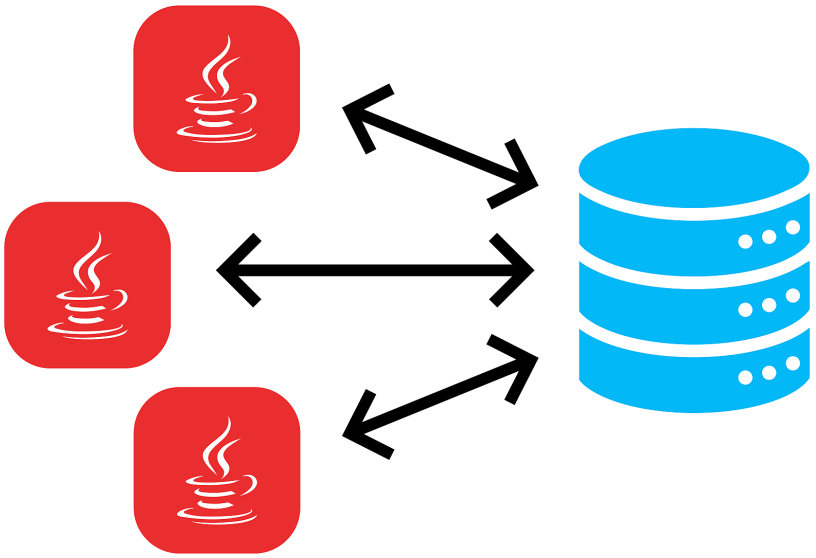
UD09

Acceso a Bases de Datos



1. Introducción

2. JDBC

Java puede conectarse con distintos SGBD y en diferentes sistemas operativos. Independientemente del método en que se almacenen los datos debe existir siempre un **mediador** entre la aplicación y el sistema de base de datos y en Java esa función la realiza **JDBC**.

> Para la conexión a las bases de datos utilizaremos la API estándar de JAVA denominada **JDBC** (Java Data Base Connection)

JDBC es un API incluido dentro del lenguaje Java para el acceso a bases de datos. Consiste en un conjunto de clases e interfaces escritas en Java que ofrecen un completo API para la programación con bases de datos, por lo tanto es la única solución 100% Java que permite el acceso a bases de datos.

JDBC es una especificación formada por una colección de interfaces y clases abstractas, que todos los fabricantes de drivers deben implementar si quieren realizar una implementación de su driver 100% Java y compatible con JDBC (JDBC-compliant driver). Debido a que JDBC está escrito completamente en Java también posee la ventaja de ser independiente de la plataforma.

> No será necesario escribir un programa para cada tipo de base de datos, una misma aplicación escrita utilizando JDBC podrá manejar bases de datos Oracle, Sybase, SQL Server, etc.

Además podrá ejecutarse en cualquier sistema operativo que posea una Máquina Virtual de Java, es decir, serán aplicaciones completamente independientes de la plataforma. Otras APIS que se suelen utilizar bastante para el acceso a bases de datos son DAO (Data Access Objects) y RDO (Remote Data Objects), y ADO (ActiveX Data Objects), pero el problema que ofrecen estas soluciones es que sólo son para plataformas Windows.

JDBC tiene sus clases en el paquete *java.sql* y otras extensiones en el paquete *javax.sql*.

2.1. Funciones del JDBC

Básicamente el API JDBC hace posible la realización de las siguientes tareas:

- Establecer una conexión con una base de datos.
- Enviar sentencias SQL.
- Manipular datos.
- Procesar los resultados de la ejecución de las sentencias.

2.2. Drivers JDBC

Los drivers nos permiten conectarnos con una base de datos determinada. Existen **cuatro tipos de drivers JDBC**, cada tipo presenta una filosofía de trabajo diferente. A continuación se pasa a comentar cada uno de los drivers:

- JDBC-ODBC bridge plus ODBC driver (tipo 1): permite al programador acceder a fuentes de datos ODBC existentes mediante JDBC. El JDBC-ODBC Bridge (puente JDBC-ODBC) implementa operaciones JDBC traduciéndolas a operaciones ODBC, se encuentra dentro del paquete *sun.jdbc.odbc* y contiene librerías nativas para acceder a ODBC.

Al ser usuario de ODBC depende de las dll de ODBC y eso limita la cantidad de plataformas en donde se puede ejecutar la aplicación.

- Native-API partly-Java driver (tipo 2): son similares a los drivers de tipo1, en tanto en cuanto también necesitan una configuración en la máquina cliente. Este tipo de driver convierte llamadas JDBC a llamadas de Oracle, Sybase, Informix, DB2 u otros SGBD. Tampoco se pueden utilizar dentro de applets al poseer código nativo.
- JDBC-Net pure Java driver (tipo 3): Estos controladores están escritos en Java y se encargan de convertir las llamadas JDBC a un protocolo independiente de la base de datos y en la aplicación servidora utilizan las funciones nativas del sistema de gestión de base de datos mediante el uso de una biblioteca JDBC en el servidor. La ventaja de esta opción es la portabilidad.
- JDBC de Java cliente (tipo 4): Estos controladores están escritos en Java y se encargan de convertir las llamadas JDBC a un protocolo independiente de la base de datos y en la aplicación servidora utilizan las funciones nativas del sistema de gestión de base de datos sin necesidad de bibliotecas. La ventaja de esta opción es la portabilidad. Son como los drivers de tipo 3 pero sin la figura del intermediario y tampoco requieren ninguna configuración en la máquina cliente. Los drivers de tipo 4 se pueden utilizar para servidores Web de tamaño pequeño y medio, así como para intranets.

3. Acceso a BB.DD.

En este apartado se ofrece una introducción a los aspectos fundamentales del acceso a bases de datos mediante código Java. En los siguientes apartados se explicarán algunos aspectos en mayor detalle, sobre todo los relacionados con las clases Statement y ResultSet.

3.1. Añadir la librería JDBC al proyecto

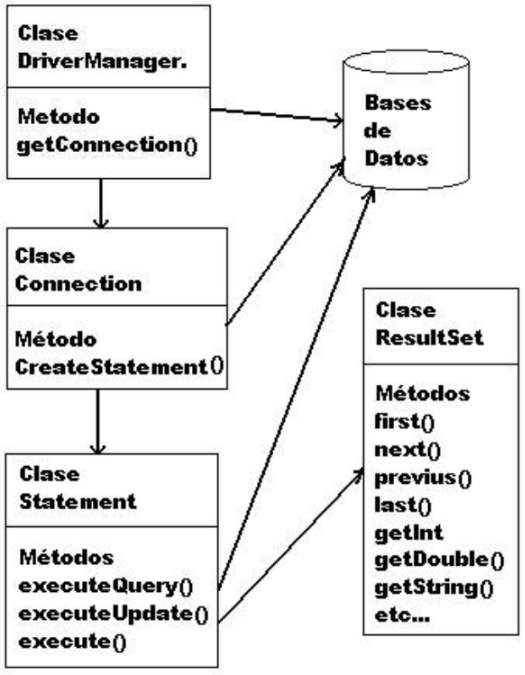
Para poder utilizar la librería JDBC en un proyecto Java primero deberemos añadirla al proyecto. Para ello debemos hacer clic derecho sobre la carpeta 'Libraries' del proyecto y seleccionar 'Add JAR/Folder'. En la ventana emergente deberemos seleccionar el archivo del driver previamente descargado mysql-connector-java-8.0.19.jar y clic en OK.

3.2. Cargar el Driver

En un proyecto Java que realice conexiones a bases de datos es necesario, antes que nada, utilizar *Class.forname(...).newInstance()* para cargar dinámicamente el Driver que vamos a utilizar. Esto solo es necesario hacerlo una vez en nuestro programa. Puede lanzar excepciones por lo que es necesario utilizar un bloque try-catch.

```
try {
    Class.forName("com.mysql.cj.jdbc.Driver").newInstance();
} catch (Exception e) {
    // manejamos el error
}
```

Hay que tener en cuenta que las clases y métodos utilizados para conectarse a una base de datos (explicados más adelante) funcionan con todos los drivers disponibles para Java (JDBC es solo uno, hay muchos más). Esto es posible ya que el estándar de Java solo los define como interfaces (interface) y cada librería driver los implementa (define las clases y su código). Por ello es necesario utilizar Class.forName(...) para indicarle a Java qué driver vamos a utilizar.



Este nivel de asbtracción facilita el desarrollo de proyectos ya que si necesitáramos utilizar otro sistema de base de datos (que no fuera MySQL) solo necesitaríamos cambiar la línea de código que carga el driver y poco más. Si cada sistema de base de datos necesitara que utilizáramos distintas clases y métodos todo sería mucho más complicado.

Las cuatro clases fundamentales que toda aplicación Java necesita para conectarse a una base de datos y ejecutar sentencias son: **DriverManager**, **Connection**, **Statement** y **ResultSet**.

3.3. Clase `DriverManager`

La clase ***java.sql.DriverManager*** es la capa gestora del driver JDBC. Se encarga de manejar el Driver apropiado y **permite crear conexiones con una base de datos** mediante el método estático **getConnection(...)** que tiene dos variantes:

DriveManager.getConnection(String url).

DriveManager.getConnection(String url, String user, String password).

Este método intentará establecer una conexión con la base de datos según la URL indicada. Opcionalmente se le puede pasar el usuario y contraseña como argumento (también se puede indicar en la propia URL). Si la conexión es satisfactoria devolverá un objeto **Connection**.

Ejemplo de conexión a la base de datos 'prueba' en localhost:

String url = "jdbc:mysql://localhost:3306/prueba";

Connection conn = DriverManager.getConnection(url,"root","");

Este método puede lanzar dos tipos de excepciones (que habrá que manejar con un try-catch):

- **SQLException**: La conexión no ha podido producirse. Puede ser por multitud de motivos como una URL mal formada, un error en la red, host o puerto incorrecto, base de datos no existente, usuario y contraseña no válidos, etc.
- **SQLTimeOutException**: Se ha superado el LoginTiemout sin recibir respuesta del servidor.

3.4. Clase `Connection`

Un objeto **java.sql.Connection representa una sesión de conexión con una base de datos**. Una aplicación puede tener tantas conexiones como necesite, ya sea con una o varias bases de datos.

El método más relevante es **createStatement()** que devuelve un objeto Statement asociado a dicha conexión que permite ejecutar sentencias SQL. El método createStatement() puede lanzar excepciones de tipo **SQLException**.

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

Cuando ya no la necesitemos es aconsejable **cerrar la conexión con close()** para liberar recursos.

```
conn.close();
```

3.5. Clase `Statement`

Un objeto **java.sql.Statement** permite **ejecutar sentencias SQL en la base de datos** a través de la conexión con la que se creó el Statement (ver apartado anterior). Los tres métodos más comunes de ejecución de sentencias SQL son executeQuery(...), executeUpdate(...) y execute(...). Pueden lanzar excepciones de tipo **SQLException** y **SQLTimeoutException**.

- **ResultSet executeQuery(String sql)**: Ejecuta la sentencia sql indicada (de tipo SELECT). Devuelve un objeto ResultSet con los datos proporcionados por el servidor.

```
1 ResultSet rs = stmt.executeQuery("SELECT * FROM vendedores");
```

- **int executeUpdate(String sql)**: Ejecuta la sentencia sql indicada (de tipo DML como por ejemplo INSERT, UPDATE o DELETE). Devuelve un el número de registros que han sido insertados, modificados o eliminados.

```
1 int nr = stmt.executeUpdate("INSERT INTO vendedores VALUES (1,'Pedro
  Gil', '2017-04-11', 15000);")
```

Cuando ya no lo necesitemos es aconsejable **cerrar el statement con close()** para liberar recursos.

```
1 stmt.close();
```

3.6. Clase `ResultSet`

Un objeto **java.sql.ResultSet** contiene un conjunto de resultados (datos) obtenidos tras ejecutar una sentencia SQL, normalmente de tipo SELECT. Es una **estructura de datos en forma de tabla** con **registros (filas)** que podemos recorrer para acceder a la información de sus **campos (columnas)**.

ResultSet utiliza internamente un cursor que apunta al 'registro actual' sobre el que podemos operar. Inicialmente dicho cursor está situado antes de la primera fila y disponemos de varios métodos para desplazar el cursor. El más común es next():

- **boolean next()**: Mueve el cursor al siguiente registro. Devuelve true si fué posible y false en caso contrario (si ya llegamos al final de la tabla).

Algunos de los métodos para obtener los datos del registro actual son:

- **String getString(String columnLabel)**: Devuelve un dato String de la columna indicada por su nombre. Por ejemplo: rs.getString("nombre")
- **String getString(int columnIndex)**: Devuelve un dato String de la columna indicada por su nombre. La primera columna es la 1, no la cero. Por ejemplo: rs.getString(2)

Existen métodos análogos a los anteriores para obtener valores de tipo int, long, float, double, boolean, Date, Time, Array, etc. Pueden consultarse todos en la documentación oficial de Java.

- **int getInt(String columnLabel) int getInt(int columnIndex)**
- **double getDouble(String columnLabel) double getDouble(int columnIndex)**
- **boolean getBoolean(String columnLabel) boolean getBoolean(int columnIndex)**
- **Date getDate(String columnLabel) int getDate(int columnIndex)**

- etc.

Más adelante veremos cómo se realiza la modificación e inserción de datos.

Todos estos métodos pueden lanzar una **SQLException**.

Veamos un ejemplo de cómo recorrer un ResultSet llamado rs y mostrarlo por pantalla:

```
while(rs.next()) {
    int id = rs.getInt("id");
    String nombre = rs.getString("nombre");
    Date fecha = rs.getDate("fecha_ingreso");
    float salario = rs.getFloat("salario");
    System.out.println(id + " " + nombre + " " + fecha + " " + salario);
}
```

4. Navegabilidad y concurrencia

Cuando invocamos a ***createStatement()* sin argumentos**, como hemos visto anteriormente, al ejecutar sentencias SQL obtendremos un ***ResultSet* por defecto en el que el curso**r** solo puede moverse hacia adelante y los datos son de solo lectura**. A veces esto no es suficiente y necesitamos mayor funcionalidad.

Por ello el método *createStatement()* está sobrecargado (existen varias versiones de dicho método) lo cual nos permite invocarlo con argumentos en los que podemos especificar el funcionamiento.

- **Statement createStatement(int resultSetType, int resultSetConcurrency)**: Devuelve un objeto Statement cuyos objetos ResultSet serán del tipo y concurrencia especificados. Los valores válidos son constantes definidas en ResultSet.

El **argumento resultSetType** indica el tipo de ResultSet:

- **ResultSet.TYPE_FORWARD_ONLY**: ResultSet por defecto, forward-only y no-actualizable.
    - Solo permite movimiento hacia delante con next().
    - Sus datos NO se actualizan. Es decir, no reflejará cambios producidos en la base de datos. Contiene una instantánea del momento en el que se realizó la consulta.
- **ResultSet.TYPE_SCROLL_INSENSITIVE**: ResultSet desplazable y no actualizable.
    - Permite libertad de movimiento del cursor con otros métodos como first(), previous(), last(), etc. además de next().
    - Sus datos NO se actualizan, como en el caso anterior.
- **ResultSet.TYPE_SCROLL_SENSITIVE**: ResultSet desplazable y actualizable.
    - Permite libertad de movimientos del cursor, como en el caso anterior.
    - Sus datos SÍ se actualizan. Es decir, mientras el ResultSet esté abierto se actualizará automáticamente con los cambios producidos en la base de datos. Esto puede suceder incluso mientras se está recorriendo el ResultSet, lo cual puede ser conveniente o contraproducente según el caso.

El **argumento resultSet.Concurrency** indica la concurrencia del ResultSet:

- **ResultSet.CONCUR_READ_ONLY**: Solo lectura. Es el valor por defecto.
- **ResultSet.CONCUR_UPDATABLE**: Permite modificar los datos almacenados en el ResultSet para luego aplicar los cambios sobre la base de datos (más adelante se verá cómo).

> El `ResultSet` por defecto que se obtiene con `createStatement()` sin argumentos es el mismo que con `createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY)`.

5. Consultas (Query)

5.1. Navegación de un `ResultSet`

Como ya se ha visto, en un objeto *ResultSet* se encuentran los resultados de la ejecución de una sentencia SQL. Por lo tanto, un objeto *ResultSet* contiene las filas que satisfacen las condiciones de una sentencia SQL, y ofrece métodos de navegación por los registros como next() que desplaza el cursos al siguiente registro del ResultSet.

Además de este método de desplazamiento básico, existen otros de desplazamiento libre que podremos utilizar siempre y cuando el ResultSet sea de tipo ResultSet.TYPE_SCROLL_INSENSITIVE o ResultSet.TYPE_SCROLL_SENSITIVE como se ha dicho antes.

Algunos de estos métodos son:

- **void beforeFirst():** Mueve el cursor antes de la primera fila.
- **boolean first():** Mueve el cursor a la primera fila.
- **boolean next():** Mueve el cursor a la siguiente fila. Permitido en todos los tipos de ResultSet.
- **boolean previous():** Mueve el cursor a la fila anterior.
- **boolean last():** Mueve el cursor a la última fila.
- **void afterLast()**. Mover el cursor después de la última fila.
- **boolean absolute(int row)**: Posiciona el cursor en el número de registro indicado. Hay que tener en cuenta que el primer registro es el 1, no el cero. Por ejemplo absolute(7) desplazará el cursor al séptimo registro. Si valor es negativo se posiciona en el número de registro indicado pero empezando a contar desde el final (el último es el -1). Por ejemplo si tiene 10 registros y llamamos absolute(-2) se desplazará al registro n.º 9.
- **boolean relative(int registros)**: Desplaza el cursor un número relativo de registros, que puede ser positivo o negativo. Por ejemplo si el cursor esrá en el registro 5 y llamamos a relative(10) se desplazará al registro 15. Si luego llamamos a relative(-4) se desplazará al registro 11.

Los métodos que devuelven un tipo boolean devolverán 'true' si ha sido posible mover el cursor a un registro válido, y 'false' en caso contrario, por ejemplo si no tiene ningún registro o hemos saltado a un número de registro que no existe.

Todos estos métodos pueden producir una excepción de tipo SQLException.

También existen otros métodos relacionados con la posición del cursor.

- **int getRow():** Devuelve el número de registro actual. Cero si no hay registro actual.
- **boolean isBeforeFirst()**: Devuelve 'true' si el cursor está antes del primer registro.
- **boolean isFirst()**: Devuelve 'true' si el cursor está en el primer registro.
- **boolean isLast()**: Devuelve 'true' si el cursor está en el último registro.
- **boolean isAfterLast()**: Devuelve 'true' si el cursor está después del último registro.

5.2. Obteniendo datos del ResultSet

Los métodos *getXXX()* ofrecen los medios para recuperar los valores de las columnas (campos) de la fila (registro) actual del *ResultSet*. No es necesario que las columnas sean obtenidas utilizando un orden determinado.

Para designar una columna podemos utilizar su nombre o bien su número (empezando por 1).

Por ejemplo si la segunda columna de un objeto ResultSet se llama "título" y almacena datos de tipo String, se podrá recuperar su valor de las dos formas siguientes:

```
1 // rs es un objeto ResultSet
2 String valor = rs.getString(2);
3 String valor = rs.getString("titulo");
```

Es importante tener en cuenta que las columnas se numeran de izquierda a derecha y que la primera es la número 1, no la cero. También que las columnas no son case sensitive, es decir, no distinguen entre mayúsculas y minúsculas.

> La información referente a las columnas de un ResultSet se puede obtener llamando al **método getMetaData()** que devolverá un objeto ResultSetMetaData que contendrá el número, tipo y propiedades de las columnas del ResultSet.

Si conocemos el nombre de una columna, pero no su índice, el método *findColumn()* puede ser utilizado para obtener el número de columna, pasándole como argumento un objeto *String* que sea el nombre de la columna correspondiente, este método nos devolverá un entero que será el índice correspondiente a la columna.

5.3. Tipos de datos y conversiones

Cuando se lanza un método *getXXX()* determinado sobre un objeto ResultSet para obtener el valor de un campo del registro actual, el driver JDBC convierte el dato que se quiere recuperar al tipo Java especificado y entonces devuelve un valor Java adecuado. Por ejemplo si utilizamos el método *getString()* y el tipo del dato en la base de datos es *VARCHAR*, el driver JDBC convertirá el dato VARCHAR de tipo SQL a un objeto *String* de Java.

Algo parecido sucede con otros tipos de datos SQL como por ejemplo DATE. Podremos acceder a él tanto con getDate() como con getString(). La diferencia es que el primero devolverá un objeto Java de tipo Date y el segundo devolverá un String.

Siempre que sea posible el driver JDBC convertirá el tipo de dato almacenado en la base de datos al tipo solicitado por el método getXXX(), pero hay conversiones que no se pueden realizar y lanzarán una excepción, como por ejemplo si intentamos hacer un getInt() sobre un campo que no contiene un valor numérico.

6. Modificación (Update)

Para poder modificar los datos que contiene un *ResultSet* necesitamos un *ResultSet* de tipo modificable. Para ello debemos utilizar la constante *ResultSet.CONCUR_UPDATABLE* al llamar al método *createStatement()* como se ha visto antes.

Para modificar los valores de un registro existente se utilizan una serie de métodos *updateXXX()* de *ResultSet*. Las *XXX* indican el tipo del dato y hay tantos distintos como sucede con los métodos *getXXX()* de este mismo interfaz: **updateString(), updateInt(), updateDouble(), updateDate(), etc.**

La diferencia es que **los métodos updateXXX() necesitan dos argumentos**:

- La columna que deseamos actualizar (por su nombre o por su numéro de columna).
- El valor que queremos almacenar en dicha columna (del tipo que sea).

Por ejemplo para modificar el campo 'edad' almacenando el entero 28 habría que llamar al siguiente método, suponiendo que rs es un objeto ResultSet:

```
1 rs.updateInt("edad", 28);
```

También podría hacerse de la siguiente manera, suponiendo que la columna "edad" es la segunda:

```
1 rs.updateInt(2, 28);
```

Los métodos updateXXX() no devuelven ningún valor (son de tipo void). Si se produce algún error se lanzará una SQLException.

Posteriormente hay que **llamar a updateRow() para que los cambios realizados se apliquen sobre la base de datos**. El Driver JDBC se encargará de ejecutar las sentencias SQL necesarias. Esta es una característica muy potente ya que nos facilita enormemente la tarea de modificar los datos de una base de datos. Este método devuelve void.

En resumen, el proceso para realizar la modificación de una fila de un *ResultSet* es el siguiente:

1. **Desplazamos el cursor** al registro que queremos modificar.
2. Llamamos a todos los métodos **updateXXX(...)** que necesitemos.
3. Llamamos a **updateRow()** para que los cambios se apliquen a la base de datos.

Es importante entender que **hay que llamar a updateRow() antes de desplazar el cursor**. Si desplazamos el cursor antes de llamar a updateRow(), se perderán los cambios.

Si queremos **cancelar las modificaciones de un registro del ResultSet** podemos llamar a **c****ancelRowUpdates()**, que cancela todas las modificaciones realizadas sobre el registro actual.

Si ya hemos llamado a updateRow() el método cancelRowUpdates() no tendrá ningún efecto.

El siguiente código de ejemplo muestra cómo modificar el campo 'dirección' del último registro de un ResultSet que contiene el resultado de una SELECT sobre la tabla de clientes. Supondremos que conn es un objeto Connection previamente creado:

```java
// Creamos un Statement scrollable y modificable
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
// Ejecutamos un SELECT y obtenemos la tabla clientes en un ResultSet
String sql = "SELECT * FROM clientes";
ResultSet rs = stmt.executeQuery(sql);
// Vamos al último registro, lo modificamos y actualizamos la base de datos
rs.last();
rs.updateString("direccion", "C/ Pepe Ciges, 3");
rs.updateRow();
```

7. Inserción (Insert)

Para insertar nuevos registros necesitaremos utilizar, al menos, estos dos métodos:

- **void moveToInsertRow()**: Desplaza el cursor al 'registro de inserción'. Es un registro especial utilizado para insertar nuevos registros en el ResultSet. Posteriormente tendremos que llamar a los métodos updateXXX() ya conocidos para establecer los valores del registro de inserción. Para finalizar hay que llamar a insertRow().
- **void insertRow()**: Inserta el 'registro de inserción' en el ResultSet, pasando a ser un registro normal más, y también lo inserta en la base de datos.

El siguiente código inserta un nuevo registro en la tabla 'clientes'. Supondremos que conn es un objeto Connection previamente creado:

```
1  // Creamos un Statement scrollable y modificable
2  Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
   ResultSet.CONCUR_UPDATABLE);
3  // Ejecutamos un SELECT y obtenemos la tabla clientes en un ResultSet
4  String sql = "SELECT * FROM clientes";
5  ResultSet rs = stmt.executeQuery(sql);
6  // Creamos un nuevo registro y lo insertamos
7  rs.moveToInsertRow();
8  rs.updateString(2,"Killy Lopez");
9  rs.updateString(3,"Wall Street 3674");
10 rs.insertRow();
```

Los campos cuyo valor no se haya establecido con updateXXX() tendrán un valor NULL. Si en la base de datos dicho campo no está configurado para admitir nulos se producirá una SQLException.

Tras insertar nuestro nuevo registro en el objeto ResultSet podremos volver a la anterior posición en la que se encontraba el cursor (antes de invocar moveToInsertRow()) llamando al método moveToCurrentRow(). Este método sólo se puede utilizar en combinación con moveToInsertRow().

8. Borrado (Delete)

Para eliminar un registro solo hay que desplazar el cursor al registro deseado y llamar al método:

- **void deleteRow()**: Elimina el registro actual del ResultSet y también de la base de datos.

El siguiente código borra el tercer registro de la tabla 'clientes':

```
// Creamos un Statement scrollable y modificable
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
// Ejecutamos un SELECT y obtenemos la tabla clientes en un ResultSet
String sql = "SELECT * FROM clientes";
ResultSet rs = stmt.executeQuery(sql);
// Desplazamos el cursor al tercer registro
rs.absolute(3)
rs.deleteRow();
```

9. Ejemplos

9.1. Ejemplo1

9.2. Ejemplo completo

Veamos un ejemplo completo de conexión y acceso a una base de datos utilizando todos los elementos mencionados en este apartado.

```
try {
    // Cargamos la clase que implementa el Driver
    Class.forName("com.mysql.cj.jdbc.Driver").newInstance();

    // Creamos una nueva conexión a la base de datos 'prueba'
    String url = "jdbc:mysql://localhost:3306/prueba?serverTimezone=UTC";

    Connection conn = DriverManager.getConnection(url,"root","");

    // Obtenemos un Statement de la conexión
    Statement st = conn.createStatement();

    // Ejecutamos una consulta SELECT para obtener la tabla vendedores
    String sql = "SELECT * FROM vendedores";

    ResultSet rs = st.executeQuery(sql);

    // Recorremos todo el ResultSet y mostramos sus datos

    while(rs.next()) {
        int id        = rs.getInt("id");
        String nombre = rs.getString("nombre");
        Date fecha    = rs.getDate("fecha_ingreso");
        float salario = rs.getFloat("salario");
        System.out.println(id + " " + nombre + " " + fecha + " " + salario);
    }
    // Cerramos el statement y la conexión
    st.close();
    conn.close();
} catch (SQLException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
}
```

10. Píldoras informáticas relacionadas

- https://www.youtube.com/playlist?list=PLNjWMbvTJAIjLRW2qyuc4DEgFVW5YFRSR
- https://www.youtube.com/playlist?list=PLaxZkGlLWHGUWZxuadN3J7KKaICRlhz5-

11. Fuentes de información

- Wikipedia
- Programación (Grado Superior) - Juan Carlos Moreno Pérez (Ed. Ra-ma)
- Apuntes IES Henri Matisse (Javi García Jimenez?)
- Apuntes AulaCampus
- Apuntes José Luis Comesaña
- Apuntes IOC Programació bàsica (Joan Arnedo Moreno)
- Apuntes IOC Programació Orientada a Objectes (Joan Arnedo Moreno)
- FXDocs
- https://openjfx.io/openjfx-docs/