Recuperación de Datos

En este ejercicio vamos a ver como recuperar datos de una tabla almacenada en la BD y al mismo tiempo veremos por un lado como hacer una primera aproximación a una modularización del código y por otro lado manejaremos ficheros como repaso del tema anterior. Veremos también una primera aproximación a los LOGS.

Empezaremos descomponiendo nuestro código en métodos lo más genéricos posibles. La idea sería acabar construyendo una librería que nos simplificase la escritura de código en los programas que desarrollemos para el control de BD relacionales.

Vamos a declarar una serie de variables a nivel de la clase para facilitar la escritura del código

```
public class SGBD {

    static Connection conexion;
    static Statement sentencia;
    public static final Logger LOG = Logger.getLogger(SGBD.class.getSimpleName());
```

conn guarda el estado de la conexión

stmt para la construcción de sentencias SQL

LOG para la gestión del Log

Por defecto cuando dejamos que el IDE escriba el código del try-catch, lo que hace es crear una línea de gestión de logs. Esto es muy útil pero solo si se maneja bien. Lo que va a hacer es escribir las condiciones de error, normalmente en rojo, en pantalla. Esto no es lo óptimo para comenzar ya que lo que hace es complicar más la salida por consola, por eso en principio preferimos sentencias System.out.println con un mensaje propio que indique de forma concisa y en vuestra propias palabras lo que ha ido mal.

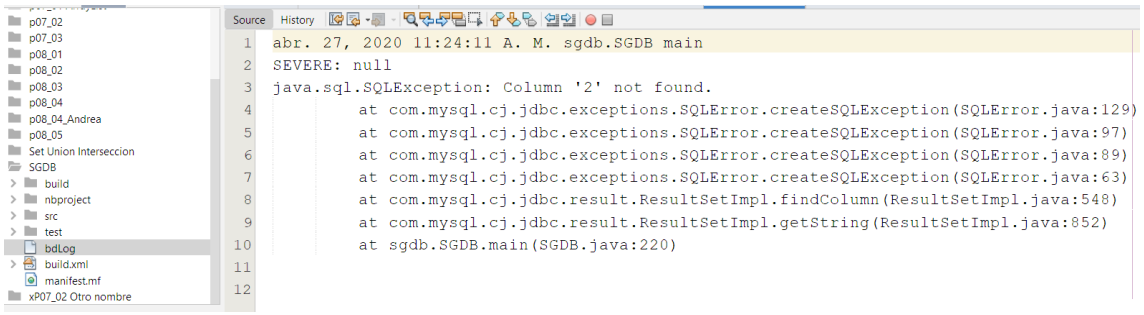
La gran utilidad de los Logs es el “desatenderse” de los errores, que estos se almacenen para su posterior revisión, y dejar una salida “limpia” por pantalla. O sea, vamos a buscar que en pantalla sólo aparezca la salida que nosotros establezcamos del programa (nuestras órdenes System.out.print, por ejemplo) y todo lo concerniente a excepciones lo redirigiremos a fichero. Pero el log no hace esto por defecto, ya que por defecto tanto la salida de error: **System.err** como la salida estándar: **System.out** están asignadas a la consola, en nuestro caso la pantalla.

Empezamos pues por la parte de **ficheros** (repaso de la 2ª evaluación). Vamos a hacer que la salida de error se redirija a fichero, y ahí es donde se grabarán nuestros logs:

```java
/**
 * Redirige la salida de error a fichero
 *
 * @param nombreFichero Nombre del fichero destino
 */
public static void redirigeSalidaError(String nombreFichero) {
  try {
    PrintStream ps = new PrintStream(new BufferedOutputStream(new FileOutputStream(
            new File(nombreFichero), true)), true);
    System.setErr(ps);
  } catch (FileNotFoundException ex) {
    System.err.println("Fichero de log no encontrado");
  }
}
```

Ahora todo lo que se manda a la salida de error, bien sea con System.err, o bien mediante los logs, se almacenará en el fichero indicado como parámetro del método.

Un ejemplo después de ejecutar una sentencia SQL mal formada. Vemos el contenido del fichero:



En la pantalla no habrá aparecido nada sobre el error. Evidentemente podemos por ejemplo utilizar la pantalla para dar una pequeña descripción del error (con System.out) y el fichero para guardar una descripción con más detalle para su análisis posterior sin complicar innecesariamente lo mostrado en pantalla al usuario (será el técnico el que analice los logs)

Vamos ahora con métodos genéricos para el manejo de la BD

1. Conectarse al SGBD, a una BD concreta:

```
/**
 * Conect a una Base de Datos
 *
 * @param url dirección y opcionalmente puerto y parámetros extra del servidor
 * MySQL/MariaDB
 * @param usuario Usuario con permnsos sobre la BD
 * @param clave Clave del usuario
 * @return true en caso de realizarse la conexión o false en otro caso.
 */
public static boolean bdConectarSGBD(String url, String usuario, String clave) {
   try {
      conexion = DriverManager.getConnection(url, usuario, clave);
   } catch (SQLException ex) {
      LOG.log(Level.SEVERE, null, ex);
      return false;
   }
   return true;
}
```

Pasamos como parámetros URL de la BD, usuario y password. En caso de no poder realizar la conexión, por un lado, se almacena en el fichero de log y por otros se devuelve false para su gestión desde el método que lo llamó (sabrá que la conexión no se pudo realizar y en función de ello ejecutará un código u otro).

2. Desconexión de la BD

```
/**
 * Cierra la conexión a la Base de Datos
 *
 * @return true si se pudo cerrar correctamente o false en caso contrario
 */
public static boolean bdCerrarConexion() {
   try {
      conexion.close();
   } catch (SQLException ex) {
      LOG.log(Level.SEVERE, null, ex);
      return false;
   }
   return true;
}
```

Al igual que antes, y lo que haremos en general, en caso de error tomamos dos medidas, por un lado, almacenamos información sobre el mismo en el fichero de log, y por otro lado devolvemos un valor (en este caso false) al método llamante para que determine que hacer (ejecutar uno u otro código, indicar algo en pantalla…) pero siempre sin estropear innecesariamente la salida del programa.

3. Crear una tabla

```
/**
 * Crea una tabla en la Base de Datos
 *
 * @param tabla Nombre de la tabla a crear
 * @param columnas Datos de las columnas que componen la tabla
 * @return true si se pudo crear la tabla o false en caso contrario
 */
public static boolean bdCrearTabla(String tabla, String[] columnas) {
  String sql = "CREATE TABLE IF NOT EXISTS " + tabla + "("
          + Arrays.toString(columnas).replace('[', ' ').replace(']', ' ') + ")";
  try {
    sentencia = conexion.createStatement();
    sentencia.executeUpdate(sql);
  } catch (SQLException ex) {
    LOG.log(Level.SEVERE, null, ex);
    return false;
  }
  try {
    sentencia.close();
  } catch (SQLException ex) {
    LOG.log(Level.SEVERE, null, ex);
  }
  return true;
}
```

Pasamos como parámetros el nombre de la tabla a crear y un array de String conteniendo el nombre y tipo de cada uno de los campos, con los que formaremos la sentencia SQL.

Importante incluir el “IF NOT EXITS” para evitar que de error en sucesivas ejecuciones del mismo código.

En caso de error grabamos en el log y devolvemos false.

4. Borrar una tabla

```
/**
 * Borra una tabla de la Base de Datos
 *
 * @param tabla Nombre de la tabla a borrar
 * @return true si se pudo borrar la tabla o false en caso contrario
 */
public static boolean bdBorrarTabla(String tabla) {
  try {
    sentencia = conexion.createStatement();
    sentencia.executeUpdate("DROP TABLE IF EXISTS " + tabla);
  } catch (SQLException ex) {
    LOG.log(Level.SEVERE, null, ex);
    return false;
  }
  try {
    sentencia.close();
  } catch (SQLException ex) {
    LOG.log(Level.SEVERE, null, ex);
  }
  return true;
}
```

Pasamos como parámetro el nombre de la tabla. Y en caso de error devolvemos false y grabamos el log.

5. Insertar datos en una tabla

```
/**
 * Inserta datos en una tabla de la Base de Datos
 *
 * @param tabla Nombre de la tabla en la que se van a insertar los datos
 * @param datos array de dos dimensiones <ol><li>primera dimensión: nombres de
 * los campos</li><li>segunda dimensión: valores de los campos. Entre '' si
 * son de algún tipo caracter</li></ol>
 * @return true si se pudo realizar la insercción o false en caso contrario
 */
public static boolean bdInsertarDatos(String tabla, String[][] datos) {
  String sql = "INSERT INTO " + tabla + " (";
  for (String[] dato : datos) {
    sql += dato[0] + ", ";
  }
  sql = sql.substring(0, sql.length() - 2) + ") VALUES(";
  for (String[] dato : datos) {
    sql += dato[1] + ", ";
  }
  sql = sql.substring(0, sql.length() - 2) + ")";
  try {
    sentencia = conexion.createStatement();
    sentencia.executeUpdate(sql);
  } catch (SQLException ex) {
    LOG.log(Level.SEVERE, null, ex);
```

```
            return false;
        }
        try {
            sentencia.close();
        } catch (SQLException ex) {
            LOG.log(Level.SEVERE, null, ex);
        }
        return true;
    }
```

Como parámetros pasamos el nombre de la tabla y un array de dos dimensiones que almacena el nombre de cada columna y su valor. Os recuerdo que, si tenéis algún campo auto incrementable, NO se debe pasar. Por otro lado, hay que recordar que los de tipo carácter deben ir encerrados entre comillas simples.

Como todas las anteriores, devuelve false en caso de error, además de escribir en el log.

6. Recuperar datos

Para finalizar por ahora, veremos un nuevo método (en el siguiente ejercicio veremos cómo actualizar datos), la necesaria para recuperar datos de la BD (una SELECT).

A diferencia de las vistas hasta este momento, para la realización de SELECTs no se emplea la sentencia executeUpdate sino executeQuery. La primera devolvía un int (el número de filas afectadas por la sentencia DML o 0 si las sentencias SQL que no devuelven nada). La sentencia executeQuery, devuelve en su lugar un conjunto de valores (las filas recuperadas). Ese conjunto es de la clase ResultSet.

El método que vamos a mostrar es el más simple. Suponemos que queremos recuperar todos los campos y sin ningún tipo de condición (SELECT * FROM tabla), y veremos cómo gestionar los datos recuperados.

```
/**
 * Recupera todas las filas de una tabla
 *
 * @param tabla Nombre de la tabla a consultar
 * @return ResulSet con las filas recuperadas o null en caso de error
 */
public static ResultSet bdRecuperarTodo(String tabla) {
    String sql = "SELECT * FROM " + tabla;
    ResultSet rs;
    try {
        sentencia = conexion.createStatement();
        rs = sentencia.executeQuery(sql);
    } catch (SQLException e) {
        LOG.log(Level.SEVERE, null, e);
        return null;
    }
    return rs;
}
```

Pasamos como parámetro el nombre de la tabla, y recuperamos el conjunto de datos resultante, que es lo que devolveremos en caso de que la ejecución no haya dado error (en caso contrario devolvemos null y grabamos el log).

¿Cómo hacemos ahora para gestionar los datos recuperados? Dentro del main tendríamos algo como:

```
ResultSet rs = bdRecuperarTodo(tabla);
if (rs != null) {
    try {
        int cont = 1;
        while (rs.next()) {
            int id = rs.getInt(1);
            String nombre = rs.getString("nombre");
            String apellido = rs.getString(3);
            String email = rs.getString("email");
            System.out.println("Persona " + cont + ": " + "\t" + id
                    + "\t" + nombre + "\t" + apellido
                    + "\t" + email);
        }
    } catch (SQLException ex) {
        LOG.log(Level.SEVERE, null, ex);
    }

}
```

Si el ResulSet recuperado no es null, es que tenemos datos sobre los que iterar. Mediante un bucle vamos recorriendo cada fila utilizando el método next() del ResultSet. La primera vez que se ejecuta devuelve la primera fila recuperada. El bucle se repite mientras que next() devuelva true.

Ahora que tenemos una fila recuperada, para acceder a los valores de sus campos utilizamos los métodos get() Hay para cada tipo de valor. En este ejemplo vemos como recuperar un entero (getInt()) y un String (getString()). Para indicar la columna a la que queremos hacer referencia tenemos 2 opciones:

1- Indicar el nombre de la columna
2- Indicar su posición (comienza a numerar en 1)

En el ejemplo combinamos ambas. Recuperamos nombre y email con el nombre del campo, y la ID y el apellido por su posición. IMPORTANTE, la posición es aquella en que pongamos los campos en la sentencia:

SELECT nombre, email, id FROM prueba2;

Aquí para recuperar el id sería: rs.getInt(3)

SELECT email, id, nombre FROM prueba2;

Mientras que aquí sería: rs.getInt(2)

¿Y qué pasa con *?

SELECT * FROM prueba2;

Pues se coge el orden que se utilizó a la hora de crear la tabla. En este caso sería rs.getInt(1).

El main completo:

```
public static void main(String[] args) throws IOException {
    String tabla = "prueba2";
    redirigeSalidaError("bdLog");
    bdConectarSGBF("jdbc:mysql://localhost/java", "root", "");
    bdCrearTabla(tabla, new String[]{
        "id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY",
        "nombre VARCHAR(30) NOT NULL",
        "apellido VARCHAR(30) NOT NULL",
        "email VARCHAR(50)"
    });
    Scanner tec = new Scanner(System.in);
    while (true) {
        System.out.print("Introduce nombre: ");
        String nombre = tec.nextLine();
        if (nombre.isEmpty()) {
            break;
        }
        System.out.print("Introduce apellido: ");
        String apellido = tec.nextLine();
        System.out.print("Introduce email: ");
        String email = tec.nextLine();

        bdInsertarDatos(tabla, new String[][]{
```

```
            {"nombre", "'" + nombre + "'"},
            {"apellido", "'" + apellido + "'"},
            {"email", "'" + email + "'"}
        });
    }

    ResultSet rs = bdRecuperarTodo(tabla);
    if (rs != null) {
        try {
            int cont = 1;
            while (rs.next()) {
                int id = rs.getInt(1);
                String nombre = rs.getString("nombre");
                String apellido = rs.getString(3);
                String email = rs.getString("email");
                System.out.println("Persona " + cont + ": " + "\t" + id
                        + "\t" + nombre + "\t" + apellido
                        + "\t" + email);
            }
        } catch (SQLException ex) {
            LOG.log(Level.SEVERE, null, ex);
        }

    }
    bdCerrarConexion();
}
```

Y esta sería una salida del programa.

```
129             return true;
130         }
131
```

Output ×

Debugger Console × | SQL 1 execution × | SGDB (run) ×

```
run:
Conexión establecida.
Introduce nombre:
Persona 1:      1       uno     dos     tres
Persona 1:      2       uno     dos     tres
Persona 1:      3       a       b       c
Persona 1:      4       fds     fdsaf   fa
BUILD SUCCESSFUL (total time: 2 seconds)
```

7. Recuperar datos, seleccionando columnas, y con claúsulas where y order by. Devolvemos ArrayList

Vamos a ver ahora otro método genérico para la recuperación de datos. En este caso permitiremos que se especifiquen las columnas a recuperar así como la condición que deben cumplir y el orden en que se mostrarán.

Además, en lugar de devolver un ResultSet, que implicaría que el usuario de la librería tendría que conocer su utilización, vamos a devolver en su lugar un ArrayList, de modo que el usuario no tenga que conocer el manejo de las instrucciones de java.sql

```
/**
 * Recupera datos de una tabla, especificando las columnas a obtener
 *
 * @param nombreTabla nombre de la tabla
 * @param columnas nombre de lascolumnas a recuperar
 * @param where condición que tienen que cumplir (null si no se quierecondición
 * @param order criterio de ordenación (null si no se quiere ordernar
 * @return Un ArrayList<ArrayList<String>> conteniendo todas las filas
 * recuperadas o null si no se pudo ejecutar la sentecia
 */
public static ArrayList<ArrayList<String>> bdRecupar(String nombreTabla,
        ArrayList<String> columnas, String where, String order) {
  ArrayList<ArrayList<String>> tabla = new ArrayList<>();
  String sql = "SELECT " + columnas.toString().replace("[", " ").replace("]", " ")
          + " FROM " + nombreTabla;
  if (where != null) {
    sql += " WHERE " + where;
  }
  if (order != null) {
    sql += " ORDER BY " + order;
  }
  ResultSet rs;
  try {
      sentencia = conexion.createStatement();
      rs = sentencia.executeQuery(sql);
      int numeroColumnas = columnas.size();
      for (ArrayList<String> row; rs.next(); tabla.add(row)) {
        row = new ArrayList<>(numeroColumnas);
        for (int c = 1; c <= numeroColumnas; ++c) {
          row.add(rs.getString(c));
        }
      }
      rs.close();
      sentencia.close();
    } catch (SQLException e) {
      LOG.log(Level.SEVERE, null, e);
      return null;
    }
    return tabla;
  }
```

```
                row.add(rs.getString(c));
              }
            }
            rs.close();
            sentencia.close();
        } catch (SQLException e) {
            LOG.log(Level.SEVERE, null, e);
            return null;
        }
        return tabla;
    }

}
```