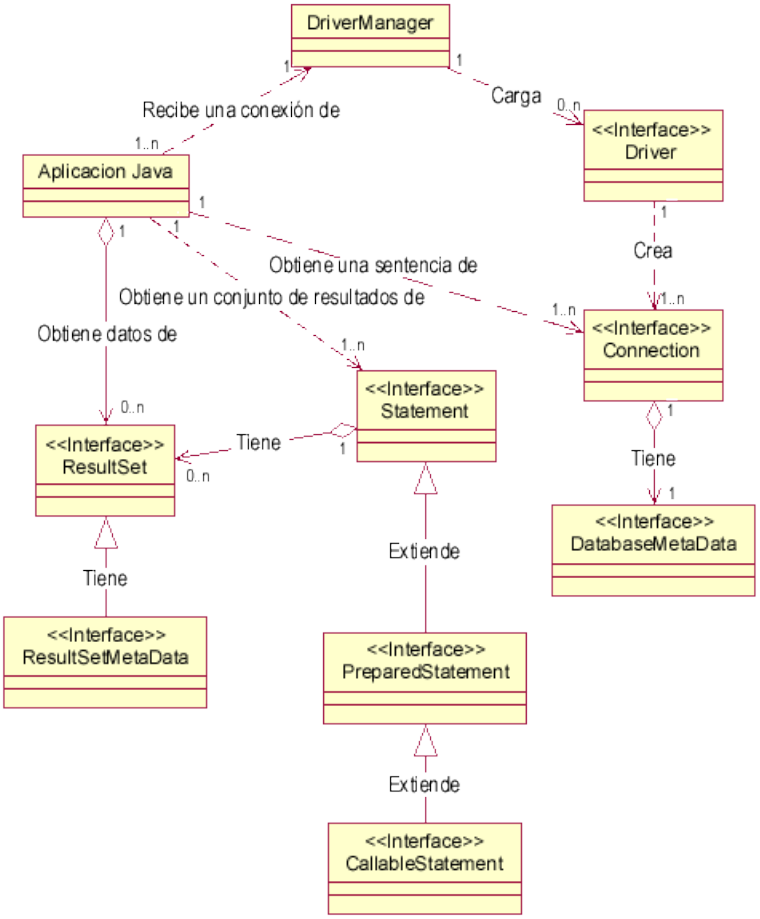
UNIDAD 13.

Bases de datos

Java utiliza la interfaz JDBC para comunicarse con las bases de datos. JDBC está compuesto por un número determinado de clases que permiten a cualquier programa escrito en java acceder a una base de datos.

Las clases relacionadas con el acceso a Bases de Datos se encuentran en el paquete **java.sql.**

JDBC define ocho interfaces para operaciones con bases de datos, de las que se derivan las clases correspondientes. La figura siguiente con nomenclatura UML, muestra la interrelación entre estas clases según el modelo de objetos de la especificación de JDBC.



Clase/Interface	*Descripción*
1. 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 a más de una base de datos.
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, típicamente procedimientos almacenados.
ResultSet	Contiene las filas o registros obtenidos al ejecutar un SELECT.
ResultSetMetadata	Permite obtener información sobre un **ResultSet**, como el número de columnas, sus nombres, etc.

La clase que se encarga de cargar inicialmente todos los drivers JDBC disponibles es DriverManager. **Una aplicación puede utilizar DriverManager para obtener un objeto de tipo conexión, Connection, con una base de datos**.

Tipos de controladores:

1. **Puente JDBC-ODBC:** Éste tipo de controlador es un controlador que crea una aplicación dependiente de la plataforma, ya que ODBC está definido para Windows. Debido a que java es por definición un lenguaje independiente de la plataforma, éste tipo de controlador es poco recomendable.
2. **100% java nativo:** Este controlador se salta la capa ODBC. Realiza las llamadas a las librerías nativas del sistema gestor de

las bases de datos de forma directa. De cualquier forma, sigue siendo dependiente de la plataforma.

3. **100% java /Protocolo nativo:** Está diseñado 100% en java y se independiza totalmente de la plataforma. El único inconveniente es que se crea una dependencia con un tipo de servidor de un sistema gestor de bases de datos. Por ejemplo, el protocolo de red que sigue Microsoft SQL Server es absolutamente diferente de el de Oracle.
4. **100% java/ Protocolo independiente:** Es el tipo de controlador más independiente de la plataforma. Traduce las órdenes JDBC directamente al protocolo de red que implementa ó emplea el sistema gestor de base de datos.

La clase DriverManager:

Se utilizará para gestionar los controladores JDBC, y es la encargada de devolvernos la conexión con la Base de Datos.

Previo a la conexión con la base de datos, es preciso cargar el conjunto de clases que permiten la conexión de java con el sistema gestor de bases de datos:

El método de clase **Class.forname(String controlador),** permite cargar el driver adecuado para el sistema gestor de bases de datos que estemos utilizando**.**

El controlador que utilizaremos para SQL Server será:

Class.*forName*("com.microsoft.jdbc.sqlserver.SQLServerDriver")

El controlador que utilizaremos para Mysql será:

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

Métodos de DriverManager:

Connection getConnection(dirección, nombre_usuario, clave)
realiza la conexión con la base de datos. Obtiene un objeto perteneciente al interfaz **Connection**. Éste objeto representa a la conexión con la base de datos.

Dicho interfaz contiene métodos para la manipulación de bases de datos, lanzamiento de sentencias SQL, obtención de resultados de dichas consultas, etc.

Métodos de Connection:

Statement createStatement(): Crea un objeto que permite lanzar sentencias SQL, para solicitud de información a la base de datos.

Commit(): Todas las modificaciones que se han hecho en la base de datos desde la última sentencia commit ó rollback se confirma que se desean realizar.

Rollback(): deshace todas las operaciones que se han hecho desde la última sentencia commit ó rollback.

Close(): cierra la conexión con la base de datos.

IsClosed(): devuelve si se ha cerrado ó no la conexión con la base de datos.

SetTransactionIsolation(int nivel): Establece el nivel de aislamiento de la transacción. Existen cinco posibles niveles, que se determinan mediante las constantes de clase:

- **TRANSACTION_NONE**
- **TRANSACTION_READ_COMMITTED**
- **TRANSACTION_READ_UNCOMMITTED**
- **TRANSACTION_REPEATABLE_READ**
- **TRANSACTION_SERIALIZABLE**

Int getTransactionIsolation(): Devuelve el nivel de aislamiento de la transacción establecido en la conexión actual.

Los métodos commit, rollback, setTranslationIsolation y getTranslationIsolation están relacionados con el término **Transacción** en bases de datos.

Una **transacción** es un conjunto de operaciones sobre una base de datos, tales que se tratan como una sóla operación.

Se dice que una transacción es **ACID,** es decir:

- A: Atomicidad. En una transacción se realizan todas las operaciones ó no se realiza ninguna.
- C: Consistencia.
- Isolation: aislado.
- D: Durability. Permanencia de los cambios.

El interfaz Statement:

Contiene métodos necesarios para la ejecución de consultas SQL:

- **ResultSet executeQuery(String consulta_select):** Ejecuta una consulta de selección y devuelve un objeto ResultSet con los datos obtenidos por la consulta de selección.
- **Int executeUpdate(String consulta_modificación):** Permite ejecutar consultas de actualización, inserción ó eliminación. Devuelve un entero que representa el número de líneas que se han visto afectadas por la consulta.

El interfaz ResultSet:

Los objetos de éste interfaz almacenan los resultados obtenidos como consecuencia de la ejecución de una consulta de **selección.**

Mediante los métodos de ésta clase es posible acceder a los datos devueltos por la consulta.

Sólo con API 2.0. (Por ahora, los drivers de SQL Server no implementan API 2.0) :

Dependiendo de cómo se cree el objeto Statement, el objeto ResultSet podrá comportarse de distintos modos:

```
Statement st= miconexión.createStatement
    (ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
```

Éstos parámetros están determinados mediante las constantes de clase de ResultSet. Algunas de ellas son:

static int	**CONCUR_READ_ONLY** : El ResultSet se creará sin la posibilidad de actualizar filas.
static int	**CONCUR_UPDATABLE** : El ResultSet se creará con la posibilidad de actualizar filas.
static int	**TYPE_FORWARD_ONLY** : El ResultSet se creará únicamente con la posibilidad de avanzar en el recorrido de los datos (no retroceso).
static int	**TYPE_SCROLL_SENSITIVE:** Permite realizar movimientos por la estructura y es sensible a los cambios realizados por otras consultas**.**
static int	**TYPE_SCROLL_INSENSITIVE:** Permite realizar movimientos por la estructura pero no es sensible a los cambios realizados por otras consultas**.**

Métodos:

Void afterLast(): Coloca el cursor después del último dato de la estructura (para realizar el recorrido hacia atrás).

Void beforeFirst(): Coloca el cursor antes del primer dato de la estructura.

Boolean first(): Coloca el cursor en la primera tupla de la estructura. Devuelve false si la consulta no ha devuelto ningún valor.

Boolean last(): Coloca el cursor en la última tupla de la estructura. Devuelve false si la consulta no ha devuelto ningún valor.

Int getRows(): devuelve el número de fila actual.

String getString (String nombreColumna): Devuelve el valor asociado a la columna que se le pasa como argumento en forma de cadena de caracteres.

Int getInt(String nombreColumna): Devuelve el valor asociado a la columna que se le pasa como argumento en un tipo entero.

Boolean previous(): posiciona el cursor a la tupla anterior. Devuelve false si no hay registro anterior.

Boolean absolute(int núm_linea): Si el num_linea es positivo, posiciona el cursor en ese número de línea, empezando a contar por la primera. Si es negativo, empieza a contar por la última línea.

Boolean relative(int despl): Posiciona el cursor "despl" filas adelante (ó atrás si es negativo). Devuelve false si dicho desplazamiento no es posible realizarlo (no existe dicho número de fila).

Void refreshRow(): Obtiene la última versión del objeto RecordSet. Esto tiene sentido en caso de que el tipo de conexión sea **TYPE_SCROLL_SENSITIVE**, ya que las modificaciones realizadas en otras consultas, se reflejan aquí.

Inserción de datos con API 2.0:

Una vez creado el objeto ResultSet tal como lo vimos anteriormente (ejecución de una consulta de selección), los pasos a seguir son los siguientes.

Suponiendo que el objeto ResultSet tiene el nombre mirs:

1. **mirs.moveToInsertRow():** método que lleva el cursor después de la última fila, en modo inserción.
2. **mirs.updateString("nombre_campo", valor);**
3. **mirs.insertRow();**

Actualización de datos con API 2.0:

Una vez creado el objeto ResultSet tal como lo vimos anteriormente (ejecución de una consulta de selección), los pasos a seguir son los siguientes.

1. Movimiento del puntero del ResultSet hacia la posición deseada (con alguno de los métodos vistos anteriormente para movimiento del cursor).
2. Suponiendo que se pretende actualizar el dato una columna que contiene datos de tipo Float: **mirs.updateFloat("Precio", 5.30);** Ésta actualización afectaría únicamente al registro referenciado por el cursor.

Borrado de una fila con API 2.0:

1. Movimiento del puntero del ResultSet hacia la posición deseada (con alguno de los métodos vistos anteriormente para movimiento del cursor).
2. **mirs.deleteRow();**

Un Ejemplo Completo (SQL Server):

Previo a la realización de las siguientes clases java, es preciso crear en **SQL Server** y posteriormente ayudados por la aplicación **DBArtisan** una base de datos llamada **Colegio**, con una única tabla, llamada **alumnos,** con la siguiente especificación:

Create database colegio; (Consola de MySQL)

	Column Name	Datatype	Nulls
1	CODIGO	int	No
2	NOMBRE	varchar(100)	No
3	APELLIDOS	varchar(100)	No
4	EDAD	int	No

Y cuya **clave primaria** es el campo **CODIGO**.

La sentencia SQL que realiza la creación de dicha tabla es:

```
CREATE TABLE ALUMNOS
(
   CODIGO    int         NOT NULL,
   NOMBRE    varchar(100) NOT NULL,
   APELLIDOS varchar(100) NOT NULL,
   EDAD      int         NOT NULL,
   PRIMARY KEY (CODIGO)
)
go
```

BD.java: Ésta clase contiene métodos relacionados con la conexión a la Base de datos, métodos para realizar la carga de los drivers correspondientes y métodos para la ejecución de consultas.

```
import java.sql.*;

public class BD {

    private Connection con;
    private Statement st;

    public BD (){
        try {

        //carga el controlador
                Class contr= Class.forName
                ("com.microsoft.jdbc.sqlserver.SQLServerDriver");

                System.out.println(contr.toString());
                }
        catch (ClassNotFoundException cnfe) {
```

```
        System.out.println("com.microsoft.jdbc.sqlserver.SQLServerDriver");

                        }
                try {
                        //String dirección. Hace referencia a una URL, un recurso
de la aplicación
                        Connection micon= DriverManager.getConnection
("jdbc:microsoft:sqlserver://localhost:1433;databaseName=Comercio", "sa",
"cursojava");

                        Statement st=micon.createStatement();

                        this.con=micon;
                        this.st=st;
                        }
                catch (SQLException sqle) {
                        System.out.println("Error al establecer la conexion");
                }
        }

        //Ejecuta una consulta de Actualización de la base de datos
        //(inserción, actualización o borrado de datos).
        public void actualiza(String datos){
                try {
                        System.out.println(datos);
                        st.executeUpdate(datos);
                }
                catch (SQLException sqle) {
                        System.out.println("Error al ejecutar la consulta");
                }
        }
        //Ejecuta una consulta de selección, devolviendo el objeto //ResultSet
        con los datos obtenidos en la consulta.
        public ResultSet selecciona(String consulta) throws SQLException {

                        return st.executeQuery(consulta);

        }

        public ResultSet getResultSet() throws SQLException{

                return st.getResultSet();

        }

        public Connection getConex() {
                return con;
        }
```

```
}
```

Principal.java: Ésta clase es la encargada de crear un objeto de la clase BD, y manejar la base de datos con los métodos existentes en dicha clase.

```
import java.sql.*;

public class Principal {

	public static void main(String [] args) throws Exception{

		ResultSet rs=null;
		BD bd= new BD();

		//Inserción de 3 registros a la tabla.
		String sentencia1="INSERT INTO ALUMNOS (CODIGO,
NOMBRE, APELLIDOS, EDAD)" +
			" VALUES (1, 'Alberto', 'Lopez Perez', 12)";
		String sentencia2="INSERT INTO ALUMNOS (CODIGO,
NOMBRE, APELLIDOS, EDAD)" +
		" VALUES (2, 'Aurora', 'Roldan Pino', 13)";
		String sentencia3="INSERT INTO ALUMNOS (CODIGO,
NOMBRE, APELLIDOS, EDAD)" +
		" VALUES (3, 'Rafa', 'Martinez Cobos', 14)";

		System.out.println("Insertando registros en la BD...");
		bd.actualiza(sentencia1);
		bd.actualiza(sentencia2);
		bd.actualiza(sentencia3);

		//Recuperación de los datos.
		String seleccion="SELECT * FROM ALUMNOS";

		try {
			//obtencion de resultados del ResultSet
			System.out.println("Obteniendo registros de la BD...");
			rs=bd.selecciona(seleccion);
			while (rs.next()) {
				int c= rs.getInt("CODIGO");
				String n= rs.getString("NOMBRE");
				String a= rs.getString("APELLIDOS");
				int e= rs.getInt("EDAD");
				System.out.println(c + " "+ n + " "+ a + " "+ e + "\n");
				}
			}
		catch (SQLException sqle) {
			System.out.println("Error en la consulta");
			}
```

```
        //Eliminación de un registro de la base de datos.
                System.out.println("Eliminando registro con código 1 de la BD...");
                sentencia1="delete from ALUMNOS where codigo=1";
                bd.actualiza(sentencia1);

                try {
                        //obtencion de resultados del ResultSet después del
borrado
                        System.out.println("Obteniendo registros de la BD después
del borrado...");
                        rs=bd.selecciona(seleccion);
                        while (rs.next()) {
                                int c= rs.getInt("CODIGO");
                                String n= rs.getString("NOMBRE");
                                String a= rs.getString("APELLIDOS");
                                int e= rs.getInt("EDAD");
                                System.out.println(c + " "+ n + " "+ a + " "+ e + "\n");
                                }
                        }
                catch (SQLException sqle) {
                        System.out.println("Error en la consulta");
                        }

                //Actualización de un campo de la base de datos.
                System.out.println("Actualizando registro con código 3 de la
BD...");
                sentencia1="update alumnos set apellidos='Torres López' where
codigo=3";
                bd.actualiza(sentencia1);
                try {
                        //obtencion de resultados del ResultSet después del
borrado
                        System.out.println("Obteniendo registros de la BD después
del borrado...");
                        rs=bd.selecciona(seleccion);
                        while (rs.next()) {
                                int c= rs.getInt("CODIGO");
                                String n= rs.getString("NOMBRE");
                                String a= rs.getString("APELLIDOS");
                                int e= rs.getInt("EDAD");
                                System.out.println(c + " "+ n + " "+ a + " "+ e + "\n");
                                }
                        }
                catch (SQLException sqle) {
                        System.out.println("Error en la consulta");
                        }
                }
        }
```

Instalación de las librerías necesarias:

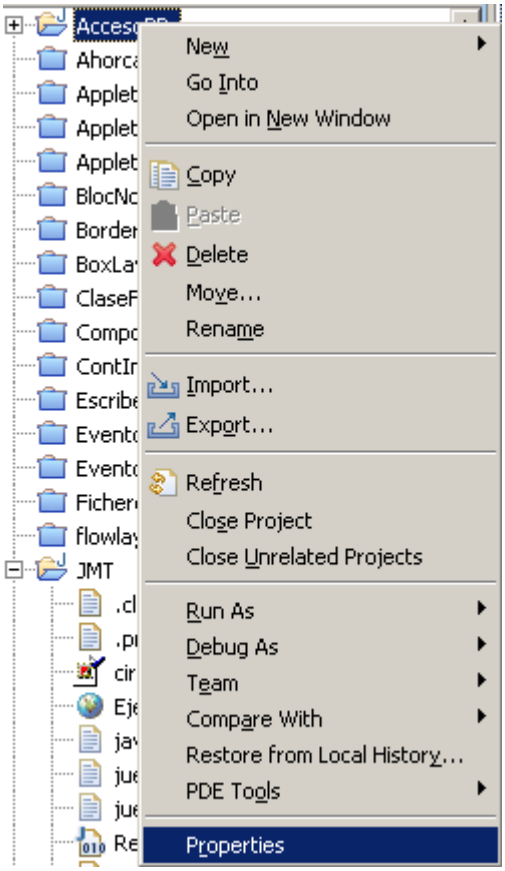
Para el correcto funcionamiento del ejemplo anterior, es preciso proporcionarle a **Eclipse** los ficheros **jar** que contienen el conjunto de clases que **SQL Server** proporciona para JDBC.

Dichos fichero son:

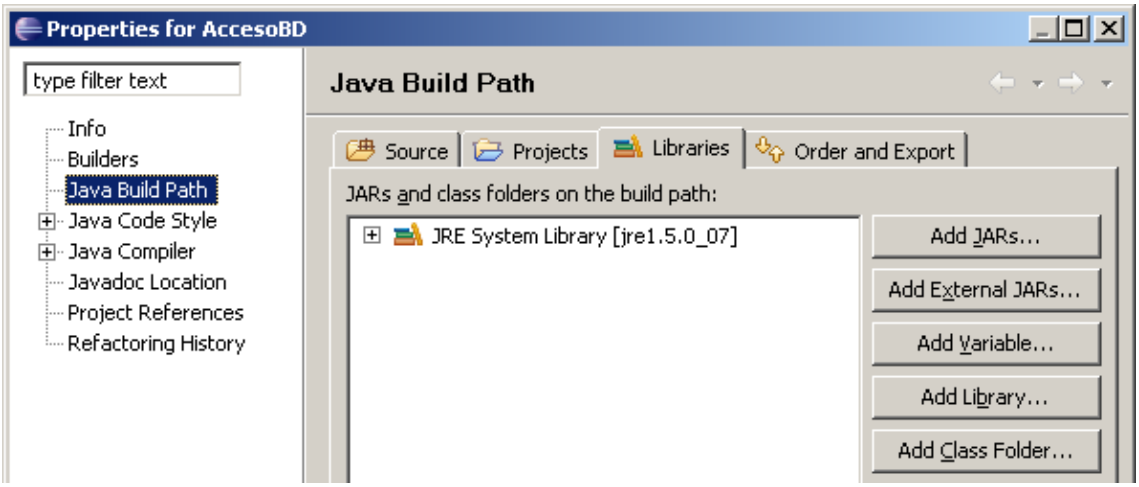
1. msbase.jar
2. mssqlserver.jar
3. msutil.jar

Los pasos a seguir en Eclipse para agregar las librerías anteriores son:

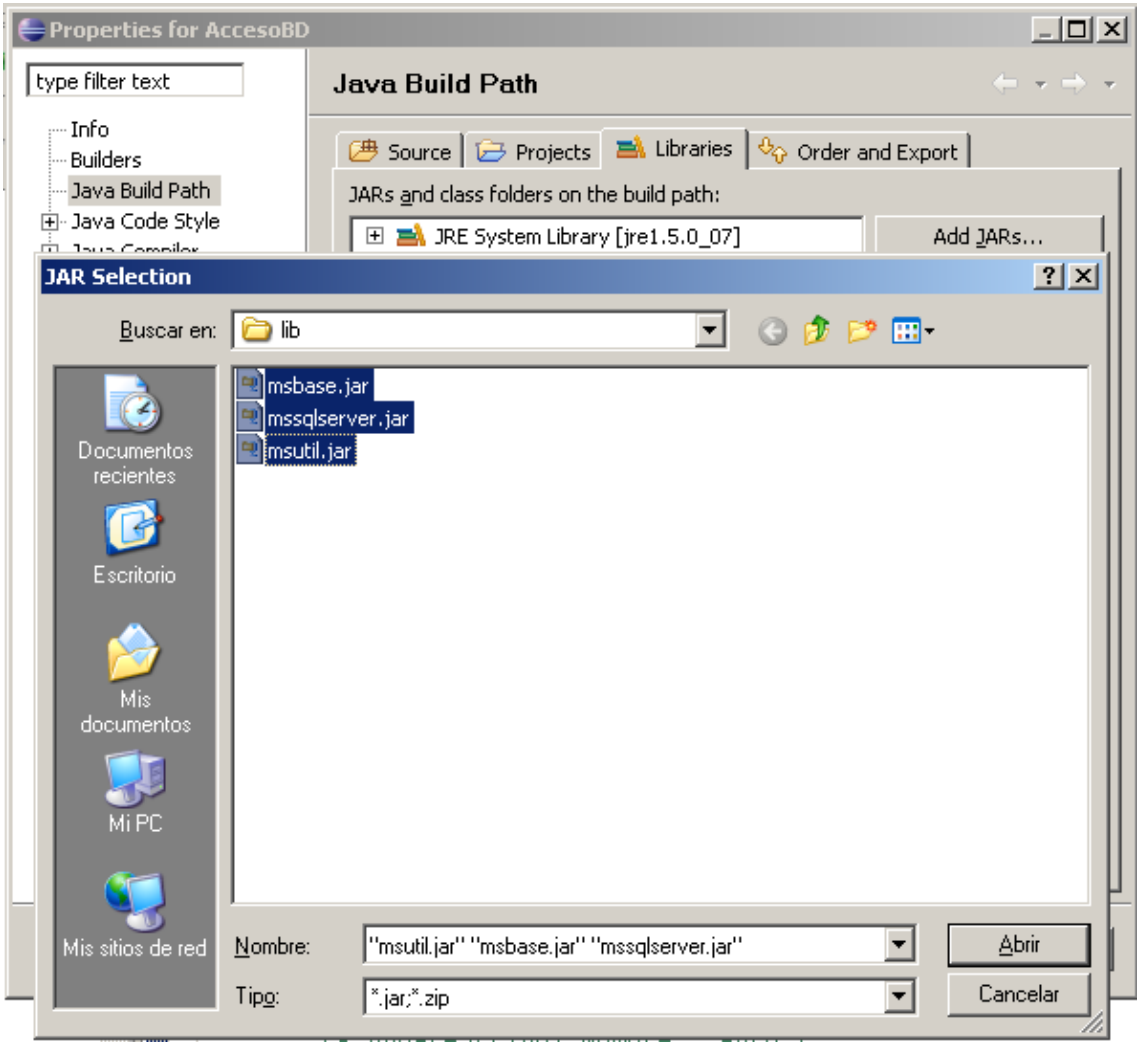
1. Acceder a las propiedades del proyecto, pulsando sobre él con el botón derecho:



2. Seleccionar “Java Build Path”, a la izquierda de la ventana:



3. Pulsar el botón “Add External JARs” a la derecha de la ventana y seleccionar los ficheros jar dondequiera que se encuentren y pulsar el botón “Abrir”:



4. Por último, pulsar el botón “ok” y el proyecto estará listo para ejecutarse.

Un Ejemplo Completo (MySQL):

Previo a la realización de las siguientes clases java, es preciso crear en **MySQL** y posteriormente ayudados por la aplicación **DBArtisan** una base de datos llamada **Colegio**, con una única tabla, llamada **alumnos,** con la siguiente especificación:

Y cuya **clave primaria** es el campo **CODIGO**.

La sentencia SQL que realiza la creación de dicha tabla es:

```
CREATE TABLE ALUMNOS
(
   CODIGO    int         NOT NULL,
   NOMBRE    varchar(100) NOT NULL,
   APELLIDOS varchar(100) NOT NULL,
   EDAD      int         NOT NULL,
   PRIMARY KEY (CODIGO)
)
go
```

BD.java: Ésta clase contiene métodos relacionados con la conexión a la Base de datos, métodos para realizar la carga de los drivers correspondientes y métodos para la ejecución de consultas.

```
import java.sql.*;

public class BD {

    private Connection con;
    private Statement st;

    public BD (){
        try {

        //carga el controlador
                Class contr= Class.forName ("com.mysql.jdbc.Driver");

                System.out.println(contr.toString());
                }
        catch (ClassNotFoundException cnfe) {
                System.out.println("com.mysql.jdbc.Driver");

                }
        try {
                //String dirección. Hace referencia a una URL, un recurso
```

```
de la aplicación
                    Connection micon= DriverManager.getConnection
("jdbc:mysql://localhost/base_de_datos", "usuario", "contraseña");

                    Statement st=micon.createStatement();

                    this.con=micon;
                    this.st=st;
                    }
                catch (SQLException sqle) {
                    System.out.println("Error al establecer la conexion");
                }
        }

        //Ejecuta una consulta de Actualización de la base de datos
        //(inserción, actualización o borrado de datos).
        public void actualiza(String datos){
                try {
                    System.out.println(datos);
                    st.executeUpdate(datos);
                }
                catch (SQLException sqle) {
                    System.out.println("Error al ejecutar la consulta");
                }
        }
        //Ejecuta una consulta de selección, devolviendo el objeto //ResultSet
        con los datos obtenidos en la consulta.
        public ResultSet selecciona(String consulta) throws SQLException {

                    return st.executeQuery(consulta);

        }

        public ResultSet getResultSet() throws SQLException{

                return st.getResultSet();

        }

        public Connection getConex() {
                return con;
        }
}
```

Principal.java: Ésta clase es la encargada de crear un objeto de la clase BD, y manejar la base de datos con los métodos existentes en dicha clase.

```
import java.sql.*;
```

```
public class Principal {

	public static void main(String [] args) throws Exception{

		ResultSet rs=null;
		BD bd= new BD();

		//Inserción de 3 registros a la tabla.
		String sentencia1="INSERT INTO ALUMNOS (CODIGO,
NOMBRE, APELLIDOS, EDAD)" +
			" VALUES (1, 'Alberto', 'Lopez Perez', 12)";
		String sentencia2="INSERT INTO ALUMNOS (CODIGO,
NOMBRE, APELLIDOS, EDAD)" +
		" VALUES (2, 'Aurora', 'Roldan Pino', 13)";
		String sentencia3="INSERT INTO ALUMNOS (CODIGO,
NOMBRE, APELLIDOS, EDAD)" +
		" VALUES (3, 'Rafa', 'Martinez Cobos', 14)";

		System.out.println("Insertando registros en la BD...");
		bd.actualiza(sentencia1);
		bd.actualiza(sentencia2);
		bd.actualiza(sentencia3);

		//Recuperación de los datos.
		String seleccion="SELECT * FROM ALUMNOS";

		try {
			//obtencion de resultados del ResultSet
			System.out.println("Obteniendo registros de la BD...");
			rs=bd.selecciona(seleccion);
			while (rs.next()) {
				int c= rs.getInt("CODIGO");
				String n= rs.getString("NOMBRE");
				String a= rs.getString("APELLIDOS");
				int e= rs.getInt("EDAD");
				System.out.println(c + " "+ n + " "+ a + " "+ e + "\n");
				}
			}
		catch (SQLException sqle) {
			System.out.println("Error en la consulta");
			}
	//Eliminación de un registro de la base de datos.
		System.out.println("Eliminando registro con código 1 de la BD...");
		sentencia1="delete from ALUMNOS where codigo=1";
		bd.actualiza(sentencia1);

		try {
			//obtencion de resultados del ResultSet después del
```

```
borrado
                    System.out.println("Obteniendo registros de la BD después
del borrado...");
                    rs=bd.selecciona(seleccion);
                    while (rs.next()) {
                        int c= rs.getInt("CODIGO");
                        String n= rs.getString("NOMBRE");
                        String a= rs.getString("APELLIDOS");
                        int e= rs.getInt("EDAD");
                        System.out.println(c + " "+ n + " "+ a + " "+ e + "\n");
                        }
                    }
                catch (SQLException sqle) {
                    System.out.println("Error en la consulta");
                    }

                //Actualización de un campo de la base de datos.
                System.out.println("Actualizando registro con código 3 de la
BD...");
                sentencia1="update alumnos set apellidos='Torres López' where
codigo=3";
                bd.actualiza(sentencia1);
                try {
                    //obtencion de resultados del ResultSet después del
borrado
                    System.out.println("Obteniendo registros de la BD después
del borrado...");
                    rs=bd.selecciona(seleccion);
                    while (rs.next()) {
                        int c= rs.getInt("CODIGO");
                        String n= rs.getString("NOMBRE");
                        String a= rs.getString("APELLIDOS");
                        int e= rs.getInt("EDAD");
                        System.out.println(c + " "+ n + " "+ a + " "+ e + "\n");
                        }
                    }
                catch (SQLException sqle) {
                    System.out.println("Error en la consulta");
                    }
                }
        }
```

Instalación de las librerías necesarias:

Para el correcto funcionamiento del ejemplo anterior, es preciso proporcionarle a **NetBeans** el fichero **jar** que contiene el conjunto de clases que My**SQL** proporciona para JDBC.

El fichero es:

- mysql-connector-java-5.0.6-bin.jar

Los pasos a seguir en **NetBeans** para agregar la librería anterior es:

1. Acceder a las propiedades del proyecto, pulsando sobre él con el botón derecho.
2. Seleccionar “Libraries”, a la izquierda de la ventana:

3. Pulsar el botón “Add JAR/FOLDER” a la derecha de la ventana y seleccionar el fichero jar donde quiera que se encuentre y Aceptar.

Cómo conectar Java y Access

La capacidad para acceder a bases de datos desde Java la ofrece la API JDBC (Java DataBase Conectivity). JDBC es un estándar para manejar bases de datos en Java. ODBC es un estándar de Windows para manejar bases de datos, de forma que cualquier programa en Windows que desee acceder a bases de datos genéricas debe usar este estándar. La necesidad de crear un estándar propio para acceder a bases de datos desde Java se explica porque el estándar ODBC está programado en C y un programa que use este estándar, por lo tanto, depende de la plataforma.

Controladores JDBC-ODBC

Necesitamos acceder a un origen de datos ODBC pero contamos con una API que usa el estándar JDBC. Para solventar este problema las empresas realizan drivers que traducen el ODBC a JDBC. Hay varios tipos de Driver, pero para nuestro ejemplo usaremos los llamados puentes JDBC-ODBC. Estamos de suerte porque el JDK de Windows incorpora el driver necesario para conectar bases de datos Access.

Crear un nuevo DSN (Data Source Name)

Para realizar la conexión a una base de datos ODBC necesitaremos crear un perfil DSN desde el panel de control y posteriormente accederemos a la base de datos a partir del nombre del perfil. En el perfil DSN lo que se hace es

indicar el driver a utilizar, así como el archivo o archivos del origen de datos. Estos son los pasos a llevar a cabo para configurar un perfil DSN.

1.- Iremos a Panel de Control. Ello se hace desde Inicio->Configuración o desde MiPC.

2.- Ahora hacemos doble-click en el icono de Fuentes de datos ODBC (32 bits).

3.- En nuestra pantalla aparecerá ahora la pestaña DSN usuario seleccionada. Para crear un nuevo perfil haremos click en Agregar...

4.- A continuación se nos pide que ingresemos el controlador que vamos a usar en el nuevo perfil. En nuestro caso será Microsoft Access Driver (*.mdb).

5.- Una vez aquí sólo nos queda dar un nombre al origen de datos y especificar el archivo .mdb de origen. Tras aceptar la ventana ya tenemos creado un perfil con lo que ya podemos comenzar a programar.

Clases, objetos y métodos básicos

Lo que necesitamos para hacer nuestro programa es la API JDBC incluida en la última versión del JDK. El paquete a utilizar y el cual deberemos importar es el paquete java.sql

Las primeras líneas de código suelen ser rutinarias ya que siempre serán muy similares.

Cargar el Driver

Lo primero es hacer una llamada al Driver JDBC-ODBC para cargarlo. Eso se consigue con las siguientes líneas de código:

```
try{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }
catch(Exception e){ System.out.println("No se ha podido cargar el Driver JDBC-ODBC"); }
```

Con esto ya tenemos cargado el Driver. Ahora básicamente trabajaremos con tres objetos. Estos objetos son: Connection, Statement y ResultSet. El objeto Connection se obtiene al realizar la conexión a la base de datos. El objeto Statement se crea a partir del anterior y nos permite ejecutar SQL para hacer consultas o modificaciones en la base de datos. En caso de hacer una consulta (SELECT ... FROM ...) se nos devolverá un objeto que representa los datos que deseamos consultar; este objeto es un objeto ResultSet (Hoja de resultados).

El objeto Connection

Debemos realizar la conexión a nuestro origen de datos. Para ello debemos crear un objeto Connection de la siguiente forma:

```
Connection con =
DriverManager.getConnection("jdbc:odbc:Nombre_Perfil_DSN",
"Nombre_Usuario", "Contraseña");
```

Los dos últimos parámetros pueden ser cadenas vacías "" a no ser que la base de datos las requiera. Con esto ya hemos realizado una conexión a nuestra base de datos. Pero esto todavía no es suficiente. Ahora vamos a crear un objeto Statement con el que podremos ejecutar y hacer consultas SQL. Si hasta ahora todo ha sido rutinario a partir de ahora vamos a poder crear código más adaptado a las necesidades de nuestro programa.

El objeto Statement

Como se ha dicho, un objeto Statement se crea a partir del objeto Connection antes obtenido. También como se ha dicho un objeto Statement nos permite hacer consultas SQL que nos devuelven una hoja de resultados. Pues bien, según como deseamos que sea esa hoja de resultados (modificable, o no, sensible a las modificaciones o no,...) deberemos crear de una forma u otra el objeto Statement. Más adelante se explican las diferencias entre crearlos de una u otra forma. De momento nos limitaremos ha hacerlo de la forma más sencilla.

```
Statement stat = con.createStatement();
```

De esta forma muy simple hemos creado un objeto Statement. Ahora podemos usar este objeto Statement para hacer modificaciones en la base de datos a través del lenguaje SQL. Para realizar modificaciones, es decir, instrucciones INSERT, UPDATE o DELETE, se usa el método executeUpdate pasando como parámetro una cadena de texto String que contenga la instrucción SQL.

Para hacer una consulta, es decir, una instrucción SELECT debemos usar otro método: el método executeQuery que como el anterior se le ha de pasar un String que contenga la instrucción. Este método nos devuelve un objeto ResultSet que contiene los datos obtenidos. Este objeto será diferente según como hayamos creado el objeto Statement. En el ejemplo anterior hemos usado un método muy simple de Connection para crear el objeto Statement, pero hay otras formas de crearlo. Estas son y estas son las características que aportan a la hoja de resultados de una consulta:

El método anteriormente utilizado es el siguiente: createStatement(), pero existe el mismo método al que le podemos pasar dos parámetros. En la documentación se expresa así: createStatement(int resultSetType, int resultSetConcurrency). Los posibles valores a pasar son campos de la clase ResultSet. Por ejemplo para el parámetro resultSetType podemos elegir que el objeto ResultSet se pueda mover (entre registros) sólo hacia delante (ResultSet.TYPE_FORWARD_ONLY), que se pueda mover pero que cuando se actualice, aunque los cambios ya hayan sido reflejados en la base de datos, el ResultSet no los 'sienta' (ResultSet.TYPE_SCROLL_INSENSITIVE) o que se pueda mover y que los cambios si que se reflejen también en él (ResultSet.TYPE_SCROLL_SENSITIVE). Para el parámetro resultSetConcurrency podemos establecer dos valores diferentes: ResultSet.CONCUR_READ_ONLY si queremos que los datos se puedan leer pero no actualizar, y ResultSet.CONCUR_UPDATABLE si queremos permitir que la base de datos sea actualizable mediante el objeto ResultSet. Si no se usa el método sin parámetros el objeto será TYPE_FORWARD_ONLY y CONCUR_READ_ONLY.

El objeto ResultSet: Hoja de resultados

Moverse por la hoja de resultados

Al hablar de posición del cursor nos referimos a la posición dentro de los datos del objeto ResultSet. Lo primero que hay que saber es que el cursor tiene tantas posiciones como filas tenga la consulta y dos más que se sitúan antes de la primera fila y después de la última. Nada más crear un objeto ResultSet, la posición del cursor es la anterior a la primera fila.

Para mover el cursor a la posición siguiente usaremos el método next() que nos devuelve una variable booleana: sí, si se ha podido mover; no, si no ha sido posible el desplazamiento. Nos devolverá false si estamos en el útimo registro (el posterior al último). Para movernos hacia atrás tenemos un método muy similar: el método previous() que al igual que el anterior nos devuelve un valor booleano que será false si estamos en el registro anterior al primero. Estos métodos nos permiten movernos de forma relativa por la hoja de resultados; si deseamos movernos a un registro determinado usaremos el método absolute(int numero_fila), que también nos devuelve un valor boolean. Además tenemos otros métodos que son: afterLast(), que mueve el cursor a la fila posterior a la última; beforeFirst() que mueve el cursor a la fila anterior a la primera; last() que mueve el cursor a la última fila; first() que mueve el cursor a la primera fila. Todos ellos devuelven un valor booleano.

Obtener datos de la hoja de resultados

Para acceder a los datos de la hoja de resultados usamos los métodos get...(int numeroColumna) o get...(String nombreColumna). Estos métodos nos devuelven el valor que indica el nombre del método (por ejemplo tenemos: getString, getInt, getDate, ...) indicando el número o el nombre de la columna. Hay que tener en cuenta que el número de columna es el número de columna en la hoja de resultados y por tanto se establece con el orden en el que se han incluido las columnas en la instrucción SELECT. Por ejemplo si hemos hecho la consulta de la siguiente forma: SELECT Nombre, Apellidos ... la columna Nombre será la primera y la columna Apellidos será la segunda

independientemente de cómo estén situadas en la base de datos. Si hacemos un SELECT * FROM ..., en ese caso las columnas estarán en el orden en el que están en la base de datos.

Modificar la base de datos con el objeto ResultSet

Hemos dicho que mediante el objeto Statement podemos ejecutar sentencias SQL para modificar la base de datos y hacer consultas, por lo que podemos usar el lenguaje SQL para hacer cualquier operación en la base de datos; pero además tenemos métodos especializados que nos permiten hacer estas mismas tareas sin usar el lenguaje SQL. Para ello se usan los métodos especializados del objeto ResultSet. Por tanto si deseamos modificar la base de datos a través de Java, sin utilizar SQL, lo primero que deberemos hacer es realizar una consulta SQL para obtener el objeto ResultSet (parece un poco paradógico). Ahora sólo nos queda usar los métodos especializados de este objeto.

Por ejemplo podemos usar deleteRow() para borrar una fila.

Para actualizar una fila se usan los métodos update...(NombreColumna, valor) o update...(NumeroColumna, valor), así podemos encontrar updateFloat, updateString, ... Para que los cambios se reflejen en la base de datos se debe llamar al método updateRow() o cancelRowUpdates() si queremos cancelar la actualización.

```
resultset.updateString("Nombre", "PEPE");
resultset.updateRow();
```

Para insertar una fila nos hemos de mover a un lugar del cursor especial con el método toInsertRow(), una vez allí usaremos los métodos update... explicados antes y para reflejar esta actualización llamar al método insertRow()

Ejemplo:

```
package Conexiones;

import java.sql.Connection;
```

```
import java.sql.DriverManager;
import java.sql.SQLException;

/**
 *
 * @author Coby
 */
public class DBConnection {

    Connection con = null;

    public DBConnection() {

        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con=DriverManager.getConnection("jdbc:odbc:Agenda","","");
            if(con!=null)
                System.out.println("Conexion completa");

        }catch(SQLException e)
        {
            System.out.println("Error al abrir la conexion");
        }
        catch(ClassNotFoundException e)
        {
            System.out.println("Error al tratar de cargar el controlador");
        }

    }

    public Connection getConexion()
    {
        return con;
    }



}
```

```
package Conexiones;

import java.sql.*;

/**
```

```
 *
 * @author Coby
 */
public class TrabajoBaseDatos {

    DBConnection objConexion;

    public TrabajoBaseDatos() {

        objConexion = new DBConnection();
    }

    public boolean Grabar(String nombre, String telefono)
    {
        try {
            Statement conector = objConexion.getConexion().createStatement();
            conector.executeUpdate("INSERT INTO Contactos(nombre,telefono)
Values('" + nombre + "','" + telefono + "')");
        } catch (SQLException ex) {
            System.out.println("Error al escribir en la base de datos: " +
ex.getMessage());
        }

        return true;
    }

    public Object[][] ObtenerDatos(String sql,String sql2)
    {
        int numeroRegistros=0;
        try {
            PreparedStatement Consulta =
objConexion.getConexion().prepareStatement(sql);
            ResultSet res = Consulta.executeQuery();
            res.next();
            numeroRegistros = res.getInt("cont");
            res.close();
        } catch (Exception e) {
        }
        Object[][] datos = new Object[numeroRegistros][3];
        try {
            PreparedStatement Consulta =
objConexion.getConexion().prepareStatement(sql2);
            ResultSet res = Consulta.executeQuery();
            int i=0;
            while(res.next())
            {
                datos[i][0]=res.getInt("id");
                datos[i][1]=res.getString("nombre");
                datos[i][2]= res.getString("Telefono");
```

```
                i++;
            }

            res.close();
        } catch (Exception e) {
        }
        return datos;
    }


}
```