TEMA XII GESTIÓN DE BASES DE DATOS RELACIONALES

"Es entretenido resolver misterios,
pero no deberías necesitar hacerlo con el código.
Simplemente deberías poder leerlo"

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.

Un SGBD (Sistema Gestor de Base de Datos) es el proceso responsable de manejar, almacenar y recuperar los datos de una bbdd.

Controladores JDBC-ODBC

Necesitamos acceder a un origen de datos y 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

- Puente JDBC – ODBC: controlador JDBC que permite la comunicación entre ambas especificaciones. No es una solución estrictamente Java.
- Native API/Part Java: controladores propios de las bases de datos, convierten las llamadas del JDBC en llamadas de la bbdd que ese está usando. Son específicos ,generalmente, del s.o. y no so estrictamente Java
- JDBC/Pure java: basado totalmente en Java, convierten las llamadas del JDBC en un protocolo independiente que puede utilizarse como interfaz para una base de datos. Su principal inconveniente es la seguridad
- Native-Protocol Pure Java: lo proporciona el fabricante de la bbdd y convierte llamadas del JDBC en un protocolo de red admitido por la bbdd.

Las dos primeras soluciones son temporales para cuando no existe un controlador JDBC para una bbdd determinada. Las otras dos son las preferidas por los desarrolladores, ya que son independientes de la plataforma utilizada.

Para nuestro ejemplo usaremos los llamados puentes JDBC-ODBC. El JDK de Windows incorpora el driver necesario para conectar bases de datos Access, no así para conectarse a mysql, por lo que debemos descargar dicho driver

Establecer una Conexión

Lo primero que tenemos que hacer es establecer una conexión con el controlador de base de datos que queremos utilizar. Esto implica dos pasos:

1. Cargar el driver
2. Hacer la conexión.

Cargar el Driver

Cargar el driver o drivers que queremos utilizar es muy sencillo y sólo implica una línea de código. Si, por ejemplo, queremos utilizar el puente JDBC-ODBC(base de datos access), se cargaría la siguiente línea de código.

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

Si queremos conectarnos a una base de datos mysql

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

Esta instrucción puede provocar una excepción que hay que controlar

```
try{
        Class.forName("com.mysql.jdbc.Driver ");
}
catch( Exception e){
        System.out.println("No se ha podido cargar el Driver"); }
```

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("url", "Nombre_Usuario", "Contraseña");

En la url indicamos la fuente de nuestros datos y su ruta. Los dos últimos parámetros pueden ser cadenas vacías "" a no ser que la base de datos las requiera.

La url depende del gestor de bbdd que usemos:

Por ejemplo si usamos access, empezará con **jdbc:odbc y** el resto de la URL será el DSN asociado a nuestra bbdd

"jdbc:odbc:Nombre_Perfil_DSN""

Si usamos mysql:

"jdbc:mysql://localhost/"+bd;

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, .accbd de origen. Tras aceptar la ventana ya tenemos creado un perfil con lo que ya podemos comenzar a programar.

Si uno de los drivers que hemos cargado reconoce la URL suministrada por el método **DriverManager.getConnection**, dicho driver establecerá una conexión con el controlador de base de datos especificado en la URL del JDBC. La clase **DriverManager**, como su nombre indica, maneja todos los detalles del establecimiento de la conexión detrás de la escena. A menos que estemos escribiendo un driver, posiblemente nunca utilizaremos ningún método del interface **Driver**, y el único método de **DriverManager** que realmente necesitaremos conocer es **DriverManager.getConnection**.
La conexión devuelta por el método **DriverManager.getConnection** es una conexión abierta que se puede utilizar para crear sentencias JDBC que pasen nuestras sentencias SQL al controlador de la base de datos. En el ejemplo anterior, **con** es una conexión abierta, y se utilizará en los ejemplos posteriores.

Crear las Sentencias JDBC

Un objeto Statement(interfaz**)** es el que envía nuestras sentencias SQL al controlador de la base de datos. Simplemente creamos un objeto Statement y lo ejecutamos, suministrando el método SQL apropiado con la sentencia SQL que queremos enviar.

- Para una sentencia SELECT, el método a ejecutar es **executeQuery**. Retorna el conjunto de registros consultados
- Para sentencias que crean o modifican tablas, el método a utilizar es **executeUpdate**. Retorna un int con el número de registros modificados
- Para sentencias que no sabemos de qué tipo son se utiliza execute, retorna un boolean que nos indica si la sentencia devuelve un conjunto de registros (true). Si es así se recuperan estos con un método de otro objeto que veremos más adelante.

Un objeto Statement siempre va **a**sociado a un objeto Connection. En el siguiente ejemplo, utilizamos nuestro objeto Connection: con para crear el objeto Statement: stmt.

Statement stmt = con.createStatement();

En este momento **stmt** existe, pero no tiene ninguna sentencia SQL que pasarle al controlador de la base de datos. Necesitamos suministrarle el mètodo que utilizaremos para ejecutar **stmt**. Por ejemplo, en el siguiente fragmento de código, suministramos **executeUpdate** con la sentencia SQL del ejemplo anterior.

String cadena =" INSERT INTO Datos VALUES ('c','c')";
Stmt.executeUpdate(cadena);

Stmt.executeUpdate("INSERT INTO Datos VALUES ('c','c'));

Para ejecutar consultas

String cadena="SELECT nombre,apellidos FROM Datos WHERE edad>18";
Stmt.executeQuery(cadena);

Para actualizar datos en una tabla usamos por tanto las sentencias SQL : INSERT, DELETE, UPDATE (update from tabla set campo='valor' where condicion) y ejecutamos método executeUpdate.

Para consultar datos en una tabla usamos la sentencia SQL: SELECT y ejecutamos el método executeQuery, en este caso tenemos que guardar el conjunto de registros obtenidos en un objeto de tipo Resulset, veamos como obtener el valor de los campos de dichos registros

Recuperar Valores desde una Hoja de Resultados

JDBC devuelve los resultados en un objeto **ResultSet**, estos resultados se muestran en forma de filas de una tabla.

ResultSet rs = stmt.executeQuery(SELECT nombre,apellidos FROM Datos WHERE edad>18");

Para acceder a los datos, iremos a una fila y recuperaremos los valores de acuerdo con sus tipos.

- El método **next** mueve el cursor a la siguiente fila y hace que esa fila (llamada fila actual) sea con la que podamos operar. Como el cursor inicialmente se posiciona justo encima de la primera fila de un objeto **ResultSet**, primero debemos llamar al método **next** para mover el cursor a la primera fila y convertirla en la fila actual. Sucesivas invocaciones del método **next** moverán el cursor de línea en línea de arriba a abajo.

  Se puede mover el cursor hacia atrás, hacia posiciones específicas y a posiciones relativas a la fila actual además de mover el cursor hacia delante, pero eso supone distinguir entre varios tipos de Resulset que veremos mas adelante

- Los métodos **getXXX** del tipo apropiado se utilizan para recuperar el valor de cada columna.

  getString, getInt ,....

Estos métodos reciben como parámetro el número de la columna que deseamos obtener (empezando en 1) (nº de columna en el objeto Resulset , no el la tabla original) o el nombre del campo de la tabla

```
String nombre,apellidos;
While ( rs.next() )
{
nombre=rs.getStrint(1);
apellidos=rs.getStrint(2);
System.out.println( nombre + “  “ + apellidos);
}
```

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 la clase 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. Veremos un ejemplo de esto en las página siguientes.

Crear un Objeto PreparedStatement

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

```
PreparedStatement updateSales = con.prepareStatement(
                    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
```

La variable updateSales contiene la sentencia SQL, "UPDATE COFFEES SET SALES = ? WHERE COF_NAME 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, tuviéramos 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. 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 resetear 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).

Mover el Cursor por una Hoja de Resultados

Una de las nuevas características del API JDBC 2.0 es la habilidad de mover el cursor en una hoja de resultados tanto hacia atrás como hacia adelante. También hay métodos que nos permiten mover el cursor a una fila particular y comprobar la posición del cursor. Para ello debemos indicar que tipo de ResulSet utilizaremos:

- TYPE_FORWARD_ONLY: se crea una hoja de resultados no desplazable, es decir, una hoja en la que el cursor sólo se mueve hacia adelante
- TYPE_SCROLL_INSENSITIVE :es desplazable pero no refleja los cambios hechos mientras estaba abierta
- TYPE_SCROLL_SENSITIVE: es desplazable y refleja los cambios hechos mientras está abierta.

Cualquiera de los tres tipos refleja los cambios cuando se cierra el ResulSet

Hemos clasificado un ResulSet atendiendo a su movilidad, pero hay otro criterio para clasificarlos:

- CONCUR_READ_ONLY: es solo de lectura
- CONCUR_UPDATABLE : es actualizable

Para indicar estos tipos de ResulSet se le pasan dos parámetros al método createStatement

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT .....");
```

Si especificamos un tipo, también debemos especificar si es de sólo lectura o actualizable. Si no se especifican constantes para el tipo y actualización de un objeto ResultSet, obtendremos automáticamente una TYPE_FORWARD_ONLY y CONCUR_READ_ONLY

Deberíamos tener en mente el hecho de que no importa el tipo de hoja de resultados que especifiquemos, siempre estaremos limitados por nuestro controlador de base de datos y el driver utilizados.

El cursor también se posiciona inicialmente delante de la primera fila. El método next es apropiado si queremos acceder a las filas una a una, yendo de la primera fila a la última, pero ahora tenemos muchas más formas para mover el cursor.

- método **previous:** que mueve el cursor una fila hacia atrás (hacia el inicio de la hoja de resultados). Devuelve false cuando el cursor se sale de la hoja de resultados (posición antes de la primera o después de la última fila)
- Los métodos **first**, **last** mueven el cursor a la fila indicada en sus nombres.
- Los métodos **aflerLast y beforeFirst** mueven el cursor al principio y final (no primera y última fila)
- El método **absolute** moverá el cursor al número de fila indicado en su argumento. Si el número es positivo, el cursor se mueve al número dado desde el principio, por eso llamar a **absolute(1)** pone el cursor en la primera fila. Si el número es negativo, mueve el cursor al número dado desde el final, por eso llamar a **absolute(-1)** pone el cursor en la última fila.

```
srs.absolute(4); //cuarta fila
Si srs tuviera 500 filas, la siguiente línea de código movería el cursor a la fila 497.
srs.absolute(-4);
```

- El método **relative**, especifica cuántas filas se moverá desde la fila actual y también la dirección en la que se moverá. Un número positivo mueve el cursor hacia adelante el número de filas dado; un número negativo mueve el cursor hacia atrás el número de filas dado. Por ejemplo, en el siguiente fragmente de código, el cursor se mueve a la cuarta fila, luego a la primera y por último a la tercera.

```
srs.absolute(4); // cursor está en la cuarta fila
. . .
srs.relative(-3); // cursor está en la primera fila
. . .
srs.relative(2); // cursor está en la tercera fila
```

- El método **getRow** permite comprobar el número de fila donde está el cursor. Por ejemplo, se puede utilizar **getRow** para verificar la posición actual del cursor en el ejemplo anterior.

```
srs.absolute(4);
int rowNum = srs.getRow(); // rowNum debería ser 4
srs.relative(-3);
int rowNum = srs.getRow(); // rowNum debería ser 1
srs.relative(2);
int rowNum = srs.getRow(); // rowNum debería ser 3
```

- Existen cuatro métodos adicionales que permiten verificar si el cursor se encuentra en una posición particular. La posición se indica en sus nombres : **isFirst, isLast, isBeforeFirst, isAfterLast.** Todos estos métodos devuelven un boolean y por lo tanto pueden ser utilizados en una sentencia condicional.

```
if (srs.isAfterLast() == false)
   srs.afterLast();
while (srs.previous()) {
   String name = srs.getString("COF_NAME");
   float price = srs.getFloat("PRICE");
   System.out.println(name + "     " + price);}
```

Hacer Actualizaciones en una Hoja de Resultados

Otra nueva característica del API JDBC 2.0 es la habilidad de actualizar filas en una hoja de resultados utilizando métodos Java en vez de tener que enviar comandos SQL.

Modificar el valor de un campo

Una actualización es la modificación del valor de una columna de la fila actual. Para ello se dispone de métodos updateXXX

Supongamos que queremos aumentar el precio del café "French Roast Decaf" a 10.99.

```
uprs.last();
uprs.updateFloat("PRICE", 10.99);
```

Las operaciones de actualización en el API JDBC 2.0 afectan a los valores de columna de la fila en la que se encuentra el cursor. Una vez situado el cursor, todos los métodos de actualización que llamemos operarán sobre esa fila hasta que movamos el cursor a otra fila.

Los métodos updateXXX de ResultSet toman dos parámetros: la columna a actualizar y el nuevo valor a colocar en ella. Al igual que en los métodos getXXX de ResultSet., el parámetro que designa la columna podría ser el nombre de la columna o el número de la columna. Existe un método updateXXX diferente para cada tipo (updateString, updateInt, etc.)

Para que la actualización tenga efecto en la base de datos y no sólo en la hoja de resultados, debemos llamar al método updateRow de ResultSet.

```
uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.updateRow();
```

Si hubiéramos movido el cursor a una fila diferente antes de llamar al método updateRow, la actualización se habría perdido. Si, por el contrario, nos damos cuenta de que el precio debería haber sido 10.79 en vez de 10.99 podríamos haber cancelado la actualización llamando al método cancelRowUpdates. Tenemos que llamar al método cancelRowUpdates antes de llamar al método updateRow; una vez que se llama a updateRow, llamar a cancelRowUpdates no hará nada. Observa que cancelRowUpdates cancela todas las actualizaciones en una fila, por eso, si había muchas llamadas a método updateXXX en la misma fila, no podemos cancelar sólo una de ellas.

```
uprs.last();
uprs.updateFloat("PRICE", 10.99);
uprs.cancelRowUpdates();
uprs.updateFloat("PRICE", 10.79);
```

```
uprs.updateRow();
```

En este ejemplo, sólo se había actualizado una columna, pero podemos llamar a un método updateXXX apropiado para cada una de las columnas de la fila. El concepto a recordar es que las actualizaciones y las operaciones relacionadas se aplican sobre la fila en la que se encuentra el cursor. Incluso si hay muchas llamadas a métodos updateXXX, sólo se hace una llamada al método updateRow para actualizar la base de datos con todos los cambios realizados en la fila actual.

Todos los movimientos de cursor se refieren a filas del objeto ResultSet, no a filas de la tabla de la base de datos. Si una petición selecciona cinco filas de la tabla de la base de datos, habrá cinco filas en la hoja de resultados, con la primera fila siendo la fila 1, la segunda siendo la fila 2, etc. La fila 1 puede ser identificada como la primera, y, en una hoja de resultados con cinco filas, la fila 5 será la última.

Insertar Filas

El primer paso será mover el cursor a la fila de inserción, lo que podemos hacer llamando al método moveToInsertRow. El siguiente paso será seleccionar un valor para cada columna de la fila. Hacemos esto llamando a los métodos updateXXX apropiados para cada valor. Finalmente, podemos llamar al método insertRow para insertar la fila que hemos rellenado en la hoja de resultados. Este único método inserta la fila simultáneamente tanto en el objeto ResultSet como en la tabla de la base de datos de la que la hoja de datos fue seleccionada.

```
uprs.moveToInsertRow();
uprs.updateString("COF_NAME", "Kona");
uprs.updateInt("SUP_ID", 150);
uprs.updateFloat("PRICE", 10.99);
uprs.updateInt("SALES", 0);
uprs.updateInt("TOTAL", 0);
uprs.insertRow();
```

Como podemos utilizar el nombre o el número de la columna para indicar la columna seleccionada, nuestro código para seleccionar los valores de columna se podría parecer a esto.

```
uprs.updateString(1, "Kona");
uprs.updateInt(2, 150);
uprs.updateFloat(3, 10.99);
uprs.updateInt(4, 0);
uprs.updateInt(5, 0);
```

Podríamos habernos preguntado por qué los métodos updateXXX parecen tener un comportamiento distinto a como lo hacían en los ejemplos de actualización. En aquellos ejemplos, el valor seleccionado con un método updateXXX reemplazaba inmediatamente el valor de la columna en la hoja de resultados. Esto era porque el cursor estaba sobre una fila de la hoja de resultados. Cuando el cursor está sobre la fila de inserción, el valor seleccionado con un método updateXXX también es automáticamente seleccionado, pero lo es en la fila de

inserción y no en la propia hoja de resultados. Tanto en actualizaciones como en inserciones, llamar a los métodos updateXXX no afectan a la tabla de la base de datos. Se debe llamar al método updateRow para hacer que las actualizaciones ocurran en la base de datos. Para el inserciones, el método insertRow inserta la nueva fila en la hoja de resultados y en la base de datos al mismo tiempo.

Podríamos preguntarnos que sucedería si insertáramos una fila pero sin suministrar los valores para cada columna. Si no suministramos valores para una columna que estaba definida para aceptar valores NULL de SQL, el valor asignado a esa columna es NULL. Si la columna no acepta valores null, obtendremos una SQLException. Esto también es cierto si falta una columna de la tabla en nuestro objeto ResultSet.

Cuando llamamos al método moveToInsertRow, la hoja de resultados graba la fila en la que se encontraba el cursor, que por definición es la fila actual. Como consecuencia, el método moveToCurrentRow puede mover el cursor desde la fila de inserción a la fila en la que se encontraba anteriormente.

Borrar Filas

Borrar una fila es la tercera forma de modificar un objeto ResultSet, y es la más simple. Todo lo que tenemos que hacer es mover el cursor a la fila que queremos borrar y luego llamar al método deleteRow.

```
uprs.absolute(4);
uprs.deleteRow();
```

La cuarta fila ha sido eliminada de uprs y de la base de datos.

El único problema con las eliminaciones es lo que ResultSet realmente hace cuando se borra una fila. Con algunos driver JDBC, una línea borrada es eliminada y ya no es visible en una hoja de resultados. Algunos drives JDBC utilizan una fila en blanco en su lugar pone (un "hole") donde la fila borrada fuera utilizada. Si existe una fila en blanco en lugar de la fila borrada, se puede utilizar el método absolute con la posición original de la fila para mover el cursor, porque el número de filas en la hoja de resultados no ha cambiado.

En cualquier caso, deberíamos recordar que los drivers JDBC manejan las eliminaciones de forma diferente. Por ejemplo, si escribimos una aplicación para ejecutarse con diferentes bases de datos, no deberíamos escribir código que dependiera de si hay una fila vacía en la hoja de resultados.

TRANSACCIONES

Hay veces que no queremos que una sentencia tenga efecto a menos que otra también suceda. Por ejemplo, cuando tenemos dos cuentas bancarias y queremos sacar dinero de una para ingresarlo en otra. Si no se realizan las dos operaciones, la bbdd podría quedar inconsistente. Una transacción es un bloque de instrucciones que deben ejecutarse como una unidad (o todas o ninguna)

Cuando se crea una conexión, está en modo auto-entrega. Esto significa que cada sentencia SQL individual es tratada como una transacción y será automáticamente entregada justo después de ser ejecutada. La forma de permitir que dos o más sentencias sean agrupadas en una transacción es desactivar el modo auto-entrega.

```
con.setAutoCommit(false);
```

Una vez que se ha desactivado la auto-entrega, no se entregará ninguna sentencia SQL hasta que llamemos explícitamente al método commit. Todas las sentencias ejecutadas después de la anterior llamada al método commit serán incluidas en la transacción actual y serán entregadas juntas como una unidad.

```
String acta= “UPDATE Cuentas SET saldo=saldo-” + cantidad + “WHERE numerocta=” + numctA;

String actb= “UPDATE Cuentas SET saldo=saldo+” + cantidad + “WHERE numerocta=” + numctB;

con.setAutoCommit(false);
st=con.createStatement();
st.executeUpdate(acta);
st.executeUpdate(actb);
con.commit();
con.setAutoCommit(true);
```

En este ejemplo, el modo auto-entrega se desactiva para la conexión con, lo que significa que las dos sentencias serán entregadas juntas cuando se llame al método commit. Siempre que se llame al método **commit** todos los cambios resultantes de las sentencias de la transacción serán permanentes.
Es bueno desactivar el modo auto-commit sólo mientras queramos estar en modo transacción. De esta forma, evitamos bloquear la base de datos durante varias sentencias, lo que incrementa los conflictos con otros usuarios.

Además de agrupar las sentencias para ejecutarlas como una unidad, las transacciones pueden ayudarnos a preservar la integridad de los datos de una tabla. Por ejemplo, supongamos que un empleado se ha propuesto introducir los nuevos precios de unos productos, pero lo retrasa unos días. Mientras tanto, los precios han subido, y hoy el propietario está introduciendo los nuevos precios. Finalmente el empleado empieza a introducir los precios ahora desfasados al mismo tiempo que el propietario intenta actualizar la tabla. Después de insertar los precios

desfasados, el empleado se da cuenta de que ya no son válidos y aborta la transacción (método **rollback**, aborta transacción y restaura los valores que había antes de intentar la actualización) para dejar los precios como estaban. El propietario podría perder sus cambios y obtener los precios viejos.

Esta clase de situaciones puede evitarse utilizando transacciones. Si un controlador de base de datos soporta transacciones, y casi todos lo hacen, proporcionará algún nivel de protección contra conflictos que pueden surgir cuando dos usuarios acceden a los datos a la misma vez.
Para evitar conflictos durante una transacción un controlador de base de datos utiliza bloqueos, mecanismos para bloquear el acceso de otros a los datos que están siendo accedidos por una transacción. Por ejemplo, un controlador de base de datos podría bloquear una fila de una tabla hasta que la actualización se haya entregado. El efecto de este bloqueo es evitar que un usuario obtenga una lectura sucia, esto es, que lea un valor antes de que sea permanente. Acceder a un valor actualizado que no haya sido entregado se considera una lectura sucia porque es posible que el valor sea devuelto a su valor anterior. Si leemos un valor que luego es devuelto a su valor antiguo, habremos leído un valor nulo.
La forma en que se configuran los bloqueos está determinado por lo que se llama nivel de aislamiento de transacción, que pude variar desde no soportar transacciones en absoluto a soportar todas las transacciones que fuerzan una reglas de acceso muy estrictas.

Un ejemplo de nivel de aislamiento de transacción es TRANSACTION-READ_COMMITTED, que no permite que se acceda a un valor hasta que haya sido entregado. En otras palabras, si nivel de aislamiento de transacción se selecciona a TRANSACTION_READ_COMMITTED, el controlador de la base de datos no permitirá que ocurran lecturas sucias. El interface Connection incluye cinco valores que representan los niveles de aislamiento de transacción que se pueden utilizar en JDBC. Normalmente, no se necesita cambiar el nivel de aislamiento de transacción; podemos utilizar el valor por defecto de nuestro controlador. JDBC permite averiguar el nivel de aislamiento de transacción de nuestro controlador de la base de datos (utilizando el método getTransactionIsolation de Connection) y permite configurarlo a otro nivel (utilizando el método setTransactionIsolation de Connection). Sin embargo, ten en cuenta, que aunque JDBC permite seleccionar un nivel de aislamiento, hacer esto no tendrá ningún efecto a no ser que el driver del controlador de la base de datos lo soporte.

Como se mencionó anteriormente, llamar al método rollback aborta la transacción y devuelve cualquier valor que fuera modificado a sus valores anteriores. Si estamos intentando ejecutar una o más sentencias en una transacción y obtenemos una SQLException, deberíamos llamar al método rollback para abortar la transacción y empezarla de nuevo. Esta es la única forma para asegurarnos de cuál ha sido entregada y cuál no ha sido entregada. Capturar una SQLException nos dice que hay algo erróneo, pero no nos dice si fue o no fue entregada. Como no podemos contar con el hecho de que nada fue entregado, llamar al método rollback es la única forma de asegurarnos.

```
String acta= “UPDATE Cuentas SET saldo=saldo-” + cantidad + “WHERE numerocta=” + numctA;

String actb= “UPDATE Cuentas SET saldo=saldo+” + cantidad + “WHERE numerocta=” + numctB;
```

```
st=con.setAutoCommit(false);
st=con.createStatement();
st.executeUpdate(acta);
st.executeUpdate(actb);
try{
con.commit();
}
Catch ( SQLException e{
   con.rollback();
)
st.close();
con.setAutoCommit(true);
```