Ejercicios Java JDBC

Contenido

Creación de bases de datos en SQLite 3

Ayuda y referencias 5

Creación de la base de datos en H2 6

Ejercicio 1. Gestión de Filósofos 13

Para H2 Embedded 14

Métodos y constructores 16

Ejercicio 2. Gestión de Filósofos (II) 17

Atributos (nuevos) 18

Métodos (nuevos) 18

Ejercicio 3. Gestión BIblioteca. ..... **Erro Marcador non definido.**

Clase **Book** implementa ***Serializable*** ..... **Erro Marcador non definido.**

Constructores ..... **Erro Marcador non definido.**

Atributos ..... **Erro Marcador non definido.**

Métodos ..... **Erro Marcador non definido.**

BookDAO ..... **Erro Marcador non definido.**

Ejercicio 4. Usuarios y productos. 19

Ejercicio 5. Gestión de vacunas 20

Clases del modelo 20

Persoa.java 20

Vacinacion.java 21

Clases DAO 21

ConnectionDB.java 21

PersoaDAO.java 21

VacinacionDAO.java 21

Clases Vista (Opcional) 22

Ejercicio 6. Paint con BASE DE DATOS 22

Clases del modelo 23

Ruta (Serializable) 23

Debuxo (Serializable) 25

Clases DAO 25
ConnectionDB 25
DebuxoDAO 26
AppDebuxo 26

Creación de bases de datos en SQLite

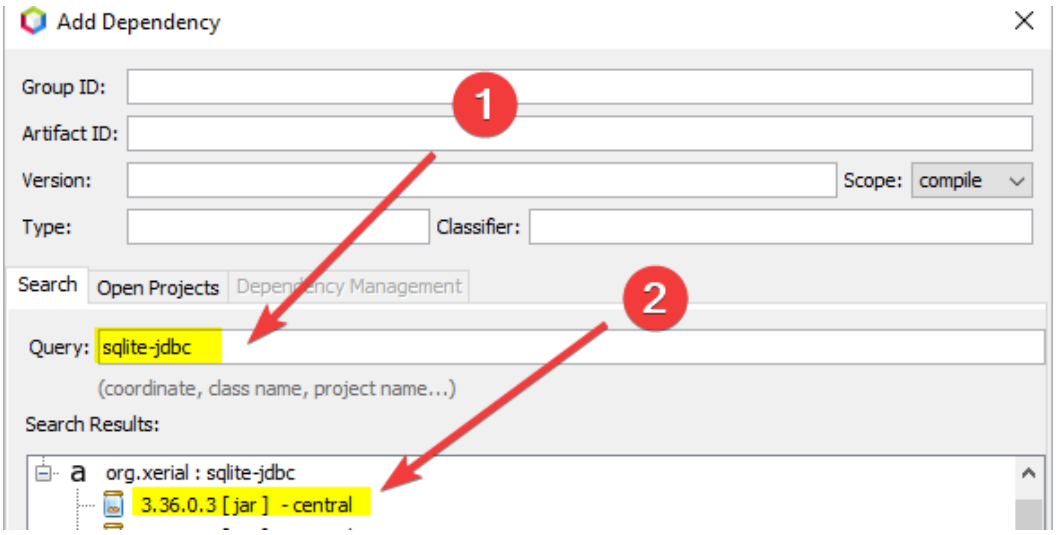
Para facilitar el trabajo de aplicaciones sencillas, existen muchos SGBD relacionales orientados a archivo *opensource* como **H2**, **SQLite**, HSQL, tinySQL, smallSQL o comerciales:

- SQLite : https://sqlite.org/index.html
- HSQLDB: https://hsqldb.org/ (HyperSQL database management system)
- H2Database
- MariaDB
- PostgreSQL
- Derby
- tinySQL: http://priede.bf.lu.lv/ftp/pub/DatuBazes/tinySQL/tinySQL.htm
- SmallSQL: http://www.smallsql.de/
- Microsoft SQL Server
- Oracle

Uno de los SGBD más empleados, sobre todo en dispositivos móviles, es **SQLLite** (https://sqlite.org/index.html).

Como trabajaremos con **dependencias a los Drivers JDBC**, cuyo **archivo *jar*** precisamos en nuestro proyecto y en el ***classpath*** de ejecución/compilación, recomendaría realizar **un proyecto Maven** en Netbeans, aunque **podría descargarse el driver JDBC de SQLite** y añadirse como biblioteca al proyecto Java.

Para trabajar con *SQLite* se precisa tener añadida la **dependencia con los Driver JDBC de SQLite**:



Puede hacerse a mano en el propio archivo pom.xml:

```
<dependencies>
    <dependency>
      <groupId>org.xerial</groupId>
      <artifactId>sqlite-jdbc</artifactId>
      <version>3.43.2.2</version>
    </dependency>
  </dependencies>
```

Los drives JDBC del *SQLite* pueden descargarse de:

https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc (Repositorio Maven)

o

https://github.com/xerial/sqlite-jdbc/releases

Podéis ver ejemplo de uso en la página:

https://github.com/xerial/sqlite-jdbc#usage

Se puede trabajar tanto con **datos en memoria** (durante la ejecución del programa) **como en archivo**:

Bases de datos en memoria

```
Connection conex = DriverManager.getConnection("jdbc:sqlite::memory:");
```

En archivo

```
Connection conex = DriverManager.getConnection("jdbc:sqlite:rutaArchivo.db");
```

El hecho de realizarlo en SQLite da **portabilidad al proyecto**, pues este SGBDR es orientado a archivo y **no precisa estar instalado como servicio en ningún computador**.

Como dice en la página del proyecto:

*«SQLite es una biblioteca "en proceso" que implanta un **motor de bases de datos SQL** autónomo, **sin servidor**, sin configuración y transaccional. El código de SQLite es de **dominio público** y, por lo tanto, es gratuito para cualquier propósito, comercial o privado. SQLite es la **base de datos más implementada en el mundo** con más aplicaciones de las que podemos contar, incluidos varios proyectos de alto perfil.»*

«SQLite es un motor de base de datos SQL incorporado ("embebido"). A diferencia de la mayoría de las otras bases de datos SQL, SQLite no tiene un proceso de servidor separado. SQLite ***lee y escribe directamente en archivos de disco normales****. Una base de datos SQL completa con múltiples tablas, índices, disparadores y vistas* ***está contenida en un solo archivo de disco****. El formato de archivo de la base de datos es multiplataforma: se puede copiar libremente una base de datos entre sistemas de 32 y 64 bits o entre arquitecturas big-endian y little-endian. Estas características hacen de SQLite una opción* ***popular para emplearlas en formato de archivo de aplicaciones*** *(formato usado para aplicaciones que requieran persistencia de datos de archivo o para intercambiar información entre programas). Los archivos de base de datos SQLite son un formato de almacenamiento recomendado por la Biblioteca del Congreso de EE. UU. Piense en SQLite no como una sustitución de Oracle pero como sí como sustitución de fopen () (apertura de archivos)»*

"SQLite es el motor de base de datos más utilizado del mundo. SQLite ***está integrado en todos los teléfonos móviles y la mayoría de las computadoras y viene incluido dentro de innumerables otras aplicaciones que la gente usa todos los días****"*

Ayuda y referencias

https://docs.oracle.com/javase/tutorial/uiswing/components/table.html

SQLite:
https://sqlite.org/index.html
https://sqlitebrowser.org/dl/

DAO:
http://acodigo.blogspot.com/2016/03/data-access-object-dao-con-jdbc.html
https://www.oracle.com/java/technologies/dataaccessobject.html
https://www.youtube.com/watch?v=CEDKxPCgosY
https://www.youtube.com/watch?v=NjY-WA-jeJ8

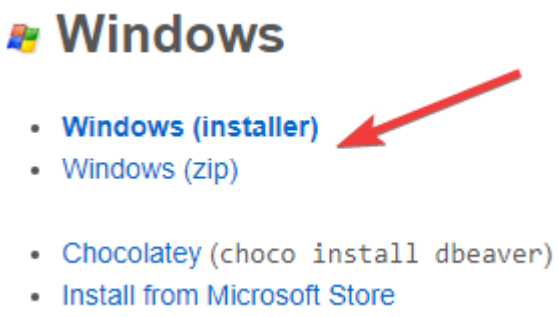
Creación de la base de datos en H2

Para la creación de la BD podemos emplear el propio Netbeans, HeidiSQL, **DBeaver** (recomendación personal en este caso) o similar.

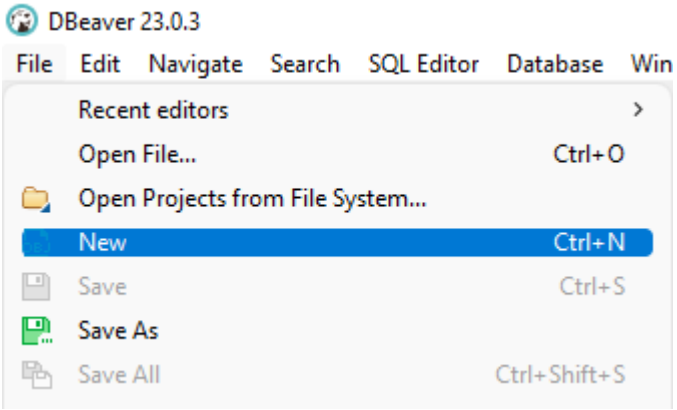
En este caso tan concreto, recomendaría usar, DBeaver, pues permite gestionar muchas bases de datos (prácticamente todas las que disponen de Drivers JDBC, pues este programa está escrito en Java):

https://dbeaver.io/

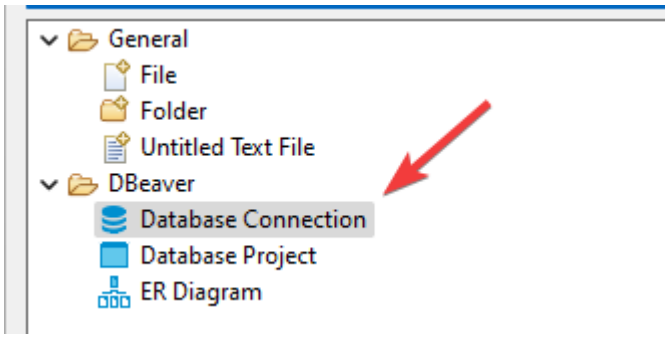
https://dbeaver.io/download/



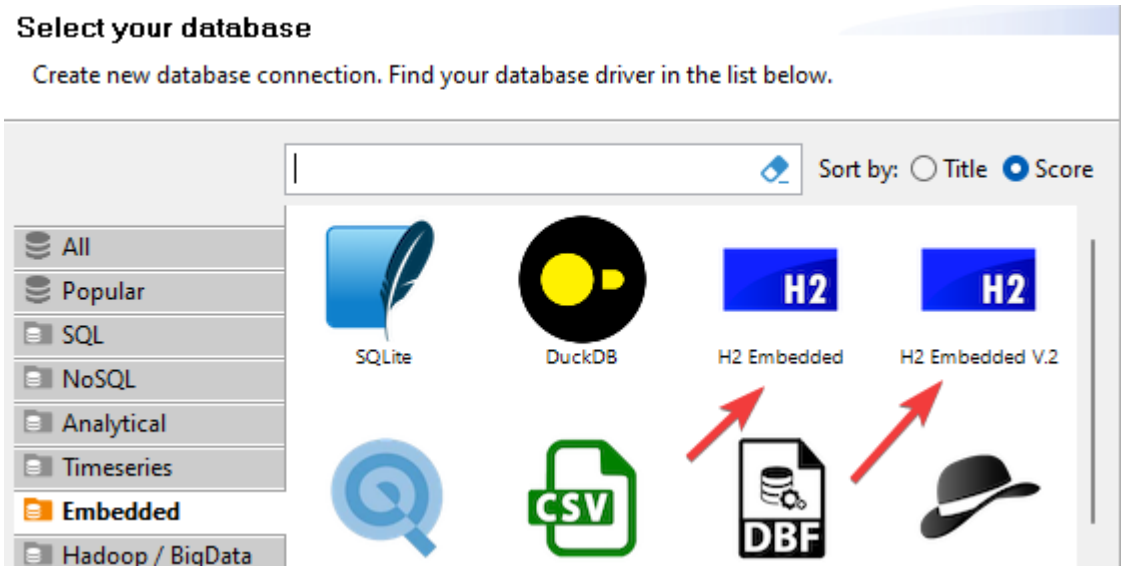
Es importante crear la base de datos y usar la biblioteca de la aplicación con el mismo formato 1.X o 2.X:

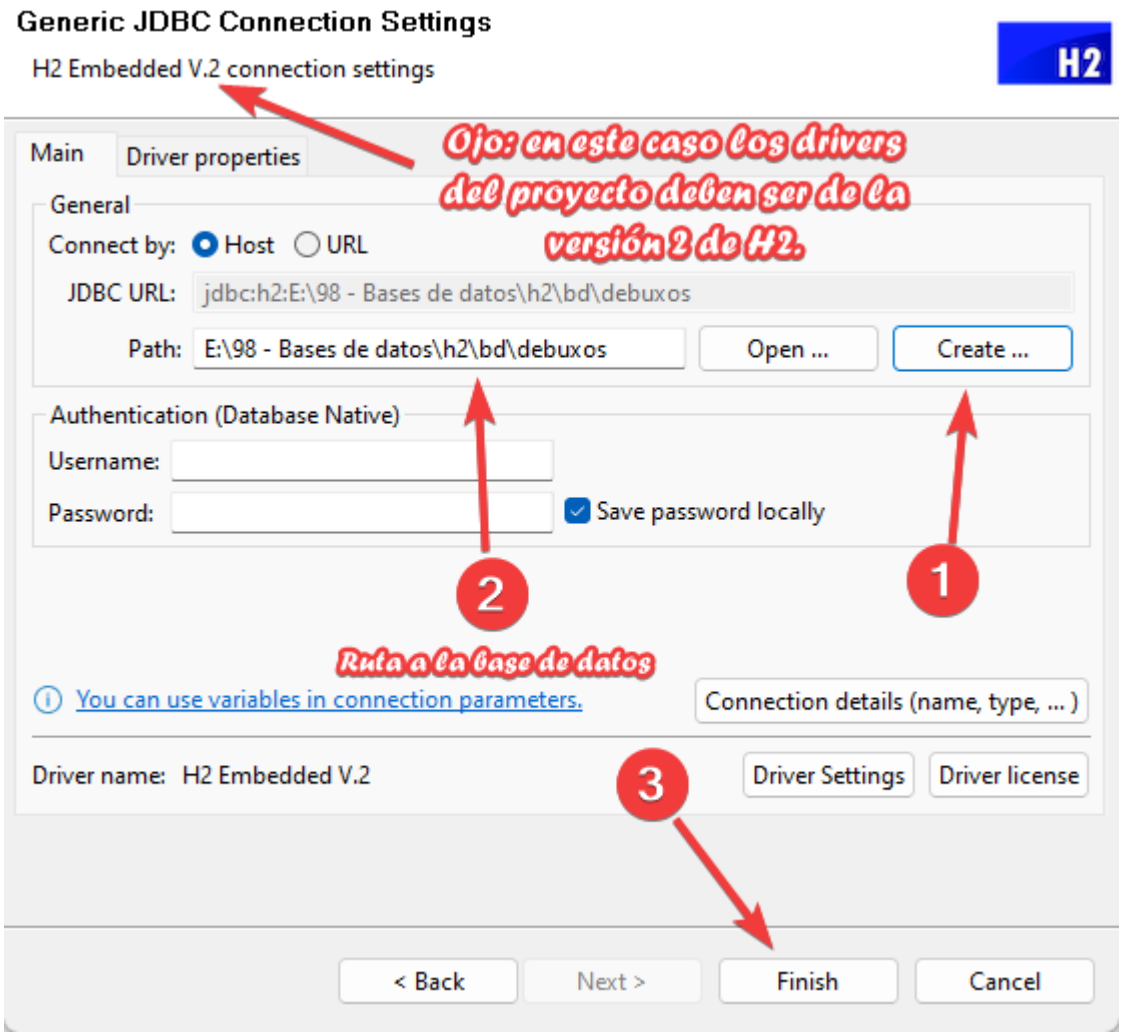


Creamos una nueva conexión:



Seleccionamos el base de datos H2 1.X o la versión 2.X, en cuyo caso debemos añadir las dependencias del proyecto con la versión correspondiente:





Para la versión 2:

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>2.1.214</version>
</dependency>
```

Para la versión 1:

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>1.4.190</version>
</dependency>
```

Por ejemplo, puedes crear la base de datos con de acuerdo con el script SQL o por medio de la interface gráfica (no entraré en detalles):

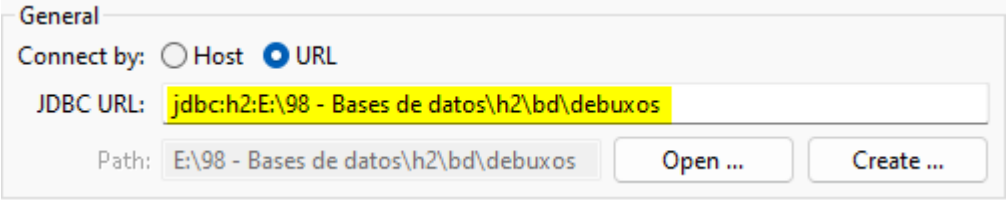
```
CREATE TABLE PUBLIC.Debuxo (
      idDebuxo INTEGER NOT NULL AUTO_INCREMENT,
      nome CHARACTER VARYING(64) NOT NULL,
      CONSTRAINT DEBUXO_PK PRIMARY KEY (idDebuxo)
);
CREATE INDEX DEBUXO_NOME_IDX ON PUBLIC.DEBUXO (nome);
COMMENT ON TABLE PUBLIC.DEBUXO IS 'Debuxo da base de datos composto
por figuras.';
COMMENT ON COLUMN PUBLIC.DEBUXO. idDebuxo IS 'Clave primaria';
COMMENT ON COLUMN PUBLIC.DEBUXO.nome IS 'Nome do debuxo';

CREATE TABLE PUBLIC.Shape (
      idDebuxo INTEGER NOT NULL,
      shape BINARY LARGE OBJECT,
      CONSTRAINT SHAPE_FK FOREIGN KEY (idDebuxo) REFERENCES
PUBLIC.Debuxo(idDebuxo) ON DELETE CASCADE ON UPDATE CASCADE
);
CREATE INDEX SHAPE_IDDEBUXO_IDX ON PUBLIC.SHAPE (idDebuxo);
COMMENT ON TABLE PUBLIC.shape IS 'Figuras de dibujo';
COMMENT ON COLUMN PUBLIC.Shape. idDebuxo IS 'Referencia ó debuxo';
COMMENT ON COLUMN PUBLIC.Shape.shape IS 'BLOB con el objeto de la
figura';
```

En el ejemplo de BD se emplea un tipo dato BLOB (binario grande), para guardar un objeto binario en la base de datos.

La configuración de la URL a la base de datos es la que aparece en las propiedades de la conexión:

*"**jdbc:h2:**RutaABaseDatos**\debuxos**"*



Los parámetros de la conexión deben ser:

JDBC_DRIVER = "***org.h2.Driver***"; *// No se precisa en JDBC versión mayor a 4.0*
DB_URL = *"**jdbc:h2:**RutaABaseDatos**\nomeBD**"*;

Ahora, podemos proceder como cualquier otro proyecto de conexión a base de datos empleando los dos parámetros (el primero no se precisa desde JDBC 4.0).

Ejercicio 1. Gestión BIblioteca

Queremos desarrollar una aplicación para una biblioteca y necesitamos interactuar con una base de datos que contiene información sobre los libros que tenemos en nuestra colección de libros.

Para ello, vamos a crear una clase **Book** que represente la entidad libro y otra clase ***BookDAO*** que nos permita realizar operaciones CRUD (Create, Read, Update y Delete) sobre la tabla **Book** en la base de datos.

Además, precisamos una clase ***ConnectionManager*** para la gestión y obtención de las conexiones a la base de datos de una manera eficiente. Emplearemos el patrón Singleton para el gestor, que en la primera versión tendrá una única conexión, pero que podremos convertir en un conjunto/pool de conexiones.

Estructura de la base de datos

Está formada por una única tabla, ***Book***. La tabla **Book** tiene la siguiente estructura:

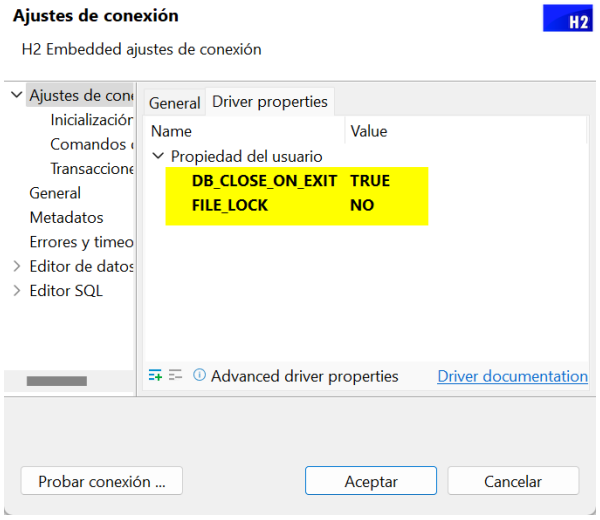
Columna	Tipo de dato	Descripción
idBook	int	Identificador único del **ejemplar** del libro
isbn	varchar(13)	Identificador del libro
title	varchar(100)	Título del libro
author	varchar(100)	Autor del libro
ano	int	Año de publicación del libro
available	boolean	Indica si el libro está disponible
portada	Blob	Portada del libro en formato binario

```
CREATE TABLE Book (
  idBook INT NOT NULL AUTO_INCREMENT,
  isbn VARCHAR(13) NOT NULL,
  title VARCHAR(255) NOT NULL,
  author VARCHAR(255) NOT NULL,
  ano INT NOT NULL,
  available BOOLEAN NOT NULL DEFAULT true,
  blob portada,
  PRIMARY KEY (id)
);
```

```
// Adapta la consulta al gestor de BD H2.
```

Para permitir el uso de nombres en ***CamelCase*** en H2 JDBC Driver versión 2, es necesario agregar la propiedad ***DATABASE_TO_UPPER=FALSE*** en la URL de conexión.

DRIVER: ***"org.h2.Driver"***
URL:
"jdbc:h2:rutaBaseDatosSinExtensión;DB_CLOSE_ON_EXIT=TRUE;FILE_LOCK=NO;DATABASE_TO_UPPER=FALSE"




```
-- PUBLIC.BOOK definition
-- Drop table
-- DROP TABLE PUBLIC.Book;

CREATE TABLE PUBLIC.Book (
	idBook INTEGER NOT NULL AUTO_INCREMENT,
	isbn CHARACTER VARYING(13) NOT NULL,
	titulo CHARACTER VARYING(255) NOT NULL,
	autor CHARACTER VARYING(255),
	anho INTEGER,
	disponible BOOLEAN DEFAULT TRUE,
	portada BINARY LARGE OBJECT,
	CONSTRAINT BOOK_PK PRIMARY KEY (idBook)
);
CREATE INDEX IdxBookISBN ON PUBLIC.Book (isbn);
CREATE INDEX IdxBookTitle ON PUBLIC.Book (titulo);
CREATE UNIQUE INDEX IdBookPK ON PUBLIC.Book (idBook);
```

ConnectionManager

Mediante el patrón Singleton crea una clase ConnectionMaganer o LIbraryConnectionMaganer 😉, con la instancia de un objeto de la propia clase como atributo privado y final. Además, debe tener un atributo privado de tipo Connection que se creará al invocar al método getConnection.

Hazlo con Thread-Safe y doble comprobación. Piensa cómo realizarlo de manera eficiente.

Para simplificar el problema inicial, crea las constantes de la URL, usuario y contraseña, preferiblemente privadas.

Clase **Book** implementa ***Serializable***

Constructores

La clase debe tener los siguientes constructores que:

- Book()
- Book(String title, String author, int year, boolean available)
- Book(String isbn, String title, String author, int year, boolean available)

Atributos

La clase **Book** debe tener los siguientes atributos:

- **idBook**: int
- **isbn**: String
- **title**: String
- **author**: String
- **year**: int
- **available**: boolean
- **portada**: byte[]

Métodos

- ***Get*** y ***set*** para cada atributo.
- ***equals*** y ***hashcode***: considerando que son iguales cuando tienen el mismo **isbn**. Además, el método *hashcode* debe devolver un valor coherente con el método *equals* (**todos los objetos iguales deben tener, al menos el mismo *hashCode***).

- ***toString***: devuelve el título, el autor y el año. Si no está disponible escribe un asterisco.

ConnectionManager

BookDAO

La clase **BookDAO** debe tener los siguientes métodos:

- **create(Book book)**: crea un nuevo registro en la tabla **Book** con la información del libro pasado como parámetro.
- **get(int idBook)**: devuelve un objeto **Book** con la información del libro que tiene el identificador pasado como parámetro.
- **update(Book book)**: Actualiza la información del registro correspondiente al libro pasado como parámetro.
- **delete(int idBook)**: elimina el registro correspondiente al libro con el identificador pasado como parámetro.
- **getAll()**: devuelve una lista de todos los libros almacenados en la base de datos

Debes implantar la gestión de sentencias de esta la clase **BookDAO** por medio de try-with-resources para manejar las PreparedStatement y los ResultSet de consultas automáticamente. La conexión no debe cerrarse pues debe

Ejercicio 2. Gestión de Filósofos

Swing, eventos, excepciones y bases de datos. Realícese una aplicación (***FilosofosView***), que es un JFrame, de consulta de información de los filósofos guardados en una **base de datos embebida** (existen muchas BBDD orientadas a archivo *opensource* como HSQL, tinySQL, smallSQL...):

- SQLite : https://sqlite.org/index.html
- HSQLDB: https://hsqldb.org/ (HyperSQL database management system)
- H2Database (*)
- MariaDB
- PostgreSQL
- Derby
- tinySQL: http://priede.bf.lu.lv/ftp/pub/DatuBazes/tinySQL/tinySQL.htm

- SmallSQL: http://www.smallsql.de/
- Microsoft SQL Server
- Oracle

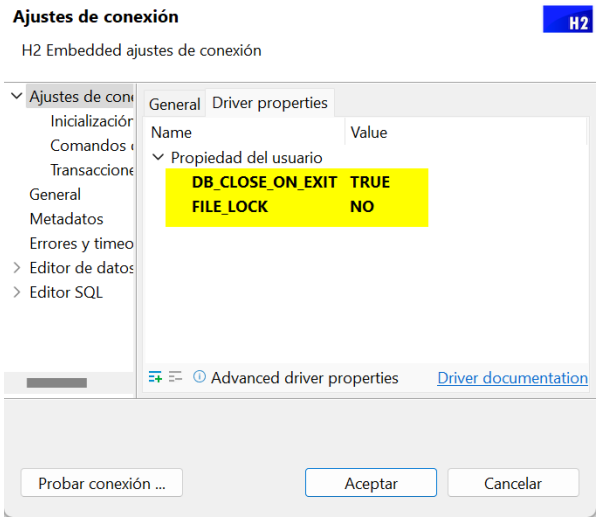
Para H2 Embedded

DRIVER: ***"org.h2.Driver"***

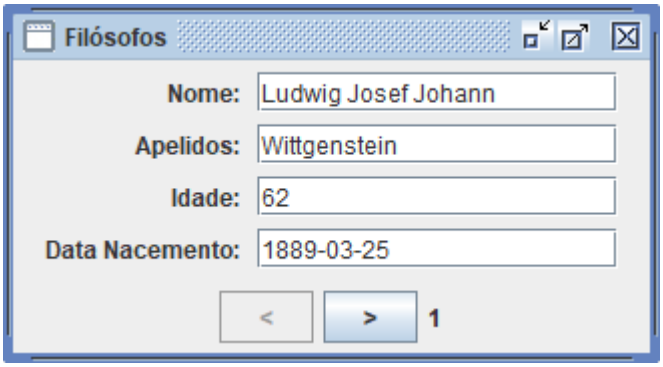
URL: ***"jdbc:h2:rutaBaseDatosSinExtensión;DB_CLOSE_ON_EXIT=TRUE;FILE_LOCK=NO"***

```
CREATE TABLE PUBLIC.FILOSOFO (
      IDFILOSOFO INTEGER NOT NULL AUTO_INCREMENT,
      NOME CHARACTER VARYING(32) NOT NULL,
      APELIDOS CHARACTER VARYING(64),
      IDADE TINYINT,
      DATANACEMENTO DATE,
      CONSTRAINT FILOSOFO_PK PRIMARY KEY (IDFILOSOFO)
);
CREATE INDEX FILOSOFO_NOME_IDX ON PUBLIC.FILOSOFO (NOME,APELIDOS);
CREATE UNIQUE INDEX PRIMARY_KEY_3 ON PUBLIC.FILOSOFO (IDFILOSOFO);
```

```
INSERT INTO PUBLIC.FILOSOFO (NOME,APELIDOS,IDADE,DATANACEMENTO) VALUES
       ('Ludwig Josef Johann','Wittgenstein',62,'1889-03-25'),
       ('Martin','Heidegger',28,'1889-09-25'),
       ('Jean-Paul','Sartre',23,'1905-06-21'),
       ('Simone','de Beauvoir',21,'1908-04-14'),
       ('Hannah','Arendt',25,'1906-10-14');
```



	IDFILOSOFO	NOME	APELIDOS	IDADE	DATANACEMENTO
1	1	Ludwig Josef Johann	Wittgenstein	62	1889-03-25
2	2	Martin	Heidegger Gadamer	28	1889-09-25
3	3	Jean-Paul	Sartre Beauvoir	23	1905-06-21
4	4	Simone	de Beauvoir Merleau-Ponty	21	1908-04-14
5	5	Hannah	Arendt Levinas	25	1906-10-14



Las constantes a la BD deben guardarse en variables estáticas y finales, DRIVER y URL.

- **Consulta**: crea 4 campos: *nome*, *apelidos*, *idade*, *dataNacemento* del tipo correspondiente.

- ***Propiedades***:
  - ***btSeguinte, btAnterior***: botones de swing que aparecen en la parte inferior de la ventana.
  - El *JLabel* que aparece al lado de los botones (***lbID***) es para mostrar el identificador del Filósofo.
  - ***campos***: array de 4 *JTextField* para mostrar los datos de la consulta.
  - ***rs***: *ResultSet* con los alumnos de la consulta. Se podrá **acceder para <u>delante y para atrás</u>**, por lo que, **<u>a la hora de crear la sentencia, *Statement*, debe hacerse cos parámetros apropiados</u>**:

https://docs.oracle.com/en/java/javase/19/docs/api/java.sql/java/sql/Connection.html#createStatement(int,int)

Connection dispone del método:

```
Statement createStatement(int resultSetType,
      int resultSetConcurrency) throws SQLException
```

En el que:

- ***resultSetType*** puede valer: `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, o `ResultSet.TYPE_SCROLL_SENSITIVE`

- ***resultSetConcurrency*** puede valer: `ResultSet.CONCUR_READ_ONLY` o `ResultSet.CONCUR_UPDATABLE`.

```
Statement st = con.createStatement(
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
```

Debe crearse en el constructor y cerrarse al salir del programa, deben cerrarse las sentencia *Statement* y la *Connection* asociadas (esta última cierra todo). Como no están establecidos como propiedades, deben obtenerse en el método que escucha el evento de salida con "*Statemet st = rs.**getStatement**()*" y "*Connection con = st.**getConnection**()*". (*Comprobadlo*)

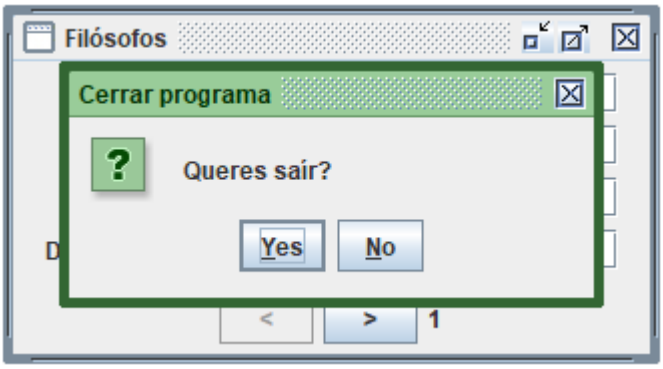
Métodos y constructores

Constructor que recoge el título de la ventana: además de construir la interface gráfica, debe realizar la conexión, la sentencia y la consulta, asignando el resultado a ***rs***. No guardaremos la referencia a la conexión ni a la consulta, sólo al cursor. **La consulta debe estar abierta durante toda la ejecución del programa** (no cierres la conexión).

La interface gráfica debe hacerse por medio de un método "***createGUI***", eso facilita a lectura y la seguridad.

El cierre de la ventana se hará con gestión de eventos, por lo que debe indicarse que la operación por defecto es no hacer nada:

*setDefaultCloseOperation(**DO_NOTHING_ON_CLOSE**);*



```java
private void sair() {
    if (JOptionPane.showConfirmDialog(this, "Queres saír?",
            "Cerrar programa", JOptionPane.YES_NO_OPTION,
            JOptionPane.QUESTION_MESSAGE
    ) == JOptionPane.YES_OPTION) {
        try {
            rs.getStatement().getConnection().close();
        } catch (SQLException ex) {

        }
        System.exit(0);
    }
}
```

- ***setValores()***: sin recoger ningún argumento, asigna los valores a los que apunta ***rs*** en ese momento a las cajas de texto. Debe lanzar una excepción, no capturarla, de tipo ***SQLException***, puesto que la obtención de los valores, puede dar lugar a excepciones. Será los métodos que hagan uso de éste los encargados de capturar la excepción:

Si el cursor no está cerrado, no es nulo y no está antes del primer registro ni después del último, debe asignar los valores a las cajas de texto y el idFilosofo a la etiqueta. Además debe habilitar o deshabilitar los botones según la posición en la que esté:

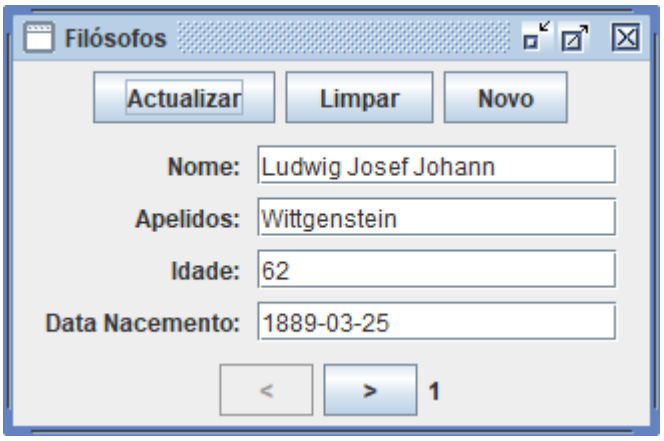
```java
btSeguinte.setEnabled(!rs.isLast());
btAnterior.setEnabled(!rs.isFirst());
```

Métodos de gestión de eventos: la propia ventana se encargará de gestionar los eventos sobre los botones y sobre el cierre de la misma, implantando las interfaces correspondientes (puedes emplear una clase anónima si lo desea). La gestión de acción sobre los botones puedes hacerlo con una expresión lambda.

*Nota: debes utilizar los métodos **next()**, **previous()**, que avanzan y retroceden el cursor; los métodos **isAfterLast()**, **isLast()**, **isBeforeFirst()** y **isFirst()**, que indican si o cursor está después del último, en el último, antes del primero o en el primer registro, respectivamente. Recuerda que al iniciar la consulta el cursor es probable que se encuentre "before first".*

Ejercicio 3. Gestión de Filósofos (II)

Realícese la aplicación ***FilososofosViewUpdate***, que es un *JFrame*, de **consulta y actualización**, **ampliación del ejercicio anterior** y que permita modificar la información almacenada en la base de datos.



Ahora el ***ResultSet*** lo abrimos en modo actualización:

```
Statement st = con.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
```

Atributos (nuevos)

- ***btActualizar, btLimpar, btNovo***: botones que aparecen en panel superior de la ventana y permiten actualizar el registro actual, limpiar la caja de texto y crear uno nuevo al final (al pulsar el botón se guarda el registro que se haya escrito y se mueve al final).

Métodos (nuevos)

- ***update(boolean isInsert)***, recoge un argumento que indica si es actualización o inserción actualizando la base de datos con lo que haya en la caja de texto.
  *Dependiendo del valor del parámetro debe actualizar o insertar la fila (previamente debe hacerse un rs.**updateXXX**(columna, valor) de los campos:*

```
if (!isInsert) {
    rs.updateRow();
} else {
    rs.insertRow();
    rs.last();
}
```

- ***clear()***, limpia las cajas de texto.

- ***Métodos y métodos de eventos***:
  - Al pulsar o botón de "btActualizar" llama al método ***update(false)***.
  - Al pulsar o botón de "btLimpar" llama al método ***clear()***.
  - Al pulsar o botón de "btNovo" llama al método ***update(true) y mueve el cursor al final***.

Ejercicio 4. Usuarios y productos.

Haz una aplicación de gestión en ***Java*** que tenga las siguientes características:

- Clase de acceso/gestión a la BD (**XestionDAO**) que está presente desde el inicio del programa ata salir.
  - *Propiedades*: conexión a la base de datos (***con***), propiedades estáticas ***DRIVER*** y ***URL*** con las cadenas de conexión a la base de datos.
  - *Métodos*:
    - ***getUser***, recoge un login y clave y devuelve el usuario (null si no existe).
    - ***getProductos***: recoge un código de marca, un precio máximo y un precio mínimo y devuelve una lista con los datos. Se los precios son negativos no devuelve nada.
    - ***getAllProductos***: devuelve todos los productos de la BD.

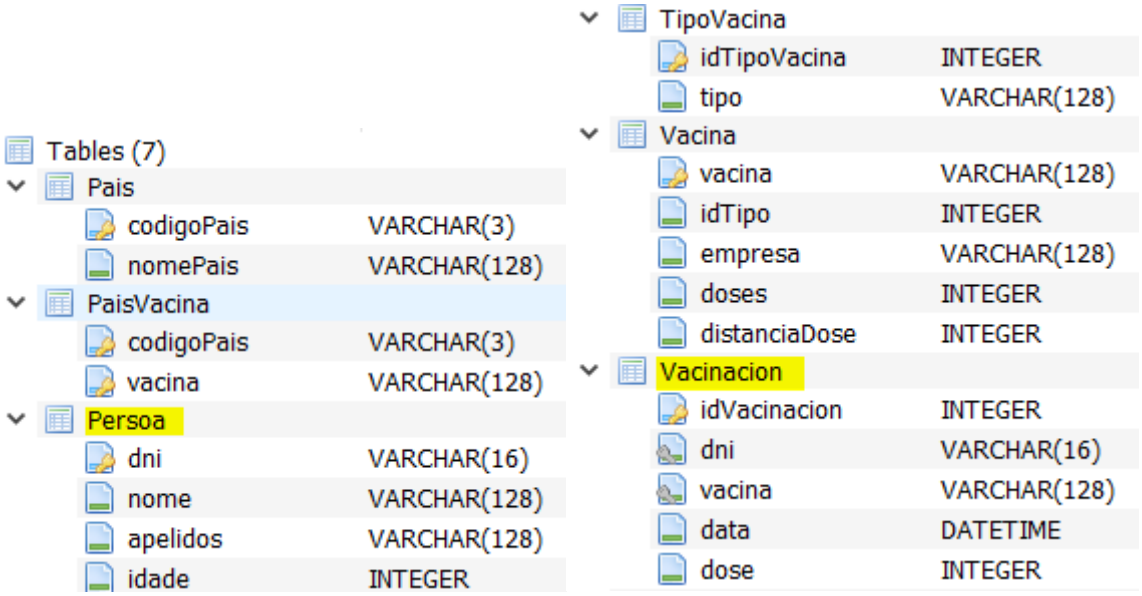
- Formulario de entrada, ***VentaEntrada***, con dos cajas de texto (*tfUsuario*, *tfClave*) y dos botones, "*Aceptar*" (btAceptar) y "Cancelar" (*btCancelar*). Esta ventana consulta de la BD si el usuario existe. Se el usuario existe debe seguir con la aplicación y mostrar a ventana de busca, ***VentaBusca***.
- Formulario de búsqueda, ***VentaBusca***, que contiene un desplegable con la lista de marcas y dos cajas de texto con el precio máximo (*tfMax*) y mínimo (*tfMin*). Debe contener dos botones, uno de *limpar* y otro de *aceptar*.

  Si pulsamos al botón "Aceptar" consultará en la base de datos los productos que se ajusten a la consulta. Se encuentra productos muestra una nueva ventana, en caso contrario debe mostrar un nuevo formulario con la lista de productos en una caja de texto.
- Ventana de dialogo de lista de productos, ***VentaProductos***, contiene una caja de texto no editable en la que se muestran los productos resultado de la búsqueda.

Ejercicio 5. Gestión de vacunas

Para la creación de la BD podemos emplear el propio Netbeans, HeidiSQL, **DB Browser for SQLite**, dbeaver o similar. En este caso tan concreto, recomendaría usar, DB Browser for SQLite: https://sqlitebrowser.org/dl/

Crea la base de datos con de acuerdo con el script *.sql proporcionado y de acuerdo con la siguiente imagen:



Los parámetros de la conexión deben ser:

JDBC_DRIVER = "***org.sqlite.JDBC***";
DB_URL = "***jdbc:sqlite:vacinas.sqlite3***";

- Crea una base de dato con el nombre ***vacinas.sqlite3*** y cópiala en el directorio raíz del proyecto (main).

Clases del modelo

Persoa.java

Atributos de Persoa: *nome*, *apelidos*, *idade*, *dni*.
Métodos get/ser para cada atributo.

Vacinacion.java

Para facilitar la práctica no trabajaremos con fechas, por lo que se precisan los atributos: ***persoa*** (persona que ha sido vacunada), **vacina**, **dose**, **idVacinacion**. Si se desea facilitar el proyecto puede guardarse sólo el DNI de la persona y no el objeto.

En la base de datos el idVacinacion es autonumérico.

Métodos get/ser para cada atributo.

Clases DAO

ConnectionDB.java

Tiene las constantes de la aplicación: **JDBC_DRIVER** y **DB_URL**.
Atributo privado:

*Connection **conexion**;*

Método:

*public Connection **getConnection**().* Este método crea la conexión a la BD si esta es nula.

Constructor por defecto.

PersoaDAO.java

Clase DAO con los métodos de acceso/modificación (CRUD) a la tabla ***Persoa***:

*public Optional<Persoa> **get**(String dni);*
*List<Persoa> **getAll**();*
*public int **save**(Persoa persoa); // devuelve si ha guardado no .*
*public int **update**(Persoa persoa); // devuelve si ha actualizado no .*
*public int **delete**(Persoa persoa) // devuelve si ha borrado no .*

VacinacionDAO.java

Clase DAO con los métodos CRUD de la tabla ***Vacinacion***:

*public Vacinacion **get**(Integer id);*
*public List<Vacinacion> **getAll**();*

*public int **save**(Vacinacion t);*
*public int **update**(Vacinacion t);*
*public int **delete**(Vacinacion t);*

Clases Vista (Opcional)

Para evitar ceñirse a un enunciado concreto, se pide **diseñar de forma libre una aplicación gráfica que contenga una tabla con los datos de vacunación de la BD**.

Los datos deben guardarse en una tabla de tipo JTable. Puede ver información y ayuda en la página:

https://docs.oracle.com/javase/tutorial/uiswing/components/table.html

Adema las, debe contener un panel superior (o una ventana emergente) en el que aparezca el DNI de la persona a buscar, un desplegable con las vacunas, y el número de dosis.

Debajo, tres botones para: insertar, borrar y actualizar.
Adema las, debe aparecer un botón de búsqueda para buscar por DNI.

Recomendación. Cree, al menos las siguientes clases:
PanelTabla
Con la tabla de datos y/o los elementos de búsqueda.
VentanaVacinacions
Con los elementos de la aplicación principal.

Ejercicio 6. Paint con BASE DE DATOS

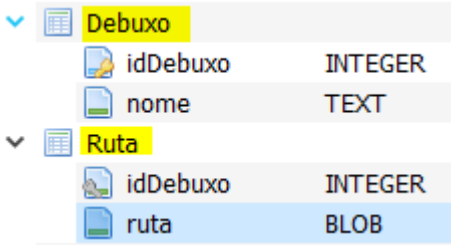
Se trata de implementar una pequeña aplicación que utilice una conexión contra una base de datos relacional, que estará **implantada en SQLite** (https://sqlite.org/index.html).

Además, a modo de repaso, aprovecharemos para personalizar como se pintan componentes Swing sobrescribiendo el método ***paintComponent***.

Para la creación de la BD podemos emplear el propio Netbeans, HeidiSQL, **DB Browser for SQLite** (recomendación personal en este caso), dbeaver o similar.

En este caso tan concreto, recomendaría usar, DB Browser for SQLite, pues está optimizado para este “gestor de bases de datos”: https://sqlitebrowser.org/dl/

Crea la base de datos con de acuerdo con el script *.sql proporcionado y de acuerdo con la siguiente imagen:



Ojo, la ruta se guarda como BLOB (binario grande), pues vamos a guardar el objeto en la base de datos.

Los parámetros de la conexión deben ser:

JDBC_DRIVER = "***org.sqlite.JDBC***";
DB_URL = "***jdbc:sqlite:paint.sqlite3***";

- Crea una base de dato con el nombre ***paint.sqlite3*** y cópiala en el directorio raíz del proyecto (***main***).

Clases del modelo

Ruta (Serializable)

El polígono/ruta es cada uno de los segmentos del dibujo.
Hay muchos modos de implantarlo, pero el más sencillo (¿interesante?) es que la clase **contenga la ruta como un objeto** de la clase *java.awt.geom.**Path2D.Float**.*

*Si lo deseáis, y os parece más sencillo, **podéis hacerlo como un ArrayList de puntos (java.awt.Point) que representan la ruta**, que se unirán al representarlo por medio de líneas. En ese caso, las propiedades, además del color y la anchura será la lista de puntos de la ruta.*

Constantes

El **color** y la **anchura** por defecto de la ruta/fragmento/polígono:
***DEFAULT_COR** = **new** Color(6,111,169); // poned el que deseéis.*
***DEFAULT_ANCHO** = 2;*

Atributos

Color (***cor***) y anchura (***anchura***) de la ruta.
Objeto de tipo ***Path2D.Float*** con la ruta (***ruta***).
Opcional: ***puntos****, lista de puntos de la ruta.*

Constructores

Un constructor que recoge el **color**, la **anchura** y el **punto inicial** de la ruta.
Un constructor que recoge el **color** y la **anchura**.
Un constructor por defecto que crea la **ruta** y la **anchura** con los valores **por defecto**.

Métodos

- **Get/set** para ***ancho*** y ***cor***.
- ***getRuta()***: devuelve la propiedad *ruta*.
- ***addPunto***: añade un punto a la ruta. En el caso de emplear una lista de puntos lo añade a la lista. Si *Ruta* contiene la ruta como un objeto de tipo Path2D debes comprobar que si es el primer punto a añadir (la posición actual es *null* en ese caso):

```
Point2D actual = ruta.getCurrentPoint();
if(actual==null){
   ruta.moveTo(p.x, p.y); // si es el inicio de la ruta la mueves a esa posición.
} else {
   ruta.lineTo(p.x, p.y); // creamos una línea al nuevo punto
}
```

- ***addPunto***: método sobrecargado que recoge las coordenadas del punto.

- ***getPuntos***: devuelve la lista de puntos de la ruta. Si se hace con un Path2D debe emplearse el método ***getPathIterator***:

```
PathIterator pathIterator = ruta.getPathIterator(new AffineTransform());
while (!pathIterator.isDone()) {
        // …
}
```

- ***toString()***: devuelve una cadena con información del color, la anchura y la lista de cada uno de los puntos. Emplee *StringBuilder* para crear la cadena como concatenación de los puntos.

Debuxo (Serializable)

Esta clase tiene la lista de rutas que componen el dibujo.

Constantes

El **nombre por defecto del dibujo**:
DEFAULT_NAME *= "New.png";*

Atributos

ArrayList de objetos de tipo Ruta (rutas/polígonos) (***rutas***).
Nombre del dibujo (***nome***).
Identificador del Dibujo, como entero, ***idDebuxo*.**

Constructores

Un constructor que recoge el ***idDebuxo*** y el ***nome***.
Un constructor que recoge el ***nome***.
Un constructor por **defecto que crea un dibujo con el nombre por defecto**.

Métodos

- ***Get/set*** para el **nombre** y el ***idDebuxo***.
- ***clear()***: limpia la lista de **rutas**.
- ***getRutas()***: devuelve la lista de rutas de dibujo.
- ***getRuta(int i)***: devuelve la ruta que está en la posición i si el valor de i está dentro de los valores permitidos.
- ***addRuta(Ruta ruta)***: añade una ruta al dibujo.
- ***toString***: devuelve una cadena con el ***nombre*** del dibujo, su ***idDebuxo*** y la ***lista de rutas***. Emplee StringBuilder para la concatenación de cadenas.

Clases DAO

ConnectionDB

Constructor por defecto.

Constantes

Tiene las constantes de la base de datos:
JDBC_DRIVER ("org.sqlite.JDBC")
DB_URL ("jdbc:sqlite:*ruta a la BD SQLite*")

Atributo

Atributo privado con la conexión a la BD:
*Connection **conexion**;*

Método

*public Connection **getConnection**()*: método crea la conexión a la BD si el atributo es nulo o está cerrada y (siempre) la devuelve.

*public static Connection **openConnection()**:* versión estática del método anterior. Crea una nueva conexión en cada llamada.

DebuxoDAO

Clase DAO con los métodos de acceso/modificación (CRUD) de la tabla ***Debuxo***:

*List<Debuxo> **getAll**(); // devuelve todos los dibujos de la base de datos. Debe recorrer todos dos dibujos y, para cada uno de ellos, recoger sus rutas.*

*public int **save**(Debuxo debuxo); // Guarda el dibujo y devuelve si lo ha guardado no (valor devuelto por executeUpdate()).*

*public int **update**(Debuxo debuxo); // devuelve si ha actualizado no. No se pide.*

*public int **delete**(Debuxo debuxo) // devuelve si ha borrado no (valor devuelto por executeUpdate()).*

*public int **deleteAll**() // borra todos los dibujos de la BD y devuelve si ha borrado no (valor devuelto por executeUpdate()).*

*public Optional<Debuxo> **get**(int idDebuxo); // devuelve el dibujo con ese id de la BD.*

AppDebuxo

Crea una pequeña aplicación que cree varios dibujos con varias rutas y los guarde en la base de datos.

Recupera los dibujos de la base de datos y muéstralos por pantalla.

Borra algún dibujo y vuelve a hacer la consulta.