Ut9.- Gestión de bases de datos relacionales.

Objetivos

En este tema veremos cómo utilizar JDBC (Java DataBase Connectivity) para almacenar información y recuperarla de una base de datos relacional a través de nuestras clases Java.

Utilizaremos los dos SGBD (Sistemas Gestores de Bases de Datos) vistos en el módulo de Bases de Datos: Oracle y MySQL.

1.- Introducción.

Hoy en día, la mayoría de aplicaciones informáticas necesitan almacenar y gestionar gran cantidad de datos.

Esos datos, se suelen guardar en bases de datos relacionales, ya que éstas son las más extendidas actualmente.

Las bases de datos relacionales permiten organizar los datos en tablas y esas tablas y datos se relacionan mediante campos clave. Además se trabaja con el lenguaje estándar conocido como SQL, para poder realizar las consultas que deseemos a la base de datos.

Una base de datos relacional se puede definir de una manera simple como aquella que presenta la información en tablas con filas y columnas.

Una **tabla es una serie de filas y columnas**, en la que **cada fila es un registro** y cada columna es un campo. Un **campo** representa un dato de los elementos almacenados en la tabla. Cada registro representa un elemento de la tabla (el equipo Real Madrid, el equipo Real Murcia, etc.)

Tradicionalmente, la programación de bases de datos ha sido como una Torre de Babel: gran cantidad de productos de bases de datos en el mercado, y cada uno "hablando" en su lenguaje privado con las aplicaciones.

Java, mediante JDBC (***Java Database Connectivity***), permite **simplificar el acceso** a bases de datos, proporcionando un lenguaje mediante el cual las aplicaciones pueden comunicarse con motores de bases de datos.

Esta API se desarrolló para el acceso a bases de datos, con tres objetivos principales:

- Ser un API con soporte de SQL: poder construir sentencias SQL e insertarlas dentro de llamadas al API de Java,
- Aprovechar la experiencia de los APIs de bases de datos existentes,
- Ser sencillo.

Autoevaluación

JDBC permite acceder a bases de datos relacionales cuando programamos con Java, pudiendo así utilizar SQL.

Verdadero Falso

1.1.- El desfase objeto-relacional.

El desfase objeto-relacional, también conocido como impedancia objeto-relacional, consiste en la diferencia de aspectos que existen entre la programación orientada a objetos y la base de datos. Estos aspectos se puede presentar en cuestiones como:

- **Lenguaje de programación**. El programador debe conocer el lenguaje de programación orientada a objetos POO y el lenguaje de acceso a datos.
- **Tipos de datos**: en las bases de datos relacionales siempre hay restricciones en cuanto a los tipos de datos que se pueden usar, que suelen ser sencillos, mientras que la programación orientada a objetos utiliza tipos de datos más complejos.
- **Paradigma de programación**. En el proceso de diseño y construcción del software se tiene que hacer una traducción del modelo orientado a objetos de clases al modelo Entidad-Relación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas o filas, lo que implica que se tengan que desarrollar dos diagramas diferentes para el diseño de la aplicación.

El modelo relacional trata con relaciones y conjuntos debido a su naturaleza matemática. Sin embargo, el modelo de Programación Orientada a Objetos trata con objetos y las asociaciones entre ellos. Por esta razón, el problema entre estos dos modelos surge en el momento de querer hacer persistir los objetos de negocio.

La escritura (y de manera similar la lectura) mediante JDBC implica:

- abrir una conexión,
- crear una sentencia en SQL y copiar todos los valores de las propiedades de un objeto en la sentencia,
- ejecutarla,
- procesar los resultados,
- cerrar la conexión.

Esto es sencillo para un caso simple, pero trabajoso si el objeto posee muchas propiedades, o bien se necesita almacenar un objeto que a su vez posee una colección de otros elementos.

Este problema es lo que denominábamos **impedancia Objeto-Relacional**, o sea, el conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito en un lenguaje de Programación Orientada a Objetos.

Podemos poner como ejemplo de desfase objeto-relacional, un Equipo de fútbol, que tenga un atributo que sea una colección de objetos de la clase Jugador. Cada jugador tiene un atributo "teléfono". Al transformar éste caso a relacional se ocuparía más de una tabla para almacenar la información, implicando varias sentencias SQL y bastante código.

1.2.- JDBC.

JDBC es un API Java que hace posible ejecutar sentencias SQL.

De JDBC podemos decir que:

- Consta de un **conjunto de clases e interfaces** escritas en Java.
- Proporciona un API estándar para desarrollar aplicaciones de bases de datos con un API Java puro.

Con **JDBC**, no hay que escribir un programa para acceder a una base de datos Access, otro programa distinto para acceder a una base de datos Oracle, MySQL, etc., sino que **podemos escribir un único programa** con el API JDBC y el programa se encargará de enviar las sentencias SQL a la base de datos apropiada. Además, y como ya sabemos, una aplicación en Java puede ejecutarse en plataformas distintas.

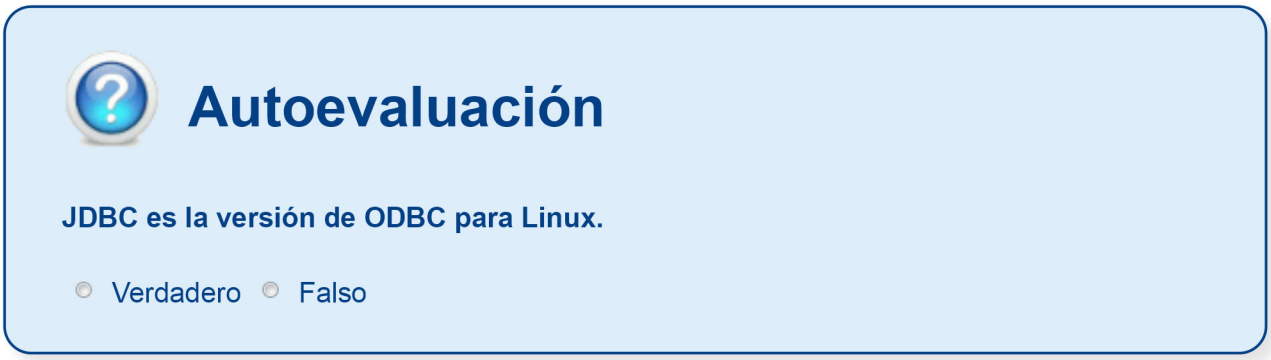
En el desarrollo de JDBC, y debido a la confusión que hubo por la proliferación de API's propietarias de acceso a datos, se buscaron los aspectos de éxito de un API de este tipo,ODBC (Open Database Connectivity).

ODBC se desarrolló con la idea de tener un estándar para el acceso a bases de datos en entornos Windows.

Aunque la industria ha aceptado ODBC como medio principal para acceso a bases de datos en Windows, ODBC no se introduce bien en el mundo Java, debido a la complejidad que presenta ODBC, y que entre otras cosas ha impedido su transición fuera del entorno Windows.

El nivel de abstracción al que trabaja JDBC es alto en comparación con ODBC.

JDBC intenta ser tan simple como sea posible, pero proporcionando a los desarrolladores la máxima flexibilidad.



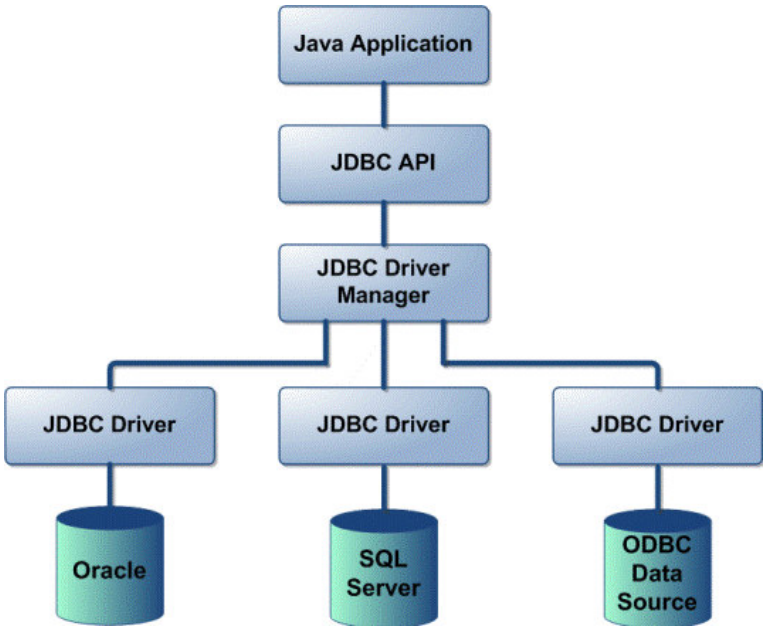
1.3.- Conectores o Drivers.

El API JDBC viene distribuido en dos paquetes:

- **java.sql**, dentro de J2SE
- **javax.sql**, extensión dentro de J2EE

Un conector o **driver** es un conjunto de clases encargadas de implementar las interfaces del API y acceder a la base de datos.

Para poder conectarse a una base de datos y lanzar consultas, una aplicación necesita tener un conector adecuado. Un conector suele ser un fichero .jar que contiene una implementación de todas las interfaces del API JDBC.



Cuando se construye una aplicación de base de datos, **JDBC oculta** los detalles específicos de cada base de datos, de modo que le programador se ocupe sólo de su aplicación.

El conector lo proporciona el **fabricante** de la base de datos o bien un tercero.

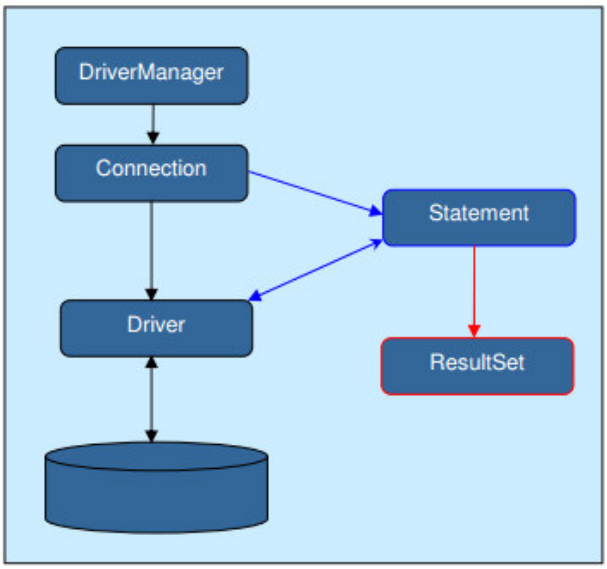
El código de nuestra aplicación no depende del driver, puesto que trabajamos contra los paquetes **java.sql** y **javax.sql**.

El uso del API JDBC ofrece las clases e interfaces para:

- Establecer una conexión con una base de datos.
- Ejecutar una sentencia SQL.
- Procesar los resultados.

2.- Establecimiento de conexiones.

A continuación se muestran las operaciones que pueden realizarse en una base de datos. Cada rectángulo representa una clase o interfaz de JDBC que tiene un role en el acceso a una BD relacional.



Todo trabajo con JDBC se inicia con la clase **DriveManager**, que es la que establece la **conexión** con las fuentes de datos, mediante el driver JDBC correspondiente. Para establecer la conexión podemos utilizar el método **getConnection()** de la clase **DriverManager**. Este método recibe como parámetro la URL de JDBC que identifica a la base de datos con la que queremos realizar la conexión.

La ejecución de este método devuelve un objeto **Connection** que representa la conexión con la base de datos.

Cuando se presenta con una URL específica, **DriverManager** itera sobre la colección de drivers registrados hasta que uno de ellos reconoce la URL especificada. Si no se encuentra ningún driver adecuado, se lanza una **SQLException**.

Las órdenes SQL (**Statements**) se envían a la base de datos a través de la conexión establecida. Cuando la orden genera resultados, éstos se almacenan en un objeto **ResultSet**.

Veamos un ejemplo comentado:

```
public static void main(String[] args){
   String driver = "com.mysql.jdbc.Driver";
   String servidor = "localhost";
   String bd = "test";
   String url = "jdbc:mysql://"+servidor+"/"+bd;
   String usuario = "root";
   String contraseña = "daw1";
   try  {
        // Carga el driver JDBC
        Class.forName(driver);
        // Establece la conexión
        Connection con = DriverManager.getConnection(url, usuario, contraseña);
        }
     } catch (ClassNotFoundException e) {
        // No encontró el driver
        System.out.println("ClassNotFoundException  "+e.getMessage());
     } catch (SQLException e){
        System.out.println(e.getMessage());
     }
   }
```

Si probamos este ejemplo con Eclipse, o cualquier otro entorno, y no hemos instalado el conector para MySQL, en la consola obtendremos el mensaje: Excepción: **java.lang.ClassNotFoundException: com.mysql.jdbc.Driver.**

2.1.- Instalar el conector de la base de datos.

Ya hemos comentado anteriormente que entre el programa Java y el Sistema Gestor de la Base de Datos (SGBD) se intercala el conector JDBC. Este conector es el que implementa la funcionalidad de las clases de acceso a datos y proporciona la comunicación entre el API JDBC y el SGBD.

La función del conector es traducir los comandos del API JDBC al protocolo nativo del SGBD.

A continuación se muestran las url de las páginas oficiales para descargar el conector (driver) que necesitamos para trabajar con MySQL y el que necesitamos para trabajar con Oracle 11g. (También están disponibles en el apartado de *Recursos del aula virtual*).

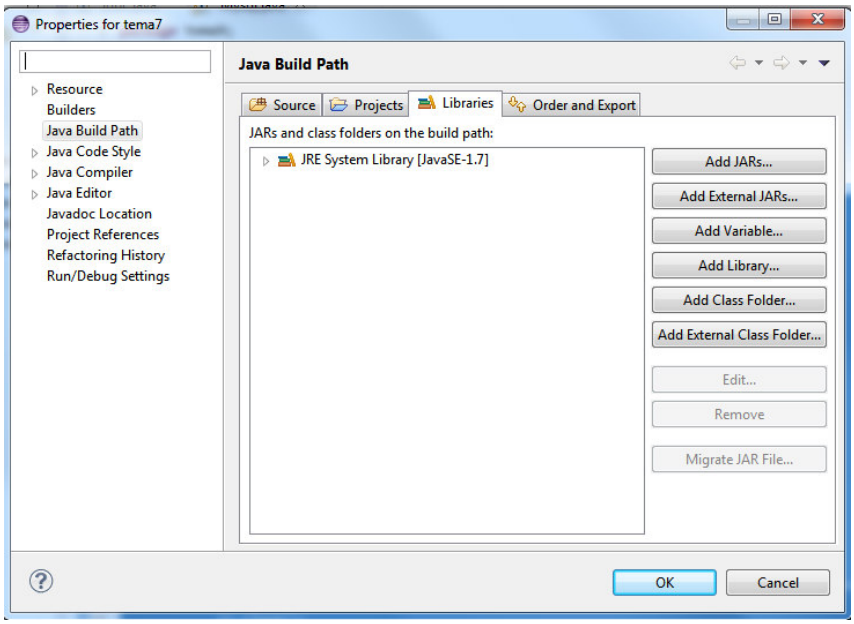
- Oracle: http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html
- MySQL: http://dev.mysql.com/downloads/connector/j/5.0.html

El proceso de instalación, tan sólo consiste en descargar un archivo, descomprimirloy extraer el fichero **.jar** que constituye el conector que necesitamos.

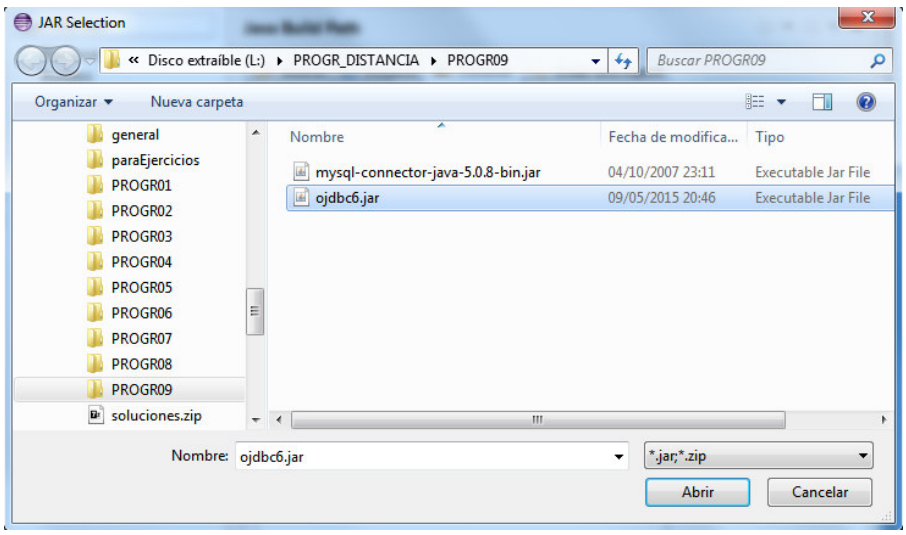
El driver JDBC no es necesario para compilar la clase Java, pero sí para ejecutarla.

Seguiremos estos pasos para añadirlo a nuestro proyecto de Eclipse:

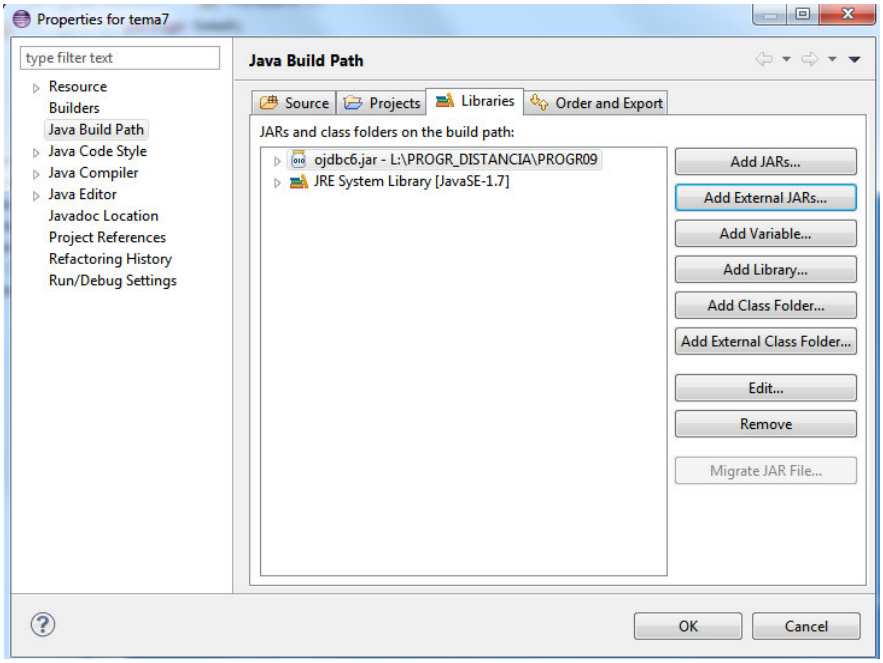
Desde el menú principal: Project - Properties - Java Build Path - Libraries



Pulsar el botón Add External JARs.. y seleccionar el driver JDBC correspondiente.



Abrir, y ya nos aparece el fichero jar en nuestra librería.



2.2.- Registrar el controlador JDBC.

Ya lo hemos visto en el ejemplo de código que poníamos antes. Registrar el controlador que queremos utilizar es tan fácil como escribir una línea de código.

Hay que consultar la documentación del controlador que vamos a utilizar para conocer el nombre de la clase que hay que emplear.

En el caso del controlador para **MySQL** es "**com.mysql.jdbc.Driver**", o sea, que se trata de la clase **Driver** que está en el paquete **com.mysql.jdbc** del conector que hemos descargado, y que has observado que no es más que una librería empaquetada en un fichero .jar. (*mysql-connector-java-5.0.8-bin.jar*)

La línea de código necesaria en este caso, en la aplicación Java que estemos construyendo es:

```
// Cargar el driver de MySQL
        Class.forName("com.mysql.jdbc.Driver");
```

En el caso del controlador para **Oracle** es "**oracle.jdbc.driver.OracleDriver**", o sea, que se trata de la clase **OracleDriver** que está en el paquete **oracle.jdbc.driver** del conector que hemos descargado, y que has observado que no es más que una librería empaquetada en un fichero .jar. (*ojdbc6.jar*)

La línea de código necesaria en este caso, en la aplicación Java que estemos construyendo es:

```
// Cargar el driver de Oracle
        Class.forName("oracle.jdbc.driver.OracleDriver");
```

Una vez cargado el controlador, ya es posible hacer una conexión al SGBD.

Hay que asegurarse de que si no utilizáramos Eclipse u otro IDE, para añadir el .jar como hemos visto, entonces el archivo .jar que contiene el controlador JDBC para el SGBD habría que incluirlo en el CLASSPATH que emplea nuestra máquina virtual, o bien en el directorio ext del JRE de nuestra instalación del JDK.

Hay una excepción en la que no hace falta ni hacer eso: en caso de utilizar un acceso mediante puente JDBC-ODBC, ya que ese driver está incorporado dentro de la distribución de Java, por lo que no es necesario incorporarlo explícitamente. Por ejemplo, sería el caso de acceder a una base de datos Microsoft Access.

3.- Ejecución de consultas sobre la base de datos.

Para operar con una base de datos ejecutando las consultas necesarias, nuestra aplicación deberá hacer las operaciones siguientes:

- **Cargar el conector** necesario para comprender el protocolo que usa la base de datos en cuestión.
- **Establecer una conexión** con la base de datos.
- **Enviar consultas** SQL y procesar el resultado.
- **Liberar los recursos** al terminar.
- **Gestionar los errores** que se puedan producir.

Podemos utilizar los siguientes tipos de sentencias:

- **Statement**: para sentencias sencillas en SQL.
- **PreparedStatement**: para consultas preparadas, como por ejemplo las que tienen parámetros.
- **CallableStatement**: para ejecutar procedimientos almacenados en la base de datos.

El API JDBC distingue dos tipos de consultas:

- **Consultas**: **SELECT**. Para las sentencias de consulta que obtienen datos de la base de datos, se emplea el método **ResultSet executeQuery(String sql)**. El método de ejecución del comando SQL devuelve un objeto de tipo **ResultSet** que sirve para contener el resultado del comando SELECT, y que nos permitirá su procesamiento.
- **Actualizaciones**: **INSERT, UPDATE, DELETE**, sentencias DDL. Para estas sentencias se utiliza el método **executeUpdate(String sql)**

Autoevaluación

Para poder enviar consultas a la base de datos hemos tenido que conectarnos a ella previamente.

Verdadero Falso

3.1.- Recuperación de información.

Las consultas a la base de datos se realizan con sentencias SQL que van **"embebidas"** en otras sentencias especiales que son propias de Java. Por tanto, podemos decir que las consultas SQL las escribimos como parámetros de algunos métodos Java que reciben el **String** con el texto de la consulta SQL.

Las consultas devuelven un **ResultSet**, que es una clase java parecida a una lista en la que se aloja el resultado de la consulta. Cada elemento de la lista es uno de los registros de la base de datos que cumple con los requisitos de la consulta.

El ResultSet no contiene todos los datos, sino que los va obteniendo de la base de datos según se van pidiendo. La razón de esto es evitar que una consulta que devuelva una cantidad muy elevada de registros, tarde mucho tiempo en obtenerse y sature la memoria del programa.

Con el **ResultSet** hay disponibles una serie de métodos que permiten movernos hacia delante y hacia atrás en las filas, y obtener la información de cada fila.

Por ejemplo, para obtener: nif, nombre y teléfono de los clientes que están almacenados en la tabla del mismo nombre, de la base de datos *test*, haríamos la siguiente consulta:

```
//Establecemos la conexion
Class.forname("com.mysql.jdbc.Driver");
String url = "jdbc:mysql://localhost/test";
Connection con = DriverManager.getConnection(url,"root","daw1");

// Preparamos la consulta y la ejecutamos
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery ("SELECT NIF, Nombre, Tfno FROM clientes");
```

El método **next()** del **ResultSet** hace que dicho puntero avance al siguiente registro. Si lo consigue, el método **next()** devuelve **true**. Si no lo consigue, porque no haya más registros que leer, entonces devuelve **false**.

El método **executeQuery** devuelve un objeto **ResultSet** para poder recorrer el resultado de la consulta utilizando un cursor.

Para obtener una columna utilizamos los métodos **get**. Hay un método **get**... para cada tipo básico Java y para las cadenas.

Un método interesante es **wasNull** que nos informa si el último valor leído con un método **get** es nulo.

Cuando trabajamos con el **ResultSet**, en cada registro, los métodos **getInt(), getString(), getDate()**, etc., nos devuelve los valores de los campos de dicho registro. Podemos pasar a estos métodos un índice (que comienza en 1) para indicar qué columna de la tabla de base de datos deseamos, o bien, podemos usar un **String** con el nombre de la columna (tal cual está en la tabla de base de datos).

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery ("SELECT NIF, Nombre, Tfno FROM clientes");
while (rs.next()){
   System.out.println("Cliente: "+rs.getString("NIF")+" -- "+getString(2)+" -- "+getString(3));
}
```

Autoevaluación

Para obtener un entero almacenado en uno de los campos de un registro, trabajando con el `ResulSet` emplearemos el método `getInt()`.

- ○ Verdadero
- ○ Falso

3.2.- Actualización de información.

Respecto a las consultas de actualización, **executeUpdate**, devuelve el número de registros insertados, registros actualizados o eliminados, dependiendo del tipo de consulta que se trate.

Supongamos que tenemos varios registros en la tabla Clientes, de una base de datos Oracle XE. Si quisiéramos actualizar el teléfono del tercer registro, que tiene idCliente=3 y ponerle como nuevo teléfono el 968610009 tendríamos que hacer:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
String url = "jdbc:oracle:thin:@localhost:1521:xe";
 // Establece la conexión
  Connection con = DriverManager.getConnection (url, "empresa", "empresa");

// Preparamos la consulta y la ejecutamos
Statement st = con.createStatement();
String sql ="UPDATE clientes SET teléfono='968610009'  WHERE idCliente=3";
int nfilas = st.executeUpdate(sql);
// Cerramos la conexión a la base de datos.
con.close();
```

3.3.- Adición de información.

Si queremos añadir un registro a la tabla Clientes, de la base de datos con la que estamos trabajando tendremos que utilizar la sentencia **INSERT INTO** de SQL. Al igual que hemos visto en el apartado anterior, utilizaremos **executeUpdate** pasándole como parámetro la consulta, de inserción en este caso.

Así, un ejemplo sería:

```
// Preparamos la consulta y la ejecutamos
Statement st = con.createStatement();
String sql =  "INSERT INTO Clientes (idCliente, NIF, nombre, cpostal, tlfno, email)"
 +  " VALUES (4, '66778998T', 'Alfredo Gates Gates','20400', '891222112', 'prueba@lalaboral.es' )" ;
st.executeUpdate(sql) ;
```

Autoevaluación

Al añadir registros a una tabla de una base de datos, tenemos que pasar como parámetro al **executeUpdate()**, una sentencia SQL del tipo: **DELETE**...

- Verdadero
- Falso

3.4.- Borrado de información.

Cuando nos interese eliminar registros de una tabla de una base de datos, emplearemos la sentencia SQL: **DELETE**.

Así, por ejemplo, si queremos eliminar el registro a la tabla ClienteS, de nuestra base de datos y correspondiente a la persona que tiene el nif: 66778998T, tendremos que utilizar el código siguiente.

```
// Preparamos la consulta y la ejecutamos
Statement st = con.createStatement();
int numReg = st.executeUpdate( "DELETE FROM clientes WHERE NIF= '66778998T' " );
 // Informamos del número de registros borrados
System.out.println ("\nSe borraron " + numReg + " registros") ;
```

Autoevaluación

Al ejecutar el borrado de un registro mediante executeUpdate(…), no podemos saber si el borrado eliminó alguna fila o no.

- Verdadero
- Falso

3.5.- Cierre de conexiones.

Las conexiones a una base de datos consumen muchos recursos en el sistema gestor por ende en el sistema informático en general.

Por ello, conviene cerrarlas con el método **close()** siempre que vayan a dejar de ser utilizadas, en lugar de esperar a que el recolector de basuras de Java las elimine.

También conviene cerrar las consultas (**Statement** y **PreparedStatement**) y los resultados (**ResultSet**) para liberar los recursos.

4.- Excepciones.

En todas las aplicaciones en general, y por tanto en las que acceden a bases de datos en particular, nos puede ocurrir con frecuencia que la aplicación no funciona, no muestra los datos de la base de datos que deseábamos, etc.

Es importante capturar las excepciones que puedan ocurrir para que el programa no aborte de manera abrupta. Además, es conveniente tratarlas para que nos den información sobre si el problema es que se está intentando acceder a una base de datos que no existe, o que el servicio MySQL no está arrancado, o que se ha intentado hacer alguna operación no permitida sobre la base de datos, como acceder con un usuario y contraseña no registrados, ...

Por tanto es conveniente emplear el método **getMessage()** de la clase **SQLException** para recoger y mostrar el mensaje de error que ha generado MySQL, lo que seguramente nos proporcionará una información más ajustada sobre lo que está fallando.

Cuando se produce un error se lanza una excepción del tipo **java.sql.SQLException.**

- Es importante que **las operaciones de acceso a base de datos** estén **dentro de un bloque try-catch** que gestione las excepciones.
- Los objetos del tipo **SQLException** tienen dos métodos muy útiles para obtener el código del error producido y el mensaje descriptivo del mismo, **getErrorCode()** y **getMessage()** respectivamente.

El método **getMessage()** imprime el mensaje de error asociado a la excepción que se ha producido, nos ayuda a saber qué ha generado el error que causó la excepción. El método **getErrorCode()**, devuelve un número entero que representa el código de error asociado. Habrá que consultar en la documentación para averiguar su significado.

Autoevaluación

El cierre de las conexiones y la gestión de excepciones sólo hay que efectuarla con bases de datos MySQL.

○ Verdadero ○ Falso