta JPQL en consulta SQL.

# Query Dinámicas

- ▶ em.createQuery para construir una query
  - ▶ toma una string parámetro conteniendo la sentencia en JPQL.
- ▶ Se obtienen los resultados con:
  - ▶ List getResultList()
  - ▶ Object getSingleResult()
  - ▶ Stream getResultStream()
  - ▶ int executeUpdate()
- ▶ Ejemplo: Seleccionar todas las instancias de una sola entidad:

```
1 Query query = em.createQuery("SELECT c FROM Cliente c");  
2 List clientes = query.getResultList();  
3 for (Object cliente : clientes) {  
4     System.out.println((Cliente) cliente);
```

- ▶ Ver otros ejemplos en código.

# Scripts para carga de datos

- ▶ Propiedad "javax.persistence.sql-load-script-source"

```
1 <property name="javax.persistence.sql-load-script-source"
2   value="src/main/resources/sql-scripts/inicial.sql" />
```

# Query Dinámicas con parámetros

- ▶ Parámetros por posición:

```
1 Query query = em.createQuery("SELECT c FROM Cliente c WHERE  
    c.id = ?1");  
2 query.setParameter(1, 4L);  
3 Cliente cliente = (Cliente) query.getSingleResult();
```

- ▶ Parámetros nombrados:

```
1 Query query = em.createQuery("SELECT c FROM Cliente c WHERE  
    c.fechaNacimiento > :fecha");  
2 query.setParameter("fecha", LocalDate.of(2003,01,01));  
3 List clientes = query.getResultList();
```

# TypedQuery

- ▶ Es la versión type-safe the TypedQuery.
- ▶ em.createQuery para construir una query
  - ▶ toma una string parámetro conteniendo la sentencia en JPQL.
  - ▶ Y el tipo del resultado de la clase
- ▶ Parámetros nombrados:

```
1 TypedQuery<Cliente> query = em.createQuery("SELECT c FROM  
     Cliente c",Cliente.class);  
2 List<Cliente> clientes = query.getResultList();  
3 for (Cliente cliente : clientes) {  
4     System.out.println(cliente);
```

# NamedQuery

- ▶ Son consultas reutilizables.
- ▶ Las consultas con nombre son estáticas e inmutables.
- ▶ Pueden ser más eficientes de ejecutar porque el proveedor de persistencia las puede traducir al inicio de la aplicación.
- ▶ Se definen como metadatos con la anotación @NamedQuery (o el equivalente XML)
- ▶ @NamedQuery tiene dos elementos:
  - ▶ el nombre de la consulta
  - ▶ su contenido.

# NamedQuery

- ▶ Se definen en la entidad

```
1 @Entity  
2 @NamedQuery(name = "buscarTodos", query = "SELECT c FROM  
3   Cliente c")  
4 @NamedQuery(name = "buscarPorId", query = "SELECT c FROM  
5   Cliente c WHERE c.id = ?1")  
6 public class Cliente {  
7   ...  
8 }
```

- ▶ Uso (version untyped)

```
1 em.createQuery("buscarTodos");
```

- ▶ Uso (version typed)

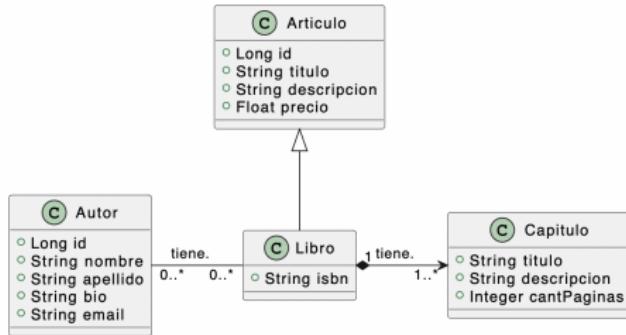
```
1 em.createNamedQuery("buscarTodos", Cliente.class)
```

## Queries orientadas a objetos (API criteria)

- ▶ Permite escribir una consulta de manera orientada a objetos y sintácticamente correcta.
- ▶ Muchos errores son detectados en tiempo de compilación, no en ejecución.
- ▶ La idea es que las palabras clave de JPQL son métodos de la API.

# Ejercicio

- ▶ Se desea implementar el siguiente modelo de objetos



- ▶ Implementar todas las clases.
- ▶ No considerar validaciones de datos.
- ▶ Definir los casos de test.
- ▶ Sugerencia: aplicar TDD.

# Ejercicio

- ▶ Dar una implementación donde:
  - ▶ Libro y Autor son las únicas entidades.
  - ▶ todos los atributos de Artículo deben ser persistentes.
  - ▶ las ids deben ser generadas automáticamente.
- ▶ Dar los casos de test.

# Callbacks

- ▶ El ciclo de vida de una entidad se divide en cuatro categorías:
  - ▶ persistencia,
  - ▶ actualización,
  - ▶ eliminación, y
  - ▶ carga
- ▶ que corresponden a las operaciones sobre la base de datos (insert, update, delete y select).
- ▶ Cada categoría tiene un evento "pre" y "post" (excepto la carga, que solo tiene "post") que puede ser interceptado por el administrador para invocar un método.
- ▶ Se usan anotaciones para indicar estos métodos.

# Callbacks

- ▶ `@PrePersist`: método a invocar antes de `persist`.
- ▶ `@PostPersist`: método a invocar después de `persist`.
- ▶ `@PreUpdate`: método a invocar antes de que el estado se actualice.
- ▶ `@PostUpdate`: método a invocar después de que el estado se actualice.
- ▶ `@PreRemove`: método a invocar antes de `remove`.
- ▶ `@PostRemove`: método a invocar después de `remove`.
- ▶ `@PostLoad`: método a invocar después de que una entidad es cargada (por `find` o una consulta)

## Reglas de Callbacks

- ▶ Pueden ser **public**, **private**, **protected** o de protegido en paquete.
- ▶ No deben ser **static** ni **final**
- ▶ Un método puede tener varias anotaciones de ciclo de vida.
- ▶ Una clase no puede tener dos métodos con la misma anotación.
- ▶ Puede arrojar excepciones no-chequeadas pero no puede arrojar chequeadas.
- ▶ Si arroja una excepción, la transacción (si existe) hace rollback.
- ▶ No puede invocar operaciones sobre un entityManger ni hacer queries.
- ▶ En jerarquías de herencia, se invocan antes los de las superclases.
- ▶ Si se usa eventos en cascada, se invocan también en cascada.

# Ejemplo

Cliente: validar datos, nombre.

## Listeners

- ▶ Los callback son útiles cuando la lógica de negocio está relacionada a una única entidad.
- ▶ Los listeners se utilizan para extraer lógica en una clase separada y compartirla entre varias entidades.
- ▶ Un listener es un POJO en el que puede definir uno o más métodos de callback.
- ▶ Para registrar un listener, se usa la anotación `@EntityListeners`

# Ejemplo

Cliente: validar datos, nombre.

## Reglas Listeners

- ▶ La clase listener debe tener un constructor público sin argumentos.
- ▶ Las firmas de los métodos son análogos a los callbacks, sólo que recibe como parámetro el objeto a verificar.

## Reglas Listeners

- ▶ Para asociar un Listener a una entidad Cliente se usa `@EntityListeners`.
- ▶ Esta anotación puede tomar un Listener o un array de Listeners.
- ▶ Cuando se definen varios y se produce el evento:
  - ▶ El proveedor ejecuta los callbacks de los listeners en el orden en el que están declarados.
  - ▶ Luego los callbacks de la propia entidad (si los hay).
- ▶ Si un método arroja excepción, no se ejecutan los demás y se hace rollback de la transacción.
- ▶ Los listeners definidos en la superclase se invocan antes que los definidos en las subclases.
- ▶ Si una entidad no quiere heredar los listeners, los puede excluir con `@ExcludeSuperclassListeners`.
- ▶ Se pueden definir defaults (pero definiendo mapeo por XML)

## Reglas Listeners

- ▶ Para asociar un Listener a una entidad Cliente se usa `@EntityListeners`.
- ▶ Esta anotación puede tomar un Listener o un array de Listeners.
- ▶ Cuando se definen varios y se produce el evento del ciclo de vida:
  - ▶ el proveedor de persistencia ejecuta los callbacks de los listeners en el orden en el que están declarados.
  - ▶ Luego los callbacks de la propia entidad (si los hay).

## Ejercicio

- ▶ Se extender la solución al problema anterior, de manera tal que la entidad Libro tenga una propiedad cantidad de páginas (que se corresponde con la suma de las páginas de sus capítulos).
- ▶ No se debe agregar una columna a la tabla Libro

# Java/Jakarta Bean Validation

- ▶ La validación de datos es una tarea común que se debe realizar en todas las capas de una aplicación.
- ▶ Requiere mucho tiempo, es propensa a errores y es difícil de mantener.
- ▶ Algunas se usan con tanta frecuencia que podrían considerarse estándar (comprobar un valor nulo, su tamaño, su rango, etc.).
- ▶ **Bean Validation** permite escribir una validación y reutilizarla en diferentes capas de aplicación.
- ▶ Permite aplicar restricciones ya definidas, y también escribir las propias reglas de validación.
- ▶ Se definen las validaciones mediante anotaciones o descriptores XML.

# Java Persistence Query Language

```
1 public class Cliente {  
2     @NotEmpty //No puede ser nulo ni vacío  
3     private String nombre;  
4     @NotEmpty  
5     private String apellido;  
6     @Email //Email valido  
7     private String email;  
8     @Past //fecha en el pasado  
9     private LocalDate fechaNacimiento;  
10    ...  
11 }
```

▶ Documentación

# Dependencias Validation (javax.)

- ▶ La API del estándar de validación

```
1 <dependency>
2   <groupId>javax.validation</groupId>
3   <artifactId>validation-api</artifactId>
4   <version>2.0.1.Final</version>
5 </dependency>
```

- ▶ La implementación de referencia de la API (esta importa también la anterior)

```
1 <dependency>
2   <groupId>org.hibernate</groupId>
3   <artifactId>hibernate-validator</artifactId>
4   <version>6.0.13.Final</version>
5 </dependency>
```

- ▶ Una implementación de la API *Expression Language* (permite interpolación de variables)

```
1 <dependency>
2   <groupId>org.glassfish</groupId>
3   <artifactId>jakarta.el</artifactId>
4   <version>3.0.2</version>
5 </dependency>
```

# Dependencias Validation (jakarta)

```
1 <dependency>
2   <groupId>org.hibernate</groupId>
3   <artifactId>hibernate-validator</artifactId>
4   <version>7.0.4.Final</version>
5 </dependency>
6
7 <dependency>
8   <groupId>org.glassfish</groupId>
9   <artifactId>jakarta.el</artifactId>
10  <version>4.0.2</version>
11 </dependency>
```

# Validación programática

- ▶ Primero necesitamos un objeto Validator, que se construye utilizando ValidatorFactory.

```
1 ValidatorFactory factory = Validation.  
     buildDefaultValidatorFactory();  
2 Validator validator = factory.getValidator();
```

- ▶ Podemos validar nuestro bean (objeto) usando el método validate
- ▶ Que devolverá un conjunto de violaciones a las restricciones definidas en el objeto.

```
1 Set<ConstraintViolation<Cliente>> violations = validator.  
     validate(cliente);
```

# Ejemplo

```
1 ValidatorFactory factory = Validation.  
2         buildDefaultValidatorFactory();  
3 Validator validator = factory.getValidator();  
4  
5 Cliente cliente = new Cliente("Juan", null, "asdf@pepe.com", "  
6     23232");  
7 Set<ConstraintViolation<Cliente>> violations = validator.  
8     validate(cliente);  
9 if (violations.isEmpty()) {  
10     System.out.println("No hay errores");  
11 } else {  
12     violations.forEach(violation -> {  
13         String message = violation.getPropertyPath()  
14             + ": " + violation.getMessage();  
15         System.out.println(message);  
16     });  
17 }
```

# Personalización de mensajes

- ▶ Puede utilizarse la propiedad `message` de la anotación de restricción.

```
1 @NotEmpty(message = "El nombre no puede ser vacío o nulo")
2 private String apellido;
```

- ▶ Es posible usar expresiones EL:

```
1 @NotEmpty(message = "no puede ser nulo")
2 @Size(min = 2, max = 20, message = "Longitud minima: {min}
   y maxima:{max}")
3 private String apellido;
4 @Email(message = "${validatedValue} no es un email valido")
   //Email valido
5 private String email;
```

# Externalización de mensajes

- ▶ Se puede definir los mensajes en un archivo ValidationMessages.properties en el directorio src/main/resources/.
- ▶ Se agregan las propiedades en este archivo, Por ejemplo:.

```
1 cliente.fechaNacimiento.past = La fecha ${validatedValue}  
no es anterior a la actual
```

- ▶ Luego, el mensaje usa esa propiedad:

```
1 @Past(message = "{cliente.fechaNacimiento.past}")  
2 private LocalDate fechaNacimiento;
```

# Integración con JPA

- ▶ En un entorno JSE, la validación se habilita por default si se proporciona la API y un proveedor de validación.
- ▶ Se debe utilizar un archivo `persistence.xml` de la versión 2.X.
- ▶ La validación de beans proporciona tres modos de operación dentro del entorno JPA:
  - ▶ AUTO Habilita la validación si hay un proveedor de validación disponible (predeterminado).
  - ▶ CALLBACK Si no hay proveedor de validación genera una excepción.
  - ▶ NONE Inhabilita la validación de beans para una unidad de persistencia particular.
- ▶ AUTO simplifica la implementación, pero puede generar problemas si la validación no se lleva a cabo.
- ▶ Buena práctica: Especificar CALLBACK o NONE

# Integración con JPA

- ▶ La validación dentro de JPA ocurre durante el procesamiento de eventos del ciclo de vida de una entidad.
- ▶ Ocurre:
  - ▶ en la etapa final de los eventos del ciclo de vida **PrePersist**, **PreUpdate** y **PreRemove**.
  - ▶ Se puede trabajar con grupos de constraints (que se deben configurar)

# Ejercicio

- ▶ Revisar el problema de libros y definir las validaciones para la entidad Libro (considerar también validaciones en Capítulo).
- ▶ Dar casos de test.

# Lombok

- ▶ Evitar código repetitivo
- ▶ Lombok actúa en el proceso de compilación y genera automáticamente el código de bytes de Java en los archivos .class a partir de anotaciones.

▶ Lombok

# Ejemplo

```
1 @Getter  
2 public class Autor {  
3     private Long id;  
4     private String nombre;  
5     private String apellido;  
6     private String bio;  
7     private String email;  
8 }
```

```
1 @Test  
2 void test() {  
3     assertNull(new Autor().getApellido());  
4 }
```

## Mapeo con descriptor XML

- ▶ En lugar de (o además de) anotaciones, se puede definir el mapeo usando descriptores XML.
- ▶ El mapeo se define en un archivo XML externo: Mapping descriptor.
- ▶ El Descriptor coincide estrechamente con las anotaciones.
- ▶ También se colocan en META-INF.

## Descriptor cliente-mapping.xml

```
1 <entity-mappings
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
4   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/
5       orm http://xmlns.jcp.org/xml/ns/persistence/orm_2_2.xsd"
6   version="2.2">
7     <entity class="org.cursoabbva.modulo4.xmlmapping.Cliente">
8       <table name="Tabla_Cliente" />
9       <attributes>
10         <id name="id">
11           <generated-value />
12         </id>
13         <basic name="apellido">
14           <column name="C_APELLIDO" nullable="false" updatable="
15             false" />
16         </basic>
17         <one-to-one name="direccion"></one-to-one>
18
19       </attributes>
20     </entity>
21   </entity-mappings>
```

## persistence.xml

```
1 <persistence-unit name="mappingxml"
2   transaction-type="RESOURCE_LOCAL">
3     <provider>org.hibernate.jpa.HibernatePersistenceProvider</
4       provider>
5     <mapping-file>META-INF/cliente-mapping.xml</mapping-file>
6     <mapping-file>META-INF/direccion-mapping.xml</mapping-file>
7
8     <exclude-unlisted-classes />
9     <properties>
10      ...
11      <properties>
</persistence-unit>
```

## Mapeo combinado

- ▶ Se pueden combinar las dos técnicas (XML y anotaciones).
- ▶ El mapeo se define en un archivo XML externo: Mapping descriptor
- ▶ El Descriptor coincide estrechamente con las anotaciones
- ▶ También se colocan en META-INF.