1º CFGS Desarrollo De Aplicaciones Multiplataforma

UD 10. Gestión de BD relacionales

Módulo: Programación

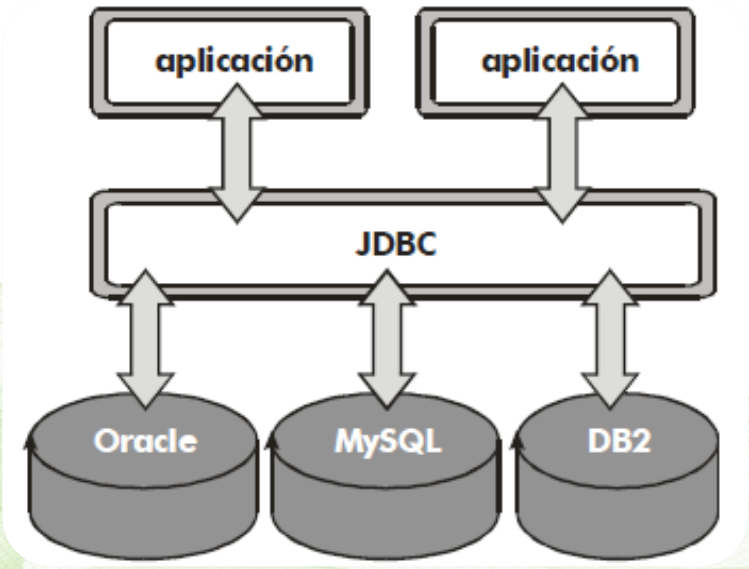
Centro de Enseñanza
Gregorio Fernández

Macarena Cuenca Carbajo

Introducción

- Java dispone del API **JDBC** (*Java DataBase Connectivity*), incluido en Java SE (Standard Edition), compuesta por un conjunto de clases que unifican el acceso a las bases de datos.
- Gracias a la utilización de JDBC, un programa Java puede acceder a cualquier base de datos sin necesidad de modificar la aplicación.
- Sin embargo, para que esto sea posible es necesario que el fabricante de bases de datos ofrezca un **driver** que cumpla la especificación JDBC.
- Un driver JDBC es una capa de software intermedia que traduce las llamadas JDBC a los APIs específicos del vendedor.

Estructura JDBC



En el diagrama se puede apreciar como la idea es que las aplicaciones sólo se tengan que comunicar con la interfaz **JDBC**.

Ésta es la encargada de comunicarse con los sistemas gestores de base de datos.

Drivers

- Entonces, para trabajar con JDBC, es necesario tener controladores (drivers) que permitan acceder a las distintas bases de datos, y así poder trabajar con ellas desde nuestros programas Java.
- Un driver JDBC, no es más que un fichero **JAR**.
- Los driver JDBC se clasifican en cuatro tipos o niveles:
  1. **Tipo 1**. **Puente JDBC-ODBC**. Traduce operaciones JDBC en llamadas a la API ODBC. No son muy productivos.
  2. **Tipo 2**. Son controladores parcialmente escritos en Java y parcialmente escritos en el código nativo que comunica con el API de la base de datos. Por tanto son específicos para una plataforma.

Drivers

3. **Tipo 3**. Son paquetes puros de Java con los que las instrucciones JDBC son pasadas a un servidor genérico de base de datos que utiliza un protocolo determinado. No requiere tener instalado en el cliente ninguna API de la base de datos.

   Este tipo de drivers, son los que mejor funcionan en redes basadas en Internet o intranet y aplicaciones con un gran número de operaciones concurrentes como consultas, búsquedas, etc.

4. **Tipo 4**. Se comunica directamente con el servidor de bases de datos utilizando el protocolo nativo del servidor.

   No requieren intermediarios entre el software JDBC y la base de datos.

¿Qué driver elijo?

- Elegir el driver JDBC correcto es importante porque tiene un impacto directo en el **rendimiento** de la aplicación.
  - El **puente JDBC-ODBC** se podría considerar únicamente como una **solución transitoria** ya que no soporta todas las características de Java, y el usuario está limitado por la funcionalidad del driver ODBC elegido.
  - En las **aplicaciones de intranet** es útil considerar los driver de **tipo 2**, pero estos drivers, como el puente, necesitan que el API nativo se instale en cada cliente. Por lo tanto, tienen los mismos problemas de mantenimiento que el driver puente. Sin embargo, los drivers de tipo 2 son más rápidos que los de tipo 1 porque se elimina la capa extra de traducción.
  - Para las aplicaciones relacionadas con **Internet**, no hay otra opción que utilizar drivers del **tipo 3** o del **tipo 4**. Los drivers de tipo 3 son los que mejor funcionan con los entornos que necesitan proporcionar conexión a gran cantidad de servidores de bases de datos y a bases de datos heterogéneas.
  - Los drivers de **tipo 4** están generalmente recomendados para aplicaciones que requieren un acceso rápido y eficiente a bases de datos. Como estos drivers traducen llamadas JDBC directamente a protocolo nativo sin utilizar ODBC o el API nativo, pueden aportar acceso a bases de datos de alto rendimiento.
  - Para adquirir los drivers es necesario ponerse en contacto con el fabricante o dirigirse a su página Web y después descargarlo. Las instrucciones de instalación las da el fabricante.
- Lo más ágil es dejar que nuestro proyecto Java use el **repositorio Maven** para gestionar la descarga del driver JDBC y sus dependencias.

Uso de JDBC

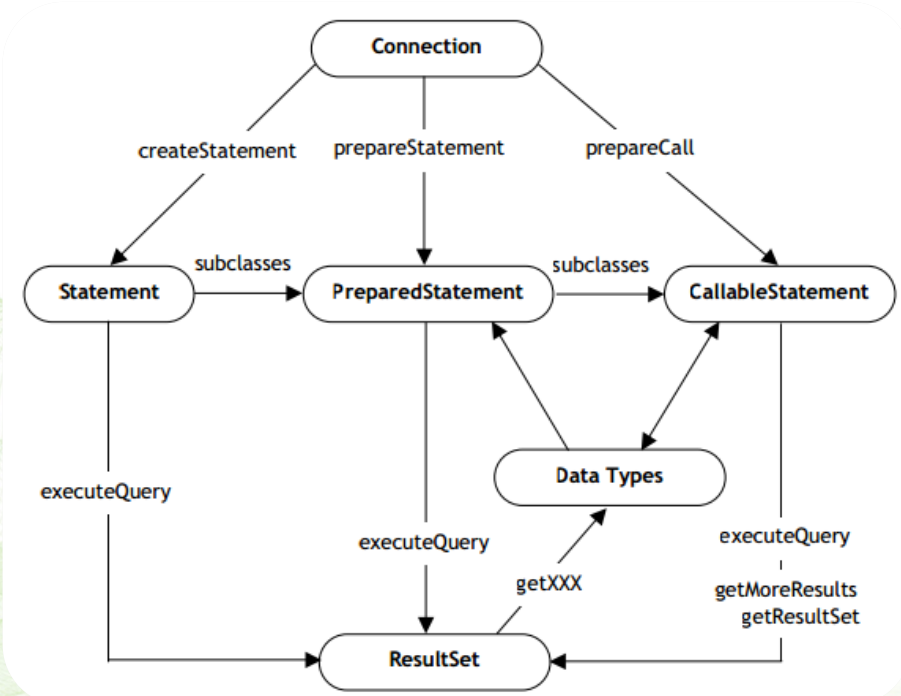
Pasos para que una aplicación Java se comunique con una base de datos:

1. ~~Registrar el driver necesario~~.
2. Conectarse a la base de datos.
3. Enviar consultas SQL, ejecutar la petición y procesar los resultados.
4. Liberar los recursos.

Las clases necesarias para usar JDBC están en los paquetes **java.sql** y **javax.sql**.

En la siguiente imagen se pueden ver las clases básicas para el trabajo con JDBC de forma nativa.

Uso de JDBC



Macarena Cuenca Carbajo

Centro de Enseñanza
Gregorio Fernández

Uso de JDBC
I – Cargar el driver

Class.forName(“nombre del driver”);

- En sistemas antiguos, para que DriverManager tuviera "registrados" los drivers, era necesario cargar la clase en la máquina virtual. Esto lo hace la instrucción **Class.forName()**.
- A partir del JDK 6, los drivers JDBC ya se registran automáticamente y no es necesario el **Class.forName**(), sólo que estén en el classpath de la JVM.

Uso de JDBC
II – Conexión

- Crear un objeto **Connection** invocando al método estático **DriverManager.getConnection** y pasarle la URL de la base de datos, el nombre de usuario y la contraseña.
- Si nos conectamos a una BD MySQL llamada "prueba":

```
String url = "jdbc:mysql://localhost:3306/prueba";
Connection con = DriverManager.getConnection(url,"root", "");
```

- Al crear la conexión se pueden lanzar excepciones **SQLException** que habrá que manejar.
- La conexión se cierra con el método **close()** de la clase Connection.

Uso de JDBC
III – Ejecutar consultas

- Para ejecutar consultas SQL en primer lugar hay que crear un objeto **Statement** *, con el método **createStatement()** sobre la conexión:

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

- Posteriormente se ejecuta la sentencia SQL utilizando alguno de los siguientes métodos del objeto Statement:
  - ✓ executeUpdate
  - ✓ executeQuery
  - ✓ execute

* Hay otras clases que derivan de **Statement** cuyo funcionamiento se explica más adelante.

Centro de Enseñanza
Gregorio Fernández

Uso de JDBC
III – Ejecutar consultas

executeUpdate:

- Permite ejecutar instrucciones SQL de tipo **UPDATE**, **INSERT** o **DELETE** y también sentencias **DDL** (CREATE TABLE por ejemplo).
- Devuelve un entero que indica el número de filas implicadas en la operación (en sentencias DDL siempre es 0).

executeQuery:

- Permite ejecutar **SELECT**.
- Este tipo de consultas devuelven una tabla con los resultados, que en Java se representa con objetos de la clase **ResultSet**.
- El método **next()** de esta clase permite avanzar de fila, y los métodos **get** permiten obtener el valor de las columnas.

execute

- Permite ejecutar procedimientos almacenados o sentencias SQL dinámicas.

Centro de Enseñanza
Gregorio Fernández

Uso de JDBC
IV – Liberar recursos

- Cuando se termina de usar un objeto Connection, un Statement o un ResultSet hay liberar los recursos que utilizan.
- Para ello se utilizan los métodos **close()**:
  - ✓ **ResultSet.close()**: libera los recursos del ResultSet. Se cierra automáticamente al cerrar el Statement que lo creó o al reejecutar el Statement.
  - ✓ **Statement.close()**: libera los recursos del Statement. Aunque los objetos Statement se cierran automáticamente por el recolector de basura de Java, la llamada al método close() hace que se libere inmediatamente la basura y se eviten posibles problemas con la memoria.
  - ✓ **Connection.close()**: finaliza la conexión con la base de datos.

Uso de JDBC
IV – Liberar recursos

- Para ello se utilizan los métodos **close()**:
  - ✓ **ResultSet.close()**: libera los recursos del ResultSet. Se cierra automáticamente al cerrar el Statement que lo creó o al reejecutar el Statement.
  - ✓ **Statement.close()**: libera los recursos del Statement. Aunque los objetos Statement se cierran automáticamente por el recolector de basura de Java, la llamada al método close() hace que se libere inmediatamente la basura y se eviten posibles problemas con la memoria.
  - ✓ **Connection.close()**: finaliza la conexión con la base de datos.
- Como hicimos con los fichero, es posible usar bloques **Try-with-resources** en los que crear la conexión a la BD, las Statement y ResultSet y de esta forma liberarnos del trabajo de realizar los cierres manualemte. Aunque habrá casos en los que tendremos que hacerlo de forma explícita.

Uso de JDBC
V – Ejemplos

executeUpdate

```
try {
    //conexión con base de datos
    …
    //ejecución de sentencia SQL
    Statement st=con.createStatement();
    System.out.println(st.executeUpdate("UPDATE clientes SET " +
    "sexo='V' WHERE sexo='H'")); //líneas afectadas
} catch (SQLException e){
    System.out.println(e.getMessage());
}
```

Uso de JDBC
V – Ejemplos

executeQuery

```
try {
    //conexión con base de datos
    …
    //ejecución de sentencia SQL
    Statement st=con.createStatement();
    ResultSet rs=st.executeQuery("SELECT * FROM empleados");
    while(rs.next()){
          System.out.println(rs.getString("Nombre") + " " +
                rs.getInt("Edad"));
    }

}catch (SQLException e){
    e.printStackTrace();
}
```

Acceso al conjunto de resultados

Centro de Enseñanza
Gregorio Fernández

Uso de JDBC
V – Ejemplos

ResultSet

- El **ResultSet** es el objeto que representa el resultado de la consulta. Internamente tiene un **cursor** (puntero) que apunta a un fila concreta. Hay que posicionar el cursor en cada fila y obtener la información de la misma.
- El **cursor** puede estar en una fila concreta. También puede estar en dos filas especiales:
  - Antes de la primera fila (*Before the First Row,* **BFR**)
  - Después de la última fila (*After the Last Row,* **ALR**)
- Inicialmente el **ResultSet** está en BFR.
- El método **next()** mueve el **cursor** hacia delante, devuelve **true** si se encuentra en una fila concreta y **false** si alcanza el ALR. Al avanzar por los registros se pueden utilizar los métodos **get** para obtener los valores de las columnas.
- Los métodos **get** tienen como nombre **get** seguido del tipo de datos a leer (**getString, getByte, getInt,...**). Se les pasa como parámetro el nombre del campo a leer (el que esté puesto en la base de datos) o el número de campo según el orden en el que aparezca en la tabla de la base de datos (el primer campo es el 1).

Uso de JDBC
V – Ejemplos

execute

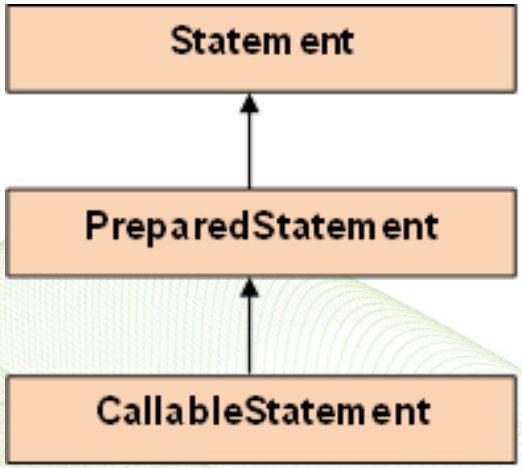
- Si no se sabe cuál es el tipo de sentencia SQL a ejecutar, debe utilizarse el método **execute**.
- Una vez que se ha procesado este método, el controlador JDBC indica qué tipos de resultados ha generado la sentencia SQL.
- De tal forma que el método devuelve:
  - **true**: si el resultado es un ResultSet como mínimo.
  - **false**: si el valor de retorno es una cuenta de actualización.
- Con esta información, se pueden utilizar los métodos **getUpdateCount** o **getResultSet** para recuperar el valor de retorno de la sentencia SQL.

Centro de Enseñanza
Gregorio Fernández

Excepciones

- Se pueden producir excepciones **SQLException** cuando trabajamos con bases de datos desde Java.
- Su uso no difiere del resto de excepciones, pero incorpora nuevos métodos interesantes para ayudarnos a determinar el origen del error:
  - ✓ **getSQLState**: devuelve el error producido y su significado según las convenciones XOPEN.
  - ✓ **getMessage**: devuelve el texto que envía el gestor de la base de datos.
  - ✓ **getErrorCode**: devuelve el código numérico que identifica el error producido, según el SGBD.

Tipos de sentencias SQL



Statement

- Para enviar consultas SQL estáticas en tiempo de ejecución.
  No acepta parámetros:

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

PreparedStatement

- Hereda de Statement.
  Permite ejecutar la misma sentencia muchas veces (la "prepara").
  Acepta parámetros:

```
PreparedStatement ps = con.prepareStatement(…);
```

CallableStatement

- Llamada a procedimientos almacenados

```
CallableStatement s = con.prepareCall(…);
```

PreparedStatement

- Permite ejecutar consultas con **parámetros** y sentencias *precompiladas*, que son aquellas sentencias que se preparan al crear el objeto y posteriormente pueden llamarse varias veces con los métodos **execute**.
- Ejemplo:

```
PreparedStatement ps = con.
    prepareStatement("INSERT INTO Libros VALUES (?,?,?)");
ps.setInt(1,23);
ps.setString(2,"El Quijote");
ps.setInt(3,45);
ps.executeUpdate();
```

PreparedStatement

- Como vemos en el ejemplo, para crear objetos **PreparedStatement** nuevos, se utiliza el método **prepareStatement**.
- A diferencia del método createStatement, la **sentencia SQL** debe suministrarse al crear el objeto PreparedStatement. En ese momento, la sentencia SQL se precompila para su utilización.
- Después para poder procesar el PreparedStatement, debe establecerse un valor en cada uno de los **marcadores de parámetro** (?). Para ello el objeto PreparedStatement proporciona varios métodos. Todos ellos tienen el formato *set<Tipo>,* donde *<Tipo>* un tipo de datos Java. Por ejemplo, setInt, setLong, setString... Casi todos estos métodos toman dos parámetros:
  - El primero es el índice que el parámetro tiene dentro de la sentencia, empezando por el 1.
  - El segundo parámetro es el valor que debe establecerse en el parámetro.
- Consulta el Javadoc del paquete java.sql para obtener más información.
- Si se intenta procesar una PreparedStatement con marcadores de parámetro que no se han establecido, se lanza una SQLException.
- El método **clearParameters** limpia el valor de todos los parámetros del objeto PreparedStatement.

CallableStatement

- Permite hacer llamadas a los **subprogramas almacenados** de la base de datos.
- Estos programas:
  - ✓ Pueden tener parámetros de entrada, de salida, o parámetros de entrada y salida.
  - ✓ Pueden tener un valor de retorno.
  - ✓ Tienen la capacidad de devolver varios **ResultSets**.
- Conceptualmente, en JDBC una llamada de subprograma almacenado es una sola llamada a la base de datos, pero el programa asociado puede procesar varias peticiones de base de datos.
- La encapsulación de gran cantidad de trabajo de base de datos en una sola llamada reutilizable es un ejemplo de utilización óptima. Solo esta llamada se transmite por la red, pero la petición puede realizar gran cantidad de trabajo en el sistema remoto.
- Este tipo de sentencias, siguen el modelo de **PreparedStatement,** que consiste en *desacoplar las fases de preparación y proceso*, permitiendo la reutilización optimizada.

CallableStatement

- Ejemplo:

```
CallableStatement cstmt =
    con.prepareCall (”{? = call obtenerNombreEmp (?)}");
cstmt.setInt(2,99);
cstmt.registerOutParameter(1,java.sql.Types.VARCHAR);
cstmt.execute();
String nombreEmple = cstmt.getString(1);
```

- El método **prepareCall** se utiliza para crear objetos **CallableStatement.**
- Al igual que en el método **prepareStatement**, la sentencia SQL debe suministrarse en el momento de crear el objeto **CallableStatement**. En ese momento, se precompila la sentencia SQL.
- El procedimiento almacenado toma un parámetro de entrada para el ID del empleado.
- A partir de esta información, obtiene el nombre del empleado devolviéndolo como valor de retorno.

CallableStatement – Parámetros

Los objetos **CallableStatement** pueden tomar tres tipos de parámetros:

- **IN:** se manejan de la misma forma que en **PreparedStatements**. Se establecen sus valores con los métodos **set**.
- **OUT:** manejan con el método **registerOutParameter**. En su forma más común, el primer parámetro es el índice de parámetro y el segundo es un tipo de SQL. Este indica al controlador JDBC qué tipo de datos debe esperar del parámetro cuando se procesa la sentencia.

  Existen otras dos variantes del método que puedes consultar en el API java.sql.
- **INOUT**: requieren que se realice el trabajo tanto para parámetros IN como para parámetros OUT. Para cada parámetro INOUT, debe llamarse a un método **set** y al método **registerOutParameter** para que pueda procesarse la sentencia. Si alguno de los parámetros no se establece o registra, se lanza una *SQLException*.
- El método **clearParameters** no afecta a los parámetros OUT.
- JDBC 3.0, tiene soporte para especificar parámetros de procedimiento almacenado por nombre y por índice.

ResultSet

- El objeto **ResultSet** representa el resultado de una consulta.
- Conceptualmente, los datos de un ResultSet pueden considerarse como una tabla con un número específico de columnas y un número específico de filas. Dentro de una fila, se puede acceder a los valores de columna.
- Se pueden crear objetos **ResultSet** a partir de objetos **Statement**, **PreparedStatement** y **CallableStatement**.
- Un **ResultSet** no solo se puede usar para recorrer las filas de una consulta utilizando su **cursor**, también se puede usar para actualizar, borrar y añadir nuevas filas.
- Para ello el **ResultSet** debe tener unas características especiales que se pueden configurar al crear el objeto **Statement, PreparedStatement** o **CallableStatement**.

ResultSet
I - Tipos

1.Tipo: *¿Cómo puede recorrerse?*

- **ResultSet.TYPE_FORWARD_ONLY**. Sólo movimiento hacia delante (por defecto).
- **ResultSet.TYPE_SCROLL_INSENSITIVE**. Puede moverse hacia delante y hacia atrás. No refleja los cambios que se produzcan en la base de datos mientras permanece abierto. Contiene los datos que se recuperaron cuando se ejecutó el comando SQL.
- **ResultSet.TYPE_SCROLL_SENSITIVE**. Puede moverse hacia delante y hacia atrás y además refleja los cambios que se produzcan en la base de datos, trabajando por tanto, con los cambios que se produzcan en la base de datos.

2.Concurrencia: *¿Puede actualizarse?*

- **ResultSet.CONCUR_READ_ONLY**. Sólo se utiliza para leer datos de la BD (por defecto).
- **ResultSet.CONCUR_UPDATABLE**. Actualizable. Los cambios que se realicen en el ResultSet pueden llevarse a la BD.
- No todos los drivers JDBC soportan la concurrencia con la base de datos.

Centro de Enseñanza
Gregorio Fernández

ResultSet
II - Desplazamiento

- Si se permite el desplazamiento, la clase **ResultSet** posee los siguientes métodos para desplazarse por los registros:
  - **next()**
  - **previous()**
  - **absolute(int pos)**
  - **relative(int fila)**
  - **first()**
  - **last()**
  - **beforeFirst()**
  - **afterLast()**
  - **isFirst()**
  - **isLast()**
  - **isBeforeFirst()**
  - **isAfterLast()**
  - **getRow()**
- Puedes consultar el API

Centro de Enseñanza
Gregorio Fernández

ResultSet
III – Recuperar datos

- El objeto ResultSet proporciona varios métodos para obtener los datos de columna correspondientes a un fila.
- Todos ellos tienen el formato **get<*Tipo*>**, siendo <*Tipo*> un tipo de datos Java. Por ejemplo: *getInt*, *getLong*, *getString, getTimestamp* y *getBlob*.
- Casi todos estos métodos toman un solo parámetro, que es el índice que la columna tiene dentro del ResultSet o bien el nombre de la columna.
- Las columnas de ResultSet están numeradas, empezando por el 1. Si se emplea el nombre de la columna y hay más de una columna que tenga ese mismo nombre en el ResultSet, se devuelve la primera.
- Algunos de los métodos get tienen parámetros adicionales. Consulta el API para obtener todos los detalles.
- En los métodos get que devuelven objetos, el valor de retorno es null cuando la columna del ResultSet es nula. En tipos primitivos, se devuelve el valor es 0 o false. Si una aplicación debe distinguir entre null, y 0 o false, puede utilizarse el método **wasNull** inmediatamente después de la llamada.

ResultSet
VI - Actualización

- Se utilizan los métodos **update<*Tipo*>**, de la clase ResultSet, que permiten modificar el contenido de una columna en la posición actual del cursor.
- Estos métodos reciben dos parámetros, el primero que indica la columna a modificar y el segundo que indica el nuevo valor para la columna
- Ejemplo:

```
rs.first(); //no situamos en el registro a modificar
rs.updateDouble("Precio",2.34); // modificamos el precio
rs.updateString("Tipo","Comestibles"); //y el tipo
rs.updateRow(); //mandamos los cambios a la BD
```

- Hasta que no se realice el último paso la BD no se actualiza.

ResultSet
V – Añadir datos

- Para añadir una nueva fila en el ResultSet (y BD) hay que emplear los métodos **update<*Tipo*>** sobre una fila especial.
- Pasos:
  1. Colocar el cursor en la fila de inserción de registros: método **moveToInsertRow**.
  2. Actualizar los datos de los campos de ese nuevo registros: métodos **update<*Tipo*>** .
  3. Añadir el registro en la base de datos con **insertRow** (se puede cancelar con *cancelRowUpdates*)
  4. Colocar el cursor en la posición anterior al movimiento de inserción: método **moveToCurrentRow**.

```
rs.moveToInsertRow();
rs.updateString(1, "Limpieza");
rs.updateInt(2,35);
rs.updateBoolean(3, true);
rs.insertRow();
rs.moveToCurrentRow();
```

ResultSet
VI – Borrar

- Para borrar la fila actual del ResultSet se utiliza el método **deleteRow()**.

VII – Actualizar

- El método **refreshRow()** del ResultSet actualiza el valor del registro actual, mostrando lo que valga en ese momento en la base de datos.
- Se utiliza por si se ha modificado el valor de los datos desde otro cliente de la base de datos.

Centro de Enseñanza
Gregorio Fernández

ResultSet vs executeUpdate

- Según lo expuesto hasta el momento, a la hora de realizar actualizaciones en la base de datos, tenemos 2 posibilidades:
  a) Hacerlo a través de un ResultSet con los métodos **insertRow(), updateRow()** y **deleteRow().**
  b) Mandar las consultas a la BD con el método **executeUpdate** de Statement.

ResultSet vs executeUpdate

- Desde mi punto de vista, creo que aunque la primera opción puede resultarnos la más atractiva, ya que no tendríamos que escribir las sentencias SQL para realizar las operaciones…
- La realidad es que este enfoque es menos escalable, pues en el caso de que tengamos varios usuarios trabajando con la base de datos, lo ideal es que compartan conexiones, para lo cual, habría que mantener una conexión ocupada el menor tiempo posible (abrir conexión – ejecutar sentencia – cerrar conexión)…
- Y esto se consigue con la segunda opción, no con la primera, ya que en este caso se mantendría la conexión ocupada todo el tiempo durante el que se está trabajando con el ResultSet.

Metadatos

- Los datos que describen la estructura de las bases de datos (metadatos) se obtienen utilizando el método **getMetaData()** de la clase **Connection**, que da como resultado un objeto **DatabaseMetaData**.

```
DatabaseMetaData metadatos=con.getMetaData();
```

- La clase **DatabaseMetadata** contiene más de 150 métodos para recuperar información de una base de datos (catálogos, esquemas, tablas, tipos de datos, columnas, procedimientos almacenados, vistas,…) así como información sobre algunas características del controlador JDBC que estemos utilizando.
- Puedes consultar el API para obtener información detallada.

Centro de Enseñanza
Gregorio Fernández

Metadatos

- Quizás lo que nos puede resultar más interesante es obtener información del conjunto de resultados obtenido tras una consulta SELECT, utilizamos el método **getMetaData()** de la clase **ResultSet**, que da como resultado un objeto de tipo **ResultSetMetaData**.
- Este objeto también dispone de métodos interesantes que podemos consultar en el API.
- Ejemplo: