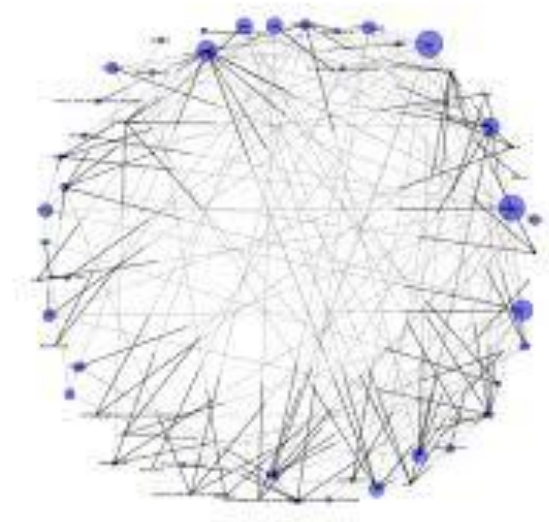


# **DAM**

## **ACCESO A DATOS**



### **UD2**

## **MANEJO DE CONECTORES**



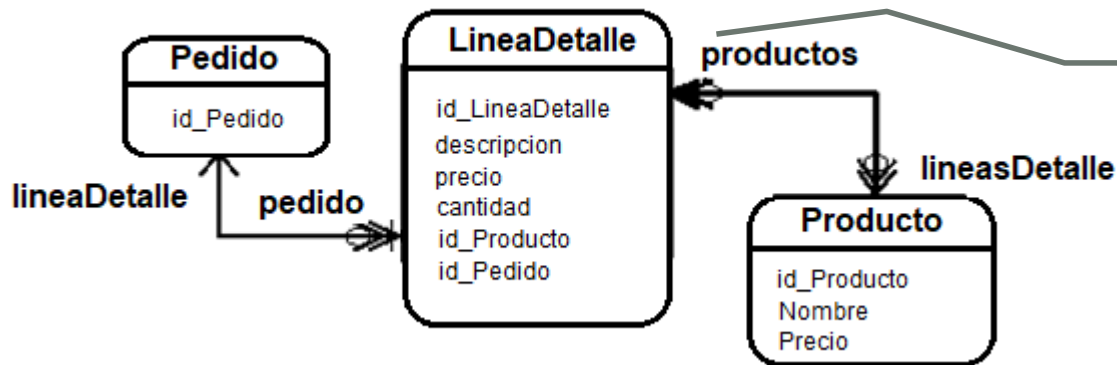
# Introducción

- En este tema analizaremos los métodos de conexión de Java con distintos tipos de bases de datos utilizando conectores.
- Un conector **conector** es el software que permite conectar una aplicación Java con una base de datos con el objetivo de:
  - Modificar la estructura y el contenido de una base de datos.
  - Realizar consultas sobre la base de datos.
- Existen **conectores diferentes** para cada **base de datos-tipo de conexión**.
- Según la base de datos que queramos explotar y el sistema de conexión, deberemos **incorporar a nuestro proyecto la librería** correspondiente.
- Los tipos de bases de datos con las que trabajaremos son:
  - bases de datos relacionales y
  - bases de datos orientadas a objetos.



# Bases de datos objeto/relacional

- Las bases de datos relacionales, basadas en tablas y relaciones, no se ajustan al tratamiento de datos que tienen como fuente objetos con atributos y comportamientos.
- El uso del paradigma de objetos en el tratamiento de datos a dado lugar a las bases de datos orientadas a objetos.
- **Bases de datos relacionales:**
  - Basadas en entidades y relaciones.
  - Uso del lenguaje SQL para su creación y explotación.
  - Orientada a los datos.
- **Bases de datos orientadas a objetos:**
  - Basada en clases y sus relaciones.
  - Orientada a los comportamientos de los objetos.



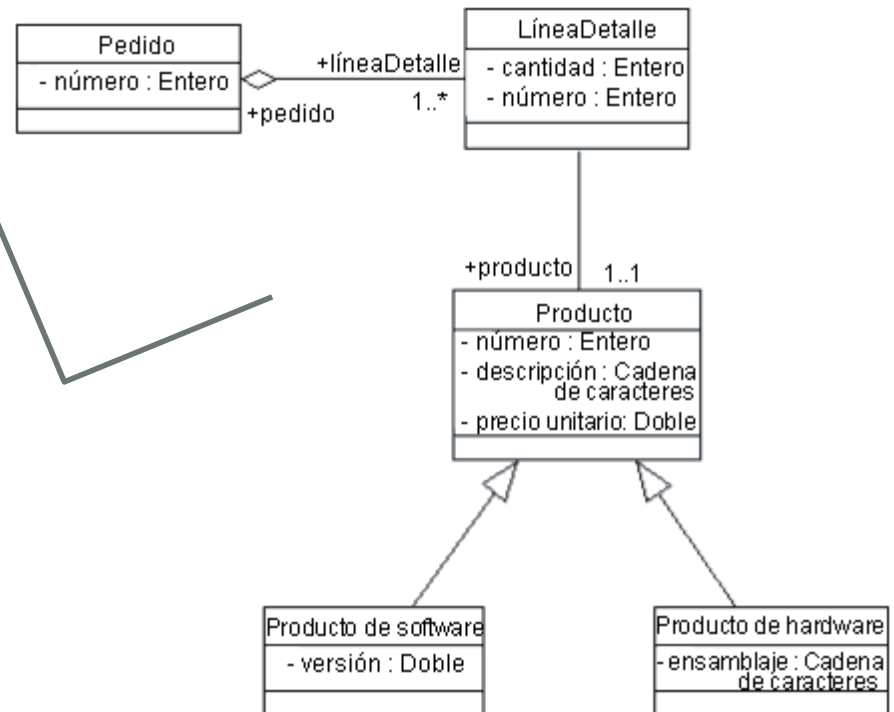
**LineaDetalle** tiene las columnas **Id\_LineaDetalle** (la clave principal), descripción, precio, cantidades y las claves externas, **Id\_Producto** y **Id\_Pedido**, que enlazan la tabla con **Pedido** y **Producto**.

*Un **Pedido** puede tener varias referencias en **LineaDetalle** y un **Producto** también.*

Una instancia de **Pedido** está compuesta por varias **LineasDetalle** y una instancia de **LineaDetalle** hace referencia a un solo **producto**.

Cada **Producto** se puede clasificar como **Producto de software** o **Producto de hardware**.

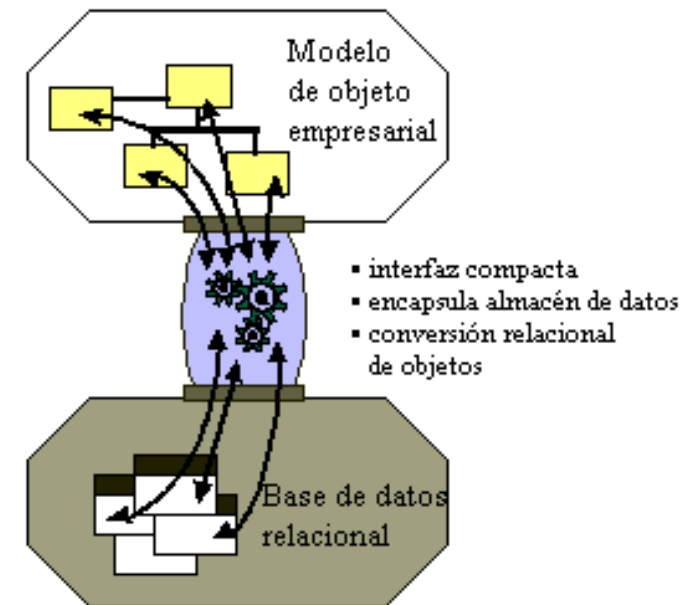
El modelo de objetos soporta la herencia. Una clase puede heredar propiedades y comportamientos de otra clase (por ejemplo, los productos **ProductoSoftware** y **ProductoHardware** heredan los atributos y los métodos de la clase **Producto**).



# Desfase objeto-relacional

- La mayoría de las aplicaciones utilizan bases de datos relacionales.
- Los desarrolladores de software OO debe implementar un modelo de datos que facilite la transición de los datos entre las **clases** y la **bases de datos relacional** donde están almacenados físicamente los datos.

En el software OO, los datos deben someterse a la conversión **relacional** → **objetos** antes de ser utilizados por la aplicación. También es necesario la traslación de la información de los objetos a las estructuras relacionales antes de almacenar la información en la base de datos. Al conjunto de dificultades que esto plantea se le conoce como **desfase objeto-relacional**.



# BASES DE DATOS EMBEBIDAS

- Están almacenadas en ficheros locales.
- No requieren de un SGBD (*Sistemas de Gestión de Bases de Datos*).
- La base de datos se inicia cuando se arranca la aplicación y finaliza cuando se cierra la misma.
- **Ventajas:**
  - No es necesario gestionar ni mantener ningún SGBD.
  - Suelen tener un modo de funcionamiento en el que mantienen los datos en memoria, incrementando el rendimiento.
- **Inconveniente:**
  - En el caso de que se compartan sus ficheros por varias aplicaciones suelen tener peor rendimiento debido a las colisiones de acceso.
- **Ejemplos:**
  - SQLite
  - DerbyDB o JavaDB
  - HSQLDB
  - H2
  - db\$0

# SQLite

- Sistema gestor de bases de datos multiplataforma.
- Está escrito en C y tiene un motor ligero.
- Las bases de datos se guardan como ficheros, con esto se facilita el traslado de las bases de datos con la aplicación con la que se usa.
- Implementa la mayor parte del estándar SQL-92, incluyendo transiciones de ases de datos, consistencia de bases de datos, *triggers* y la mayor parte de consultas complejas.
- Es un proyecto de dominio público.
- Se puede usar desde programas en Java, C, C++, PHP, Visual Basic, Perl,...



# SQLite. Instalación y uso

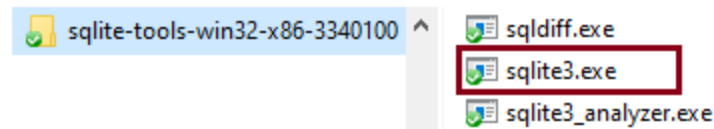
- Para instalar SQLite debemos seguir los siguientes pasos:
  - Descargar la versión apropiada para nuestro sistema operativo, Windows, en este caso. Usar el siguiente enlace.

[sqlite-tools-win32-x86-3340100.zip](#)  
(1.76 MiB)

A bundle of command-line tools for managing SQLite database files, including the [command-line shell](#) program, the [sqldiff.exe](#) program, and the [sqlite3\\_analyzer.exe](#) program.

(sha3: 9f42a533e5dd04c98e575cb95ef6458eachb43dbaa3cc0c51558a6a626f2a3acd)

- Descomprimir el fichero descargado.
- Ejecutar el fichero **sqlite3.exe** para crear/modificar/consultar bases de datos





# SQLite. Creación de una BD

## Crear agenda.db

```
D:\sqlite-tools-win32-x86-3340100>sqlite3 agenda.db
SQLite version 3.34.1 2021-01-20 14:10:07
Enter ".help" for usage hints.
sqlite>
```

## Ver las tablas creadas

```
sqlite> SELECT * FROM sqlite_master WHERE type = "table";
table|contactos|contactos|2|CREATE TABLE contactos (
nick TEXT,
nombre TEXT,
apellidos TEXT,
direccion TEXT,
telefono1 NUMERIC,
telefono2 NUMERIC)
```

## Crear tabla contactos

```
sqlite> CREATE TABLE contactos (
...> nick TEXT,
...> nombre TEXT,
...> apellidos TEXT,
...> direccion TEXT,
...> telefono1 NUMERIC,
...> telefono2 NUMERIC);
```

## Salir de la base de datos

```
sqlite> .quit
```

## Insertar registros en la tabla contactos

```
sqlite> INSERT INTO contactos VALUES ('adp','Ana','Pin','',65456765, 345675433);
sqlite> INSERT INTO contactos VALUES ('teri','Teresa','Arias','C/Sol nº 21',, 345675433);
sqlite> INSERT INTO contactos VALUES ('lolo','Manolo','Silva','C/Murias nº 2','348975433','666376234');
```

## Mostrar el contenido de la tabla contactos

```
sqlite> SELECT * FROM contactos;
adp|Ana|Pin||65456765|345675433
teri|Teresa|Arias|C/Sol nº 21|345675433|654376234
lolo|Manolo|Silva|C/Murias nº 2|348975433|666376234
```

Se ha creado una BD que se  
almacena en el fichero  
**agenda.db**

Enlaces de consulta:

<https://josejuansanchez.org/bd/unidad-04-sqlite/index.html>

<https://www.sqlite.org/datatype3.html>

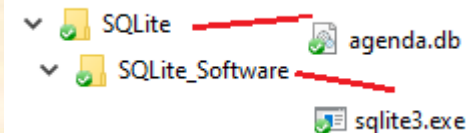
# SQLite. Consulta de una BD

```
D:\SQLite_Software>sqlite3 abrir sqlite
SQLite version 3.34.1 2021-01-20 14:10:07
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.

sqlite> .open ../agenda.db abrir la BD

sqlite> select * from contactos;
adp|Ana|Pin||65456765|345675433
teri|Teresa|Arias|C/Sol nº 21|345675433|654376234
lolo|Manolo|Silva|C/Murias nº 2|348975433|666376234
sqlite>
```

En estos accesos suponemos la siguiente estructura de carpetas y localización de ficheros implicados.



# DerbyDB o JavaDB

- **JavaDB** es una distribución de Oracle de la base de datos de código libre **DerbyDB**.
- Soporta el estándar ANSI/ISO SQL a través de JDBC y Java EE.
- Estas librerías están incluidas en el JDK.
- Almacena la base de datos en múltiples archivos, lo que puede resultar útil para escalar el almacenamiento.
- Solo podemos utilizarlo en Java, no desde otros lenguajes.



# DerbyDB. Instalación y uso

- Para instalar **DerbyDB** debemos seguir los siguientes pasos:
  - Descargar la versión apropiada para nuestro sistema operativo, Windows, en este caso. Usar el siguiente enlace para descargar el fichero [db-derby-10.15.2.0-bin.zip](#) para Java 9 y superiores.



- Descomprimir el fichero descargado.
- Ejecutar el fichero **ij.bat** de la carpeta **bin** del software descomprimido para crear/modificar/consultar bases de datos

```
D:\db-derby-10.15.2.0-bin\bin>ij.bat
Versi n de ij 10.15
ij>
```

**L nea de comandos  
de DerbyDB**

# DerbyDB. Creación de una BD

## Crear una nueva base de datos

```
ij> connect 'jdbc:derby:../empresa;create=true';
```

## Crear la tabla *empleados*

```
ij> CREATE TABLE empleados (  
> id_user INTEGER NOT NULL PRIMARY KEY,  
> nombre VARCHAR(25),  
> apellidos VARCHAR(50),  
> direccion VARCHAR(75),  
> localidad VARCHAR(50),  
> telefono VARCHAR(9)  
> );  
0 filas insertadas/actualizadas/suprimidas  
ij>
```

## Insertar registros en la tabla *empleados*

```
ij> INSERT INTO empleados VALUES (11,'Adolfo','Meana','','Gijón','456786543');  
1 fila insertada/actualizada/suprimida  
ij> INSERT INTO empleados VALUES (12,'Jose','Rodriguez','','Mieres','656786576');  
1 fila insertada/actualizada/suprimida  
ij> INSERT INTO empleados VALUES (13,'Xiana','Pereira','','Oviedo','675849364');  
1 fila insertada/actualizada/suprimida  
ij>
```

## Mostrar el contenido de la tabla *empleados*

```
ij> SELECT * FROM empleados;
```

ID_USER	NOMBRE	APELLIDOS	LOCALIDAD	DIRECCION	TELEFONO
11	Adolfo	Meana	Gijón		456786543
12	Jose	Rodriguez	Mieres		656786576
13	Xiana	Pereira	Oviedo		675849364

3 filas seleccionadas  
ij>

## Salir de la base de datos

```
ij> exit  
>
```

*Pulsar CTRL+C  
para finalizar*

Se ha creado una BD que se  
almacena en el fichero  
**empresa.db**

# DerbyDB. Consulta de una BD

Conectarse a la base de datos ya creada *empresa* y consulta de todos los registros de la tabla empleados

```
ij> connect 'jdbc:derby:../empresa';
```

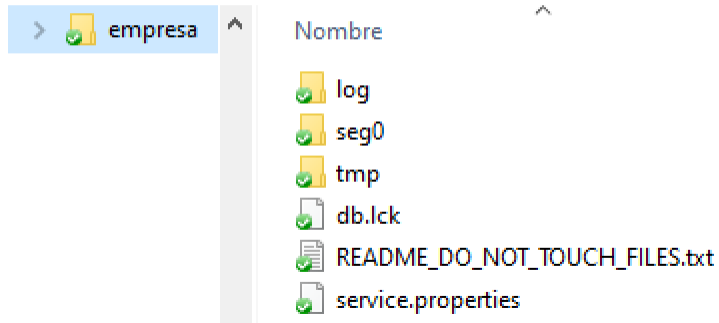
```
ij> SELECT * FROM empleados;
```

ID_USER	NOMBRE LOCALIDAD	APELLIDOS	TELEFONO	DIRECCION
11	Adolfo Gijón	Meana	456786543	
12	Jose Mieres	Rodriguez	656786576	
13	Xiana Oviedo	Pereira	675849364	

3 filas seleccionadas

```
ij>
```

Aspecto de la base de datos creada desde el administrador de archivos



Almacenamiento de la base de datos **empresa** en el disco duro.



# HSQLDB (*Hyperthreaded Structured Query Language database*)

- Sistema gestor de bases de datos relacionales escrito en Java.
- El paquete OpenOffice la incluye desde su versión 2.0
- Puede mantener la base de datos en memoria o en ficheros en disco.
- Se distribuye bajo la licencia BSD (*Berkeley Software Distribution*) es una licencia muy cercana a dominio público.
- Soporta la mayor parte de las características y funciones incluidas en el estándar SQL:2011.

HyperSQL

# HSQLDB. Instalación y uso

- Para instalar SQLite debemos seguir los siguientes pasos:
  - Descargar la versión apropiada para nuestro sistema operativo, Windows, en este caso. Usar el siguiente enlace <https://sourceforge.net/projects/hsqldb/files/> para descargar el fichero **hsqldb-2.5.1.zip**.

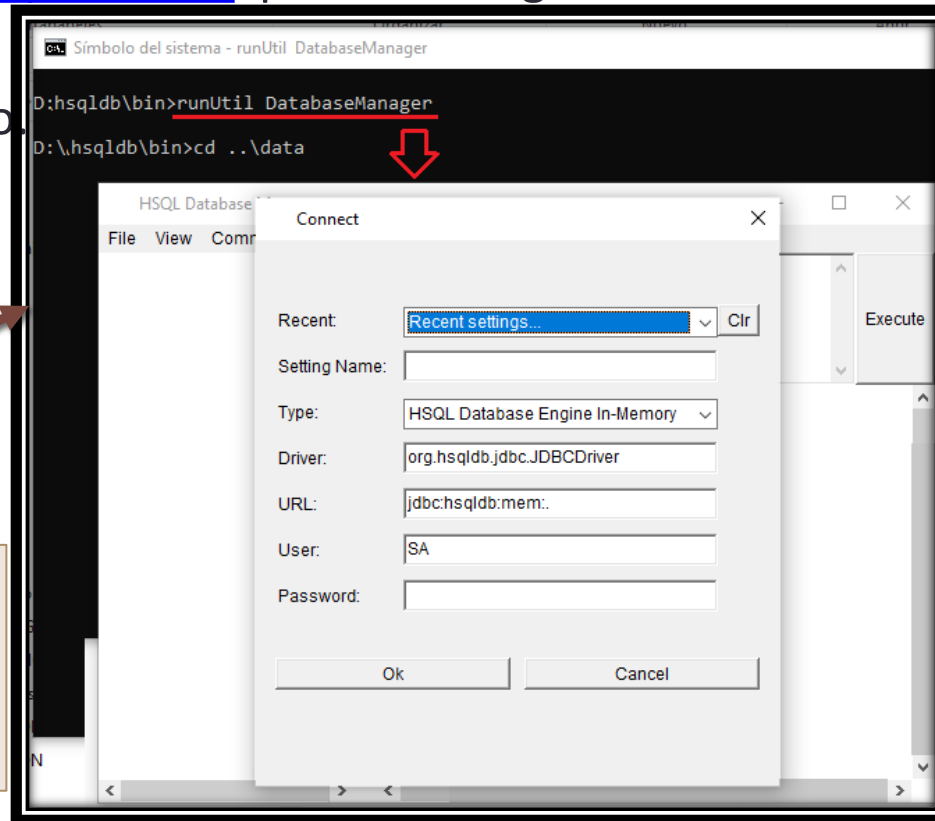
- Descomprimir el fichero descargado

Ejecutar el fichero **/bin/runUtil.bat** con el parámetro **DatabaseManager** para abrir el gestor de base de datos **HSQLDB** mediante una interfaz gráfica.

**Setting Name:** Nombre de la configuración que se puede seleccionar en las siguientes sesiones.

**Type:** *HSQLB Database Engine Standalone* → si la BD existe, la abre y, sino, la crea.

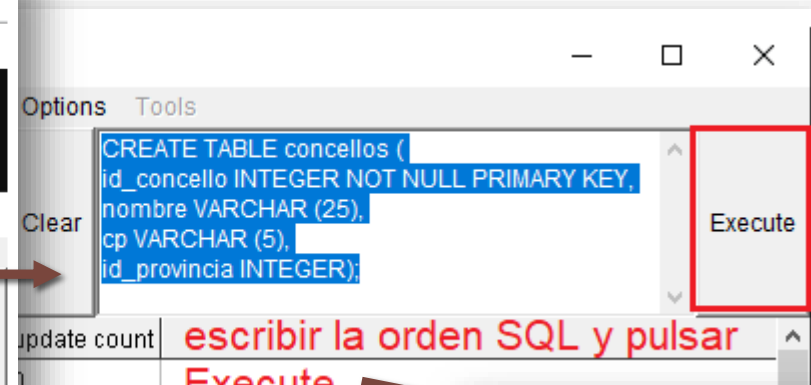
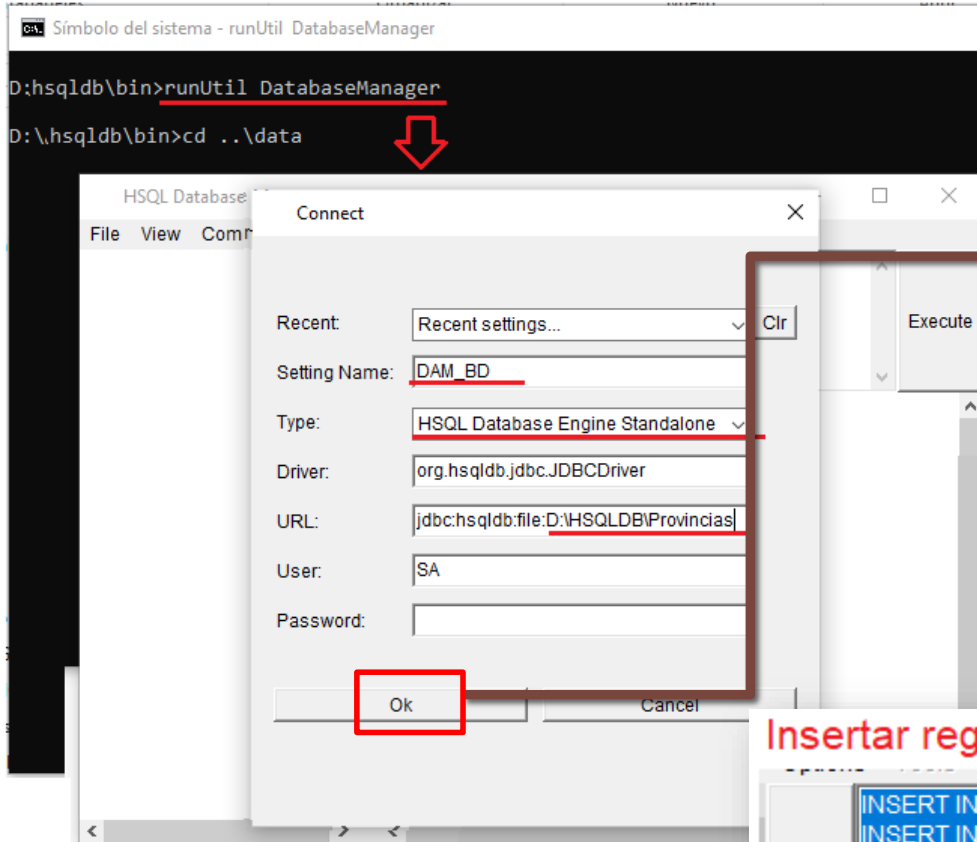
**URL:** Ruta y nombre de la base de datos.





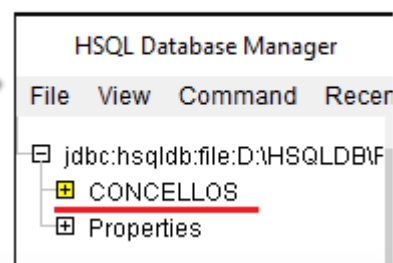
# HSQLDB. Creación de una BD

Crear una nueva DB llamada Provincias en D:\HSQLDB

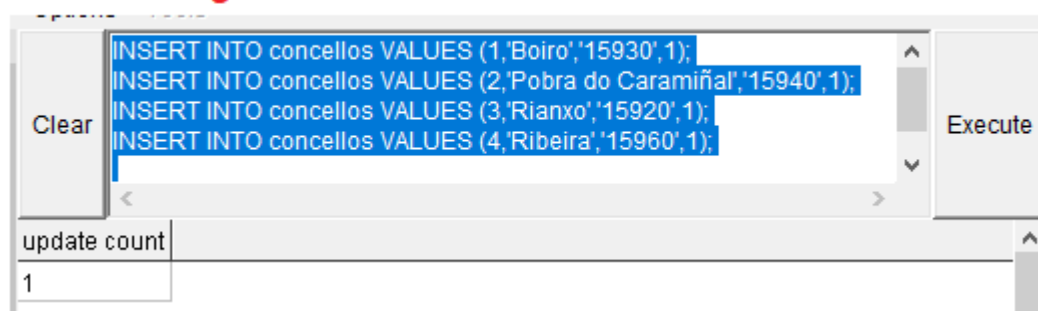


escribir la orden SQL y pulsar  
Execute

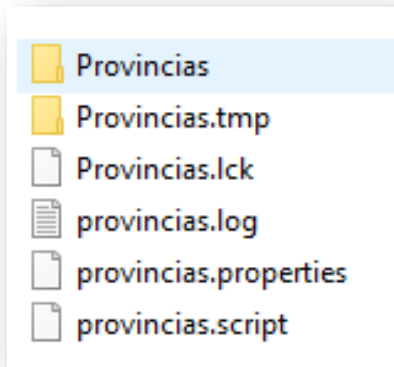
View-Refresh tree



Insertar registros



# HSQLDB. Consulta de una BD



En el cuadro de consultas de la interfaz gráfica de HSQLDB se puede ejecutar cualquier consulta SQL

La estructura de la base de datos en el disco duro de la base de datos **provincias** es la siguiente

# H2

- Es un sistema gestor de bases de datos relacionales programado en Java.
- Disponible como software de código abierto, bajo la licencia pública de Mozilla (*Mozilla Public License / Eclipse Public License*)



# H2. Instalación y uso

- En el enlace del *site* oficial de H2 descargamos la última versión del fichero [Windows Installer](#).
- Ejecutamos el fichero descargado e indicamos la ruta de instalación.



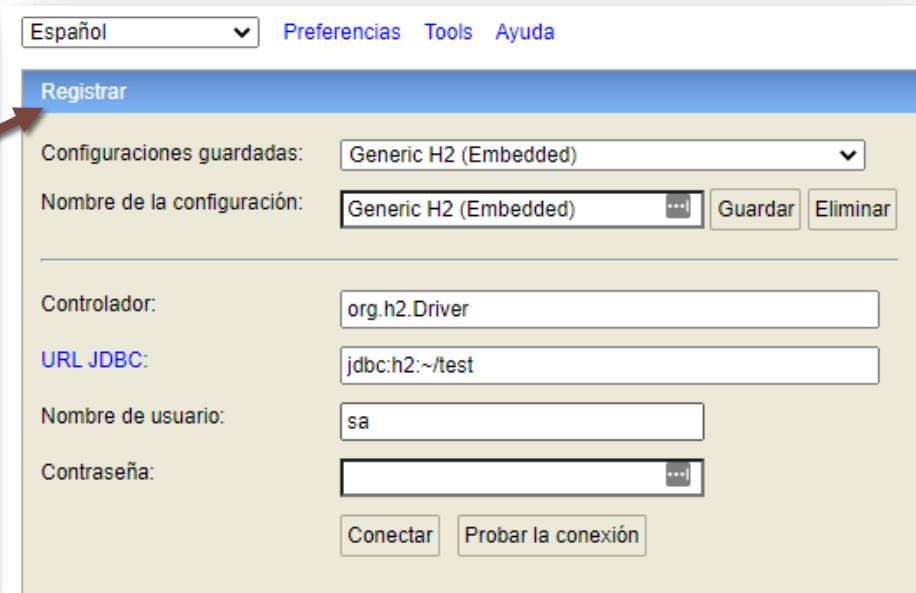
## Downloads

Version 1.4.200 (2019-10-14)

[Windows Installer](#) (SHA1 checksum: 12710a463318cf23c0e0e3f7d58a0f232bd39cfe)

Ejecutar el fichero **/bin/h2.bat** de sus ficheros instalados en la ruta indicada y se abre una **consola de administración** en el navegador por defecto.

**Setting Name:** Nombre de la configuración que se puede seleccionar en las siguientes sesiones.  
**URL JDBC:** Ruta y nombre de la base de datos.

A screenshot of the H2 console administration interface. At the top, there is a language dropdown menu set to 'Español' and links for 'Preferencias', 'Tools', and 'Ayuda'. Below this is a 'Registrar' (Register) tab. The form contains several fields: 'Configuraciones guardadas' (Saved configurations) with a dropdown menu showing 'Generic H2 (Embedded)'; 'Nombre de la configuración' (Configuration name) with a text box containing 'Generic H2 (Embedded)' and buttons for 'Guardar' (Save) and 'Eliminar' (Delete); 'Controlador' (Driver) with a text box containing 'org.h2.Driver'; 'URL JDBC' with a text box containing 'jdbc:h2:~/test'; 'Nombre de usuario' (Username) with a text box containing 'sa'; and 'Contraseña' (Password) with a masked text box. At the bottom, there are buttons for 'Conectar' (Connect) and 'Probar la conexión' (Test connection). A red arrow points from the text in the previous block to the 'Registrar' tab.

# H2. Creación y consulta de una BD

Crear/Abrir una base de datos file:D:/H2/Productos

Registrar

Configuraciones guardadas: AD\_Ejemplo

Nombre de la configuración: AD\_Ejemplo

---

Controlador: org.h2.Driver

URL JDBC: jdbc:h2:D:/H2/Pedidos

Nombre de usuario: sa

Contraseña:

Se abre en el navegador una página de administración donde podemos ejecutar cualquier comando SQL

Auto commit ☒ Número máximo de filas: 1000     Instrucción SQL:

jdbc:h2:D:/H2/Productos

INFORMATION\_SCHEMA

Usuarios

H2 1.4.200 (2019-10-14)

## H2. Ejercicio

- En la base de datos **Pedidos**, crear una tabla llamada **Productos** según los campos que se indican en el siguiente enlace <https://diego.com.es/sql-principios-basicos>.
- Para consultar <https://www.adictosaltrabajo.com/2010/07/06/h-2-base-datos/>

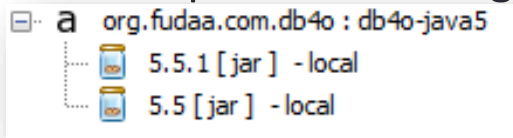
# Db4o (*Data Base 4 Objects*)

- Motor de bases de datos orientado a objetos.
- Está disponible para entornos Java y .NET.
- Posee una licencia dual GPL/Comercial
- Características
  - Se consigue evitar el desfase objeto-relacional.
  - No existe un lenguaje SQL para manipular los datos. Para ello se usan métodos delegados.
  - Para instalarlo solo hay que añadir una librería.
  - Se guarda como un único fichero de base de datos con la extensión “.yap”.

db4o

# Db4o. Instalación

- Crear un proyecto para trabajar con una base de datos **db4o** en el que habrá que enlazar con la librería adecuada y crear la clase que generará los objetos que se almacenarán en la base de datos
  - Las librerías necesarias para el uso de la base de datos **db4o v5** desde un proyecto **Maven** en Java se pueden descargar del repositorio de Maven.



- Para descargar la versión 8 utilizar un paquete que se encuentra en el aula virtual. Para los proyectos Maven de Java es necesario seguir unas instrucciones específicas.
- La base de datos db4o se usa para almacenar objetos de forma directa, por lo tanto vamos a definir una clase para trabajar con ella. La clase se llamará **Jugador** y tendrá las siguientes características:
  - **Propiedades:** **nombre** (*String*) y **puesto** (*int*).
  - Con un **constructor** que permite dar valor a las dos propiedades.
  - Con **getter** y **setter** para las dos propiedades.
  - Un método **toString()** que muestre la información de la clase.



# Db4o. Uso I

- Para crear/abrir una base de datos de llama al método **Db4o.openFile(*String* nameDB)** que crea la base de datos si ésta no existe o la abre en caso contrario. El nombre de la base de datos puede incluir la ruta y debe tener la extensión **.YAP**.

```
//acceso a Db4o
```

```
ObjectContainer db = Db4o.openFile(nameDB);
```

```
try {
```

```
    //ejecutar las operaciones en la base de datos
```

```
}
```

```
finally {
```

```
    db.close();
```

```
}
```

# Db4o. Uso II

Almacenar un objeto  
en la BD

```
static void almacenarObjetos(String nombreJugador, int puestoJugador) {  
    Jugador prototipo = new Jugador(nombreJugador, puestoJugador);  
    db.store(prototipo);  
    System.out.println("Almacenado " + prototipo);  
}
```

```
static void recuperarTodas() {  
    Jugador auxJugador = new Jugador ();  
    ObjectSet<Jugador> result = db.queryByExample(auxJugador);  
    listResult(result);  
}
```

Obtener en un  
ObjectSet todos  
los objetos  
almacenados en la  
base de datos

```
static void listResult(ObjectSet result) {  
    System.out.println("Existen " + result.size() + " objetos");  
    while (result.hasNext()) {  
        Jugador aux = (Jugador) result.next();  
        System.out.println(aux.toString());  
    }  
}
```

Recorrer el  
ObjectSet y  
mostrar la  
información  
por pantalla

# Db4o. Uso II

Consultar por *nombre*

```
static void consultaPorNombre(String nombre) {  
    Jugador prototipo = new Jugador(nombre);  
  
    ObjectSet<Jugador> result = db.queryByExample(prototipo);  
  
    listResult(result);  
}
```

Eliminar objetos

```
static void eliminarObjetos() {  
    ObjectSet<Jugador> result = db.queryByExample(new Jugador("Otilia"));  
    Jugador found = (Jugador) result.next();  
  
    db.delete(found);  
  
    System.out.println("Eliminado " + found.toString);  
    recuperarTodas();  
}
```



Implementar un  
proyecto que realice  
las operaciones  
indicadas

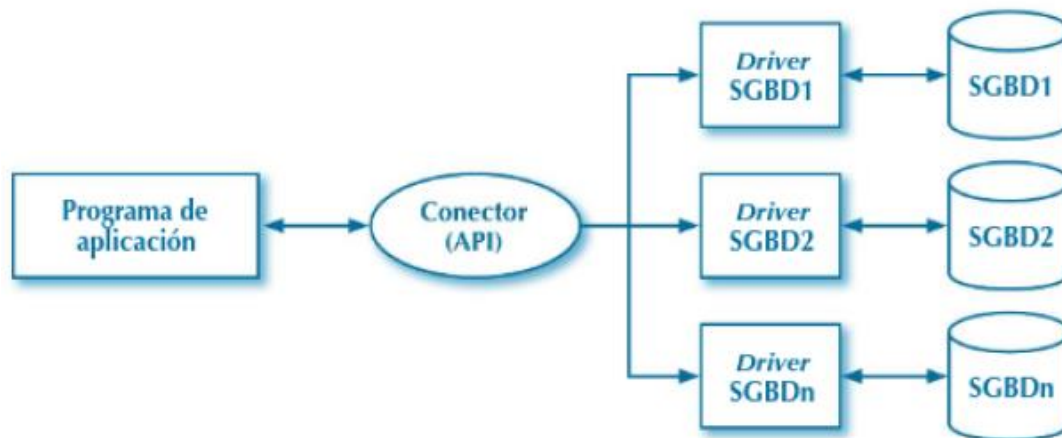
# CONECTORES. Introducción

- Los sistemas gestores de bases de datos, **SGBD**, tienen sus propios lenguajes y mecanismos de operación con las bases de datos que gestionan.
- Para poder manipular, desde Java, una base de datos debemos instalar una librería que realice la función de **conector**.
- Los conectores aportan una **API** que puede ser utilizada desde un lenguaje de programación para interactuar con una base de datos.
- Cada tipo de base de datos tiene su conector específico.



# Conectores de bases de datos relacionales. Drivers

- Existen multitud de bases de datos relacionales que utilizan lenguaje SQL.
- Cada base de datos tiene su versión de SQL y sus propias interfaces a bajo nivel.
- Mediante el uso de *drivers*, que tratan las particularidades de cada SGBD, se pueden usar conectores que aportan a los lenguajes de programación una interfaz uniforme.



# Conectores de bases de datos relacionales. Arquitectura

- El conector de una base de datos relacional, además de una API, incluye una arquitectura que enlaza con un driver que accede a la base de datos específica.
- Las arquitecturas más utilizadas son :
  - **ODBC** (*Open Database Connectivity*): API para distintos lenguajes.
  - **JDBC** (*Java Database Connectivity*): API para Java.
- Existen otras arquitecturas de acceso a datos como
  - **OLE-DB** (*Object Linking and Embedding or Databases*) de Microsoft que permite trabajar con distintos orígenes de datos,
  - **ADO** (*Active Data Objects*) creada por Microsoft que permite ser utilizada desde lenguajes de script como VBScript o Jscript.
- Los conectores que no se basan en drivers necesitan una librería específica para cada base de datos/sistema. Por ejemplo, el entorno servidor *LAMP* (Linux, Apache, MySQL, PHP) permite ejecutar código PHP en un módulo de un navegador Web Apache, mediante una extensión MySQLi que proporciona una API de acceso directo a bases de datos MySQL. Este sistema permite el acceso a las bases de datos desde entornos Web.

# ODBC

- **ODBC proporciona acceso a cualquier base de datos desde cualquier aplicación, sin importar el SGBD que lo almacena.**
- Existen conectores **OBDC** para hojas de cálculo, ficheros CSV y almacenes XML.
- **Cada SGBD compatible con ODBC** proporciona una **librería** para permitir la **conexión con el programa cliente**. Desde el programa cliente se realizan llamadas a la librería correspondiente y ésta es la encargada de acceder al *SGBD* y realizar las operaciones demandadas y devolverlas al programa.
- Existe una gran variedad de drivers **ODBC** para facilitar el acceso a distintos tipos de bases de datos. Se puede consultar en <http://www.databasedrivers.com/odbc/>.
- **OBDC** fue creado por **Microsoft** y se usa tanto en **Windows** como en **Linux**.
- Los pasos para efectuar una operación con una base de datos son los siguientes:
  1. Configurar la interfaz ODBC definiendo un *handler*.
  2. Abrir la conexión con la BD.
  3. Enviar las consultas SQL *queries*.
  4. Desconexión de la base de datos



# JDBC

- La API **JDBC** **está incluida en el paquete JDK** y proporciona una **librería estándar para el acceso a fuentes de datos**, que son en su mayoría, bases de datos relacionales.
- Existen conectores **JDBC** para hojas de cálculo, ficheros CSV y almacenes XML.
- Posee una API de aplicaciones y una arquitectura estándar que los fabricantes tienen en cuenta a la hora de implementar los drivers específicos de cada SGBD.
- *Existe un driver **JDBC** para ODBC que se puede utilizar cuando no se dispone de un driver propio **JDBC** pero si de un driver ODBC para una fuente de datos concreta.*
- Los pasos para efectuar una operación con una base de datos son los siguientes son similares a los de ODBC.
- Enlaces para ampliar información:
  - [Bases de datos soportadas por JDBC.](#)
  - [Tutoriales JDBC de Oracle.](#)



# JDBC. Ejecutar órdenes SQL

- La API de **JDBC** está disponible en el paquete **java.sql**.
- Se podrán ejecutar operaciones propias de **SQL**:
  - **DML** (*Data Manipulation Language*):
    - **SELECT** se ejecuta con **executeQuery()**, que devuelve una lista de filas en un **ResultSet**, sobre el que se puede iterar para obtener los resultados uno a uno.
    - **UPDATE**, **DELETE** e **INSERT** se ejecutan con **executeUpdate**, que devuelve el número de filas afectadas por la operación.
  - **DDL** (*Data Definition Language*):
    - Se ejecutan con **execute()**.

Crear conexión	<code>Connection laConexion = DriverManager.getConnection(cadenaConexion);</code>
Crear objeto para ejecutar sentencia SQL	<code>Statement laSentencia = laConexion.createStatement();</code>
Ejecutar sentencia DML	<pre>ResultSet elResultSet = laSentencia.executeQuery(laOrdenSql); //Siendo laOrdenSql una instrucción SQL de tipo SELECT u otra DML  while (elResultSet.next()){     ... } elResultSet.close();</pre>
Cerrar Statement	<code>laSentencia.close();</code>
Cerrar Connection	<code>laConexion.close();</code>

# JDBC: Objeto Connection

- Cada objeto **Connection** representa una conexión física con la base de datos.
- Se pueden especificar más propiedades además del usuario y la contraseña al crear una conexión.
- Estas propiedades se pueden especificar:
- Codificadas en la URL Usando métodos **getConnection(...)** sobrecargados de la clase **DriverManager**

## Ejemplo 1

```
String url = "jdbc:mysql://localhost:3306/sample";
String name = "root";
String password = "pass";
Connection c = DriverManager.getConnection(url, user, password);
```

## Ejemplo 2

```
String url = "jdbc:mysql://localhost:3306/sample?user=root&password=pass";
Connection c = DriverManager.getConnection(url);
```

## Ejemplo 3

```
String url = "jdbc:mysql://localhost:3306/sample";
Properties prop = new Properties();
prop.setProperty("user", "root");
prop.setProperty("password", "pass");
Connection c = DriverManager.getConnection(url, prop);
```

# Sentencias SQL I

- Con **JDBC** se pueden usar diferentes tipos de **Statement**.

- **Statement.**

- SQL estático en tiempo de ejecución, no acepta parámetros

```
Statement stmt = conn.createStatement();
```

- **PreparedStatement**

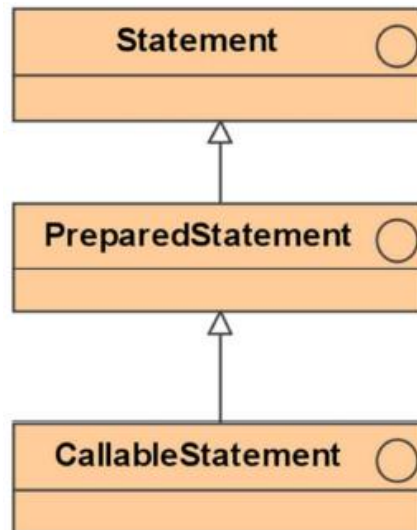
- Para ejecutar la misma sentencia muchas veces la “prepara”.
  - Acepta parámetros

```
PreparedStatement ps = conn.prepareStatement(...);
```

- **CallableStatement**

- Llamadas a procedimientos almacenados.

```
CallableStatement s = conn.prepareCall(...);
```



# Sentencias SQL II

- **Uso de Statement:** Tiene diferentes métodos para ejecutar una sentencia.
  - **executeQuery(...).** Se usa para sentencias **SELECT**. Devuelve un **ResultSet**
  - **executeUpdate(...).** Se usa para sentencias **INSERT**, **UPDATE**, **DELETE** o sentencias **DDL**. Devuelve el número de filas afectadas por la sentencia.
  - **execute(...).** Método genérico de ejecución de consultas. Puede devolver uno o más **ResultSet** y uno o más contadores de filas afectadas.

- **Uso de PreparedStatement:** Los PreparedStatement se utilizan:

- Cuando se requieren parámetros.
- Cuando se ejecuta muchas veces la misma sentencia:
  - La sentencia se prepara al crear el objeto.
  - Puede llamarse varias veces a los métodos **execute**.

```
PreparedStatement ps = conn.prepareStatement("INSERT INTO
Libros VALUES (?, ?, ?)");
ps.setInt(1, 23);
ps.setString(2, "Bambi");
ps.setInt(3, 45);
ps.executeUpdate();
```

- **Uso de CallableStatement:** Permite hacer llamadas a los procedimientos almacenados de la base de datos.

```
CallableStatement cstmt = conn.prepareCall("{call getEmpName (?,
?)}");
cstmt.setInt(1, 111111111);
cstmt.registerOutParameter(2, java.sql.Types.VARCHAR);
cstmt.execute();
String empName = cstmt.getString(2);
```

# ResultSet I

- Un **ResultSet** es una clase Java que accede al resultado de una consulta a una Base de Datos.
- El **ResultSet** no almacena todos los datos de la consulta, los va leyendo a medida que se solicitan.
- El **ResultSet** Tiene una estructura similar a una tabla de una base de datos.
- El siguiente ejemplo devuelve de recorrido de un **ResultSet** que es el resultado de buscar un *libro* en la base de datos que tiene los siguientes campos: *titulo* y *precio*.

```
while (elResultSet.next()) {  
    resultado += " TITULO: " + elResultSet.getString("TITULO");  
    resultado += " PRECIO: " + elResultSet.getDouble("PRECIO");  
    resultado += "\n";  
}
```

Si conocemos el nombre de las columnas

```
while (resul.next()) {  
    for (int i = 1; i <= resul.getMetaData().getColumnCount(); i++) {  
        System.out.printf("%s ", resul.getString(i));  
    }  
}
```

Si NO conocemos el nombre de las columnas

No se puede saber el nº de filas de un ResultSet antes de recorrerlo.

# ResultSet. Metadatos

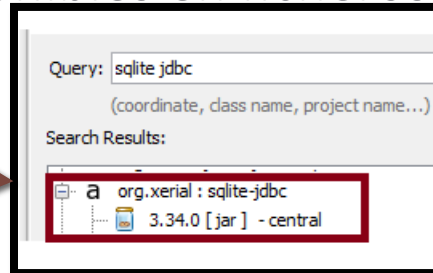
- Los metadatos nos aportan información de la estructura de las tablas de la base de datos, así tenemos:
  - **getColumnCount()**, que nos dice el número de columnas que hay en la consulta.
  - **columnName(id)**, que dado un **id** de columna nos devuelve su nombre.
- *Si no conocemos el nombre de los campos devueltos por un **ResultSet** podemos utilizar el siguiente código:*

```
static void mostrarCamposTabla(String nombreTabla, Connection miConexion)
    throws SQLException {
    try (Statement elStatement = miConexion.createStatement()) {
        String sql = "SELECT * FROM " + nombreTabla;
        try (ResultSet resul = elStatement.executeQuery(sql)) {
            // Recorremos todas las columnas(campos) del ResultSet y
            //mostramos los nombre de los campos
            for (int i = 1; i <= resul.getMetaData().getColumnCount(); i++) {
                System.out.print(resul.getMetaData().getColumnName(i) + "\t");
            }
        }
    }
}
```

# JDBC. Apertura y cierre de conexiones. Excepciones

- Los drivers de **JDBC** están disponibles en ficheros de tipo JAR.

- En Maven, debemos añadir la dependencia mediante










- Las conexiones se pueden establecer de dos formas:
  - Mediante la clase **DriverManager** y su método **getConnection** (**String URL\_conexión**). Este método carga automáticamente el driver correspondiente al proyecto. La cadena de conexión será diferente según el tipo de base de datos con la que se conecte.
  - En el caso de el driver no se carga automáticamente en memoria debemos cargarlo de forma manual con el método: **Class.forName(nombre\_\_de\_la\_clase)**.
- Para las clases que necesitan ser cerradas (*Connection*, *Statement* y *ResultSet*) podemos utilizar **try/con recursos** para capturar la excepción **SQLException**, que es la que se puede producir.

# JDBC. Conexión SQLite

- Analizar el proyecto <https://github.com/estherff/AD-UD2-Conectores-SQLite>
- Características del proyecto:
  - Se abre la conexión utilizando el patrón **Singleton** que garantiza que la instancia de la conexión con la base de datos será única en todo el proyecto.
  - Se utiliza **try/con\_recursos** para evitar tener que gestionar el cierre de los distintos objetos.
  - Se envían el control de todas las excepciones al método **main**. Para ello utiliza la sentencia **throws** en los distintos métodos en que se capturan excepciones.
  - En el menú del proyecto se utiliza una excepción propia, *NumeroFueraRangoException*, que se lanza cuando el número que se selecciona del menú está fuera de rango.





```
public class ConexionSingleton {

    private String cadenaConexion;
    private Connection laConexion;

    private static ConexionSingleton laConexionSingelton;

    /**
     * Devuelve una conexión de la base de datos y garantiza que solo existirá una
     * instancia de la misma.
     *
     * @return una instancia de la conexión a una base de datos
     */
    public static Connection getConnection(String cadenaConexion) throws SQLException {
        if (Objects.isNull(laConexionSingelton)) {
            laConexionSingelton = new ConexionSingleton(cadenaConexion);
        }
        return laConexionSingelton.laConexion;
    }

    /**
     * Constructor privado que es llamado por el método público y estático de la
     * clase para garantizar la existencia de una única instancia de la misma.
     * @param url
     * @param baseDatos
     */
    private ConexionSingleton (String cadenaConexion) throws SQLException {
        this.cadenaConexion = cadenaConexion;
        this.laConexion = DriverManager.getConnection (cadenaConexion);
    }
}
```

# Patrón Singleton. Crear conexión

- El patrón **Singleton** (*instancia única*) está diseñado para limitar la creación de objetos a una instancia única.
- El objetivo del patrón **Singleton** es garantizar que una clase sólo tenga una instancia y proporcionar un punto global a la misma.
- Usaremos el patrón **Singleton** para crear una única instancia de la conexión de la base de datos.
- Las ideas principales del patrón **Singleton** son:
  - Crear un **constructor privado**.

```
private ConexionSingleton (String cadenaConexion) throws SQLException {  
    this.cadenaConexion = cadenaConexion;  
    this.laConexion = DriverManager.getConnection (cadenaConexion);  
}
```

- Crea un **objeto estático** de la propia clase.

```
private static ConexionSingleton laConexionSingelton;
```

- Crear un **método público estático** llamado **getNombreInstancia()** con las siguientes características:
  - Devuelve una instancia del objeto estático de la propia clase.
  - Llama al constructor en el caso de que la instancia sea **null**, en caso contrario devuelve la instancia existente.

```
public static Connection getConnection(String cadenaConexion)  
    throws SQLException {  
    if (Objects.isNull(laConexionSingelton)) {  
        laConexionSingelton = new ConexionSingleton(cadenaConexion);  
    }  
    return laConexionSingelton.laConexion;  
}
```

# Patrón Singleton. Abrir conexión

- El método **getConnection** () de la clase **ConexionSingleton** devuelve una conexión a la base de datos en función de la cadena de conexión establecida.
- Tenemos la garantía de que siempre existirá solamente una conexión abierta.

```
String cadenaConexion = "jdbc:sqlite:"D:\\SQLite\\matriculas.db";

//Establece la conexión con una base de datos SQLite.
//No es necesario cerrar la conexión al usar try con recursos
try (Connection laConexion = ConexionSingleton.getConnection(cadenaConexion)) {

    //...instrucciones de manejo de la base de datos
} catch (SQLException e) {
    System.out.println("Se ha producido una excepcion " + e.getMessage()
        + " El programa se cerrará. /n Consulte con el responsable");
}
```

# JDBC. Conexión SQLite. Práctica 1

- Añade al proyecto ya existente funcionalidad para:
  - Añadir un registro a una tabla.
  - Buscar por tipo de campo en una tabla.

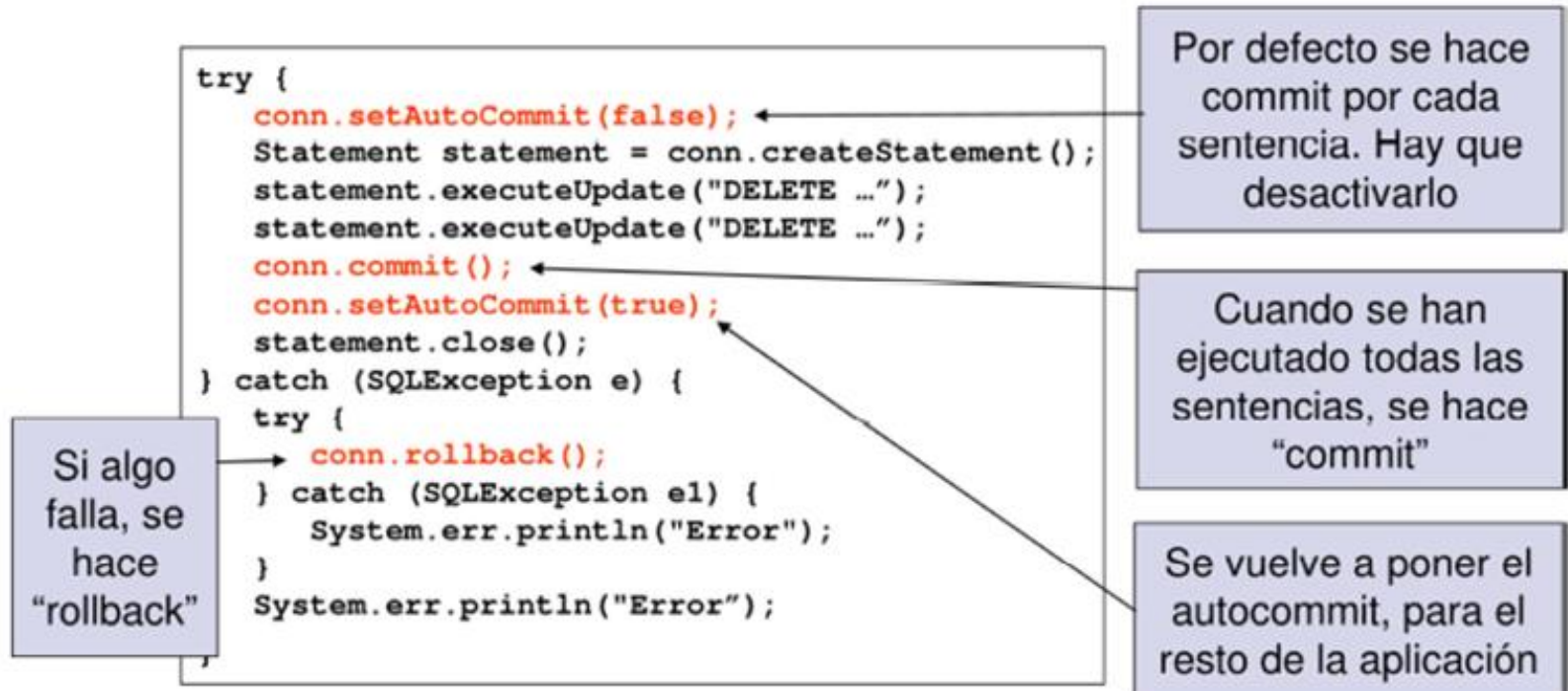
# Transacciones

- Las transacciones tratan un conjunto de sentencias como un bloque, de tal forma que si falla una sentencia del grupo, la ejecución del programa vuelve al punto de inicio del bloque.
- Las instrucciones que se incluyen en una transacción o se ejecutan todas o ninguna.
- Las transacciones permiten proteger la integridad de la base de datos garantizando, por ejemplo, que al agregar un registro de una tabla que está relacionada con otra, se agregue la información adecuada en las dos tablas correctamente o no se realice la operación.
- **Por ejemplo**, si vamos al banco y ordenamos una transferencia para pagar una compra que hemos realizado por Internet, el proceso en sí está formado por un conjunto (o bloque) de operaciones que deben ser realizadas para que la operación global tenga éxito:
  1. Comprobar que nuestra cuenta existe es válida y está operativa.
  2. Comprobar si hay saldo en nuestra cuenta.
  3. Comprobar los datos de la cuenta del vendedor (que existe, que tiene posibilidad de recibir dinero, etc...).
  4. Retirar el dinero de nuestra cuenta
  5. Ingresar el dinero en la cuenta del vendedor.
  - Dentro de este proceso hay cinco operaciones, las cuales deben ejecutarse todas correctamente o la operación no es válida.

# Transacciones en Java

- Los siguientes métodos de la interfaz **Connection** son utilizados para gestionar las **transacciones** en una base de datos.
  - **void setAutoCommit(boolean valor)**: Para iniciar una transacción deshabilitamos el modo auto-commit mediante el método **setAutoCommit(false)**. Esto nos da el control sobre lo que se realiza y cuándo se realiza.
  - **void commit()**: Una llamada al método **commit()** realizará todas las instrucciones emitidas desde la última vez que se invocó el método **commit()** o desde que deshabilito el modo auto-commit con **setAutoCommit(false)**.
  - **void rollback()**: Una llamada a **rollback()** deshacerá todos los cambios realizados desde el último **commit()**. Una vez se ha emitido una instrucción **commit()**, esas transacciones no pueden deshacerse con **rollback()**.

# Transacciones. Ejemplo





# Transacciones. Práctica II

- Crear una clase **Contactos** que sea una réplica de la estructura de la base de datos de **Contactos**.
- Introducir una opción en el proyecto que permita **introducir varios *contactos* y almacenarlos en un **ArrayList** de la clase **Contactos****. Al finalizar la operación, se deberán almacenar los contactos almacenados en el **ArrayList** en la base de datos.
- La operación de añadir varios registros a la base de datos deberá estar dentro de una **transacción**.



# Práctica III

- Crear una base de datos H2 llamada BIBLIOTECA que contenga una tabla llamada LIBROS cuya orden SQL de creación es:

```
CREATE TABLE LIBROS
```

```
(IDLIBRO INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
TITULO VARCHAR(50) NOT NULL,  
AUTOR VARCHAR(50) NOT NULL,  
PRECIO FLOAT);
```

- La tabla debe ser creada mediante la instrucción anterior.
- El programa deberá tener un menú con las siguientes opciones:

```
1.- CREAR BASE DE DATOS  
2.- ALTAS  
3.- BAJAS  
4.- CONSULTAS  
5.- FINALIZAR
```

```
1.- Ver Libro por TITULO  
2.- Ver Libro por ID  
3.- Ver Libros por AUTOR  
4.- Ver Libros por PRECIO  
5.- Ver todos los libros  
6.- Finalizar
```

En las **ALTAS** debemos dar la posibilidad de introducir varios libros.  
Crear un **ArrayList** de una clase **Libros** y usar **transacciones**