**Base de datos**

* Una base de datos es una colección de datos clasificados y estructurados, que son guardados en uno o varios ficheros.
* Para crear y manipular B.D. existen varios sistemas administradores de B.D. como Access, MySQL , SQL Server, Oracle, etc.
* Cada B.D. tiene su forma de recibir las consultas y su forma de devolver los resultados.
* SQL es un lenguaje estándar para interactuar con bases de datos relacionales y es soportado por casi todos los sistemas administradores de B.D.

**ODBC**

En una aplicación java que realiza acceso a B.D. se intenta escribir código que sea independiente del propietario de la B.D.

Para ello debemos establecer la conexión con la B.D. de la forma más general posible.

Microsoft creo un drive o controlador bastante estándar **ODBC** (protocolo estándar para el acceso a servidores SQL de B.D.) para acceder a cualquier B.D.

* ODBC debe instalarse en cada ordenador. Panel de control->Herramientas administrativas ->origen de datos
* Nos permite utilizar una interfaz única para los distintos tipos de B.D. generalmente se utilizan con Windows.
* En nuestro programa utilizaremos siempre nombres lógicos, abstrayéndonos de futuros cambios en los niveles inferiores.

**JDBC**

Conjunto de clases diseñadas en java para poder acceder a cualquier B.D., el acceso se realiza a través de drivers semejantes a ODBC.

* ODBC escrito en C no portable
* JDBC escrito en java portable.
  + JDBC es autoinstalable y portable
  + JDBC permite el acceso a varios servidores.

**Puente JDBC-ODBC**

Proporciona acceso a B.D., desde JDBC a través de un drive ODBC aprovechando lo que ya existe.

* Se utiliza para acceder a B.D. con aplicaciones Windows.
* Tiene que estar el drive ODBC instalado en la máquina.
* Accede a B.D locales no en red.

**Conexión a base de datos con JDBC o el puente JDBC-ODBC**

* Debemos establecer una conexión con la base de datos. Esta conexión se realiza a través de un controlador o drive que traduce los mensajes del sistema de B.D. a mensajes de alto nivel en un lenguaje específico y viceversa.
* Realizar las consultas necesarias a la base de datos.
* Recibir los resultados obtenidos con las consultas, para posteriormente poder procesarlos.
* El conjunto de clases de JDBC esta en dos paquetes java.sql y javax.sql
* Para ello debemos trabajar con las clases :

Driver, DriverManger, DriverPropertyInfo, Connection, DataBaseMetaData, Statement, PrepareStatement, ResultSet, ResultsetMetaData.

* Debemos definir cuatro objetos:

**DriverManager :** controlador

**Connection conexión :** conexión

**Statement sentenciaSQL :** envío sentencias sql.

**ResultSet resultado:** resultado.

**Cargar drive**:

Permite conectarnos a una base de datos**.** Cargamos los controladores específicos para la base de datos a utilizar.

El método forName de la clase Class nos crea un objeto de la clase del controlador.

**Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); Para Access**

**Class.forName("com.mysql.jdbc.Driver” ); Para MySQL**

El constructor del objeto controlador llama al método **registerDriver** para registrar el controlador.

Algunas veces veremos la llamada

**Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();**

Para crear el objeto, pero no hace falta pues se crea automáticamente.

**Realizar la conexión**

Una vez registrado el controlador, se solicita a DriverManager que proporcione una conexión

* Clase **DriverManager**: maneja todos los detalles del establecimiento de la conexión. Devuelve una conexión abierta que gestiona los controladores instalados en la máquina virtual de java.
* Tenemos que pasarle la Url del controlador adecuado.
* Para hacer referencia a la base de datos lo podemos hacer directamente mediante la IP, usando jdbc, o mediante una referencia al puente jdbc-odbc.

**String usuario=””;**

**String contaseña=””;**

**Connection conexion = DriverManager.getConnection ( URL, usuario, contraseña);**

Para Access:

**conexion=DriverManager.getConnection("jdbc:odbc:NombreLogico”,””,”“);**

**o**

**conexión=DriverManager.getConnection("jdbc:odbc:NombreLogico”,usuario, contraseña);**

NombreLogico: establecido en el origen de datos.

Para MYSQL

**conexion=DriverManager.getConnection**

**("jdbc:mysql://127.0.0.1:3306/nombreBD",””,””);**

Nombre servidor Ip: 127.0.0.1

Puerto mysql: 3306

Nombre base de datos: nombreBD

**Obtención del objeto Statement**

Una vez que disponemos de la conexión adecuada, creamos una instancia de la clase Statement por medio del método **createStatement**. Con el objeto **Statement**  enviamos nuestras sentencias SQL al controlador de la base de datos.

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

Podemos pasarle dos argumentos que serán dos constantes:

* El primero:

Por defecto es **TYPE\_FORWARD\_ONLY** es decir vas del primer registro al último y no puedes volver atrás.

Con las otras dos podemos desplazarnos en todos los sentidos.

**TYPE\_SCROLL\_INSENSITIVE**, no refleja los cambios hechos mientras estaba abierta

y **TYPE\_SCROLL\_SENSITIVE**. Refleja los cambios hechos mientras estaba abierta

Los tres tipos de hojas de resultados harán visibles los resultados si se cierran y se vuelve a abrir.

* El Segundo:

resultado es de sólo lectura: **CONCUR\_READ\_ONLY** por defecto

o actualizable **CONCUR\_UPDATABLE**

Para poder modificar la tabla pondríamos:

**sentenciaSql=conexion.createStatement(**

**java.sql.ResultSet.TYPE\_SCROLL\_INSENSITIVE,**

**java.sql.ResultSet.CONCUR\_UPDATABLE)**

**Ejecutar sentencias SQL**

Una vez que tenemos creado el objeto **Statement**, podemos utilizarlo para realizar las sentencias SQL necesarias para el manejo de la tabla.

* Si no devuelve nada usamos el método **execute()**.

**Crear tabla:**

**sentenciaSQl.execute**(“CREATE TABLE nombreTabla

( id\_alumno INTEGER , apellidos VARCHAR(24) NOT NULL , nombre VARCHAR(18) NOT NULL) ”);

* Si devuelve datos: Tendremos que crear un objeto **ResultSet** que será el que almacena todos los datos obtenidos de la base de datos. Proporcionan un número variable de filas y columnas**,** las filas empiezan por el valor 1.

**Consulta** sentencia **executeQuery**():

**ResultSet resultado = SentenciaSQL.executeQuery** ("SELECT \* FROM nombreTabla");

Utilizaremos el método **executeQuery()** siempre que queramos ejecutar sentencia Select de sql y el método **executeUpdate()** para actualizar las tablas de la base de datos (modificar, insertar y borrar) el cual devuelve un entero que indica cuantas líneas de la tabla fueron actualizadas.

**Insertar** para escribir datos en la tabla**:**

**sentenciaSQL.executeUpdate**(“INSERT INTO nombreTabla “ +

“VALUES (“ + cod\_alumno + ” , ‘ ” + apellido+ ” ’ , ‘ “ + nombre + “ ‘ “ +…..”)”);

**Modificar** datos, cuando encontremos fila:

**sentenciaSQL.executeUpdate(** “UPDATE nombreTabla SET col1=cambio, col2=cambio,….WHERE codigo= “codigoBuscar” “);

**Borrar** registro:

**sentenciaSQL.executeUpdate(“ DELETE FROM nombreTabla WHERE codicion”);**

**Borrar tabla:**

**sentenciaSQL.executeUpdate(“DROP TABLE nombreTabla”);**

**Para recorrer la tabla ResultSet**

* Cada objeto Resulset contiene un cursor que inicialmente se encuentra, en la posición anterior a la primera fila de la tabla.
* Existen una serie de métodos que nos permiten mover el cursor a lo largo de la tabla.

**Métodos interfaz ResultSet**

* **beforeFirst**(): Mover cursor antes 1ª fila.
* **first():** mover cursor 1ª fila.
* **last**(): mover cursor última fila.
* **afterLast**(): después última fila.
* **previous**(): a la fila anterior a la actual.
* **next():** a la fila siguiente.
* **absolute(fila):** a una fila especificada.
* **relative(n**): mover n filas a partir de la actual. Todos estos métodos devuelven true o false.
* **getRow():** devuelve un int que indica el nº de fila donde está el cursor.
* **isLast()** devuelve true si el cursor está en la última fila.
* **isFirst(), isBeforeFrist**(). Etc.

Para obtener los valores de las filas:

* **getString(String**): recupera la columna especificada por String. resultado.getString(nombre);
* **getString(int**): recupera la columna de esa posición.
* **getObject(int)**
* Para otro tipo de columnas: **getInt(). getLong(), getFloat**().etc.

Para **modificar**

resultado=sentenciaSQL.executeQuery( “SELECT \* FROM tabla”);

while(resultado.next())

{ if(condicioBusqueda) {

resultado.updateString(1, “nuevoNombre”);

resultado.updateString(2,direccion”);

**resultado.updateRow();** } }

Para **insertar**.

**resultado.moveToInsertRow();**

resultado.updateString(1,”nombre”);

resultado.updateString(2, “direccion”);

……..

**resultado.insertRow():**

Para **borrar**:

**resultado.deleteRow();**

#### Para consultar presentando los datos en un JTable:

Para meter los datos en el **JTable**, [usaremos un DefaultTableModel](http://www.chuidiang.com/chuwiki/index.php?title=JTable)

try{

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

Connection conexion= DriverManager.getConnection("jdbc:odbc:producto1","","");

Statement sentenciaSql=conexion.createStatement

(java.sql.ResultSet.TYPE\_SCROLL\_INSENSITIVE,

java.sql.ResultSet.CONCUR\_UPDATABLE);

ResultSet resultado= sentenciaSql.executeQuery("SELECT \* FROM tabla");

//obtenemos el tipo del modelo de JTable, para poder añadir líneas, si no

//sabemos cuantas líneas son

DefaultTableModel modelo = (DefaultTableModel)tabla.getModel();

// Creamos las columnas.  
modelo.addColumn("id");  
modelo.addColumn("nombre");  
modelo.addColumn("direccion");

// Bucle para cada resultado en la consulta  
while (resultado.next())  
{  
   // Se crea un vector que será una de las filas de la tabla.   
   Vector fila = new Vector();

fila.addElement(resultado.getString(1));

fila.addElement(resultado.getString(2));

fila.addElement(resultado.getString(3));

modelo.addRow(fila);

}

conexion.close();

} catch (ClassNotFoundException e) {

System.out.println(e.getMessage());

}

catch(SQLException ee){

System.out.println(ee.getMessage()); } }

Con esto basta. Todo lo que hagamos en el **DefaultTableModel** se refresca automáticamente en el **JTable**. Por ello, según se va ejecutando este código, en el **JTable** se irán añadiendo automáticamente los datos.

#### LOS METADATOS

El código que acabamos de hacer está claro que es muy dependiente de la tabla de base de datos que estamos consultando. Hay que añadir tantas columnas como campos tenga el resultado de base de datos. El nombre de las columnas depende de qué estemos consultando, es decir, un nombre de persona, una marca de coche, etc.

Sin embargo, existe una forma más o menos automática de hacer esto. Dado un **ResultSet** que obtenemos como consulta de base de datos, podemos interrogarle sobre cuántas columnas tiene y cuáles son los nombres de sus campos. Con ello, podemos hacer el código anterior de forma genérica, de tal manera que sólo teniendo el **ResultSet** y **DefaultTableModelo**, podemos rellenarlo sin saber en absoluto qué consulta ni qué tablas se han consultado.

Para obtener esta información, hay que pedirle los **MetaDatos** al **ResulSet**. El código que lo hace es este

ResultSetMetaData metaDatos = resultado.getMetaData();

La clase **ResultSetMetaData** contiene toda la información sobre los campos de base de datos contenidos en el **ResultSet**. Necesitamos preguntarle por el número de columnas y por la etiqueta para cada una de las columnas. Este es el código que lo hace

// Se obtiene el número de columnas.  
int numeroColumnas = metaDatos.getColumnCount();

**Métodos de getMetaData()**

Podemos obtener información de un conjunto de resultados, como: número de filas, número de columnas, nombre de sus columnas, tipos, etc. A través del método **getMetaData** del objeto **ResultSe**t.

* int numColumnas= resultado.getMetaData().getColumnCount();
* int getColumnDisplaySize(int col) tamaño máximo del campo.
* int getColumnName(int col) nombre de una columna.
* int getColumnType(int col) tipo SQL de un campo.
* String getColumnTypeName(int col) nombre del tipo del campo.
* boolean isWritable(int col) indica si podemos escribir en el.
* boolean isreadOnly(int col) si es solo de lectura

## Utilizar Sentencias Preparadas

Algunas veces es más conveniente o eficiente utilizar objetos **PreparedStatement** para enviar sentencias SQL a la base de datos. Este tipo especial de sentencias se deriva de una clase más general, Statement, que ya conocemos.

### Cuándo utilizar un Objeto PreparedStatement

Si queremos ejecutar muchas veces un objeto **Statement**, reduciremos el tiempo de ejecución si utilizamos un objeto **PreparedStatement**, en su lugar.

La característica principal de un objeto **PreparedStatement** es que, al contrario que un objeto **Statement**, se le entrega una sentencia SQL cuando se crea. La ventaja de esto es que en la mayoría de los casos, esta sentencia SQL se enviará al controlador de la base de datos inmediatamente, donde será compilado. Como resultado, el objeto **PreparedStatement** no sólo contiene una sentencia SQL, sino una sentencia SQL que ha sido precompilada. Esto significa que cuando se ejecuta la **PreparedStatement**, el controlador de base de datos puede ejecutarla sin tener que compilarla primero.

Aunque los objetos **PreparedStatement** se pueden utilizar con sentencias SQL sin parámetros, probablemente nosotros utilizaremos más frecuentemente sentencias con parámetros. La ventaja de utilizar sentencias SQL que utilizan parámetros es que podemos utilizar la misma sentencia y suministrar distintos valores cada vez que la ejecutemos.

### Crear un Objeto PreparedStatement

Al igual que los objetos **Statement**, creamos un objeto **PreparedStatement** con un objeto **Connection**. Utilizando nuestra conexión **con**, podríamos escribir lo siguiente para crear un objeto **PreparedStatement** que tome dos parámetros de entrada.

PreparedStatement updateSales = con.prepareStatement(

"UPDATE cafe SET SALES = ? WHERE nombre LIKE ?");

La variable **updateSales** contiene la sentencia SQL, **"UPDATE cafe SET SALES = ? WHERE nombre LIKE ?"**, que también ha sido, en la mayoría de los casos, enviada al controlador de la base de datos, y ha sido precompilado.

### Suministrar Valores para los Parámetros de un PreparedStatement

Necesitamos suministrar los valores que se utilizarán en los lugares donde están las marcas de interrogación, si hay alguno, antes de ejecutar un objeto **PreparedStatement**. Podemos hacer esto llamado a uno de los métodos **setXXX** definidos en la clase **PreparedStatement**. Si el valor que queremos sustituir por una marca de interrogación es un **int** de Java, podemos llamar al método **setInt**. Si el valor que queremos sustituir es un **String** de Java, podemos llamar al método **setString**, etc. En general, hay un método **setXXX** para cada tipo Java.

Utilizando el objeto **updateSales** del ejemplo anterior, la siguiente línea de código selecciona la primera marca de interrogación para un **int** de Java, con un valor de 75.

updateSales.setInt(1, 75);

Cómo podríamos asumir a partir de este ejemplo, el primer argumento de un método **setXXX** indica la marca de interrogación que queremos seleccionar, y el segundo argumento el valor que queremos ponerle. El siguiente ejemplo selecciona la segunda marca de interrogación con el string "**Colombian**".

updateSales.setString(2, "Colombian");

Después de que estos valores hayan sido asignados para sus dos parámetros, la sentencia SQL de **updateSales** será equivalente a la sentencia SQL que hay en string **updateString** que utilizando en el ejemplo anterior. Por lo tanto, los dos fragmentos de código siguientes consiguen la misma cosa.

Código 1.

String updateString = "UPDATE COFFEES SET SALES = 75 " +

"WHERE COF\_NAME LIKE 'Colombian'";

stmt.executeUpdate(updateString);

Código 2.

PreparedStatement updateSales = con.prepareStatement(

"UPDATE COFFEES SET SALES = ? WHERE COF\_NAME LIKE ? ");

updateSales.setInt(1, 75);

updateSales.setString(2, "Colombian");

updateSales.executeUpdate().

Utilizamos el método **executeUpdate** para ejecutar ambas sentencias **stmt** **updateSales**. Observa, sin embargo, que no se suministran argumentos a **executeUpdate** cuando se utiliza para ejecutar **updateSales**. Esto es cierto porque **updateSales** ya contiene la sentencia SQL a ejecutar.

Mirando esto ejemplos podríamos preguntarnos por qué utilizar un objeto **PreparedStatement** con parámetros en vez de una simple sentencia, ya que la sentencia simple implica menos pasos. Si actualizáramos la columna **SALES** sólo una o dos veces, no sería necesario utilizar una sentencia SQL con parámetros. Si por otro lado, tuvieramos que actualizarla frecuentemente, podría ser más fácil utilizar un objeto **PreparedStatement**, especialmente en situaciones cuando la utilizamos con un bucle while para seleccionar un parámetro a una sucesión de valores. Veremos este ejemplo más adelante en esta sección.

Una vez que a un parámetro se ha asignado un valor, el valor permanece hasta que lo resetee otro valor o se llame al método **clearParameters**. Utilizando el objeto **PreparedStatement**: **updateSales**, el siguiente fragmento de código reutiliza una sentencia prepared después de resetar el valor de uno de sus parámetros, dejando el otro igual.

updateSales.setInt(1, 100);

updateSales.setString(2, "French\_Roast");

updateSales.executeUpdate();

// changes SALES column of French Roast row to 100

updateSales.setString(2, "Espresso");

updateSales.executeUpdate();

// changes SALES column of Espresso row to 100 (the first

// parameter stayed 100, and the second parameter was reset

// to "Espresso")

### Utilizar una Bucle para asignar Valores

Normalmente se codifica más sencillo utilizando un bucle **for** o **while** para asignar valores de los parámetros de entrada.

El siguiente fragmento de código demuestra la utilización de un bucle **for** para asignar los parámetros en un objeto **PreparedStatement**: **updateSales**. El array **salesForWeek** contiene las cantidades vendidas semanalmente. Estas cantidades corresponden con los nombres de los cafés listados en el array **coffees**, por eso la primera cantidad de **salesForWeek** (175) se aplica al primer nombre de café de **coffees** ("**Colombian**"), la segunda cantidad de **salesForWeek** (150) se aplica al segundo nombre de café en **coffees** ("**French\_Roast**"), etc. Este fragmento de código demuestra la actualización de la columna **SALES** para todos los cafés de la tabla **COFFEES**

PreparedStatement updateSales;

String updateString = "update COFFEES " +

"set SALES = ? where COF\_NAME like ?";

updateSales = con.prepareStatement(updateString);int [] salesForWeek = {175, 150, 60, 155, 90};

String [] coffees = {"Colombian", "French\_Roast", "Espresso",

"Colombian\_Decaf", "French\_Roast\_Decaf"};

int len = coffees.length;

for(int i = 0; i < len; i++) {

updateSales.setInt(1, salesForWeek[i]);

updateSales.setString(2, coffees[i]);

updateSales.executeUpdate();

}

Cuando el propietario quiera actualizar las ventas de la semana siguiente, puede utilizar el mismo código como una plantilla. Todo lo que tiene que haces es introducir las nuevas cantidades en el orden apropiado en el array **salesForWeek**. Los nombres de cafés del array **coffees** permanecen constantes, por eso no necesitan cambiarse. (En una aplicación real, los valores probablemente serían introducidos por el usuario en vez de desde un array inicializado).

### Valores de retorno del método executeUpdate

Siempre que **executeQuery** devuelve un objeto **ResultSet** que contiene los resultados de una petición al controlador de la base datos, el valor devuelto por **executeUpdate** es un **int** que indica cuántas líneas de la tabla fueron actualizadas. Por ejemplo, el siguiente código muestra el valor de retorno de **executeUpdate** asignado a la variable **n**.

updateSales.setInt(1, 50);

updateSales.setString(2, "Espresso");

int n = updateSales.executeUpdate();

// n = 1 because one row had a change in it

La tabla **COFFEES** se ha actualziado poniendo el valor **50** en la columna **SALES** de la fila correspondiente a **Espresso**. La actualización afecta sólo a una línea de la tabla, por eso **n** es igual a **1**.

Cuando el método **executeUpdate** es utilizado para ejecutar una sentecia DDL, como la creación de una tabla, devuelve el **int**: **0**. Consecuentemente, en el siguiente fragmento de código, que ejecuta la sentencia DDL utilizada pra crear la tabla **COFFEES**, **n** tendrá el valor **0**.

int n = executeUpdate(createTableCoffees); // n = 0

Observa que cuando el valor devuelto por **executeUpdate** sea **0**, puede significar dos cosas: (1) la sentencia ejecutada no ha actualizado ninguna fila, o (2) la sentencia ejecutada fue una sentencia DDL.

**Generar URL de conexión**

El formato general de la URL de conexión es

jdbc:sqlserver://[serverName[\instanceName][:portNumber]][;property=value[;property=value]]

donde:

* jdbc:sqlserver:// (obligatorio) es el subprotocolo y es constante.
* serverName (opcional) es la dirección del servidor con el que se establece la conexión. Puede ser un DNS o una dirección IP o bien un localhost o 127.0.0.1 para el equipo local. Si no se especifica en la URL de conexión, es necesario especificar el nombre del servidor en la colección de propiedades.
* instanceName (opcional) es la instancia para establecer la conexión con serverName. Si no se especifica, se establece una conexión con la instancia predeterminada.
* portNumber (opcional) es el puerto para establecer la conexión con serverName. El valor predeterminado es 1433. Si usa el valor predeterminado, no es necesario especificar el puerto ni el signo ":" precedente en la dirección URL.

**Crear una conexión a SQLServer**

* Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
* String connectionUrl = "jdbc:sqlserver://localhost:1433;" +  
  "databaseName=NOMBRE DE LA BASE DE DATOS ;user=sa ; password=tuclave;");
* Connection con = DriverManager.getConnection(connectionUrl);
* Esta técnica creará una conexión a una base de datos usando el primer controlador disponible de la lista de controladores que se pueda conectar correctamente con la URL dada.

Lo primero que hay que hacer para conectar una aplicación Java con la base de datos Microsoft SQL Server 2005, es bajar el controlador que nos va a permitir manejar la conexión.

Para ello vamos al [Controlador JDBC de Microsoft SQL Server 2005](http://www.microsoft.com/downloads/details.aspx?familyid=E22BC83B-32FF-4474-A44A-22B6AE2C4E17&displaylang=es),, hacemos clic en el botón Download y en la página que nos abre, donde nos muestra la licencia, elegimos si queremos descargar la versión para Windows o para Linux.  
Una vez que tenemos descargado el archivo **sqljdbc\_1.2.2828.100\_enu.exe** (si elegimos Windows), lo ejecutamos y elegimos en que carpeta queremos descomprimir el archivo. Luego, debemos buscar el archivo **sqljdbc.jar** y lo copiamos a la carpeta Lib de nuestro proyecto donde tenemos el resto de nuestras librerías.

**try{  
Class.forName(“com.microsoft.sqlserver.jdbc.SQLServerDriver”); // 2005 version  
Connection con = DriverManager.getConnection(“jdbc:sqlserver://localhost:1433;**

**databaseName=p1;user=a;password=123456;”);**

**con.close();//se cierra la conexion con la base de datos**

**}  
catch ( SQLException excepcionSql){}**

**Connection con = DriverManager.getConnection("jdbc:sqlserver://NOMBREDB:1433", "usuario", "contraseña");**

Debes modificar el código a el puerto que está utilizando sql , nombre de la BD, Usuario y contraseña.