Tema 10. Bases de datos

Principales conceptos que se abordan en este tema

¿Utilizarías un procesador de textos que no te da la opción de guardar el documento que estás editando? Como es obvio, a nadie se le ocurre hacer un programa así. De hecho, casi todos los programas hoy día tienen la opción de guardar los datos, sean o no procesadores de texto.

Hasta ahora, ya conoces como abrir un archivo y utilizarlo como "almacén" para los datos que maneja tu aplicación. Utilizar un archivo para almacenar datos es la forma más sencilla de persistencia, porque en definitiva, **la persistencia es hacer que los datos perduren en el tiempo**.

Hay muchas formas de hacer los datos de una aplicación persistentes, y muchos niveles de persistencia. Cuando los datos de la aplicación solo están disponibles mientras la aplicación se esta ejecutando, tenemos un nivel de persistencia muy bajo, y ese era el caso de Antonio: su aplicación no almacenaba los datos en ningún lado, y en posteriores ejecuciones los datos no podían ser utilizados, pues solo estaban disponibles mientras no cerraras la aplicación.

Lo deseable es que los datos de nuestra aplicación tengan un nivel de persistencia lo mayor posible. Tendremos un mayor nivel de persistencia si los datos "sobreviven" varias ejecuciones, o lo que es lo mismo, si nuestros datos se guardan y luego son reutilizables con posterioridad. Tendremos un nivel todavía mayor si "sobreviven" varias versiones de la aplicación, es decir, si guardo los datos con la versión 1.0 de la aplicación y luego puedo utilizarlos cuando este disponible la versión 2.0.

Pero lo verdaderamente interesante de esta unidad, es la forma de hacer persistentes los datos. En esta unidad te vamos a proponer un enfoque totalmente diferente a almacenar los datos en meros archivos: vamos a utilizar bases de datos relacionales para hacer los datos de tu aplicación persistentes.

Pero para hacer esto, primero tienes que aprender qué son las bases de datos, en adelante llamadas BD, y aprender a usarlas en tu aplicación. Comprobarás que con las BD hacer los datos persistentes es más fácil y práctico de lo que imaginabas.

Número de sesiones

Se estiman un total de 14 horas

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 (NSS, nombre, etc.) Cada registro representa un elemento de la tabla (el equipo Real Madrid, el equipo Real Murcia, etc.)

No se permite que pueda aparecer dos o más veces el mismo registro, por lo que uno o más campos de la tabla forman lo que se conoce como **clave primaria**.

El sistema gestor de bases de datos, en inglés conocido como: **Database Management System (DBMS)**, gestiona el modo en que los datos se almacenan, mantienen y recuperan.

En el caso de una base de datos relacional, el sistema gestor de base de datos se denomina: **Relational Database Management System (RDBMS)**.

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 base de datos,** proporcionando un lenguaje mediante el cual las aplicaciones pueden comunicarse con motores de bases de datos. Sun desarrolló este API para el acceso a bases de datos, con tres objetivos principales en mente:

- 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.

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 diseñar 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 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 y así almacenar el objeto. 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. Se necesita crear mucho más código, además del tedioso trabajo de creación de sentencias SQL.

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 con u lenguajes 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 pura.

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, 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 propietarios de acceso a datos, Sun buscó 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 entorno 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, la intención de Sun fue que supusiera la base de partida para crear librerías de más alto nivel.

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.**

JDBC ofrece las clases e interfaces para:

- Establecer una conexión a una base de datos.
- Ejecutar una consulta.
- Procesar los resultados.

Ejemplo:

```
// Establece la conexión
Connection con = DriverManager.getConnection ("jdbc:odbc:miBD", "miLogin", "miPassword");

// Ejecuta la consulta
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT nombre, edad FROM Jugadores");

// Procesa los resultados
while (rs.next()) {
    String nombre = rs.getString("nombre");
    int edad = rs.getInt("edad");
}
```

En principio, todos los conectores deben ser compatibles con ANSI SQL-2 Entry Level (ANSI SQL-2 se refiere a los estándares adoptados por el American National Standards Institute (ANSI) en 1992. Entry Level se refiere a una lista específica de capacidades de SQL). Los desarrolladores de conectores pueden establecer que sus conectores conocen estos estándares.

1.4 Instalación de la base de datos

Lo primero que tenemos que hacer, para poder realizar consultas en una base de datos, es obviamente, instalar la base de datos. Dada la cantidad de productos de este tipo que hay en el mercado, es imposible explicar la instalación de todas. Así que vamos a optar por una, en concreto por MySQL ya que es un sistema gestor gratuito y que funciona en varias plataformas.

2. Creación de las tablas en una base de datos

En Java podemos conectarnos y manipular bases de datos utilizando JDBC. Pero la creación en sí de la base de datos debemos hacerla con la herramienta específica para ello. Normalmente será el administrador de la base de datos, a través de las herramientas que proporcionan el sistema gestor, el que creará la base de datos. No todos los drivers JDBC soportan la creación de la base de datos mediante el lenguaje de definición de datos (**DDL**).

Normalmente, cualquier sistema gestor de bases de datos incluye asistentes gráficos para crear la base de datos con sus tablas, claves, y todo lo necesario.

También, como en el caso de MySQL, o de Oracle, y la mayoría de sistemas gestores de bases de datos, se puede crear la base de datos, desde la línea de comandos de MySQL o de Oracle, con las sentencias SQL apropiadas.

Vamos utilizar el sistema gestor de base de datos MySQL, por ser un producto gratuito y de fácil uso, además de muy potente.

Debes conocer

Puedes descargarte e instalar la herramienta gráfica que permite entre otras cosas trabajar para crear tablas, editarlas, añadir datos a las tablas, etc., con MySQL. Aquí puedes descargarlo:

Descarga de MySQL WorkBench.

2.1 Lenguaje SQL

¿Cómo le pedimos al Sistema Gestor de Bases de Datos Relacional (SGBDR), en concreto en este caso, al de MySQL, que nos proporcione la información que nos interesa de la base de datos?

Se utiliza el **lenguaje SQL** (Structured Query Language) para interactuar con el SGBDR.

SQL es un lenguaje **no procedimental** en el cual se le indica al SGBDR **qué** queremos obtener **y no cómo hacerlo.** El SGBDR analiza nuestra orden y si es correcta sintácticamente la ejecuta.

El estudio de SQL nos llevaría mucho más que una unidad, y es objeto de estudio en otros módulos de este ciclo formativo. Pero como resulta imprescindible para poder continuar, haremos una mínima introducción sobre él.

Los comandos SQL se pueden dividir en dos grandes grupos:

- Los que se utilizan para **definir las estructuras de datos**, llamados comandos **DDL (Data Definition Language)**. Los que se utilizan para
- Los que se utilizan para **operar con los datos almacenados en las estructuras**, llamados **DML (Data Manipulation Language).**

Para saber más

En este enlace encontrarás de una manera breve pero interesante la historia del SQL

La primera fase del trabajo con cualquier base de datos comienza con sentencias DDL, puesto que antes de poder almacenar y recuperar información debemos definir las estructuras donde agrupar la información. Las estructuras básicas con las que trabaja SQL son las tablas.

Como hemos visto antes, una tabla es un conjunto de celdas agrupadas en filas y columnas donde se almacenan elementos de información.

Antes de llevar a cabo la creación de una tabla conviene planificar: nombre de la tabla, nombre de cada columna, tipo y tamaño de los datos almacenados en cada columna, información adicional, restricciones, etc.

Hay que tener en cuenta también ciertas restricciones en la formación de los nombres de las tablas: longitud. Normalmente, aunque dependen del sistema gestor, suele tener una longitud máxima de 30 caracteres, no puede haber nombres de tabla duplicados, deben comenzar con un carácter alfabético, permitir caracteres alfanuméricos y el guión bajo '_', y normalmente no se distingue entre mayúsculas y minúsculas.

Por ejemplo para crear una tabla de departamentos podríamos hacer:

```
CREATE TABLE departa (
                    cod_dep number(3),
                    nombre varchar2(15) not null,
                    loc varchar2(10),
                    constraint dep_pk primary key (cod_dep),
                    constraint dep_loc check (loc in ('Madrid', 'Barcelona', 'Murcia'))
);
```

donde creamos la tabla con cod_dep como clave primaria. Además, se añade una restricción para comprobar que cuando se esté dando de alta un registro, lo que se escriba en el campo **loc** sea Madrid, Barcelona o Murcia.

```
CREATE TABLE emp (
            cod_emp number(3),
            nombre varchar2(10) not null,
            oficio varchar2(11),
            jefe number(3),
            fecha_alta date,
            salario number(10),
            comision number(10),
            cod_dep number(3),
            constraint emp_pk primary key (cod_emp),
```

```
            constraint emp_fk foreign key (cod_dep) references departa(cod_dep) on delete cascade,
            constraint emp_ck check (salario > 0)
);
```

En el caso de esta tabla, se puede ver que hay un restricción para comprobar que el salario sea mayor que 0.

Y una tabla de empleados, teniendo en cuenta el departamento en el que trabajen:

Para saber más

En el siguiente enlace encontrarás algunos de los comandos SQL más utilizados.

3. Establecimiento de conexiones

Cuando queremos acceder a una base de datos para operar con ella, lo primero que hay que hacer es conectarse a dicha base de datos.

En Java, para establecer una conexión con una base de datos 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`**.

Veamos un ejemplo comentado:

```
public static void main(String[] args) {
   try {
        // Cargar el driver de mysql
        Class.forName("com.mysql.jdbc.Driver");
        // Cadena de conexión para conectar con MySQL en localhost, seleccionar la base de datos llamada test
        // con usuario y contraseña del servidor de MySQL: root y admin
        String connectionUrl = "jdbc:mysql://localhost/test?user=root&password=admin";
        // Obtener la conexión
        Connection con = DriverManager.getConnection(connectionUrl);
   } catch (SQLException e) {
        System.out.println("SQL Exception: "+ e.toString());
   } catch (ClassNotFoundException cE) {
        System.out.println("Excepción: "+ cE.toString());
   }
}
```

Si probamos este ejemplo con NetBeans, 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`**.

3.1 Instalar el conector en la base de datos

En la siguiente presentación vamos a ver cómo descargarnos el conector o driver que necesitamos para trabajar con MySQL. Como verás, tan sólo consiste en descargar un archivo, descomprimirlo y desde NetBeans añadir el fichero **.jar** que constituye el conector que necesitamos.

Estos serían los pasos necesarios a realizar:

- A través del navegador accedemos a la dirección: http://www.mysql.com/downloads/connector/j/
- Se pulsa en el botón azul Download que se ve a la derecha de la pantalla. Está disponible en .zip y en .tar.
- En la pantalla que aparece después, pinchamos en la parte de abajo para poder seleccionar los servidores de descarga donde se aloja el conector JDBC. Ahora, elegimos uno de lo servidores que aparecen y pinchamos en FTP que aparece hacia la derecha de la pantalla. Una vez descargado y descomprimido el archivo, nos fijamos en un fichero que nos interesa que es el .jar.
- En NetBeans, situándonos en el nombre el proyecto pulsamos el botón derecho del ratón.
- En el menú contextual que aparece seleccionamos Properties.
- Seleccionamos el nodo de las Librerías del proyecto.
- Pinchamos en el botón Add JAR/Folder.
- Buscamos y elegimos el fichero comentado anteriormente, el .jar.
- Tan solo queda pulsar Ok y hemos acabado.

Por tanto, como ya hemos comentado anteriormente, 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.

3.2 Registrar el controlador JDBC

Al fin y al cabo ya lo hemos visto en el ejemplo de código que poníamos antes, pero incidimos de nuevo. 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.

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");
```

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

Hay que asegurarse de que si no utilizáramos NetBeans 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 en el classpath de una aplicación Java. Por ejemplo, sería el caso de acceder a una base de datos Microsoft Access.

4. 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)`

4.1 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, apellidos* y *teléfono* de los clientes que están almacenados en la tabla del mismo nombre, de la base de datos *notarbd* que se creó anteriormente, haríamos la siguiente consulta:

```
// Preparamos la consulta y la ejecutamos
Statement s = n.createStatement();
ResultSet rs = s.executeQuery ("SELECT NIF, NOMBRE, APELLIDOS, TELÉFONO FROM CLIENTE");
```

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**.

Para saber más

Puedes consultar todos los métodos que soporta `ResultSet`, además de más información, en la API:

ResultSet (Java SE 11 & JDK 11)

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

Para obtener una columna del registro 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**.

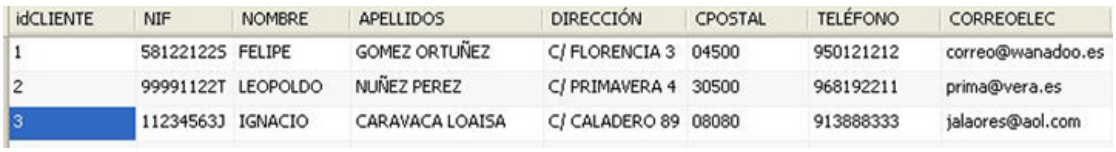
Cuando trabajamos con el `ResultSet`, en cada registro, los métodos `getInt()`, `getString()`, `getDate()`, etc., nos devuelve los valores de los campos de dicho registro. Podemos pasar a estos métodos un índice (que comienza en 1) para indicar qué columna de la tabla de base de datos deseamos, o bien,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

```
// Obtener la conexión
Connection con = DriverManager.getConnection(connectionUrl);

// Preparamos la consulta
Statement s = con.createStatement();
ResultSet rs = s.executeQuery ("SELECT NIF, NOMBRE, APELLIDOS, TELÉFONO FROM CLIENTE");

// Iteramos sobre los registros del resultado
while (rs.next())
       System.out.println(rs.getString("NIF") + " " +
                          rs.getString (2) + " " +
                          rs.getString (3) + " " +
                          rs.getString (4));
```

4.2 Actualización de información

Respecto a las consultas de actualización, **`executeUpdate`**, retornan 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 Cliente, de la base de datos ***notarbd*** con la que seguimos trabajando.



idCLIENTE	NIF	NOMBRE	APELLIDOS	DIRECCIÓN	CPOSTAL	TELÉFONO	CORREOELEC
1	58122122S	FELIPE	GOMEZ ORTUÑEZ	C/ FLORENCIA 3	04500	950121212	correo@wanadoo.es
2	99991122T	LEOPOLDO	NUÑEZ PEREZ	C/ PRIMAVERA 4	30500	968192211	prima@vera.es
3	11234563J	IGNACIO	CARAVACA LOAISA	C/ CALADERO 89	08080	913888333	jalaores@aol.com

Si quisiéramos actualizar el teléfono del tercer registro, que tiene idCLIENTE=3 y ponerle como nuevo teléfono el 968610009 tendríamos que hacer:

```
String connectionUrl = "jdbc:mysql://localhost/notarbd?" + "user=root&password=admin";

// Obtener la conexión
Connection con = DriverManager.getConnection(connectionUrl);

// Preparamos la consulta y la ejecutamos
Statement s = con.createStatement();
s.executeUpdate("UPDATE CLIENTE SET teléfono='968610009' WHERE idCLIENTE=3");

// Cerramos la conexión a la base de datos.
con.close();
```

4.3 Adición de información

Si queremos añadir un registro a la tabla Cliente, 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 s = con.createStatement();
s.executeUpdate( "INSERT INTO CLIENTE" +
             " (idCLIENTE, NIF, NOMBRE, APELLIDOS, DIRECCIÓN, CPOSTAL, TELÉFONO, CORREOELEC) VALUES " +
                      "  (4,  '66778998T',  'Alfredo',  'Gates  Gates',  'C/  Pirata  23','20400',  '891222112',
'prueba@eresmas.es' )") ;
```

4.4 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 Cliente, 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 s = con.createStatement();
numReg = res.executeUpdate( "DELETE FROM CLIENTE WHERE NIF= '66778998T' " );

// Informamos del número de registros borrados
System.out.println ("\nSe borró " + numReg + " registro\n") ;
```

4.5 Cierre de las 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 garbage collector de Java las elimine.

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

4.6 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, que aunque esté en inglés, 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.