Práctica 2

ACCESO A BASES DE DATOS CON JDBC

Tabla de Contenidos

1. Creación de una base de datos en MySQL

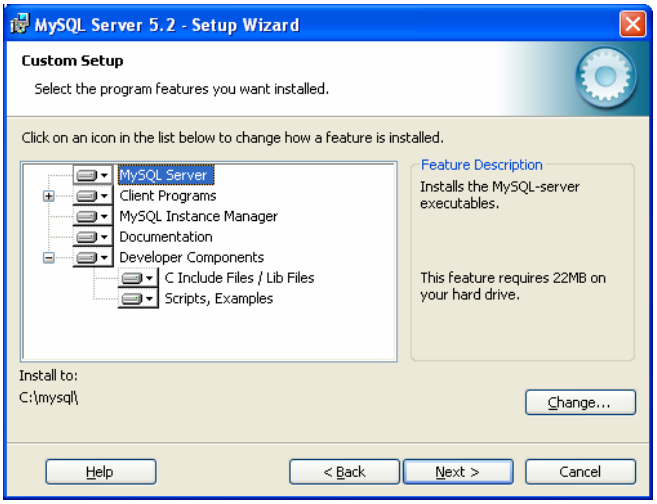
MySQL es un servidor de bases de datos multihilo y multiusuario, robusto y muy rápido. Permite la creación de bases de datos relacionales que pueden ser consultadas a través de SQL (Structure Query Language) estándar. Para ampliar información e instalarlo[1], se puede consultar la dirección http://www.mysql.com.

1.1. Iniciar y utilizar el servidor MySQL

- Instalación

Con el objetivo de simplificar esta labor, descargaremos la versión que incluye el instalador[2]. Durante la instalación, seleccione la instalación “Custom” para incluir toda la documentación y ayuda; así mismo seleccione como ruta de instalación: C:\mysql

[1] http://dev.mysql.com/downloads/mysql/
[2] http://dev.mysql.com/get/Downloads/MySQL-5.0/mysql-5.0.27-win32.zip/from/http://mysql.rediris.es/



Una vez terminada la instalación, seleccione la casilla para configurar el servidor MYSQL t de clic en finalizar.

Seleccionaremos la **configuración detallada** y pondremos los siguientes valores:

- Developer Machine
- Multifunctional Database
- Disco por defecto (C:)
- Decision Support (DSS)/OLAP
- Enable TCP/IP Networking (Port Number 3306)
- Enable Strict Mode
- Standard Character Set
- Install AS Windows Service (MySQL)
    - Launch the MySQL Server automaticaly
- Include Bin Directory in Windows PATH
- Modify Security Settings
    - New root password: admin

- Iniciarlo

Desde el Panel de Control → Herramientas Administrativas → Servicios… MySQL; es posible Iniciar, Detener y reiniciar el servidor de MySQL.

- Conectarse

```
C:\> mysql –u root -p
```

- Desconectarse

```
mysql> QUIT
```

1.2. Creación de una Base de Datos y sus tablas por medio de un fichero script

En MySQL se puede crear un fichero script que contenga todos los comandos que necesitamos ejecutar. Este fichero sólo debe ser un fichero texto que contenga las órdenes terminadas en punto y coma.

A continuación ponemos un ejemplo de un fichero **BDCoches.txt** que permite la creación de una bases de datos llamada **DatosCoches** y dos tablas relacionadas **Propietarios** y **Coches**,

donde *Propietarios* es la tabla principal de la relación con *DNI* como clave principal y *Coches* es la tabla relacionada con *Matrícula* como clave principal:

```
CREATE DATABASE DatosCoches;
USE DatosCoches;
CREATE TABLE PROPIETARIOS (DNI VARCHAR(10),
Nombre VARCHAR(40),
Edad INTEGER,
UNIQUE KEY(DNI));
CREATE TABLE COCHES (Matricula VARCHAR(10) ,
          Marca VARCHAR(20),
          Precio INTEGER,
          DNI VARCHAR (10),
UNIQUE KEY(Matricula),
          FOREIGN KEY (DNI) References Propietarios(DNI));
INSERT INTO Propietarios values('1A','Pepe',30);
INSERT INTO Propietarios values('1B','Ana',40);
INSERT INTO Propietarios values('1C','Maria',50);
INSERT INTO Coches values('MA-1111','Opel',1000,'1A');
INSERT INTO Coches values('MA-2222','Renault',2000,'1A');
INSERT INTO Coches values('BA-3333', 'Seat', 3000,'1B');
Describe Propietarios;
Describe Coches;
Select * from Propietarios;
Select * from Coches;
Select * from Propietarios, Coches
where Propietarios.DNI=Coches.DNI;
```

El comando que se utiliza para poder ejecutar todas las órdenes que se encuentran en este fichero desde MySQL es:

```
mysql> SOURCE c:\BDCoches.txt
```

2. Operaciones sobre la BD desde Java con JDBC

2.1. Introducción a JDBC

JDBC es un API de Java para ejecutar sentencias SQL. Está formado por un conjunto de clases e interfaces programadas con el propio Java. Permite interactuar con bases de datos, de forma transparente al tipo de la misma. Es decir, es una forma única de programar el acceso a bases de datos desde Java, independiente del tipo de la base de datos. JDBC realiza llamadas directas a SQL. Para más información sobre JDBC se puede consultar la dirección http://java.sun.com/javase/technologies/database/index.jsp

Existen cuatro categorías de drivers que soportan la conectividad JDBC[3], por ejemplo: puente JDBC-ODBC, drivers de red, drivers nativos . En la práctica utilizaremos un driver específico que permite la comunicación directa de JDBC con MySQL. Su nombre es MySQL Connector/J 5.0 y es un driver nativo que convierte llamadas JDBC al protocolo de red utilizado por la base de datos MySQL. Se puede encontrar más información en: http://www.mysql.com/products/connector/j/

Este *driver* es necesario tenerlo instalado en el cliente y cambiar la variable *CLASSPATH* para que contenga la ruta del fichero *.jar* o de las carpetas *org* y *com* del *driver*. Para utilizarlo desde un entorno como el Eclipse, es necesario cambiar las propiedades del proyecto y añadir el *.jar* del driver como un *jar* externo.

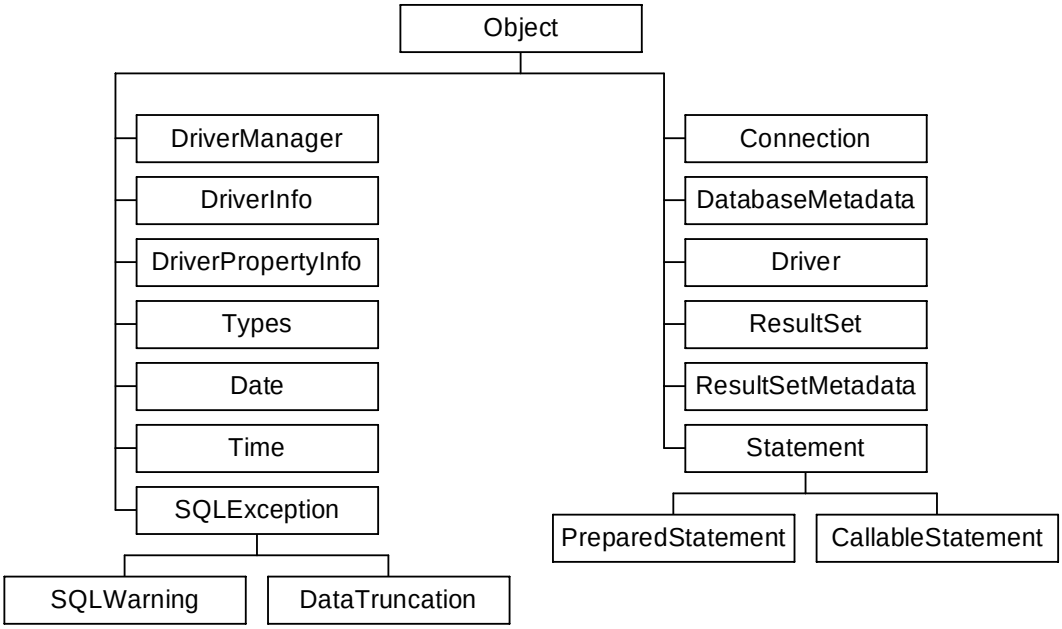
2.2. Paquete java.sql

Las clases que conforman el API JDBC[4] se encuentran agrupadas en el paquete java.sql. Este paquete contiene clases para cargar los drivers, realizar las conexiones a las bases de

[3] http://java.sun.com/products/jdbc/driverdesc.html
[4] http://java.sun.com/j2se/1.3/docs/guide/jdbc/index.html

datos, consultar los datos y manejar un conjunto de registros. También posee las clases para el manejo de excepciones que se produzcan en el acceso a bases de datos. A continuación mostramos un esquema en el cual se encuentran las interfaces y clases fundamentales del paquete *java.sql*.

Clases del Paquete java.sql



La interface Driver es la interface que todos los drivers deben implementar. Cada driver que quiera ser utilizado para conectarse a una base de datos desde Java debe suministrar una clase que implemente la interface *Driver*. Cuando esta clase es cargada en la aplicación de Java, esta debe crear una instancia de sí misma y registrarse en el *DriverManager*. Para cargar y registrar un driver desde el código se hace con:

```
Class.forName("URL  del Driver")
```

La clase DriverManager proporciona el servicio básico para manejar un conjunto de drivers JDBC. Como parte de su inicialización, esta clase intenta cargar las clases driver referenciadas en la propiedad del sistema "jdbc.drivers". El método más importante de esta clase es *getConnection*, a través de cual la clase *DriverManager* intentará localizar un driver apropiado entre los que estén cargados en el proceso de inicialización y aquellos cargados explícitamente en el código. A continuación mostramos la sintaxis del método fundamental de esta clase:

```
DriverManager.getConnection(String URL, String usuario, String password)
```

Este método intenta establecer la conección a la URL dada y devuelve un objeto *Connection* como resultado.

La interface Connection representa una conexión o sesión con una base de datos específica. Las sentencias SQL son ejecutadas y retornan sus resultados dentro del contexto de una conexión. Una conexión a una base de datos permite acceder a la información contenida en las tablas, soporta la gramática SQL, los procedimientos almacenados, etc.

La clase Types define las constantes que se utilizarán para identificar tipos genéricos SQL, estas constantes son llamadas tipos JDBC. Esta clase no puede ser instanciada. A

continuación describimos algunas de las constantes que representan los tipos más usados y su tipo equivalente en Java.

Constantes de la clase Types	Tipo de dato Java
Types.BIT	boolean
Types.TINYINT	byte
Types.SMALLINT	short
Types.INTEGER	int
Types.BIGINT	long
Types.FLOAT	double
Types.REAL	float
Types.DOUBLE	double
Types.NUMERIC	java.math.BigDecimal
Types.DECIMAL	java.math.BigDecimal
Types.CHAR	java.lang.String
Types.VARCHAR	java.lang.String
Types.LONGVARCHAR	java.lang.String
Types.DATE	java.sql.Date
Types.TIME	java.sql.Time
Types.BINARY	byte []
Types.VARBINARY	byte []

Las clases Date y Time se utilizan para representar los valores de fecha y hora desde el código de Java permitiendo a JDBC identificarlos como SQL DATE y SQL TIME. La forma de representación es especificando el valor de la fecha o la hora en milisegundos pasados desde el 1 de Enero de 1970.

Las interfaces DatabaseMetaData y ResulSetMetaData permitirán obtener información referente al diseño y estructura de la base de datos y de los *ResultSet* que se obtengan respectivamente. La clase SQLException representa a las excepciones relacionadas con el acceso a base de datos y proporciona información acerca del error que ha ocurrido.

2.3. Pasos para crear una aplicación JDBC

a) Cargar el driver JDBC.

b) Conectarse a la Base de Datos utilizando la clase Connection.

c) Crear sentencias SQL, utilizando objetos de tipo Statement.

d) Ejecutar las sentencias SQL a través de los objetos de tipo Statement.

e) En caso que sea necesario, procesar el conjunto de registros resultante utilizando la clase ResultSet.

2.4. Cargar el driver JDBC

Para conectarse a una base de datos a través de JDBC desde una aplicación Java, lo primero que se necesita es cargar el driver que se encargará de convertir la información que se envía a través de la aplicación a un formato que lo entienda la base de datos. Esta parte del código sería la única que dependería del tipo de driver y del tipo de base de datos.

La sintaxis para cargar el driver es:

```
Class.forName("Clase  del driver").newInstace();
```

Por ejemplo, si el driver que se emplea es el puente JDBC-ODBC, entonces el código para cargarlo sería:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

En esta práctica utilizaremos el driver nativo MySQLConector/J para acceder a MySQL. Entonces el código será:

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

El código para cargar el driver podría lanzar una excepción de tipo *ClassNotFoundException* que debe ser capturada.

Recordamos que este *driver* es necesario tenerlo instalado en el cliente y cambiar la variable *CLASSPATH* para que contenga la ruta del fichero *.jar* o de las carpetas *org* y *com* del *driver*. Para utilizarlo desde un entorno como el Eclipse, es necesario cambiar las propiedades del proyecto y añadir el *.jar* del driver como un *jar* externo.

2.5. Conectarse a una Base de Datos

Para conectarse a una fuente de datos específica, una vez cargado el driver, se utiliza una URL que indicará la base de datos con la sintaxis:

```
jdbc:subprotocolo:parámetros
```

El subprotocolo indica una forma de conexión a una base de datos que puede ser soportada por uno o más drivers. El contenido y sintaxis de los parámetros depende del subprotocolo. La sintaxis general de conexión es:

```
Connection con = DriverManager.getConnection(URL, usuario, password)
```

A continuación se irá desarrollando un ejemplo con las diferentes funcionalidades para acceder a la base de datos *DatosCoches*. La conexión a la base de datos y la consulta de la misma se desarrollará en una clase java llamada AccesoDatos. Las pruebas se realizarán desde otra clase PruebaAccesoDatos que solo contendrá una función *main* desde donde se llamará a todas las funciones de *AccesoDatos*. Por ejemplo, para conectarse a través del *driver* de *MySQL* a la base de datos *DatosCoches* el código sería:

```
import java.sql.*;
public class AccesoDatos {
    Connection con;
    Statement  st;
    ResultSet rs;
    public void abrirConexion() {
        try {
            String userName=”root”;
            String password=”admin”;
            String url="jdbc:mysql://localhost/DatosCoches";
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            con = DriverManager.getConnection(url, userName, password);
            System.out.println("Conexión a la BD");
        }
        catch (Exception e) {
            System.out.println("Error en conexión ");
        }
    }


    //Para cerrar la conexión una vez terminadas las consultas
    public void cerrarConexion() {
        try {
            con.close();
            System.out.println("Conexión cerrada");
        }
        catch (SQLException e) {
            System.out.println("Error al cerrar conexión");
        }
    }
}
...
```

2.6. Consultar la Base de Datos

Para recuperar información de una de base de datos se utiliza la clase Statement , cuyos objetos se crean a partir de una conexión y tienen el método *executeQuery* para ejecutar consultas SQL de tipo SELECT devolviendo como resultado un conjunto de registros en un objeto de la clase ResultSet.

Supongamos que se tiene una conexión a la base de datos *DatosCoches* cuyas tablas son *Propietarios* y *Coches* y que queremos obtener los registros de la tabla *Coches* ordenados según el precio de mayor a menor, el código sería:

```
...
public void obtenerDatosTabla() {
try {
    st =con.createStatement();
    rs = st.executeQuery("SELECT * FROM Coches ORDER BY precio DESC");
    System.out.println("Tabla abierta");
    ...
} catch (SQLException e) {
    System.out.println("Error al Abrir tabla ");
}
```

Una sentencia creada de esta forma devuelve un *ResultSet* en el que sólo puede haber desplazamiento hacia adelante.

Para luego acceder al conjunto de registros que se encuentran en el *ResultSet*, es necesario el método *next* del objeto *ResultSet* para movernos por los registros y métodos específicos para extraer la información de cada tipo de campo con la forma *getXXXX*. A continuación mostramos una tabla con los métodos de los tipos más comunes:

Métodos de ResultSet	Tipo SQL
getInt	INTEGER
getLong	BIG INT
getFloat	REAL
getDouble	FLOAT
getBignum	DECIMAL
getBoolean	BIT
getString	VARCHAR
getString	CHAR
getDate	DATE
getTime	TIME
getTimesstamp	TIME STAMP
getObject	cualquier tipo

Para cada método *getXXXX*, el driver JDBC debe hacer conversiones entre el tipo de la base de datos y el tipo Java equivalente. El driver no permite conversiones inválidas aunque si permite que todos los tipos puedan ser leídos desde Java como cadenas con el método *getString*.

Por otra parte, cuando se recorre el *ResultSet* es necesario conocer cuando se llega al final del mismo y esto se controla con el método *next* que además de moverse al siguiente registro, devuelve falso cuando se ha pasado del último registro. Un ejemplo de código para recorrer un *ResulSet* visualizando su contenido por consola es:

```
...
public void mostrarDatosCoches() {
      try {
            while (rs.next()) {
                  String strMat = rs.getString("Matricula");
                  String strMarca = rs.getString("Marca");
                  int intPrecio = rs.getInt("Precio");
```

```
                    System.out.println(strMat + ", " + strMarca + ", " +
                    intPrecio);
              }
       } catch (Exception e) {
              System.out.println("Error al visualizar datos");
       }
}
```

2.7. Consultas de Actualización

Para actualizar la base de datos con sentencias SQL de tipo UPDATE, INSERT o DELETE, es necesario al igual que en el caso de SELECT, tener una *Connection* y crear una *Statement* a partir de la misma. La diferencia es que en vez de llamar al método *executeQuery* para ejecutar la consulta, se llama al método *executeUpdate* que no devuelve un *ResultSet* como resultado, sino que devuelve la cantidad de registros afectados. A continuación se muestran varios métodos con ejemplos de este tipo de consultas.

```
//Método para modificar la tabla Coches pasando la matrícula del //coche que
se quiere modificar y el nuevo precio. Las cadenas en la //condición es
necesario ponerlas entre comillas simples.
public void modificar(String m, int p)
{
       try {
              Statement s2=con.createStatement();
              s2.executeUpdate("Update Coches set Precio="+ p +
              " where Matricula like '" + m + "%'");
              System.out.println("Elemento modificado correctamente");
       }catch (SQLException e)
       {
              System.out.println("Error al modificar");
       }
}

//Método para borrar el coche cuya matricula se pasa como argumento
public void borrar(String m)
{
       try{
              Statement s2=con.createStatement();
              s2.executeUpdate(
              "DELETE FROM Coches where Matricula like '"+m+"%'");
              System.out.println("Elemento Borrado");
       }catch(SQLException e)
       {
              System.out.println("Error al Borrar");
       }
}

//Método que permite insertar un nuevo registro en la tabla Coches,
//pasándole como argumento la matrícula, marca, precio del coche y
//dni del propietario
public void insertar(String m, String mar, int p, String d)
{ try{
       Statement s1 = con.createStatement();
       s1.executeUpdate(
       "INSERT INTO Coches (Matricula,Marca,Precio,DNI) values ('"
       + m + "', '" + mar + "'," + p + ",'" + d + "')");
       System.out.println("Elemento insertado");
}
catch(SQLException e)
{
       System.out.println("Error al insertar ");
}
}
```

2.8. Sentencias de tipo PreparedStatement

Cuando se realiza la misma operación varias veces, es mejor utilizar la clase PreparedStatement para una ejecución eficiente. Esta eficiencia está dada porque la consulta

que se ejecute a través de un objeto de la clase *PreparedStatement* será precompilada por el motor SQL de la fuente de datos a la cuál se accede.

Esta clase también permite una forma más cómoda de ejecutar consultas a las cuales hay que pasar muchos parámetros.

En estas consultas, los signos de interrogación representan los parámetros. Para sustituir cada signo de interrogación se utiliza un método *setXXXX(pos, valor)* cuyo nombre depende del tipo del parámetro. El argumento pos indica la posición del signo de interrogación que se quiere sustituir, empieza en 1.

```
//Método para insertar un registro en la tabla Propietarios. Los
//argumentos del método son el DNI, nombre y edad.
public void insertar2(String dni, String n, int ed) {
try {
       PreparedStatement ps = con.prepareStatement(
       "insert into Propietarios values (?,?,?) ");
       ps.setString(1, dni);
       ps.setString(2, n);
       ps.setInt(3, ed);
       //En este caso, el método executeUpdate devuelve la cantidad de
       //elementos insertados.
       if (ps.executeUpdate()!=1)
              throw new Exception("Error en la Inserción");
}catch (Exception e) {
       System.out.println("Error al Insertar ");
}
}
```

2.9. Utilización de Metadata

Como ya fue comentado anteriormente, existen clases en el paquete *java.sql* que permiten acceder a la información sobre el diseño y la estructura de la base de datos como un todo o de un *ResultSet* obtenido a partir de una consulta concreta. A este tipo de información se le llama metadata y las clases que nos permitirán obtenerlo son *DatabaseMetaData* y *ResultSetMetaData*.

2.9.1. Información de una Base de Datos

Cuando se necesita conocer sobre las capacidades, o el vendedor de una base de datos, se puede preguntar al objeto *Connection* por su metadata. Existen muchas preguntas que se pueden hacer, entre ellas tenemos el tipo base de datos, la cantidad máxima de conexiones que permite la base de datos, etc. El fragmento de código que se encuentra a continuación, muestra como obtener esta información.

```
Public void bd() throws SQLException{
       DatabaseMetaData dbMet = con.getMetaData();
       if (dbMet==null)
              System.out.println("No hay información de MetaData");
       else {
              System.out.println("Tipo de la BD: " +
              dbMet.getDatabaseProductName());
              System.out.println("Versión : " +
              dbMet.getDatabaseProductVersion());
              System.out.println("Cantidad máxima de conexiones activas: " +
              dbMet.getMaxConnections());
}
```

2.9.2. Información de un ResultSet

Se puede obtener información de la estructura de un conjunto de registros resultantes de una consulta. Esto puede ser muy útil para acceder a tablas de una base de datos de las cuales no se tenga información sobre su estructura. Utilizando la clase ResultSetMetaData podremos determinar la cantidad de columnas o campos que contiene un *ResultSet*, el tipo y nombre de cada campo, sin el campo es solo lectura, etc. La función siguiente muestra la estructura de una tabla que le pasemos como argumento.

```
public void estructuraTabla(String strTbl) {
	try {
		Statement st = con.createStatement();
		ResultSet rs = st.executeQuery("Select * from " + strTbl);
		//Obtiene el metadata del ResultSet
		ResultSetMetaData rsmeta = rs.getMetaData();
		//Obtiene la cantidad de columnas del ResultSet
		int col = rsmeta.getColumnCount();
		for (int i = 1; i <= col; i++) {
			System.out.println("Campo " +
			//Devuelve el nombre del campo i
			rsmeta.getColumnLabel(i) + "\t"
			//Devuelve el tipo del campo i
			+ "Tipo: " + rsmeta.getColumnTypeName(i));
		}
	}
	catch (Exception e) {
		System.out.println("Error en Metadata ");
	}
}
```

También es posible mediante la utilización de la información del *ResultSetMetaData* mostrar la información de cualquier tabla sin tener la estructura previamente.

```
	public void verCualquierTabla(String strTbl) {
		try {
			Statement st = con.createStatement();
			ResultSet rs = st.executeQuery("Select * from " +
			strTbl);
			ResultSetMetaData meta = rs.getMetaData();
			int col = meta.getColumnCount();
			//Mientras haya registros
			while (rs.next()) {
				for (int i = 1; i <= col; i++) {
					//Mostrar el dato del campo i
					System.out.print(rs.getString(i)
					+ "\t");
				}
				System.out.println("");
			}
		}catch (Exception e) {
			System.out.println("Cualquier " + e.toString());
		}
	}
```

2.10. Clase que prueba el acceso a datos

Para probar todas las funciones que se han hecho en la clase *AccesoDatos* se puede desarrollar la clase *PruebaAccesoDatos* con una función main:

```
package pjdbc;

import java.sql.SQLException;

public class PruebaAccesoDatos {

	public static void main (String[] args) throws SQLException{
		AccesoDatos AD = new AccesoDatos();
		AD.abrirConexion();
		AD.obtenerDatosTabla();
		AD.mostrarDatosCoches();
		AD.modificar("BA-3333", 5000);
		AD.borrar("MA");
		AD.insertar("AA-0005", "Ford", 4500, "1A");
		AD.insertar2("X25", "Jose", 54);
		AD.bd();
		AD.estructuraTabla("Propietarios");
		AD.verCualquierTabla("Coches");
		AD.cerrarConexion();


	}


}
```

3. Ejercicios propuestos

En el ejemplo que se ha ido desarrollando no se han verificado las restricciones que se exigen cuando se accede a una base de datos con tablas relacionadas entre si. A continuación proponemos dos ejercicios que complementarían esta práctica:

- Función que dado el dni del propietario, liste sus datos y los coches que posee.
- Modificar la función de insertar coches para que verifique que el *dni* del propietario ya existe en la tabla *propietarios*, si no existe, que no permita la inserción del nuevo coche.
- Función que permita borrar de la BD a un propietario (borrando también los coches de este propietario).