**CONFIGURAR LAS CONEXIONES DE LAS EMBEBIDAS EN ECLIPSE**

2.4 PROTOCOLOS DE ACCESO A BASES DE DATOS

En tecnologías de base de datos podemos encontrarnos con dos normas de conexión a una base de datos SQL:

* **ODBC** (*Open Database Connectivity*): define una API (*Application Program Interface - Interfaz para Programas de Aplicación*) que pueden usar las aplicaciones para abrir una conexión con una base de datos, enviar consultas, actualizaciones y obtener los resultados. Las aplicaciones pueden usar esta API para conectarse a cualquier servidor de base de datos compatible con ODBC. Está escrito en C.
* **JDBC** (*Java Database Connectivity* - *Conectividad de base de Datos con Java*) define una API que pueden usar los programas Java para conectarse a los servidores de bases de datos relacionales.

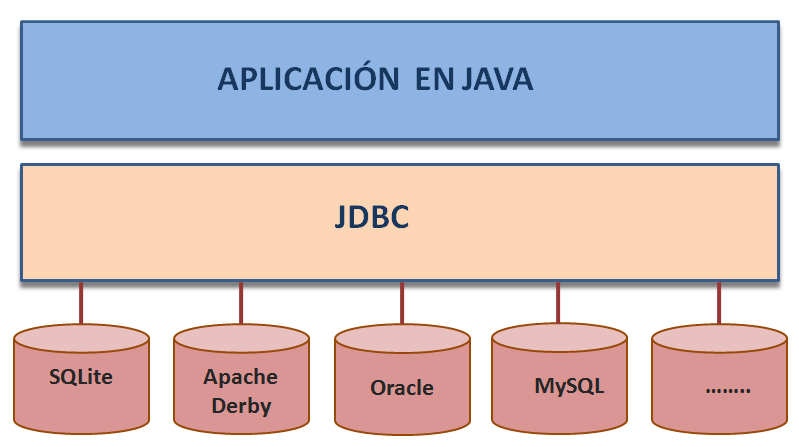
Hay muchos orígenes de datos que no son bases de datos relacionales, algunos puede que ni si quiera sean bases de datos, tal es el caso de los archivos planos y los almacenes de correo electrónico. **OLE-DB** (*Object Linking and Embedding for Databases - Enlace e incrustación de objetos para bases de datos*) de Microsoft es una API de C++ con objetivos parecidos a los de ODBC pero para orígenes de datos que no son bases de datos. OLE-DB proporciona estructuras para la conexión con orígenes de datos, ejecución de comandos y devolución de resultados en forma de conjunto de filas. Sin embargo se diferencia de ODBC en algunos aspectos. Los programas OLE-DB pueden negociar con los orígenes de datos para averiguar las interfaces que soportan. En ODBC los comandos siempre están en SQL, en OLE-DB pueden estar en cualquier lenguaje soportado por el origen de datos, puede que algunos orígenes soporten SQL o un subconjunto limitado de SQL y otros ofrezcan el acceso a los datos de los archivos planos sin ninguna capacidad de consulta.

La API **ADO** (*Active Data Objects – Objetos activos de datos*) creada por Microsoft ofrece una interfaz sencilla de utilizar con la funcionalidad OLE-DB, que puede llamarse desde los lenguajes de guiones como VBScript y JScript.

2.6 ACCESO A DATOS MEDIANTE JDBC.

JDBC proporciona una librería estándar para acceder a fuentes de datos principalmente orientados a bases de datos relacionales que usan SQL. No sólo provee una interfaz sino que también define una arquitectura estándar, para que los fabricantes puedan crear los drivers que permitan a las aplicaciones Java el acceso a los datos.

**JDBC** dispone de una interfaz distinta para cada base de datos (véase Figura 2.8), es lo que llamamos **driver** (controlador o conector). Esto permite que las llamadas a los métodos Java de las clases JDBC se correspondan con el API de la base de datos.



**Figura 2.8**. Acceso mediante JDBC.

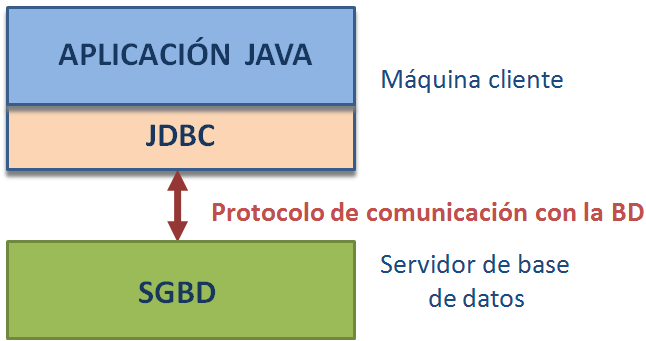
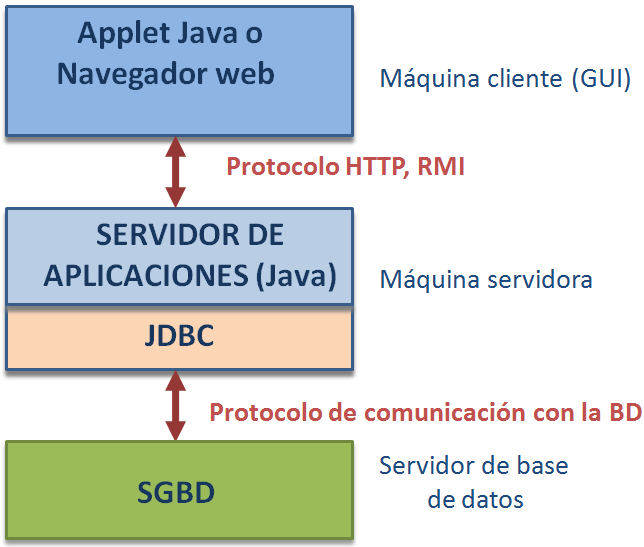
JDBC consta de un conjunto de clases e interfaces que nos permite escribir aplicaciones Java para gestionar las siguientes tareas con una base de **datos relacional:**

* Conectarse a la base de datos.
* Enviar consultas e instrucciones de actualización a la base de datos.
* Recuperar y procesar los resultados recibidos de la base de datos en respuesta a las consultas.

2.6.1 Dos modelos de acceso a bases de datos.

La API JDBC es compatible con los modelos tanto de dos como de tres capas para el acceso a la base de datos. En el **modelo de dos capas**, un applet o una aplicación Java “hablan” directamente con la base de datos, esto requiere un driver JDBC residiendo en el mismo lugar que la aplicación (Figura 2.9). Desde el programa Java se envían sentencias SQL al sistema gestor de base de datos para que las procese y los resultados se envían de vuelta al programa. La base de datos puede encontrarse en otra máquina diferente a la de la aplicación y las solicitudes se hacen a través de la red (arquitectura cliente-servidor). El driver será el encargado de manejar la comunicación a través de la red de forma transparente al programa.

En el **modelo de tres capas**, los comandos se envían a una capa intermedia que se encargará de enviar los comandos SQL a la base de datos y de recoger los resultados de la ejecución de las sentencias. Es decir, **tenemos una aplicación o applet corriendo en una máquina y accediendo a un driver de base de datos situado en otra máquina,** véase Figura 2.10. En este caso los drivers no tienen que residir en la máquina cliente.

**Figura 2.9**. Modelo de dos capas. **Figura 2.10**. Modelo de tres capas.

Un **servidor de aplicaciones** es una implementación de la especificación JEE (*Java 2 Platform Enterprise Edition*).**JEE es un entorno centrado en Java para desarrollar, construir y desplegar aplicaciones empresariales multicapa basadas en la Web.** Existen diversas implementaciones, cada una con sus propias características. Algunas de ellas son las siguientes: *BEA WebLogic, IBM WebSphere, Oracle IAS, Borland AppServer*, etc.

2.6.2 Tipos de drivers.

Existen 4 tipos de conectores (drivers o controladores) JDBC:

* **Tipo 1. JDBC-ODBC Bridge** (*JDBC-ODBC bridge plus ODBC driver*): permite el acceso a bases de datos JDBC mediante un driver ODBC. Convierte las llamadas al API de JDBC en llamadas ODBC. **Exige la instalación y configuración de ODBC en la máquina cliente.**

Open DataBase Connectivity (**ODBC**) es un estándar de acceso a las bases de datos

* **Tipo 2. Native** (*Native-API partly-Java driver*): controlador escrito parcialmente en Java y en código nativo de la base de datos. Traduce las llamadas al API de JDBC Java en llamadas propias del motor de base de datos. **Exige instalar en la máquina cliente código binario propio del cliente de base de datos y del sistema operativo**.
* **Tipo 3. Network** (*JDBC-Net pure Java driver*): controlador de Java puro que utiliza un protocolo de red (por ejemplo HTTP) para comunicarse con el servidor de base de datos.

Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red independiente de la base de datos y a continuación son traducidas por un software intermedio (*Middleware*) al protocolo usado por el motor de base de datos. El driver JDBC no comunica directamente con la base de datos, comunica con el software intermedio, que a su vez comunica con la base de datos. Son útiles para aplicaciones que necesitan interactuar con diferentes formatos de bases de datos, ya que usan el mismo driver JDBC sin importar la base de datos específica. **No exige instalación en cliente.**

* **Tipo 4. Thin** (*Native-protocol pure Java driver*): controlador de Java puro con protocolo nativo. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red usado por el motor de base de datos. **No exige instalación en cliente**.

**Los tipos 3 y 4 son la mejor forma para acceder a bases de datos JDBC**. Los tipos 1 y 2 se usan normalmente cuando no queda otro remedio, porque el único sistema de acceso final al gestor de bases de datos es ODBC (es decir, no existen drivers disponibles para el SGBD); pero exigen instalación de software en el puesto cliente. En la mayoría de los casos la opción más adecuada será el tipo 4.

2.6.3 Cómo funciona JDBC.

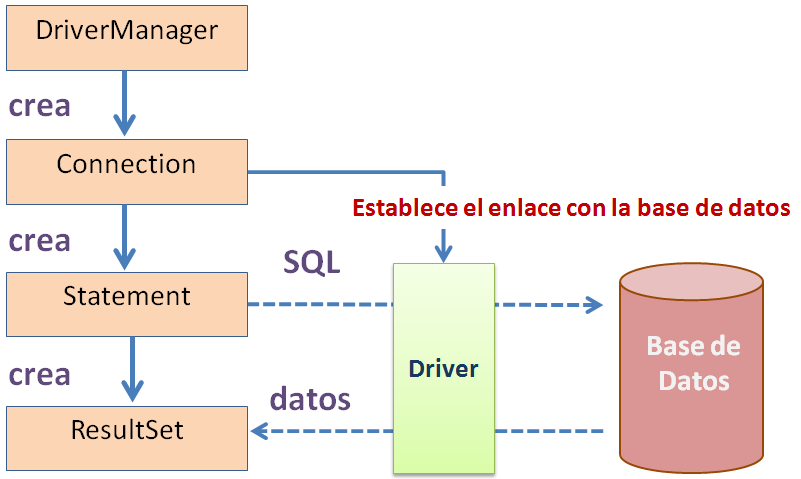
**JDBC define varias interfaces que permite realizar operaciones con bases de datos; a partir de** ellas se derivan las clases correspondientes.

Estas están definidas en el paquete **java.sql**. La siguiente tabla muestra las clases e interfaces más importantes:

|  |  |
| --- | --- |
| **CLASE E INTERFACE** | **DESCRIPCIÓN** |
| Driver | Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto. |
| DriverManager | Permite gestionar todos los drivers instalados en el sistema. |
| DriverPropertyInfo | Proporciona diversa información acerca de un driver. |
| Connection | Representa una conexión con una base de datos. Una aplicación puede tener más de una conexión. |
| DatabaseMetadata | Proporciona información acerca de una Base de Datos, como las tablas que contiene, etc. |
| Statement | Permite ejecutar sentencias SQL sin parámetros. |
| PreparedStatement | Permite ejecutar sentencias SQL con parámetros de entrada. |
| CallableStatement | Permite ejecutar sentencias SQL con parámetros de entrada y salida, como llamadas a procedimientos almacenados. |
| ResultSet | Contiene las filas resultantes de ejecutar una orden SELECT. |
| ResultSetMetadata | Permite obtener información sobre un **ResultSet**, como el número de columnas, sus nombres, etc. |
| *http://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html* | |

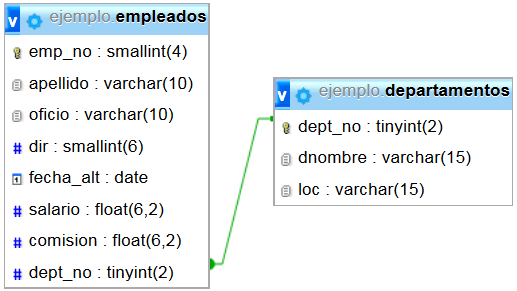
La Figura 2.11 muestra las 4 clases principales que usa cualquier programa Java con JDBC. El trabajo con JDBC comienza con la clase **DriverManager** que es la encargada de establecer las conexiones con los orígenes de datos a través de los drivers JDBC. El funcionamiento de un programa con JDBC requiere los siguientes pasos:

1. Importar las clases necesarias.
2. Cargar el driver JDBC.
3. Identificar el origen de datos.
4. Crear un objeto **Connection**.
5. Crear un objeto **Statement**.
6. Ejecutar una consulta con el objeto **Statement**.
7. Recuperar los datos del objeto **ResultSet**.
8. Liberar el objeto **ResultSet**.
9. Liberar el objeto **Statement**.
10. Liberar el objeto **Connection**.



**Figura 2.11**. Funcionamiento de JDBC.

Para el siguiente ejemplo JAVA creamos antes desde MySQL una base de datos y un usuario con nombre ***ejemplo***, la clave del usuario es la misma. Este usuario tendrá todos los privilegios sobre esta base de datos. A continuación creamos las siguientes tablas e insertamos datos en ellas, las relaciones se muestran en la Figura 2.12:



**Figura 2.12**. Base de datos *ejemplo*.

CREATE TABLE **departamentos** (

dept\_no TINYINT(2) NOT NULL PRIMARY KEY,

dnombre VARCHAR(15),

loc VARCHAR(15)

) ENGINE=InnoDB;

CREATE TABLE **empleados** (

emp\_no SMALLINT(4) NOT NULL PRIMARY KEY,

apellido VARCHAR(10),

oficio VARCHAR(10),

dir SMALLINT,

fecha\_alt DATE ,

salario FLOAT(6,2),

comision FLOAT(6,2),

dept\_no TINYINT(2) NOT NULL,

CONSTRAINT FK\_DEP FOREIGN KEY (dept\_no) REFERENCES

departamentos(dept\_no)

) ENGINE=InnoDB;

**El siguiente programa ilustra los pasos de funcionamiento de JDBC accediendo a la base de datos anterior y mostrando el contenido de la tabla *departamentos*:**

import java.sql.\*;

**public class Main** {

public static void main(String[] args) {

try {

**//Cargar el driver**

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

**//Establecemos la conexion con la BD**

Connection conexion = DriverManager.getConnection

("jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");

**// Preparamos la consulta**

Statement sentencia = conexion.createStatement();

String sql = "SELECT \* FROM departamentos";

ResultSet resul = sentencia.executeQuery(sql);

**//Recorremos el resultado para visualizar cada fila**

//Se hace un bucle mientras haya registros y se van mostrando

while (resul.next()) {

System.out.printf("%d, %s, %s %n",

resul.getInt(3),

resul.getString(1),

resul.getString(2));

}

resul.close(); **// Cerrar ResultSet**

sentencia.close(); **// Cerrar Statement**

conexion.close(); **// Cerrar conexión**

} catch (ClassNotFoundException cn) {

cn.printStackTrace();

} catch (SQLException e) {

e.printStackTrace();

}

}// fin de main

}// fin de la clase

La ejecución muestra la siguiente salida:

10, CONTABILIDAD, SEVILLA

20, INVESTIGACIÓN, MADRID

30, VENTAS, BARCELONA

40, PRODUCCIÓN, BILBAO

Para poder probar el programa hemos de obtener el JAR que contiene el driver MySQL (en el ejemplo se ha utilizado ***mysql-connector-java-8.0.17.jar***) e incluirlo en el CLASSPATH o **añadirlo a nuestro IDE**, por ejemplo en **Eclipse** pulsamos en el proyecto con el botón derecho del ratón y seleccionamos ***Build Paths-> Add External Archives*** para localizar el archivo JAR. Desde la URL *http://www.mysql.com/products/connector/* se puede descargar el conector.

Desde esta URL seleccionaremos las distintas versiones: <https://downloads.mysql.com/archives/c-j/>

Si ejecutamos el programa desde la línea de comandos hemos de asegurarnos que el lugar donde se encuentra el archivo JAR se encuentre definido en la variable CLASSPATH.

Se puede observar que en nuestro programa Java, todos los *import* que necesitamos para manejar la base de datos están en el paquete **java.sql.\***.

También se ha incluido todo el programa en un **try-catch** ya que casi todos los métodos relativos a base de datos pueden lanzar la excepción **SQLException**.

La llamada al método ***forName()*** para cargar el driver puede lanzar la excepción **ClassNotFoundException** si este no se encuentra.

Cargar el driver:

En primer lugar se carga el driver, con el método ***forName()*** de la clase **Class**, se le pasa un objeto *String* con el nombre de la clase del driver como argumento. En el ejemplo como se accede a una base de datos MySQL necesitamos cargar el driver ***com.mysql.jdbc.Driver***:

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

Establecer la conexión:

A continuación se establece la conexión con la base de datos, el servidor MySQL debe estar arrancado, usamos la clase **DriverManager** con el método ***getConnection()*** de la siguiente manera:

Connection conexion = DriverManager.getConnection

("jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");

La sintaxis del método ***getConnection()*** es la siguiente:

public static Connection **getConnection**

**(String url, String user, String password)** throws SQLException

El primer parámetro del método ***getConnection()*** representa la URL de conexión a la base de datos. Tiene el siguiente formato para conectarse a MySQL:

**jdbc:mysql://nombre\_host:puerto/nombre\_basedatos**

Donde

* **jdbc:mysql** indica que estamos utilizando un driver JDBC para MySQL.
* **nombre\_host** indica el nombre del servidor donde está la base de datos. Aquí puede ponerse una IP o un nombre de máquina que esté en la red. Si especificamos *localhost* como nombre de servidor, estamos indicando que el servidor de base de datos se encuentra en la misma máquina en la que se ejecuta el programa Java.
* **puerto** es el puerto predeterminado para las bases de datos MySQL, por defecto es **3306**. Si no se pone se asume este valor.
* **nombre\_basedatos** es el nombre de la base de datos a la que nos vamos a conectar y que debe existir en MySQL. En este caso el nombre es *ejemplo*.

El segundo parámetro es el nombre de usuario que accede a la base de datos, en este caso se llama *ejemplo*.

El tercer parámetro es la clave del usuario, que en este caso también es *ejemplo*.

Ejecutar sentencias SQL:

A continuación se realiza la consulta, para ello recurrimos a la interfaz **Statement** para crear una sentencia. Para obtener un objeto **Statement** se llama al método ***createStatement()*** de un objeto **Connection** válido. La sentencia obtenida (o el objeto obtenido) tiene el método ***executeQuery()*** que sirve para realizar una consulta a la base de datos, se le pasa un *String* en el que está la consulta SQL, en el ejemplo *“SELECT \* FROM departamentos”*:

**Statement** sentencia = **conexion.createStatement()**;

String sql = "SELECT \* FROM departamentos";

**ResultSet** resul = **sentencia.executeQuery(sql)**;

El resultado nos lo devuelve como un **ResultSet**, que es un objeto similar a una lista en la que está el resultado de la consulta. Cada elemento de la lista es uno de los registros de tabla *departamentos*. **ResultSet** no contiene todos los datos, sino que los va consiguiendo de la base de datos según se van pidiendo. Por ello, el método ***executeQuery()*** puede tardar poco, aunque recorrer los elementos del **ResultSet** puede no ser tan tan rápido.

**Resultset** tiene internamente un puntero que apunta al primer registro de la lista. Mediante el método ***next()*** el puntero avanza al siguiente registro. Para recorrer la lista de registros usaremos dicho método dentro de un bucle *while* que se ejecutará mientras ***next()*** devuelva *true* (es decir, mientras haya registros):

while (**resul.next()**) {

System.out.printf("%d, %s, %s %n",

resul.getInt(1),

resul.getString(2),

resul.getString(3) );

}

Los métodos ***getInt()*** y ***getString()*** nos van devolviendo los valores de los campos de dicho registro. Entre paréntesis se pone la posición de la columna en la tabla, es decir la columna que deseamos. También se puede poner una cadena que indica el nombre de la columna (se hará referencia a estos métodos más adelante):

System.out.printf("%d, %s, %s %n",

resul.getInt("dept\_no"),

resul.getString("dnombre"),

resul.getString("loc") );

**ResultSet** dispone de varios métodos para mover el puntero del objeto **ResultSet**:

|  |  |
| --- | --- |
| **Método** | **Función** |
| boolean next() | Mueve el puntero del objeto **ResultSet** una fila hacia adelante a partir de la posición actual. Devuelve *true* si el puntero se posiciona correctamente y *false* si no hay registros en el **ResultSet**. |
| boolean first() | Mueve el puntero del objeto **ResultSet** al primer registro de la lista. Devuelve *true* si el puntero se posiciona correctamente y *false*. |
| boolean last() | Mueve el puntero del objeto **ResultSet** al último registro de la lista. Devuelve true si el puntero se posiciona correctamente y false si no hay registros. |
| boolean previous() | Mueve el puntero del objeto **ResultSet** al registro anterior de la lista. Devuelve *true* si el puntero se posiciona correctamente y *false* si se coloca antes del primer registro. |
| void beforeFirst() | Mueve el puntero del objeto **ResultSet** justo antes del primer registro. |
| int getRow() | Devuelve el número de registro actual. Para el primer registro del objeto **ResultSet** develve 1, para el segundo 2 y así sucesivamente. |

El siguiente código muestra el número de filas recuperadas en la consulta y seguidamente muestra los datos de cada fila acompañada del número de fila:

Statement sentencia = conexion.createStatement();

String sql = "SELECT \* FROM departamentos";

ResultSet resul = sentencia.executeQuery(sql);

**resul.last();** //Nos situamos en el último registro

System.out.println ("NÚMERO DE FILAS: " + **resul.getRow()**);

**resul.beforeFirst();** //Nos situamos antes del primer registro

//Recorremos el resultado para visualizar cada fila

while (**resul.next()**)

System.out.printf("Fila %d: %d, %s, %s %n",

**resul.getRow()**,

resul.getInt(1),

resul.getString(2),

resul.getString(3) );

La salida muestra la siguiente información:

NÚMERO DE FILAS: 4

Fila 1: 10, CONTABILIDAD, SEVILLA

Fila 2: 20, INVESTIGACIÓN, MADRID

Fila 3: 30, VENTAS, BARCELONA

Fila 4: 40, PRODUCCIÓN, BILBAO

Liberar recursos:

Por último se liberan todos los recursos y se cierra la conexión:

resul.close(); //Cerrar ResultSet

sentencia.close(); //Cerrar Statement

conexion.close(); //Cerrar conexión

ACTIVIDAD 2.6

Tomando como base el programa que ilustra los pasos de funcionamiento de JDBC obtener el APELLIDO, OFICIO y SALARIO de los empleados del departamento 10.

Realiza otro programa Java que visualice el Apellido del empleado con máximo salario, visualiza también su SALARIO y el nombre de su departamento.

2.7 ESTABLECIMIENTO DE CONEXIONES.

Hemos visto en ejemplos anteriores como se realiza la conexión con una base de datos MySQL utilizando JDBC y el puente JDBC-ODBC, a continuación vamos a ver como conectarnos a través de JDBC a las bases de datos embebidas estudiadas anteriormente. Hemos de crear la base de datos *ejemplo* con las tablas *empleados* y *departamentos* y vamos a utilizar el mismo programa Java, sólo cambiaremos la carga del driver y la conexión a la base de datos.

En Windows supongamos que la base de datos *ejemplo* la tenemos en las carpetas *D:\DB\SQLite, D:\DB\HSQLDB\ejemplo, D:\DB\H2* y *D:\DB\DERBY*.

En Linux en las carpetas */home/usuario/DB/SQLITE/, /home/usuario/DB/HSQLDB/, /home/usuario/DB/H2* y */home/usuario/DB/DERBY/*. Para realizar las pruebas necesitaremos tener el conector Java (fichero JAR) correspondiente para cada una de las bases de datos.

Conexión a SQLite.

Para conectarnos a SQLite necesitamos la librería el jar sql, se puede descargar desde esta URL, y existen varias versiones:

<http://www.java2s.com/Code/Jar/s/Downloadsqlitejdbc372jar.htm>

Partimos del programa Java inicial que recorre la tabla *departamentos* (su nombre es *Main.java*) de la base de de datos *ejemplo* de MySQL. En el entorno gráfico que usemos para ejecutar el programa incluimos el fichero JAR o lo incluimos en el CLASSPATH si lo ejecutamos desde la línea de comandos.

En el programa Java cambiamos dos cosas: la carga del driver, en este caso se llama **org.sqlite.JDBC** y la conexión a la base de datos:

Class.forName("org.sqlite.JDBC");

Connection conexion = DriverManager.getConnection

("jdbc:sqlite:D:/DB/SQLITE/ejemplo.db");

**Si la base de datos no existe LA CREA.**

El ejemplo en Linux es similar, solo habría que cambiar en la conexión la carpeta donde se encuentra la base de datos: *"jdbc:sqlite:/home/usuario/DB/SQLITE/ejemplo.db"*.

Conexión a Apache Derby.

Para conectarnos a Apache Derby necesitamos la librería **derby.jar** (que se encuentra en la carpeta lib donde se instaló Derby). El driver para la conexión a la base de datos se llama: **org.apache.derby.jdbc.EmbeddedDriver**:

Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

Connection conexion = DriverManager.getConnection

("jdbc:derby:D:/DB/DERBY/ejemplo");

**Si se desea crear la BD se añade créate=true, por ejemplo::**

String sURL = "jdbc:derby:memory:myDB**;create=true**";

con = DriverManager.getConnection(sURL);

Conexión a HSQLDB.

Para conectarnos a HSQLDB necesitamos la librería **hsqldb.jar** que se puede obtener de la carpeta *lib* obtenida al descomprimir el fichero **hsqldb-2.3.3.zip**. En este caso el driver se llama **org.hsqldb.jdbcDriver** y la conexión a la base de datos es la siguiente:

Class.forName("org.hsqldb.jdbcDriver" );

Connection conexion = DriverManager.getConnection

("jdbc:hsqldb:file:D:/DB/HSQLDB/ejemplo/ejemplo");

Si no nos funciona la última versión probamos a descargar otra versión inferior desde esta URL:

<http://hsqldb.org/download/>

Conexión a H2.

Para conectarnos a H2 necesitamos el jar de la versión descargada, por ejemplo, **h2-1.4.191.jar** que se puede obtener de la carpeta ***bin*** que se encuentra al descomprimir el fichero de la versión descargada. El driver se llama **org.h2.Driver** y la conexión a la base de datos es la siguiente

Class.forName("org.h2.Driver" );

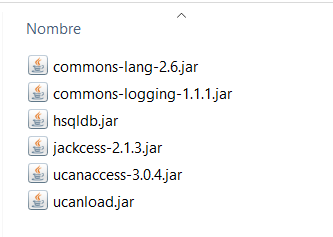
Connection conexion = DriverManager.getConnection

("jdbc:h2:D:/DB/H2/ejemplo/ejemplo","sa","");

En este caso, es necesario incluir el nombre del usuario y la clave en la conexión. El nombre es “*sa*“ y la clave se dejó en blanco cuando se creó la base de datos.

Conexión a Access.

Para Acces se utilizan varios jar:



Para conectarnos a una base de datos Access necesitamos las siguientes librerías (versión 3.0): **commons-lang-2.6.jar, commons-logging-1.1.1.jar, hsqldb.jar, jackcess-2.1.2.jar, ucanaccess-3.0.2.jar,** y **ucanload.jar.**

La última versión se puede descargar de aquí: [*http://ucanaccess.sourceforge.net/site.html*](http://ucanaccess.sourceforge.net/site.html)*, o* [*https://sourceforge.net/projects/ucanaccess/files/*](https://sourceforge.net/projects/ucanaccess/files/)

Al acceder al sitio podremos descargar un fichero similar a *UCanAccess-3.0.4-src.zip* con ejemplos, o el fichero *UCanAccess-3.0.4-bin.zip* que contiene los jar.

Para conectarnos a una base de datos access escribiremos:

Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");

Connection conn = DriverManager.getConnection

("jdbc:ucanaccess://mibasedatosaccess");

Por ejemplo, quiero conectarme a la base de datos ***ejemplo.mdb***, que la tengo guardada en la carpeta raíz del proyecto, en la conexión escribiré lo siguiente:

Connection conn = DriverManager.getConnection

("***jdbc:ucanaccess://./ejemplo.mdb***");

Conexión a MySQL.

Para conectarnos a MySQL necesitamos la librería **mysql-connector-java-5.1.38-bin.jar**, que podemos descargar desde la URL *http://dev.mysql.com/downloads/connector/j/*. Se descarga un fichero ZIP, y dentro de él se encuentra el JAR. El driver se llama **com.mysql.jdbc** y la conexión es la siguiente:

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

Connection conexion = DriverManager.getConnection

("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");

Será necesario seguir las instrucciones que marque el conector, pues puede que su declaración cambie con las versiones del conector.

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

Class.forName("com.mysql.cj.jdbc.Driver");

Conexión a MariaDB.

Class.*forName* ("org.mariadb.jdbc.Driver");

Connection conexion = DriverManager.*getConnection* ("jdbc:mariadb://localhost:3306/ejemplo", "root", "");

Conexión a Oracle.

Para conectarnos mediante JDBC usamos el driver *JDBC Thin*. Se puede descargar desde la página web de Oracle (es necesario comprobar antes la versión de la base de datos instalada), desde la dirección *https://www.oracle.com/database/technologies/appdev/jdbc-downloads.html*.

Para versiones más antiguas de Oracle se utiliza el **ojdbc6.jar**. Necesitamos saber el nombre de servicio que usa la base de datos para incluirlo en la URL de la conexión. Normalmente para la versión *Express Edition* el nombre es *XE*. El driver se llama **oracle.jdbc.driver.OracleDriver**, y la conexión a la base de datos es la siguiente:

Class.forName ("oracle.jdbc.driver.OracleDriver");

Connection conexion = DriverManager.getConnection

("jdbc:oracle:thin:@localhost:1521:XE",

"ejemplo24", "ejemplo24");

-- USER SQL

CREATE USER ejemplo24 IDENTIFIED BY "ejemplo24"

DEFAULT TABLESPACE "USERS"

TEMPORARY TABLESPACE "TEMP";

-- QUOTAS

-- ROLES

GRANT "CONNECT" TO ejemplo24 ;

GRANT "RESOURCE" TO ejemplo24 ;

-- SYSTEM PRIVILEGES

GRANT CREATE VIEW TO ejemplo24 ;", "ejemplo");