1.- Persistencia en BD Relacionales mediante JDBC

Contenido

1 Introducción.

La historia de las bases de datos relacionales ha transcurrido íntimamente ligada a la historia de las aplicaciones de arquitectura distribuida y en particular, arquitecturas cliente-servidor. Hace poco más de un cuarto de siglo, la persistencia de las aplicaciones formaba parte del desarrollo de las mismas, que se concebían como un todo monolítico. El almacenamiento y la recuperación de los datos se mezclaban y difuminaban con la explotación de las mismas. Las bases de datos solían disponer de lenguajes de programación propios que incorporaban llamadas y sentencias específicas de acceso a los datos.

En realidad, esta situación no representaba muchas ventajas ni para las empresas desarrolladoras de los SGBD ni para las empresas usuarias. Las primeras se encontraban que mantener el desarrollo de un lenguaje de programación resultaba realmente costoso si no se quería quedar rápidamente desfasado. Las empresas usuarias, por otra parte, se encontraban ligadas a un lenguaje de programación que no siempre les daba respuesta a todas sus exigencias. Además, plantearse cualquier cambio de sistema gestor de datos representaba una meta imposible, dado que implicaba tener que programar de nuevo todas las aplicaciones de la empresa. Había que desvincular los lenguajes de desarrollo de los SGBD.

1.1 ODBC.

A medida que las teorías de datos relacionales iban cogiendo fuerza y las redes ganaban adeptos gracias al incremento de la eficiencia a precios realmente competitivos, comenzaron a implementarse unos sistemas gestores de bases de datos basados en la tecnología cliente-servidor.

La tecnología cliente-servidor permitió aislar los datos y los programas específicos de acceso a las mismas, del desarrollo de la aplicación (figura 1.4). La razón principal de esta división fue seguramente posibilitar el acceso remoto a los datos de cualquier ordenador conectado a la red. Lo cierto, sin embargo, es que este hecho empujó los sistemas de bases de datos a desarrollarse de una forma aislada y a crear protocolos y lenguajes específicos para poderse comunicar remotamente con las aplicaciones que corrían en ordenadores externos.

Poco a poco, el software alrededor de las bases de datos creció espectacularmente intentando dar respuesta a un máximo abanico de demandas a través de sistemas altamente configurables. Es lo que hoy en día se conoce como middleware o capa intermedia de persistencia. Es decir, el conjunto de aplicaciones, utilidades, bibliotecas, protocolos y lenguajes, situados tanto en la parte servidor como en la parte cliente, que permiten conectarse remotamente a una base de datos para configurarlo o explotar sus datos.

La llegada de los estándares

Inicialmente, cada empresa desarrolladora de un SGBD implementaba soluciones propietarias específicas para su sistema, pero pronto se dieron cuenta de que colaborando conjuntamente podían sacar mayor rendimiento y avanzar mucho más rápidamente.

Sosteniéndose en la teoría relacional y en algunas implementaciones tempranas de las empresas IBM y Oracle, se desarrolló el lenguaje de consulta de datos llamado SQL. Este reto supuso, sin duda, un gran paso, pero las aplicaciones necesitaban API con funciones que permitieran hacer llamadas desde lenguaje de desarrollo para enviar las consultas realizadas con el SQL (figura 1.5).

Cada SGBD tiene su propia conexión y su propio API.

El grupo llamado SQL Access Group, en el que participaban prestigiosas empresas del sector como Oracle, Informix, Ingres, DEC, Sun o HP, definió un API universal con independencia del lenguaje de desarrollo y la base de datos a conectar (figura 1.6).

En 1992 , Microsoft y Simba implementan el ODBC (Open Data Base Connectivity), un API basado en la definición de SQL Acces Group, que se integra en el sistema operativo de Windows y que permite añadir múltiples conectores a varias bases de datos SQL de forma muy sencilla y transparente , ya que los conectores son autoinstalables y totalmente configurables desde las mismas herramientas del sistema operativo (figura 1.7).

La llegada del ODBC representó un avance sin precedentes en el camino hacia la interoperabilidad entre bases de datos y lenguajes de programación. La mayoría de empresas desarrolladoras de sistemas gestores de bases de datos incorporaron los drivers de conectividad a las utilidades de sus sistemas y los lenguajes de programación más importantes desarrollaron bibliotecas específicas para soportar la API ODBC.

La situación actual.

Actualmente, ODBC sigue siendo una adecuada iniciativa de conexión a SGDB relacionales. Su desarrollo sigue liderado por Microsoft , pero existen versiones en otros sistemas operativos ajenos a la compañía como UNIX / LINUX o MAC . Los lenguajes más populares de desarrollo mantienen actualizadas las bibliotecas de comunicación con las sucesivas versiones que han ido apareciendo y la mayoría de SGBD disponen de un controlador ODBC básico.

1.2 JDBC.

Casi de forma simultánea a ODBC, la empresa Sun Microsystems, en 1997 sacó a la luz JDBC, un API conector de bases de datos, implementado específicamente para usar con el lenguaje Java. Se trata de un API bastante similar a ODBC en cuanto a funcionalidad, pero adaptado a las especificidades de Java. Es decir, la funcionalidad se encuentra capsulada en clases (ya que Java es un lenguaje totalmente orientado a objetos) y además, no depende de ninguna plataforma específica, de acuerdo con la característica multiplataforma defendida por Java.

Este conector será la API que trabajaremos en detalle en esta unidad, ya que Java no dispone de ninguna biblioteca específica ODBC. Las razones esgrimidas por Sun son que ODBC no se puede utilizar directamente en Java ya que está implementado en C y no es orientado a objetos. En otras palabras, usar ODBC desde el lenguaje Java necesitaría en cualquier caso una biblioteca que adaptara el API ODBC a los requerimientos Java.

Sun Microsystem ha optado por una solución que permite hacer ambas cosas a la vez, por un lado ha implementado un conector específico de Java que puede comunicarse directamente con cualquier base de datos usando controladores (drivers) de manera muy similar a como se hace en ODBC, y por el otro lado, incorpora de serie un driver especial que actúa de adaptador entre la especificación JDBC y la especificación ODBC. Este controlador se suele llamar también puente (bridge en inglés) JDBC-ODBC. Usando este driver podremos enlazar cualquier aplicación Java con cualquier conexión ODBC. Actualmente, la gran mayoría de SGBD disponen de drivers JDBC, pero en caso de tener que trabajar con un sistema que no tenga, si dispone de controlador ODBC, podremos utilizar el puente JDBC-ODBC para lograr la conexión desde Java.

Arquitectura JDBC

Al igual que ODBC, cada desarrollador de sistemas gestores de bases de datos implementa sus propios controladores JDBC. Para conseguir la interoperabilidad entre controladores, la biblioteca estándar JDBC contiene un gran número de interfaces sin las clases que implementan. Cada controlador de cada fabricante incorpora las clases que implementan las interfaces de la API JDBC. De esta manera, el controlador utilizado será totalmente transparente a la aplicación, es decir, durante el desarrollo de la aplicación no será necesario saber cuál será el driver que finalmente se utilizará para la explotación de los datos, ya que la aplicación declarará exclusivamente las interfaces de la API JDBC (figura 1.8). De esta manera se consigue independizar la aplicación de los controladores permitiendo la sustitución del controlador original por cualquier otro compatible JDBC sin prácticamente necesidad de tener que modificar el código de la aplicación.

JDBC es una especificación formada por una colección de interfaces y clases abstractas, que deben implementar todos los fabricantes de drivers que quieran realizar una implementación de su driver 100% Java y compatible con JDBC (JDBC-compliant driver).

1.3 Funciones de JDBC

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

- Establecer una conexión con la base de datos.
- Enviar sentencias a la base de datos (SQL, manipulación de datos, creación de tablas, permisos, etc.)
- Procesar los resultados de la ejecución de las sentencias.

2 Descripción general del API JDBC

Dentro del API de JDBC existen dos conjuntos de interfaces, por un lado están aquellos interfaces utilizados por programadores de aplicaciones que utilicen el acceso a bases de datos (este es nuestro caso y es en el que nos vamos a centrar) y por otro lado existe una parte de la API de JDBC que es de un nivel más bajo y que es utilizado por los programadores de drivers.

De esta enumeración de interfaces, clases y excepciones los más importantes son:

- **java.sql.DriverManager**: es la clase gestora de los drivers. Esta clase se encarga de cargar y seleccionar el driver adecuado para realizar la conexión con una base de datos determinada.
- **java.sql.Connection**: representa una conexión con una base de datos.
- **java.sql.Statement**: actúa como un contenedor para ejecutar sentencias SQL sobre una base de datos. Este interfaz tiene otros dos subtipos: java.sql.PreparedStatement para la ejecución de sentencias SQL precompiladas a las que se le pueden pasar parámetros de entrada; y java.sql.CallableStatement que permite ejecutar procedimientos almacenados de una base de datos.
- **java.sql.ResultSet**: controla el acceso a los resultados de la ejecución de una consulta, es decir, de un objeto Statement. Permite también la modificación de estos resultados.
- **java.sql.SQLException**: para tratar las excepciones que se produzcan al manipular la base de datos, ya sea durante el proceso de conexión, desconexión u obtención y modificación de los datos.
- **java.sql.BatchUpdateException**: excepción que se lanzará cuando se produzca algún error a la hora de ejecutar actualizaciones en batch sobre la base de datos.
- **java.sql.Warning**: nos indica los warnings o avisos que se produzcan al manipular y realizar operaciones sobre la base de datos.
- **java.sql.Driver**: este interface lo deben implementar todos los fabricantes de drivers que deseen construir un driver JDBC. Representa un driver que podemos utilizar para establecer una conexión con la base de datos.
- **java.sql.Types**: realizan la conversión o mapeo de tipos estándar del lenguaje SQL a los tipos de datos del lenguaje Java.
- **java.sql.ResultSetMetaData**: este interfaz ofrece información detallada relativa a un objeto ResultSet determinado.
- **java.sql.DatabaseMetaData**: ofrece información detallada sobre la base de datos a la que nos encontramos conectados.

3 Trabajar con JDBC en un proyecto.

Para utilizar la funcionalidad de JDBC en un proyecto Eclipse hay que descargar el driver y configuar el proyecto adecuadamente:

Descargar el driver

Necesitamos disponer del driver y saber exactamente cuál es el nombre de la clase que lo contiene.

En nuestro caso vamos a utilizar bases de datos MySQL y por tanto necesitamos el driver JDBC de MySQL.

El driver se encuentra empaquetado en un **fichero .jar** llamado mysql-connector-java-*version* -bin. Tendremos que descargar este fichero a nuestro equipo desde la página oficial del fabricante (MySql).

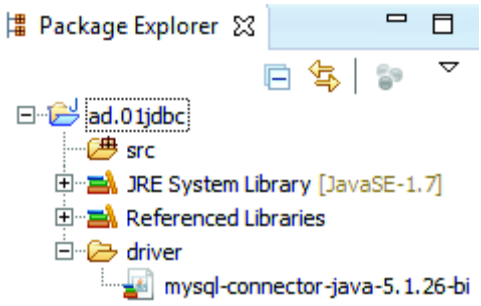
Según las especificaciones que se pueden leer en su documentación la clase que contiene el driver se llama *com.mysql.jdbc.Driver*.

Crear el proyecto y agregar el driver

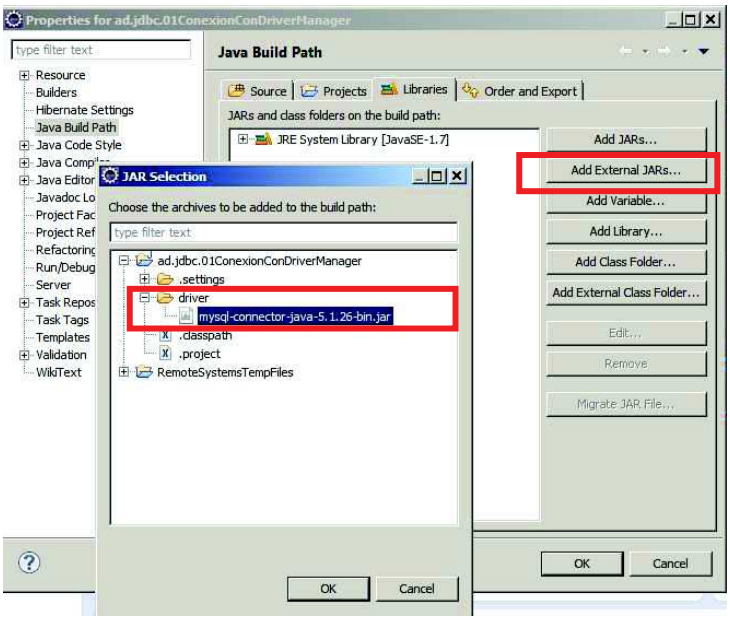
Para que la aplicación funcione, el driver JDBC debe de estar presente en el equipo del cliente. En eclipse, cuando se exporta el proyecto para distribuirlo al cliente, las **librerías** utilizadas quedan empaquetadas convenientemente y acompañan al proyecto. Lo que haremos, por tanto, es añadir el driver JDBC como librería al proyecto.

No es necesario que el driver JDBC esté dentro de una carpeta del proyecto para incluirlo como librería, pero por facilidad a la hora de trabajar para poder trasladar los proyectos de un ordenador a otra de forma sencilla, ubicaremos el driver JDBC en una paquete del proyecto al que llamaremos *driver*.

1. Crea un proyecto llamado ad.01jdbc
2. Añádele una carpeta (new – Folder), de nombre *driver.*
3. Copia en dicha carpeta el fichero mysql-connector-java-*version* –bin.jar que has descargado.



4. Agrega el driver al proyecto como biblioteca (library). Para ello, accede a la ventana de propiedades del proyecto (botón derecho sobre el proyecto en el Package Explorer) y añade el correspondiente archivo JAR, localizándolo en su ubicación en disco.



4 Conectar a BBDD con JDBC usando DriverManager

Vamos a ver cómo establecer una conexión con la base de datos utilizando la clase Driver Manager.

El código del ejemplo completo es el siguiente:

```
public class _01ConectarConsDriverManager {
   static String DRIVER = "com.mysql.jdbc.Driver";
   static String URL = "jdbc:mysql://localhost:3306/ciclismo";
   static String USR = "root";
   static String PWD = "mysql";

   public static void main(String[] args) {
      Connection con = null;
      try{
         Class.forName(DRIVER).newInstance();
         System.out.println("Registrado");

         con = DriverManager.getConnection(URL, USR, PWD);
         System.out.println("Conectado");

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

      }catch(Exception e){
         e.printStackTrace();
      }finally{
         try{
            if(con!=null && !con.isClosed())
               con.close();
               System.out.println("Desconectado");
         } catch(SQLException e){
            e.printStackTrace();
         }
      }
   }
}
```

1. **Registrar el driver.**

```
// new com.mysql.jdbc.Driver();
Class.forName(driver).newInstance();
```

La clase java.sql.DriverManager es el nivel o capa gestora del API JDBC, trabaja entre el usuario y los drivers. Tiene en cuenta los drivers disponibles y a partir de ellos establece una conexión entre una base de datos y el driver adecuado para esa base de datos.

DriverManager mantiene una lista de clases Driver que han sido registradas. Las clases Driver se pueden registran utilizando el método DriverManager.registerDriver(). Sin embargo, las clases Driver están escritas (o deberían estarlo) de forma que se registran automáticamente ellas mismas cuando se crea el objeto de la clase Driver. Es decir, que cuando se carga la clase del driver, se llama automáticamente a DriverManager.registerDriver(). No tiene que hacerlo explícitamente el programador.

Para crear la clase no usaremos el operador new como viene siendo habitual. A cambio utilizaremos el método estático *forName(clase).newInstance()*, dando como parámetro el nombre de la clase.

forName() permite crear una clase conocido su nombre, dinámicamente, por programa. La ventaja en este caso, frente a utilizar el operador new, radica en que el nombre de la clase podría estar contenido en un fichero de configuración de la aplicación. De esta manera,si el nombre del driver cambiase no sería necesario redistribuir la aplicación sino, simplemente, actualizar el nombre en dicho fichero.

```
//Registrar el driver para Mysql
String driver = "com.mysql.jdbc.Driver";
Class.forName(driver).newInstance();

// Registrar driver para Oracle
String driver = "oracle.jdbc.OracleDriver";
Class.forName(driver).newInstance();
```

Registrar el driver puede producir una serie de excepciones, de ahí el bloque try-catch:

- *forName()* lanza *ClassNotFoundException* si la clase no se encuentra.
- *newInstance()* lanza *InstantiationException* si la clase o el constructor vacío no son accesibles.
- *newInstance()* lanza *IllegalAccessException* si se trata de una clase abstracta o un interface.

2. **Establecer la conexión**

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

Una vez que las clases de los drivers se han cargado y registrado para el DriverManager están disponibles para establecer una conexión con una base de datos. La conexión se realiza mediante una llamada al método DriverManager.getConnection(), que devuelve un objeto Connection.

Una sola aplicación puede tener una o más conexiones con una misma base de datos, o puede tener varias conexiones con diferentes bases de datos. Connection es un interfaz, ya que como ya se había dicho anteriormente, JDBC ofrece una plantilla o especificación que deben implementar los fabricantes de drivers de JDBC.

Para poder conectarse se debe de especificar la URL de la base de datos, para ello se siguen la siguiente nomenclatura:

jdbc:fabricante://servidor:puerto/id_basedatos

jdbc:fabricante:@servidor:puerto:id_basedatos

Por ejemplo el URL para una base de datos Mysql

j**dbc:mysql://localhost:3306/empresa**

Para Oracle

jdbc:oracle:thin:@localhost:1521:pracs

3. Cerrar la conexión.

Para finalizar, es importante cerrar todas las conexiones abiertas antes de abandonar la aplicación, porque si no, el sistema gestor mantendrá en memoria la conexión malgastando recursos. Los objetos Connection disponen de los métodos close() y isClosed(). El primero permite cerrar la conexión, el segundo permite determinar si la conexión sigue abierta o ya ha sido cerrada.

El cierre de la conexión puede provocar una SQLException.

En resumen, para utilizar una base de datos a través de JDBC tenemos que :

a) Añadir el driver a nuestro proyecto
b) Registrar el driver
c) Establecer la conexión
d) Realizar operaciones sobre la base de datos
e) Cerrar la conexión.

5 Conectar a BBDD con JDBC usando DataSource

Hemos visto como obtener una conexión utilizando DriverManager.getConnection(). Sin embargo esta no es la forma que desde Java se indica como más adecuada.

El establecimiento y cierre de una conexión con la base de datos es una tarea "pesada". Abrir una conexión, realizar una tarea contra la base de datos y cerrar la conexión puede ser lento. En ocasiones, quizás, más que la propia acción que realizamos sobre la base de datos. Por otro lado, mantener una única conexión y compartirla o reutilizarla, podría acarrear problemas de concurrencia de hilos en el acceso a la conexión.

Para evitar estos problemas se utiliza DataSource. javax.sql.Datasource es un interfaz. DataSource.getConnection() proporciona el mismo tipo de conexiones que DriverManager.getConnection().

El API de Java no proporciona ninguna clase que implemente este interfaz, por lo que habrá descargar y utilizar alguno de terceros.

5.1 Tipos de DataSource

Un distribuidor de software podría proporcionar tres tipos de DataSources:

- Implementaciones básicas de DataSource, que no ofrecen la posibilidad de pooling (agrupamiento) ni de transacciones distribuidas.
- Implementaciones de Datasource que soportan pooling de conexiones: Producen objetos Connection que se reutilizan en lugar de volverse a crear cada vez.
- Implementaciones de Datasource que soportan pooling de conexiones y realización de transacciones distribuidas: Producen objetos Connection que, además permiten la realización de operaciones que acceden a más de un SGBD.

5.2 Pool de conexiones

En un DataSource que soporta *pooling* de conexiones, la clase mantiene abiertas varias conexiones a base de datos. Cuando alguien necesita una conexión a base de datos, en vez de abrirla directamente con DriverManager.getConnection(), se la pide al *pool* usando su método getConnection(). El *pool* coge una de las conexiones que ya tiene abierta, la marca como que alguien la está usando para no dársela a nadie más y la devuelve. La siguiente llamada a este método getConnection(), buscará una conexión libre para marcarla como ocupada y la devolverá ... y así sucesivamente.

Cuando el que ha pedido la conexión termina de usarla, normalmente después de una transacción con la base de datos o varias seguidas, llama al método Connection.close(). Esta conexión que nos ha sido entregada por el *pool*, realmente no se cierra con esta llamada. El

método close() únicamente avisa al *pool* que ya hemos terminado con la conexión, de forma que sin cerrarla, la marca como libre para poder entregársela a otro que lo pida.

Es muy importante cerrar las conexiones para que se puedan reutilizar.

5.3 apache dbcp : BasicDataSource

commons-dbcp, de apache, es una implementación sencilla de un *pool de conexiones*. La clase BasicDataSource

Dicha librería es commons-dbcp (http://commons.apache.org/dbcp/downloads.html). Esta librería necesita a su vez la librería commons-pool (http://commons.apache.org/pool/downloads.html)

Para utilizarlas en un proyecto tenemos que descargarlas y añadirlas a las librerías, tal y como hemos hecho con el driver de jdbc.

Para usar *BasicDataSource* no tenemos más que hacer un *new* de esa clase y pasarle los parámetros adecuados de nuestra conexión con los métodos *setXXXXX()* disponibles para ello. El siguiente ejemplo muestra cómo hacerlo:

```
public class _02ConectarConDataSource {
   final static String DRIVER = "com.mysql.jdbc.Driver";
   final static String URL = "jdbc:mysql://localhost:3306/ciclismo";
   final static String USR = "root";
   final static String PWD = "mysql";
   public static void main(String[] args) {
      Connection con = null;

      //Creamos datasource y configuramos sus parámetros.
      BasicDataSource ds = new BasicDataSource();
      ds.setDriverClassName(DRIVER);
      ds.setUrl(URL);
      ds.setUsername(USR);
      ds.setPassword(PWD);

      //Realizamos la conexión
      try {
         con = ds.getConnection();
         System.out.println("Conectado");
      } catch (SQLException e) {
         e.printStackTrace();
      }finally{
         //Cerramos la conexión
         try{
            if(con!=null && !con.isClosed())
               con.close();
               System.out.println("Desconectado");
         } catch(SQLException e){
            e.printStackTrace();
         }
      }

   }
}
```

También se podría utilizar un bloque try-catch with resources, de manera que la conexión se cerraría automáticamente. **Se propone como ejercicio**.

5.4 Fichero de propiedades

Es posible crear una DataSource sin tener que indicar por programa cual es la url, el nombre del driver, usuario, etc. , sino utilizando un fichero de propiedades.

Disponer de un fichero de propiedades presenta algunas ventajas: permitiría, por ejemplo, modificar la ubicación de la base de datos que utiliza el programa ya en funcionamiento sin tener que modificar el programa. El administrador del sistema del cliente, simplemente tendría que modificar el fichero de propiedades.

Seguiremos los siguientes pasos:

1. Creamos una carpeta (*Folder)* en el proyecto de nombre *configuracion*
2. En *configuración*, creamos un fichero de texto de nombre PropiedadesCiclismo, con el siguiente contenido:

   # Propiedades para la conexion a base de datos ciclismo
   driverClassName=com.mysql.jdbc.Driver
   url=jdbc:mysql://localhost:3306/ciclismo
   username=root
   password=mysql

   En el fichero especificamos los valores para el driver, la url, etc. Observa que a la izquierda del = aparece el nombre de la propiedad. Esta coincide con la palabra que usábamos en los métodos set del DataSource, pero comenzando en minúscula. Es decir, para setUrl, la propiedad será url, para setPassword la propiedad será password, etc.

   Para acceder a las propiedades

   Properties propiedades = new Properties();
   propiedades.load(new FileInputStream("configuracion//PropiedadesCiclismo.txt"));

Properties es una clase descendiente de *HashTable*, que se carga con pares (nombre de la propiedad, valor de la propiedad) a través del método load. *Properties* se puede cargar a partir de un fichero de texto, pero también a partir de un xml.

Finalmente, para la creación del *DataSource* se utiliza la clase *BasicDataSourceFactory*, como sigue:

DataSource ds = BasicDataSourceFactory.createDataSource(propiedades);

Este sería el código completo

```
public class _03ConexionFicheroPropiedades {
   public static void main(String[] args) {
      Connection con = null;
      try{
         Properties propiedades = new Properties();
         propiedades.load(new FileInputStream("configuracion\\PropiedadesCiclismo.txt"));

         DataSource ds = BasicDataSourceFactory.createDataSource(propiedades);
         con = ds.getConnection();
         System.out.println("Conexion realizada");
```

```
        } catch(FileNotFoundException e){
            e.printStackTrace();
        } catch(IOException e){
            e.printStackTrace();
        } catch (Exception e){
            e.printStackTrace();
        }finally{
            //Cerramos la conexión
            try{
                if(con!=null && !con.isClosed())
                    con.close();
                    System.out.println("Desconectado");
            } catch(SQLException e){
                e.printStackTrace();
            }
        }
    }
```

Ejercicio de investigación: Realiza un programa que conecte con la base de datos tomando los datos de la conexión de un fichero xml.

6 El interfaz Statement

En el punto anterior veíamos como establecer una conexión con una base de datos. En el actual vamos a dar un paso más en el acceso a bases de datos a través de JDBC. Una vez establecida la conexión podremos enviar sentencias SQL contra la base de datos a la que nos hemos conectado, para ello contamos con el interfaz Statement.

Un objeto Statement es utilizado para enviar sentencias SQL a una base de datos. Existen tres tipos de objetos que representan una sentencia SQL, cada uno de ellos actúa como un contenedor para ejecutar un tipo determinado de sentencias SQL sobre una conexión a una base de datos.

Un objeto Statement se crea con el método createStatement() de la clase Connection, como se puede observar en el siguiente fragmento de código:

```
stmt = conexion.createStatement();
```

El objeto Statement proporciona básicamente 3 métodos, execute(), executeUpdate() y executeQuery(), que actúan cómo conductores de información con la base de datos. La diferencia entre cada uno de estos métodos la mostramos en la siguiente tabla:

Método	Uso recomendado
executeQuery()	Se utiliza con sentencias SELECT y devuelve un ResultSet
executeUpdate()	Se utiliza con sentencias INSERT, UPDATE y DELETE, o bien, con sentencias DDL SQL.
execute()	Utilizado para cualquier sentencia DDL, DML o comando especifico de la base de datos

En el siguiente fragmento de código se muestra como crear una tabla y añadir registros utilizando executeUpdate().

```
public class _04ExecuteUpdate {
   public static void main(String[] args) {
      Connection con = null;
      Statement stmt = null;
      try{
         Properties propiedades = new Properties();
         propiedades.load(new FileInputStream("configuracion\\PropiedadesCiclismo.txt"));

         DataSource ds = BasicDataSourceFactory.createDataSource(propiedades);
         con = ds.getConnection();
         System.out.println("Conexion realizada");

         //Creamos una tabla
         stmt = con.createStatement();
         String sql = "CREATE TABLE representante (" +
                           "id int(10) unsigned NOT NULL auto_increment," +
                           "nombre varchar(50) NOT NULL default ''," +
                           "fecha_ingreso date NOT NULL default '0000-00-00'," +
                           "salario float NOT NULL default '0'," +
                     "PRIMARY KEY (id))";
         stmt.executeUpdate(sql);
         System.out.println("Tabla creada");

         //Añadimos datos a la tabla representantes
         sql = "INSERT INTO representante VALUES (1,'pepe', '2005-01-01', 12000)";
         stmt.executeUpdate(sql);
         System.out.println("Inserciones realizadas");

      } catch(FileNotFoundException e){
         e.printStackTrace();
      } catch(IOException e){
         e.printStackTrace();
      } catch (Exception e){
         e.printStackTrace();
      }finally{
         //Cerramos el statement
         try{
            if(stmt !=null && !stmt.isClosed())
               stmt.close();
               System.out.println("Desconectado");
         } catch(SQLException e){
            e.printStackTrace();
         }
         //Cerramos la conexión
         try{
            if(con!=null && !con.isClosed())
               con.close();
               System.out.println("Desconectado");
         } catch(SQLException e){
            e.printStackTrace();
         }
      }
   }
}
```

Éste otro realiza una consulta sobre la tabla equipo de la base de datos ciclismo para determinar cuántos equipos hay.

```
public class _05QueryNumeroEquipos {
   public static void main(String[] args) {
      Connection con = null;
      Statement stmt = null;
      ResultSet rs = null;
      try{
         Properties propiedades = new Properties();
         propiedades.load(new FileInputStream("configuracion\\PropiedadesCiclismo.txt"));

         DataSource ds = BasicDataSourceFactory.createDataSource(propiedades);
```

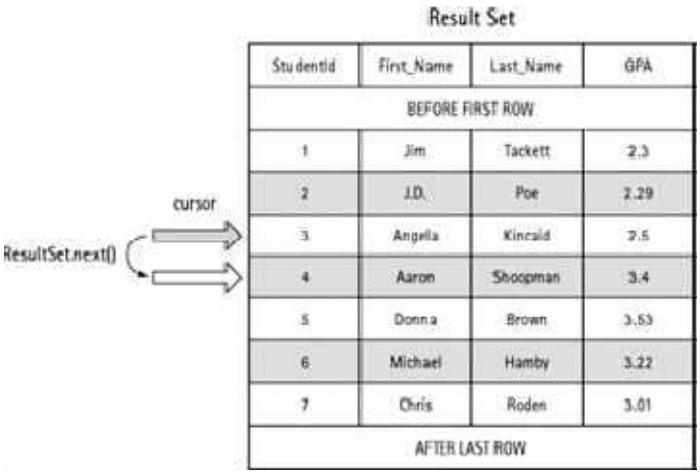
```
            con = ds.getConnection();
            System.out.println("Conexion realizada");

            // Realizamos la consulta
            stmt = con.createStatement();
            String sql = "SELECT count(*) FROM equipo";
            rs = stmt.executeQuery(sql);
            rs.first();
            System.out.println(rs.getInt(1));

        } catch(FileNotFoundException e){
            e.printStackTrace();
        } catch(IOException e){
            e.printStackTrace();
        } catch (Exception e){
            e.printStackTrace();
        }finally{
            //Cerramos el statement
            try{
                if(stmt !=null && !stmt.isClosed())
                    stmt.close();
                    System.out.println("Desconectado");
            } catch(SQLException e){
                e.printStackTrace();
            }
            //Cerramos el ResultSet
            try{
                if (rs!=null && !rs.isClosed()){
                    rs.close();
                    System.out.println("ResultSet cerrado");
                }
            }catch (SQLException e){
                e.printStackTrace();
            }
            //Cerramos la conexión
            try{
                if(con!=null && !con.isClosed())
                    con.close();
                    System.out.println("Desconectado");
            } catch(SQLException e){
                e.printStackTrace();
            }
        }
    }
}
```

7 ResultSet

El resultado de la ejecución del método executeQuery() es un objeto de tipo ResultSet.

Tras la consulta, el objeto ResultSet contiene las filas resultado de la ejecución de la sentencia SQL. Éste se puede recorrer para recuperar los datos que contiene, utilizando una serie de métodos que proporciona la clase.

El aspecto que suele tener un ResultSet es el de una tabla con cabeceras de columnas y los valores correspondientes devueltos por una consulta en las filas. Un ResultSet mantiene un cursor que apunta a la fila actual de datos. El cursor se mueve hacia abajo cada vez que se llama al método **next()**. Inicialmente está posicionado antes de la primera fila, de esta forma, la primera llamada a next() situará el cursor en la primera fila, pasando a ser la fila actual.



Cada llamada sucesiva al método next() avanza el cursor a la fila siguiente, hasta que éste se sitúe tras la última fila.

Además de este método de desplazamiento básico, según el tipo de ResultSet podremos realizar desplazamientos libres utilizando métodos como last(), relative() o previous(), ...

Para recuperar la información que contiene una fila se utiliza una serie de métodos **getXXX()** que devuelven información de las columnas de la fila actual.

Las filas del ResultSet son devueltas de arriba a abajo según se va desplazando el cursor con las sucesivas llamadas al método next(). Un cursor es válido hasta que el objeto ResultSet o su objeto padre Statement se cierra.

En el siguiente ejemplo se ilustra la manera de recorrer un ResultSet y recuperar información para mostrar los nombres de todos los equipos de la tabla equipos:

```
public class _06RecorridoBasicoResultSet {
	. . .
	. . .
	try{
	   Properties propiedades = new Properties();
	   propiedades.load(new FileInputStream("configuracion\\PropiedadesCiclismo.txt"));

	   DataSource ds = BasicDataSourceFactory.createDataSource(propiedades);
	   con = ds.getConnection();

	   //Realizamos la consulta
	   stmt = con.createStatement();
	   String sql = "SELECT * FROM equipo";
	   rs = stmt.executeQuery(sql);

	   //Recorremos el ResultSet
	   while (rs.next()){
	      System.out.println(rs.getString(1));
	   }

	} catch(FileNotFoundException e){

	. . .
	. . .
```

7.1 Necesidad de liberar los recursos.

Las instancias de Connection y las de Statement almacenan en memoria, mucha información relacionada con las ejecuciones realizadas. Además, mientras quedan activas mantienen en el SGBD un conjunto importante de recursos abiertos, destinados a servir de forma eficiente las peticiones de los clientes. El cierre de estos objetos permite liberar recursos tanto del cliente como del servidor.

En realidad , los Statements son objetos vinculados íntimamente al objeto Connection que les ha instanciado y si no se cierran específicamente , permanecerán activos mientras la conexión continúe activa, incluso más allá de haber desaparecido la variable que los referenciaba . Es más, la complejidad de estos objetos hace que aunque se haya cerrado la conexión, los objetos Statements que no se habían cerrado expresamente permanezcan más tiempo en memoria que los objetos cerrados previamente, ya que el garbage collector de Java tendrá que hacer más comprobaciones para asegurar que ya no dispone de dependencias ni internas ni externas y se puede eliminar. Es por ello que se recomienda proceder siempre a cerrarlo manualmente utilizando la operación close.

El cierre de los objetos Statement asegura la liberación inmediata de los recursos y la anulación de las dependencias . Si en un mismo método debemos cerrar un objeto Statement y la conexión que la ha instanciado , habrá que cerrar en primer lugar el Statement y después la instancia Connection. De hacerlo al revés, cuando intentáramos cerrar el Statement nos saltaría una excepción de tipo SQLException, ya que el cierre de la conexión la habría dejado inaccesible. Además de respetar el orden, asegurar la liberación de los recursos situando las operaciones de cierre dentro de un bloque finally. De esta manera, a pesar de que se produzcan errores, no se dejarán de ejecutar las instrucciones de cierre. Hay que tener en cuenta todavía un detalle más cuando sea necesario realizar el cierre de varios objetos a la vez. En este caso, a pesar de que las situásemos una tras otra, todas las instrucciones de cierre dentro del bloque finally, no sería suficiente garantía para asegurar la ejecución de todos los cierres , ya que , si mientras se produce el cierre de uno de los objetos se lanza una excepción, los objetos invocados en una posición posterior a la del que se ha producido el error no se cerrarán. La solución de este problema pasa por evitar el lanzamiento de cualquier excepción durante el proceso de cierre. Una posible forma es capsular cada cierre en un try-catch independiente, tal y como hemos visto en el ejemplo anterior.

7.2 Tipos de ResultSet

En método createStatement() del interfaz Connection, se encuentra sobrecargado, si utilizamos su versión sin parámetros, a la hora de ejecutar sentencias SQL sobre el objeto Statement que se ha creado, se obtendrá el tipo de objeto ResultSet por defecto, es decir, se obtendría un tipo de cursor de sólo lectura y con movimiento únicamente hacia adelante. Pero la otra versión que ofrece el interfaz Connection del método createStatement() ofrece dos parámetros que nos permiten definir el tipo de objeto ResultSet que se va a devolver como resultado de la ejecución de una sentencia SQL, como se muestra en el fragmento de código siguiente:

```
//Uso del métodoexecuteQuery
stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);
String sql = "select * from equipo";
rs = stmt.executeQuery(sql);
```

El primero de los parámetros indica el tipo de objeto ResultSet que se va a crear, y el segundo de ellos indica si el ResultSet es sólo de escritura o si permite modificaciones, este parámetro también se denomina tipo de concurrencia. Si especificamos el tipo de objeto Resultset es obligatorio indicar si va a ser de sólo lectura o no. Si no indicamos ningún parámetro en el método createStatement(), se creará un objeto ResultSet con los valores por defecto.

Los tipos de ResultSet distintos que se pueden crear dependen del valor del primer parámetro, estos valores se corresponden con constantes definidas en el interfaz ResultSet. Estas constantes se describen a continuación:

- **TYPE_FORWARD_ONLY**: se crea un objeto ResultSet con movimiento únicamente hacia delante (forward-only). Es el tipo de ResultSet por defecto.
- **TYPE_SCROLL_INSENSITIVE**: se crea un objeto ResultSet que permite todo tipo de movimientos. Pero este tipo de ResultSet, mientras está abierto, no será consciente de los cambios que se realicen sobre los datos que está mostrando, y por lo tanto no mostrará estas modificaciones.
- **TYPE_SCROLL_SENSITIVE**: al igual que el anterior permite todo tipo de movimientos, y además permite ver los cambios que se realizan sobre los datos que contiene.

Los valores que puede tener el segundo parámetro que define la creación de un objeto ResultSet, son también constantes definidas en el interfaz ResultSet y son las siguientes:

- **CONCUR_READ_ONLY**: indica que el ResultSet es sólo de lectura. Es el valor por defecto.
- **CONCUR_UPDATABLE**: permite realizar modificaciones sobre los datos que contiene el ResultSet.

7.3 Obteniendo datos

Los métodos getXXX() (getInt(), getString(), …) ofrecen los medios para recuperar los valores de las columnas (campos) de la fila (registro) actual del ResultSet. Dentro de cada columna no se necesita que las columnas sean recuperadas utilizando un orden determinado.

Para designar una columna podemos utilizar su nombre o bien su número de orden. Por ejemplo si la segunda columna de un objeto **rs** de la clase ResultSet se llama *título* y almacena datos de tipo String, se podrá recuperar su valor de las formas que muestra el siguiente fragmento de código:

```
String valor = rs.getString(2);
String valor = rs.getString("titulo");
```

Se debe señalar que las columnas se numeran de izquierda a derecha empezando con la columna 1, y que los nombres de las columnas no son case sensitive, es decir, no distinguen entre mayúsculas y minúsculas.

La información referente a las columnas que contiene el ResultSet se encuentra disponible llamando al método getMetaData(), este método devolverá un objeto ResultSetMetaData que contendrá el número, tipo y propiedades de las columnas del ResultSet, mas adelante veremos en detalle la interfaz ResultSetMetaData.

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.

7.4 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 a un objeto String de Java, por lo tanto el valor de retorno de getString() será un objeto de la clase String.

Esta conversión de tipos se puede realizar gracias a la clase java.sql.Types. En esta clase se definen lo que se denominan tipos de datos JDBC, que se corresponde con los tipos de datos SQL estándar. Esto nos permite abstraernos del tipo SQL específico de la base de datos con la que estamos trabajando, ya que los tipos JDBC son tipos de datos SQL genéricos.

Normalmente estos tipos genéricos nos servirán para todas nuestras aplicaciones JDBC. La clase Types está definida como un conjunto de constantes (final static), estas constantes se utilizan para identificar los tipos SQL. Si el tipo del dato SQL que tenemos en nuestra base de datos es específico de esa base de datos, y no se encuentra entre las constantes definidas por la clase Types, se utilizará el tipo Types.OTHER.

La correspondencia de tipos entre la base de datos y Java se muestra en la siguiente tabla:

Tipo SQL	Tipo Java
BIT(1)	java.lang.Boolean
BIT(> 1)	byte[]
TINYINT	java.lang.Boolean if the configuration property

	"tinyInt1isBit" is set to "true" (the default) and the storage size is "1", or java.lang.Integer if not.
BOOL	BOOLEAN
SMALLINT[(M)] [UNSIGNED]	java.lang.Integer
MEDIUMINT[(M)] [UNSIGNED]	java.lang.Integer
INT,INTEGER[(M)] [UNSIGNED]	java.lang.Integer , if UNSIGNED java.lang.Long
BIGINT[(M)] [UNSIGNED]	java.lang.Long , if UNSIGNED java.math.BigInteger
FLOAT[(M,D)]	java.lang.Float
DOUBLE[(M,B)]	java.lang.Double
DECIMAL[(M[,D])	java.math.BigDecimal
DATE	java.sql.Date (**No confundir con java.Util.Date**)
DATETIME	java.sql.Timestamp
TIMESTAMP[(M)]	java.sql.Timestamp
TIME	java.sql.Time
YEAR[(2|4)]	java.sql.Date
CHAR(M)	java.lang.String (unless the character set for the column is BINARY , then byte[] is returned.
VARCHAR(M) [BINARY]	java.lang.String (unless the character set for the column is BINARY , then byte[] is returned.
BINARY(M)	byte[]
VARBINARY(M)	byte[]
TINYBLOB	byte[]
TINYTEXT	java.lang.String
BLOB	byte[]
TEXT	java.lang.String
MEDIUMBLOB	byte[]
MEDIUMTEXT	java.lang.String

LONGBLOB	byte[]
LONGTEXT	java.lang.String
ENUM('value1','value2',...)	java.lang.String
SET('value1','value2',...)	java.lang.String

A continuación se muestra una tabla resumen con los métodos getXXX() y su correspondiente valor devuelto

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR
getByte	X	x	x	x	x	x	x	x	x	x	x
getShort	x	X	x	x	x	x	x	x	x	x	x
getInt	x	x	X	x	x	x	x	x	x	x	x
getLong	x	x	x	X	x	x	x	x	x	x	x
getFloat	x	x	x	x	X	x	x	x	x	x	x
getDouble	x	x	x	x	x	X	X	x	x	x	x
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x
getBoolean	x	x	x	x	x	x	x	x	x	X	x
getString	x	x	x	x	x	x	x	x	x	x	X
getBytes											
getDate											x
getTime											x
getTimestamp											x
getAsciiStream											x
getUnicodeStream											x
getBinaryStream											
getObject	x	x	x	x	x	x	x	x	x	x	x

	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	VARCHAR
getByte	x	x						
getShort	x	x						
getInt	x	x						
getLong	x	x						
getFloat	x	x						
getDouble	x	x						
getBigDecimal	x	x						
getBoolean	x	x						
getString	X	x	x	x	x	x	x	x
getBytes			X	X	x			
getDate	x	x				X		x
getTime	x	x					X	x
getTimestamp	x	x				x	x	X
getAsciiStream	x	X	x	x	x			
getUnicodeStream	x	X	x	x	x			
getBinaryStream			x	x	X			
getObject	x	x	x	x	x	x	x	x

Donde una x se indica la posibilidad de hacerlo y con X se indica que es la opción más recomendable.

7.5 Desplazamientos

En este apartado vamos a comentar las distintas formas que tenemos dedesplazarnos dentro de un objeto ResultSet, dependiendo del tipo de ResultSet.También veremos en este apartado como mostrar los contenidos de cada registro (fila) de un objeto ResultSet utilizando los métodos getXXX().

Como ya se había mencionado anteriormente, el método next() del interfaz ResultSet lo vamos a utilizar para desplazarnos al registro siguiente dentro de un ResultSet. El método next() devuelve un valor booleano (tipo boolean de Java), true si el registro siguiente existe y false si hemos llegado al final del objeto ResultSet, es decir, no hay más registros. Este era el único método que ofrecía la interfaz ResultSet en la versión 1.0 de JDBC, pero en JDBC 2.0, además de tener el método next(), disponemos de los siguientes métodos para el desplazamiento y movimiento dentro de un objeto ResultSet:

- **boolean absolute(int registro)**: desplaza el cursor al número de registros indicado. Si el valor es negativo, se posiciona en el número registro indicado pero empezando por el final. Este método devolverá false si nos hemos desplazado después del último registro o antes del primer registro del objeto ResultSet. Para poder utilizar este método el objeto ResultSet debe ser de tipo TYPE_SCROLL_SENSITIVE o de tipo TYPE_SCROLL_INSENSITIVE, a un ResultSet que es de cualquiera de estos dos tipos se dice que es de tipo scrollable. Si a este método le pasamos un valor cero se lanzará una excepción SQLException.

- **void afterLast()**: se desplaza al final del objeto ResultSet, después del último registro. Si el ResultSet no posee registros este método no tienen ningún efecto. Este método sólo se puede utilizar en objetos ResultSet de tipo scrollable, como sucede con muchos de los métodos comentados en estos puntos.

- **void beforeFirst()**: mueve el cursor al comienzo del objeto ResultSet, antes del primer registro. Sólo se puede utilizar sobre objetos ResultSet de tipo scrollable.

- **boolean first()**: desplaza el cursos al primer registro. Devuelve true si el cursor se ha desplazado a un registro válido, por el contrario, devolverá false en otro caso o bien si el objeto ResultSet no contiene registros. Al igual que los métodos anteriores, sólo se puede utilizar en objetos ResultSet de tipo scrollable.

- **boolean last()**: desplaza el cursor al último registro del objeto ResultSet. Devolverá true si el cursor se encuentra en un registro válido, y false en otro caso o si el objeto ResultSet no tiene registros. Sólo es válido para objetos ResultSet de tipo scrollable, en caso contrario lanzará una excepción SQLException.

- **void moveToCurrentRow()**: Tras una inserción, mueve el cursor a la posición en que se encontraba antes de la inserción. Este método sólo tiene sentido cuando estamos situados dentro del ResultSet en un registro que se ha insertado. Este método sólo es válido utilizarlo con objetos ResultSet que permiten la modificación, es decir, están definidos mediante la constante CONCUR_UPDATABLE.

- **boolean previous()**: desplaza el cursor al registro anterior. Es el método contrario al método next(). Devolverá true si el cursor se encuentra en un registro o fila válidos, y false en caso contrario. Sólo es válido este método con objetos ResultSet de tipo scrollable, en caso contrario lanzará una excepción SQLException.

- **boolean relative(int registros)**: mueve el cursor un número relativo de registros, este número puede ser positivo o negativo. Si el número es negativo el cursor se desplazará hacia el principio del objeto ResultSet el número de registros indicados, y si es positivo se desplazará hacia el final de objeto ResultSet correspondiente. Este método sólo se puede utilizar si el ResultSet es de tipo scrollable.

También existen otros métodos dentro del interfaz ResultSet que están relacionados con el desplazamiento:

- **boolean isAfterLast()**: indica si nos encontramos después del último registro del objeto ResultSet. Sólo se puede utilizar en objetos ResultSet de tipo scrollable.

- **boolean isBeforeFirst()**: indica si nos encontramos antes del primer registro del objeto ResultSet. Sólo se puede utilizar en objetos ResultSet de tipo scrollable.

- **boolean isFirst()**: indica si el cursor se encuentra en el primer registro. Sólo se puede utilizar en objetos ResultSet de tipo scrollable.

- **boolean isLast()**: indica si nos encontramos en el último registro del ResultSet. Sólo se puede utilizar en objetos ResultSet de tipo scrollable.

- **int getRow()**: devuelve el número de registro actual. El primer registro será el número 1, el segundo el 2, etc. Devolverá cero si no hay registro actual.

7.6 Métodos UpdateXXX()

Para poder modificar los datos que contiene un ResultSet debemos crear un ResultSet de tipo modificable, para ello debemos utilizar la constante ResultSet.CONCUR_UPDATABLE dentro del método createStatement().

Aunque un ResultSet que permite modificaciones suele permitir distintos desplazamientos, es decir, se suele utilizar la constante ResultSet.TYPE_SCROLL_INSENSITIVE o ResultSet.TYPE_SCROLL_SENSITIVE, pero no es del todo necesario ya que también puede ser del tipo "sólo hacia delante" (forward-only).

Para modificar los valores de un registro existente se utilizan una serie de métodos updateXXX() del interfaz ResultSet. Las XXX indican el tipo del dato al igual que ocurre con los métodos getXXX() de este mismo interfaz. El proceso para realizar la modificación de una fila de un ResultSet es el siguiente: nos situamos sobre el registro que queremos modificar y lanzamos los métodos updateXXX() adecuados, pasándole como argumento los nuevos valores. A continuación lanzamos el método update Row() para que los cambios tengan efecto sobre la base de datos.

El método updateXXX() recibe dos parámetros, el campo o columna a modificar y el nuevo valor. La columna la podemos indicar por su número de orden o bien por su nombre, igual que en los métodos getXXX().

El siguiente fragmento de código muestra como se puede modificar el campo dirección del último registro de un ResultSet que contiene el resultado de una SELECT sobre la tabla de clientes:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
//Ejecutamos la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
System.out.println("Situamos el cursor al final");
//Nos situamos en el último registro del ResultSet y hacemos la modificación
rs.last();
rs.updateString("direccion", "C/ PepeCiges, 3");
rs.updateRow();
```

Si nos desplazamos dentro del ResultSet antes de lanzar el método updateRow(), se perderán las modificaciones realizadas. Y si queremos cancelar las modificaciones lanzaremos el método cancelRowUpdates() sobre el objeto ResultSet, en lugar del método updateRow().

Una vez que hemos invocado el método updateRow(), el método cancelRowUpdates() no tendrá ningún efecto. El método cancelRowUpdates() cancela las modificaciones de todos los campos de un registro, es decir, si hemos modificado dos campos con el método updateXXX() se cancelarán ambas modificaciones.

7.7 Métodos insertRow y deleteRow

Además de poder realizar modificaciones directamente sobre las filas de un ResultSet, también podemos añadir nuevas filas (registros) y eliminar las existentes. Estos métodos son: moveToInsertRow() y deleteRow().

El primer paso para insertar un registro o fila en un ResultSet es mover el cursor (puntero que indica el registro actual) del ResultSet, esto se consigue mediante el método moveToInsertRow().

El siguiente paso es dar un valor a cada uno de los campos que van a formar parte del nuevo registro, para ello se utilizan los métodos updateXXX() adecuados. Para finalizar el proceso se lanza el método insertRow(), que creará el nuevo registro tanto en el ResultSet como en la tabla de la base de datos correspondientes. Hasta que no se lanza el método insertRow(), la fila no se incluye dentro del ResultSet, es una fila especial denominada "fila de inserción" (insertrow) y es similar a un buffer completamente independiente del objeto ResultSet.

El siguiente fragmento de código da de alta un nuevo registro en la tabla de clientes:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
//Ejecutamos la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
//Creamos un nuevo registro en la tabla de clientes
rs.moveToInsertRow();
rs.updateString(2,"Killy Lopez");
rs.updateString(3,"Wall Street 3674");
rs.insertRow();
```

Si no facilitamos valores a todos los campos del nuevo registro con los métodos updateXXX(), ese campo tendrá un valor NULL, y si en la base de datos no está definido ese campo para admitir nulos se producirá una excepción SQLException.

Cuando hemos insertado nuestro nuevo registro en el objeto ResultSet, podremos volver a la antigua posición en la que nos encontrábamos dentro del ResultSet, antes de haber lanzado el método moveToInsertRow(), llamando al método moveToCurrentRow(), este método sólo se puede utilizar en combinación con el método moveToInsertRow().

Además de insertar filas en nuestro objeto ResultSet también podremos eliminar filas o registros del mismo. El método que se debe utilizar para esta tarea es el método deleteRow(). Para eliminar un registro no tenemos que hacer nada más que movernos a ese registro, y lanzar el método deleteRow() sobre el objeto ResultSet correspondiente.

El siguiente fragmento de código borra el último registro de la tabla de clientes:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
//Ejecutamos la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
//Nos situamos al final del ResultSet
rs.last();
rs.deleteRow();
```

8 Objetos PreparedStatement

Este interfaz, al igual que el interfaz Statement, nos permite ejecutar sentencias SQL sobre una conexión establecida con una base de datos. Pero en este caso vamos a ejecutar sentencias SQL más especializadas, estas sentencias SQL se van a denominar sentencias SQL precompiladas y van a recibir parámetros de entrada.

El interfaz PreparedStatement hereda del interfaz Statement y se diferencia de él de dos maneras:

- Las instancias de PreparedStatement contienen sentencias SQL que ya han sido compiladas. Esto es lo que hace a una sentencia "prepared" (preparada).
- La sentencia SQL que contiene un objeto PreparedStatement puede contener uno o más parámetros de entrada. Un parámetro de entrada es aquél cuyo valor no se especifica cuando la sentencia es creada, en su lugar la sentencia va a tener un signo de interrogación (?) por cada parámetro de entrada.

Antes de ejecutarse la sentencia se debe especificar un valor para cada uno de los parámetros a través de los métodos setXXX() apropiados. Estos métodos setXXX() los añade el interfaz PreparedStatement.

Debido a que las sentencias de los objetos PrepareStatement están precompiladas su ejecución será más rápida que la de los objetos Statement. Por lo tanto, una sentencia SQL que va a ser ejecutada varias veces se suele crear como un objeto PreparedStatement para ganar en eficiencia. También se utilizará este tipo de sentencias para pasarle parámetros de entrada a las sentencias SQL.

Al heredar del interfaz Statement, el interfaz PreparedStatement hereda todas funcionalidades de Statement. Además, añade una serie de métodos que permiten asignar un valor a cada uno de los parámetros de entrada de este tipo de sentencias.

Los métodos execute(), executeQuery() y executeUpdate() son sobrecargados y en esta versión, los objetos PreparedStatement no toman ningún tipo de argumentos, de esta forma, a estos métodos nunca se les deberá pasar por parámetro el objeto String que representaba la sentencia SQL a ejecutar. En este caso, un objeto PreparedStatement ya es una sentencia SQL por sí misma, a diferencia de lo que ocurría con las sentencias Statement que poseían un significado sólo en el momento en el que se ejecutaban.

Para crear un objeto PreparedStatement se debe lanzar el método prepareStatement() del interfaz Connection sobre el objeto que representa la conexión establecida con la base de datos.

En el siguiente fragmento de código se puede ver como se crearía un objeto PreparedStatement que representa una sentencia SQL con dos parámetros de entrada. El objeto pstmt contendrá la sentencia SQL indicada, la cual, ya ha sido enviada al DBMS y ya ha sido preparada para su ejecución. En este caso se ha creado una sentencia SQL con dos parámetros de entrada.

```
conn = DriverManager.getConnection(jdbcUrl,"root","");
pstmt = conn.prepareStatement("update facturas set serie=? where id=?");
```

Los interrogantes indican los parámetros que deben de asignarse a esta sentencia SQL.

8.1 Utilización de parámetros de entrada

Antes de poder ejecutar un objeto PreparedStatement se debe asignar un valor para cada uno de sus parámetros. Esto se realiza mediante la llamada a un método setXXX(), donde XXX es el tipo apropiado para el parámetro. Por ejemplo, si el parámetro es de tipo long, el método a utilizar será setLong().

El primer argumento de los métodos setXXX() es la posición ordinal del parámetro al que se le va a asignar valor, y el segundo argumento es el valor a asignar. Por ejemplo en el siguiente fragmento de código crea un objeto PreparedStatement y acto seguido le asigna al primero de sus parámetros un valor de tipo String y al segundo un valor de tipo long.

```
String jdbcUrl = "jdbc:mysql://localhost:3306/empresa";
conn = DriverManager.getConnection(jdbcUrl,"root","");
pstmt = conn.prepareStatement("update facturas set serie=? where id=?");
pstmt.setString(1, "B");
pstmt.setLong(2, 1);
pstmt.executeUpdate();
```

Una vez que se ha asignado unos valores a los parámetros de entrada de una sentencia, el objeto PreparedStatement se puede ejecutar múltiples veces, hasta que sean borrados los parámetros con el método clearParameters(), aunque no es necesario llamar a este método cuando se quieran modificar los parámetros de entrada, sino que al lanzar los nuevos métodos setXXX() los valores de los parámetros serán reemplazados. Para finalizar se llama al método executeUpdate() del objeto PreparedStatement y de esta forma se verán reflejados los cambios en la base de datos.

9 Transacciones

En algunos momentos podemos necesitar que varias sentencias SQL se ejecuten como un todo, y si falla alguna de ellas se deshagan los cambios realizados por las sentencias y se vuelva a la situación anterior. Así por ejemplo, al realizar una transferencia de una cuenta de un banco a otra cuenta, las dos sentencias encargadas de realizar la actualización en el saldo de cada cuenta se deben ejecutar, si una de ellas falla se deben deshacer los cambios realizados. Estas dos sentencias se deben ejecutar en una misma transacción.

Las sentencias que se ejecutan dentro de una misma transacción, para que la transacción se lleve a cabo, todas las sentencias se deben ejecutar con éxito, si falla alguna se deshacen los cambios que se hayan podido realizar para evitar cualquier tipo de inconsistencia en la base de datos, las transacciones permiten preservar la integridad de los datos. Se puede decir que dentro de una transacción se da un "todo o nada".

Cuando establecemos una conexión a una base de datos con JDBC, por defecto se crea en modo autocommit. Esto quiere decir que cada sentencia SQL modifica la base de datos nada más ejecutarse, es decir, se trata como una transacción que contiene una única sentencia.

Si queremos agrupar varias sentencias SQL en una misma transacción utilizaremos el método setAutoCommit() del interfaz Connection, pasándole como parámetro el valor false, para indicar que una sentencia no se ejecute automáticamente.

Una vez que se ha deshabilitado el modo auto-commit, las sentencias que ejecutemos no modificarán la base de datos hasta que no llamemos al método commit() del interfaz Connection. Todas las sentencias que se han ejecutado previamente a la llamada del método commit() se incluirán dentro de una misma transacción y se llevarán a cabo sobre la base de datos como un todo.

Si retomamos el ejemplo de la transferencia entre dos cuentas y si suponemos que el número de cuenta de la cuenta de origen de la transferencia es el 1, y el de destino el 2 el siguiente código efectuaría estas dos modificaciones dentro de una transacción:

```
int transfe=5000;
int cuentaOrigen=1;
int cuentaDestino=2;
conexion.setAutoCommit(false);
String sql = "UPDATE Cuentas SET cantidad = ? WHERE NCuenta= ?”
PreparedStatement sentenciaResta=conexion.prepare Statement();

sentenciaResta.executeUpdate(sql);
Statements entenciaSuma=conexion.createStatement();
sentenciaSuma.executeUpdate("UPDATE Cuentas SET
cantidad=cantidad+"+transfe+" WHERE NCuenta="+cuentaDestino);
conexion.commit();
conexion.setAutoCommit(true);
}catch(SQLException ex){
        conexion.rollback();
        System.out.println("La transacción a fallado.");
        conexion.setAutoCommit(true);
`
```

En este caso si todo va bien (no se produce ninguna excepción) la transferencia se realiza cuando se llama al método commit() del interfaz Connection. En caso de producirse algún error, se aborta la transacción mediante el método rollback() del interfaz Connection.

10 Manejo de errores en JDBC

Una buena gestión de errores es una de las tareas más importantes cuando se está desarrollando una aplicación. Se debe de tener en cuenta cada uno de los posibles errores que puede causar el sistema para dar la máxima seguridad de los datos con los que la aplicación está trabajando.

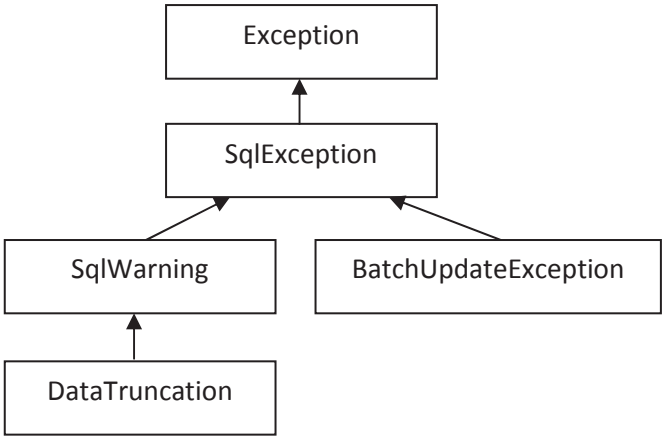
Cuando se está trabajando con JDBC se puede dar el caso de que existan errores, también llamados en el lenguaje Java Exceptions. Interactuar con la Base de Datos es un proceso complejo en el que intervienen muchos componentes y todos ellos deben garantizar que el sistema funcione a la perfección. Cuando algún componente de estos falla dará una Exception que debemos de capturarla en nuestra aplicación para tratar de resolver el problema.

Aquí mostramos una lista pequeña de algunas áreas que pueden ocasionar problemas:

- La Conectividad en la red
- Derechos de acceso y autenticación de usuarios
- Sentencias SQL inválidas
- Errores debidos a la carga de los drivers
- Conversiones entre tipos inválidas
- Etc..

Todos estos problemas y muchos más pueden ser susceptibles de lanzar una excepción, además casi todos los métodos JDBC lanzan una SQLException por si ocurre algún error. En otras palabras que toda sentencia en JDBC debe de ir acompañado de los bloques try-catch o propagar la excepción al método que lo ha invocado.

Cuando trabajamos con JDBC podemos encontrar 4 tipos de excepciones SQLException, SQLWarning, BatchUpdateExceptionyDataTruncation. La clase SQLException hereda de Exception y por tanto tiene toda la funcionalidad para manejar excepciones Java. La clase SQLWarning y BatchUpdateException heredan de SQLException. La clase DataTruncation hereda de SQLWarning. La siguiente figura muestra la relación entre cada una de estas clases.



10.1 SQLException

Casi todos los métodos del API JDBC lanzan una SQLException,normalmente cuando estamos programando el acceso a través de JDBC,encuadramos el código JDBC en bloques try-catch como mostramos en elsiguiente fragmento de código:

```
try{
        //Proceso JDBC....
}catch(SQLException se){
        //Excepción producida por JDBC
        se.printStackTrace();
}catch(Exception e){
        //Otra excepcion
        e.printStackTrace();
}
```

La clase SQLException especializa a la clase Exception, **añadiendo mucha más información acerca posibles errores**. La tabla que mostramos a continuación muestra la información más útil que podemos consultar cuando ha ocurrido una SQLException:

Información	Explicación
Error message	Cadena que contiene una breve descripción del error que se ha producido

Error code	Codigo especifico del vendedor. Depende de la base de datos que haya por debajo.
SQLState	Codigo de error estandarizado, basado en la especificación X/Open SQL
Cause	Una SQLException puede estar relacionada con una o varias *causas* que no son más que uno o más objetos Throwable que provocaron el lanzamiento de la SQLException. Con getCause() se pueden recuperar estos objetos de forma recursiva, hasta que getCause() devuelva null

El siguiente fragmento de código muestra cómo podemos obtener información cuando se produce una SQLException. En este caso se está intentando consultar los valores de una tabla que no existe en la base de datos.

```
public static void main(String[] args) {
        …
        …
        try{
                …
                …
                rs = stmt.executeQuery("select * from no_table_exisits");
        }catch(SQLException se){
                System.out.println("Code: " + se.getErrorCode());
                System.out.println("SqlState: " + se.getSQLState());
                System.out.println("Error Message: " + se.getMessage());
                se.printStackTrace();

                Throwable t = ex.getCause();
                while(t != null) {
                        System.out.println("Cause: " + t);
                        t = t.getCause();
                }
        }catch(Exception e){
                e.printStackTrace();
        } //end try
                …
        }
```

Puede ser conveniente realizar un procesamiento de las excepciones en función de su código de error.

10.2 SQLWarning

La clase SQLWarnings una subclase de SQLException. Que se produzca un SQLWarning no produce la parada del programa. Informan de posibles anomalías. Podría producirse un warning si, por ejemplo, tratamos de modificar los permisos de un usuario y no se consigue.

Los principales objetos que lanzan SQLWarning son Connection, Statement y ResultSet. Estas clases tienen un método getWarnings() que devuelven el primero de los warnings que se han producido sobre el objeto (puede haberse producido más de uno). El resto de warning se recuperan recursivamente a partir del primero.

Ejemplo

```
Statement stmt  =con.executeUpdate(….);
SQLWarning w = stmt.getWarnings();
while (w != null) {
        System.out.println("Message: " + w.getMessage());
        System.out.println("SQLState: " + w.getSQLState());
        System.out.print("Vendor error code: ");
        System.out.println(w.getErrorCode());
        System.out.println("");
        w = w.getNextWarning();
  }
```

Por ejemplo, una posible situación que lance un evento SQLWarning puede ser que intentemos crear un objeto ResultSetupdateable pero que nuestro driver no lo permita. Esto provocará que se lance un evento SQLWarning indicando que nuestro driver no soporta ResultSetupdateables y trabajará en modo normal, pero la aplicación seguirá su curso.

10.3 DataTruncation

La excepción DataTruncation ocurre cuando intentamos escribir o recuperar algún dato de la base de datos que por alguna razón la longitud del mismo ha sido truncada. Por ejemplo, este error puede ocurrir si modificamos la propiedad Statement.setMaxFieldSize() con un valor pequeño.

Normalmente cuando se produce un DataTruncation sobre una lectura de datos sobre la base de datos normalmente se suele lanzar un SQLWarning silencioso, sin embargo si estamos realizando operaciones de escritura normalmente se produce una SQLException. De todas formas estas implementaciones dependen mucho del fabricante del driver, así que lo mejor es echar mano de la documentación sobre el manejo de errores.

11 DatabaseMetadata

Esta interfaz ofrece información sobre la base de datos a la que nos hemos conectado, trata a la base de datos como un todo. A partir de esta interfaz vamos a obtener una gran cantidad de información sobre la base de datos con la que hemos establecido una conexión.

Para obtener esta información esta interfaz aporta un gran número de métodos diferentes. Muchos de estos métodos devuelven objetos ResultSet conteniendo la información correspondiente, por lo tanto, deberemos usar los métodos getXXX() para recuperar la información.

Estos métodos los podemos dividir en los que devuelven una información sencilla, tipos de Java, y los que devuelven una información más compleja, como puede ser un objeto ResultSet.

Algunas de las cabeceras de estos métodos toman como parámetro patrones de cadenas, en los que se pueden utilizar caracteres comodín. "%" identifica una cadena de 0 o más caracteres y "_" se identifica con un solo carácter, de esta forma, sólo los datos que se correspondan con el patrón dado serán devueltos por el método lanzado.

Si un driver no soporta un método de la interfaz DatabaseMetaData se lanzará una excepción SQLException, y en el caso de que el método devuelva un objeto ResultSet, se obtendrá un ResultSet vacío.

Para obtener un objeto DatabaseMetaData sobre el que lanzar los métodos que nos darán la información sobre el DBMS se debe lanzar el método getMetaData() sobre el objeto Connection que se corresponda con la conexión a la base de datos de la que queremos obtener la información, como se puede observar en el siguiente fragmento de código:

```
Connection conn = DriverManager.getConnection(jdbcUrl,"root","");
DatabaseMetadatad bmd = conn.getMetaData();
```

11.1 Obteniendo información de una Base de Datos

La interfaz DatabaseMetadata nos ofrece una información detallada sobre la base de datos. Pero lo que sí vamos a hacer es ver algunos de estos métodos a través de un ejemplo.

Vamos a realizar una aplicación Java que se conecte a una base de datos de MySQL. Una vez conectados a la base de datos vamos a obtener algunos datos de interés sobre ella. Los métodos que se utilizan de esta interfaz se han dividido en cinco métodos diferentes, atendiendo a la información que obtenemos a través de los mismos, producto, driver, funciones, tablas y procedimientos.

En el siguiente fragmento de código podemos ver alguno de estos métodos:

```
Connection con= null;
try{
	//Conexión -------------
	Properties propiedades = new Properties();
	propiedades.load(new FileInputStream("PropiedadesCiclismo.txt"));
	DataSource ds = BasicDataSourceFactory.createDataSource(propiedades);
	con = ds.getConnection();
	System.out.println("Conexion realizada");

	//DatabaseMetadata
	DatabaseMetaData dbmd = con.getMetaData();

	//Informaciondela base dedatos
	infoProducto(dbmd);

	//Informaciondel driver
	infoDriver(dbmd);

	//Informaciondelasfuncionessoportadaspor el sgbd
	infoFunciones(dbmd);

	//Informaciondelastablas, vistas, etc ...
	infoTablas(dbmd);

	//Informaciondelosproc. almacenados
	infoProcedimientos(dbmd);

} catch (Exception e){
	e.printStackTrace();
} finally {
	try{
		if(con != null) con.close();
	} catch (SQLException e){
		e.printStackTrace();
	}
}
```

El primero de los métodos es el método infoProducto(). Este método nos ofrece información general sobre el sistema gestor de base de datos: su nombre, versión, URL de JDBC, usuario conectado, características que soporta, etc. A continuación mostramos el fragmento de cómo se ha realizado el método:

```
public static void infoProducto(DatabaseMetaData dbmd) throws SQLException{
        //Nombre del SGBD
        String producto = dbmd.getDatabaseProductName();
        //Versióndel SGBD
        String version = dbmd.getDatabaseProductVersion();
        //Usuario conectado
        String nombreUsr = dbmd.getUserName();
        //Url de la base de datos
        String url = dbmd.getURL();

        System.out.println();
        System.out.println("------ INFORMACION SOBRE EL DBMS -------");
        System.out.println("Base datos: " + producto);
        System.out.println("Versión: " + version);
        System.out.println("Usuario actual " + nombreUsr);
        System.out.println("URL: " + url);
}
```

El siguiente método que se lanza es el método infoDriver(). Dentro de este método se obtienen información sobre el driver que se utiliza para realizar la conexión, la información que se obtiene es: el nombre del driver, versión, versión inferior y superior:

```
public static void infoDriver(DatabaseMetaDatadbmd) throws SQLException{
        String driver = dbmd.getDriverName();
        String driverVersion = dbmd.getDriverVersion();

        System.out.println();
        System.out.println("------ INFORMACION SOBRE EL DRIVER -------");
        System.out.println("Driver: " + driver);
        System.out.println("Version: " + driverVersion);
}
```

La información acerca de las funciones que ofrece nuestra base de datos la obtenemos a partir del método infoFunciones(). Este método nos indicará las funciones que soporta nuestro sistema gestor de base de datos en lo que respecta a cadenas de caracteres, funciones numéricas, fecha y hora y funciones del sistema.

```
public static void infoFunciones(DatabaseMetaDatadbmd) throws SQLException{
        String funcionesCadenas = dbmd.getStringFunctions();
        String funcionesSistema = dbmd.getSystemFunctions();
        String funcionesTiempo = dbmd.getTimeDateFunctions();
        String funcionesNumericas = dbmd.getNumericFunctions();

        System.out.println();
        System.out.println("------ FUNCIONES QUE SOPORTA EL DBMS -------");
        System.out.println(" Funciones de Cadenas: "+funcionesCadenas);
        System.out.println(" Funciones Numéricas: "+funcionesNumericas);
        System.out.println(" Funciones del Sistema: "+funcionesSistema);
        System.out.println(" Funciones de Fecha y Hora: "+funcionesTiempo);
}
```

En los métodos vistos hasta ahora, los métodos que se han utilizado del interfaz DatabaseMetaData devuelven valores que se corresponden con objetos String o tipos int de Java. En este nuevo método, que nos devuelve información sobre las tablas de la base de datos, la información se devuelve dentro de un objeto **ResultSet** con un formato determinado.

La llamada al método getTables() es más complicada que las llamadas al resto de los métodos del interfaz DatabaseMetaData vistos hasta ahora. Los dos primeros parámetros del método getTables() se utilizan para obtener las tablas de un catálogo, en nuestro caso no vamos a utilizar esta opción y por lo tanto le pasamos un null. El segundo parámetro, el esquema, lo pasaremos también a null. El tercer parámetro es el patrón de búsqueda que se va a aplicar al nombre de las tablas, si utilizamos el carácter especial % obtendremos todas las tablas. El último de los parámetros indica el tipo de tabla que queremos obtener.

Este parámetro es un array con todos los tipos de tablas de las que queremos obtener información, los tipos de tabla son: *TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS y SYNONYM*. En este ejemplo se quiere recuperar información sobre las tablas de usuario y de sistema, por lo que sólo tendremos un array con dos elementos, el primero de ellos contendrá la cadena TABLE y el segundo la cadena SYSTEM TABLE.

En el objeto ResultSet devuelto por el método getTables() encontraremos en cada fila del mismo información sobre cada tabla. Este ResultSet devuelto tiene un formato determinado, cada columna contiene una información determinada de la tabla que se corresponde con la fila actual del ResultSet.

Los nombres de estas columnas son:

- **TABLE_CAT**: catálogo de la tabla.
- **TABLE_SCHEM**: esquema de la tabla.
- **TABLE_NAME**: nombre de la tabla.
- **TABLE_TYPE**: tipo de la tabla.
- **REMARKS**: comentarios acerca de la tabla.

Para recuperar cualquier columna de este ResultSet utilizaremos el método getString() del interfaz ResultSet, ya que toda la información que contiene el ResultSet es de tipo cadena de
caracteres.

```
public static void infoTablas(DatabaseMetaData dbmd) throws SQLException{
       ResultSetrsTablas = dbmd.getTables(null,null,null,
                                          new String[]{"TABLE","SYSTEM_TABLE"});
       System.out.println();
       System.out.println("------ TABLAS-------");
       while(rsTablas.next()){
              System.out.println(rsTablas.getString("TABLE_CAT"));
              System.out.println(rsTablas.getString("TABLE_SCHEM"));
              System.out.println(rsTablas.getString("TABLE_NAME"));
              System.out.println(rsTablas.getString("TABLE_TYPE"));
              System.out.println(rsTablas.getString("REMARKS"));
              System.out.println("------------");
       }
}
```

```
public static void infoProcedimientos(DatabaseMetaDatadbmd) throws SQLException{
        ResultSet rsProced = dbmd.getProcedures(null,null,null);
        System.out.println();
        System.out.println("------ PROCEDIMIENTOS ALMACENADOS-------");
        while(rsProced.next()){
                System.out.println(rsProced.getString("PROCEDURE_CAT"));
                System.out.println(rsProced.getString("PROCEDURE_SCHEM"));
                System.out.println(rsProced.getString("PROCEDURE_NAME"));
                System.out.println(rsProced.getString("PROCEDURE_TYPE"));
                System.out.println(rsProced.getString("REMARKS"));
                System.out.println("------------");
        }
}
```

El método infoProcedimientos() es muy similar al anterior, en lugar de recuperar información sobre las tablas de la base de datos, vamos a obtener información sobre los procedimientos almacenados de la base de datos.

El método getProcedures() del interfaz DatabaseMetaData también devuelve un objeto ResultSet. Los parámetros son los mismos que en getTables(), a excepción del array de String que, en este caso, no se cotempla. Los nombres de las columnas del ResultSet devuelto por el método getProcedures() son:

- **PROCEDURE_CAT**: catálogo del procedimiento.
- **PROCEDURE_SCHEM**: esquema del procedimiento.
- **PROCEDURE_NAME**: nombre del procedimiento.
- **REMARKS**: comentarios acerca del procedimiento.
- **PROCEDURE_TYPE**: tipo de procedimiento.

12 ResultSetMetadata

Este interfaz tiene una finalidad muy parecida al visto en el tema anterior. El interfaz ResultSetMetaData nos ofrece una serie de métodos que nos permiten obtener información sobre las columnas que contiene el ResultSet. Es un interfaz más particular que el anterior, un objeto DatabaseMetaData es común a toda la base de datos, y cada objeto ResultSetMetaData es particular para cada ResultSet.

El número de métodos que ofrece este interfaz es mucho menor que el que ofrecía el interfaz DatabaseMetaData. Un objeto ResultSetMetaData lo obtendremos lanzando el método getMetaData() sobre el objeto ResultSet sobre el que queramos consultar la información de sus columnas.

El interfaz ResultSetMetaData proporciona métodos para recuperar información del ResultSet. A continuación mostramos una tabla con los más usados:

Método	Descripción
getTableName()	Devuelve un String con el nombre de la tabla que se ha consultado en el ResultSet.
getColumnCount()	Devuelve un int con el número de columnas que se ha consultado en el ResultSet
getColumnName(int n)	Devuelve un String con el nombre de la columna en la posición n
getColumnType(int n)	Devuelve el tipo JDBC de la columna en la posición n
getColumnTypeName(int n)	Devuelve el nombre del tipo JDBC que se corresponde con el tipo de la columna en la posición n.

A continuación vamos a mostrar un fragmento de código en el que, usando ResultSetMetaData, se va a consultar la tabla ***puerto*** de la base de datos *ciclismo* y se va a mostrar información de cada una de las columnas como el nombre de la columna, el tipo y otra información relevante.

```
...
Statement stm = con.createStatement();
ResultSetrs = stm.executeQuery("SELECT * FROM puerto");

// DatabaseMetadata
ResultSetMetaData rsmd = rs.getMetaData();

int numColumnas = rsmd.getColumnCount();
for(int i = 1; i<=numColumnas ;i++){
       System.out.println("Columna       " + i);
       System.out.println("Nombre:       " + rsmd.getColumnName(i));
       System.out.println("Tipo:         " + rsmd.getColumnType(i));
       System.out.println("Tipo:         " + rsmd.getColumnTypeName(i));
       System.out.println("Display Size: " + rsmd.getColumnDisplaySize(i));
       System.out.println("ClassName:    " + rsmd.getColumnClassName(i));
       System.out.println("Label:        " + rsmd.getColumnLabel(i));
       System.out.println("Precision:    " + rsmd.getPrecision(i));
       System.out.println("Escala:       " + rsmd.getScale(i));
       System.out.println("Nullable:     " + rsmd.isNullable(i));
       System.out.println("Auto increm:  " + rsmd.isAutoIncrement(i));
       System.out.println("-----------");
}
...
```

13 Ampliación 1: Objetos CallableStatement

El último tipo de sentencias que podemos utilizar en JDBC son las sentencias CallableStatement. Este interfaz hereda del interfaz PreparedStatement y ofrece la posibilidad de manejar parámetros de salida y de realizar llamadas a procedimientos almacenados de la base de datos.

Un objeto CallableStatement ofrece la posibilidad de realizar llamadas a procedimientos almacenados de una forma estándar para todos los DBMS. Un procedimiento almacenado se encuentra dentro de una base de datos; la llamada a un procedimiento es lo que contiene un objeto CallableStatement. Esta llamada está escrita con una sintaxis de escape, esta sintaxis puede tener dos formas diferentes: una con un parámetro de resultado, es un tipo de parámetro de salida que representa el valor devuelto por el procedimiento y otra sin ningún parámetro de resultado. Ambas formas pueden tener un número variable de parámetros de entrada, de salida o de entrada/salida. Una interrogación representará al parámetro.

La sintaxis para realizar la llamada a un procedimiento almacenado es la siguiente:

{callnombre_del_procedimiento[(?,?,...)]}

Si devuelve un parámetro de resultado:

{?=callnombre_del_procedimiento[(?.?...)]}

La sintaxis de una llamada a un procedimiento sin ningún tipo de parámetros sería:

{callnombre_del_procedimiento}

El interfaz CallableStatement hereda los métodos del interfaz Statement, que se encargan de sentencias SQL generales, y también los métodos del interfaz PreparedStatement, que se encargan de manejar parámetros de entrada.

13.1 Utilización de parámetros

Una sentencia CallableStatement puede tener parámetros de entrada, de salida y de entrada/salida. Para pasarle parámetros de entrada a un objeto CallableStatement, se utilizan los métodos setXXX() que heredaba del interfaz PreparedStatement.

Si el procedimiento almacenado devuelve parámetros de salida, el tipo JDBC de cada parámetro de salida debe ser registrado antes de ejecutar el objeto CallableStatement correspondiente. Para registrar los tipos JDBC de los parámetros de salida se debe lanzar el método CallableStatement.register- OutParameter.

Después de ejecutar la sentencia, se pueden recuperar los valores de estos parámetros llamando al método getXXX() adecuado. El método getXXX() debe recuperar el tipo Java que se correspondería con el tipo JDBC con el que se registró el parámetro. A los métodos getXXX() se le pasará un entero que indicará el valor ordinal del parámetro a recuperar.

En el siguiente fragmento de código se registra un parámetro de salida, ejecuta el procedimiento llamado por el objeto CallableStatement y luego recupera el valor de los parámetros de salida.

```
String jdbcUrl = "jdbc:mysql://localhost:3306/empresa";
Connection conn = DriverManager.getConnection(jdbcUrl,"root","");
//Preparamos la llamada al procedimiento
CallableStatementcstmt = conn.prepareCall("{call compras_clientes(?,?)}");
cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
cstmt.setInt(1, 360);
cstmt.execute();
```

14 Ampliación 2: JDBC BatchProcessing

El interfaz Statement también soporta el procesamiento de múltiples sentencias DML en modo batch. De esta forma minimizamos el número de llamadas a la base de datos que a veces se hace tan crítico en algunas aplicaciones.

Por ejemplo, supongamos una aplicación que necesite realizar muchas sentencias de tipo INSERT y de tipo UPDATE para controlar un determinado sistema. Si no utilizamos procesamiento batch podemos sobrecargar en exceso con llamadas a la base de datos y disminuir el rendimiento general de la aplicación. Para ello si utilizamos el procesamiento batch podemos agrupar varias sentencias INSERT y UPDATE y lanzarlas hacia la Base de Datos en una única llamada. Para ello es necesario saber si nuestra base de datos soporta el procesamiento batch, podemos consultar si nuestra base de datos soporta procesamiento Batch con el interfaz DatabaseMetaData y llamar al método supportBatchUpdates(), si devuelve true quiere decir que nuestra base de datos soporta procesamiento batch. En nuestro caso sí que tenemos soporte batch.

```
//Comprobarsoporte batch
Connection conexion = DriverManager.getConnection(jdbcUrl,"root","");
DatabaseMetaDatadm = conexion.getMetaData();
System.out.println("Soportaprocesamiento batch ->
"+dm.supportsBatchUpdates());
```

Para utilizar el procesamiento batch sobre el interfaz Statement llamamos al método addBatch() que recibe como parámetro la sentencia SQL y este proceso lo repetimos para cada una de las sentencias SQL que queramos añadir en batch. Una vez tengamos todas las sentencias llamamos al método executeBatch() del interfaz Statement. Previamente debemos haber desactivado el autocommit con el método setAutoCommit() y finalmente invocar al método commit() para realizar los cambios.

En el siguiente ejemplo se crean 3 artículos nuevos en modo batchsobre la base de datos empresa en MySQL:

```
Connection conn = DriverManager.getConnection(jdbcUrl,"root","");
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
//Añadimos sentencias SQL a modo Batch
String sql = "insert into articulosvalues(8,'HD 120G',100.0,'HD120',2)";
stmt.addBatch(sql);
sql = "insert into articulos values(9,'HD 160G',120.0,'HD160',2)";
stmt.addBatch(sql);
sql = "insert into articulos values(10,'HD 200G',140.0,'HD200',2)";
stmt.addBatch(sql);
int result[] = stmt.executeBatch();
conn.commit();
```

14.1 BatchUpdateException

Los objetos Statement, PreparedStatement y CallableStatement lanzan excepciones BatchUpdateException si ocurre un error cuando se está trabajando en modo batch.

Esta clase hereda de SQLException y por tanto contiene la misma información, sin embargo añade propiedades especificas que te ayudan a determinar cuál ha sido la sentencia que ha provocado el error.