UD3. MANEJO DE CONECTORES (Bases de Datos Relacionales)

1. Introducción.

Actualmente, las **bases de datos relacionales** constituyen el sistema de almacenamiento probablemente más extendido, aunque otros sistemas de almacenamiento de la información se estén abriendo paso poco a poco.

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 o relación es una colección de objetos del mismo tipo (filas o tuplas).
En cada tabla de una base de datos se elige una **clave primaria** para identificar de manera unívoca a cada fila de la misma.

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**) o en español “Sistema Gestor de bases de Datos Relacional” (**SGBDR)**.

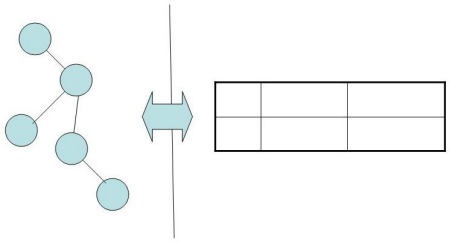
Tradicionalmente, la programación de bases de datos ha sido como una torre de Babel: gran cantidad de productos de bases de datos en el mercado y cada uno “hablando” en su lenguaje privado con las aplicaciones.

Java, mediante **JDBC** (**Java Database Connectivity**), permite simplificar el acceso a bases de datos relacionales, 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 API’s de bases de datos existentes.
- **Ser lo más sencillo posible**.

1.1 El desfase objeto-relacional.

El **modelo relacional o bases de datos relacionales** no están diseñadas para almacenar objetos, sino que trata con **relaciones o tablas** debido a su **naturaleza matemática**. En cambio, **el modelo de POO 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 la aplicación.



Sin

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 pueden presentar en cuestiones como:

- **Lenguaje de programación:** el programador debe conocer el lenguaje de programación orientado a objetos (POO) y el lenguaje de acceso a datos.
- **Tipos de datos:** en las bases de datos relacionales siempre hay restricciones en el uso de tipos, mientras que la programación orientada a objetos utiliza tipos de datos mas 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.

Sin embargo, el paradigma relacional y el paradigma orientado a objetos pueden ser “amigos”. Cada vez que los objetos deben extraerse o almacenarse de la base de datos relacional se requiere un mapeo desde las estructuras provistas en el modelo de datos a las provistas por el entorno de programación. En la siguiente unidad trataremos este tema mediante herramientas diseñadas para facilitar este proceso.

Al trabajar con JDBC implica (tanto para lecturas como para escrituras):

- Abrir o establecer una conexión con la base de datos.
- Crear una sentencia en SQL.
- 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 complicado 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 POO.

2. Protocolos de acceso a bases de datos

En tecnologías de bases de datos podemos encontrar las siguientes normas de conexión a la BD:

- **ODBC (Open Database Connectivity):** define una API que pueden usar las aplicaciones para abrir una conexión con una base de datos, enviar consultas, actualizaciones y obtener resultados. Las aplicaciones pueden usar esta API para conectarse a cualquier servidor de bases de datos compatible con ODBC.
- **JDBC (Java Database Connectivity):** define una API que pueden usar los programas Java para conectarse a los servidores de bases de datos realcionales.

ODBC había sido desarrollado por Microsoft con la idea de tener un estándar para el acceso a bases de datos en entorno Windows.

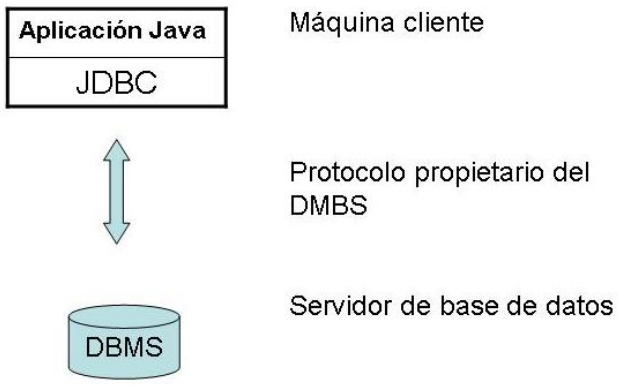
Aunque la industria aceptó ODBC como medio principal para acceso a bases de datos en Windows, la verdad es que no se introduce bien en el mundo Java, debido a la complejidad que presenta ODBC, y que entre otras cosas ha impedido su transición fuera del entorno Windows.

La idea en el desarrollo de JDBC era intentar ser tan sencillo como fuera posible, pero proporcionando a los desarrolladores la máxima flexibilidad.

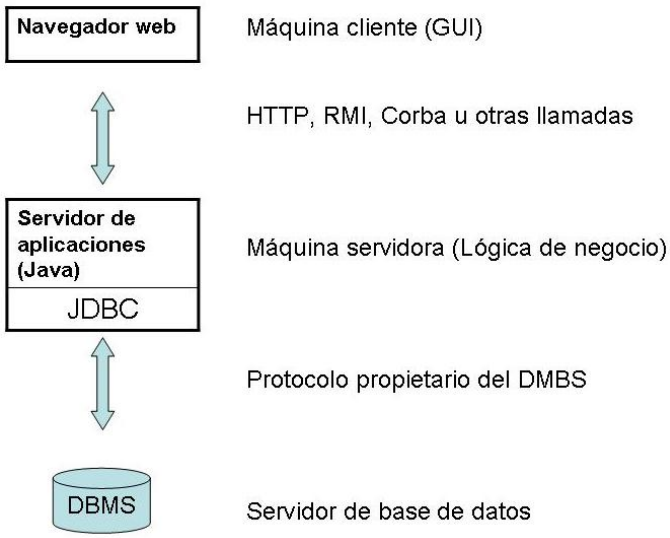
2.1 Arquitectura JDBC.

El **API JDBC** soporta dos modelos de procesamiento para acceso a bases de datos: de dos y tres capas.

En el **modelo de dos capas**, una aplicación se comunica directamente a la fuente de datos. Esto necesita un conector JDBC que pueda comunicar con la fuente de datos específica a la que acceder.
Los comandos o instrucciones del usuario se envían a la base de datos y los resultados se devuelven al usuario. La fuente de datos puede estar ubicada en otra máquina a la que el usuario se conecte por red. A esto se denomina configuración cliente/servidor, con la máquina del usuario como cliente y la máquina que aloja los datos como servidor.



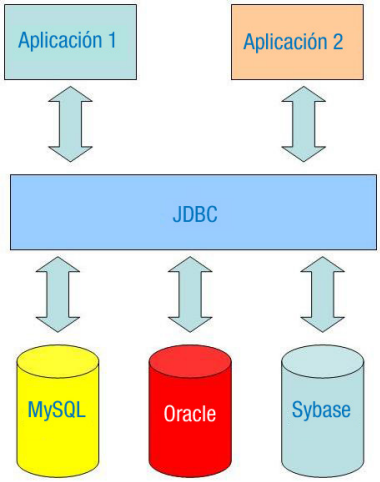
En el **modelo de tres capas**, los comandos se envían a una capa intermedia de servicios, la cual envía los comandos a la fuente de datos. La fuente de datos procesa los comandos y envía los resultados de vuelta la capa intermedia, desde la que luego se le envían al usuario.



El API JDBC viene distribuido en dos paquetes:

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

2.2 Conectores o Drivers. Tipos



Un conector o driver es un conjunto de clases encargadas de implementar los 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 driver 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 lo específico de cada base de datos**, de modo que el 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 mediante los paquetes **java.sql y javax.sql**.

JDBC ofrece las clases e interfaces para:

- Establecer una conexión a una base de datos (BD).
- Ejecutar una consulta e instrucciones de actualización a la base de datos.
- Recuperar y procesar los resultados recibidos de la BD como respuesta a las consultas.

Ejemplo:

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

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

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

Hay **cuatro tipos de drivers JDBC**: Tipo 1, Tipo 2, Tipo 3 y Tipo 4.

Tipo 1. JDBC-ODBC Bridge (puente JDBC-ODBC).

Proporcionan un puente entre el API JDBC y el API ODBC. El driver JDBC-ODBC Bridge traduce las llamadas JDBC a llamadas ODBC y las envía a la fuente de datos ODBC. Exige la instalación y configuración de ODBC en la máquina cliente.

Tipo 2. API nativa

Convierten las llamadas JDBC a llamadas específicas del motor de bases de datos.
El conector tipo 2 se comunica directamente con el servidor de bases de datos, por lo que es necesario que haya código en la máquina cliente, es decir, exige instalar en la máquina cliente código binario propio del cliente de base de datos y del sistema operativo.

Como inconveniente, señalar que la librería de la bases de datos del vendedor necesita cargarse en cada máquina cliente. Por esta razón los drivers tipo 2 no pueden usarse para Internet.

Tipo 3: JDBC-Network.

Este controlador utiliza un protocolo de red para comunicarse con un servidor de base de datos. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red. No exige instalación en cliente.

Tipo 4: Protocolo nativo.

En este caso se trata de conectores que convierten directamente las llamadas JDBC al protocolo de red usando por el sistema gestor de la base de datos. Esto permite una llamada directa desde la máquina cliente al servidor del sistema gestor de base de datos y es una solución excelente para acceso en intranets.

No se necesita instalar ningún software especial en el cliente o en el servidor. Como inconveniente, de este tipo de conectores, el usuario necesita un driver diferente para cada base de datos.

Los tipos 3 y 4 son la mejor forma de acceder a bases de datos mediante JDBC, en la mayoría de los casos la opción más adecuada es el tipo 4. Los tipos 1 y 2 se usan normalmente cuando no queda otro remedio, porque el único sistema de acceso final al gestor de bases de datos (SGBD) es ODBC (es decir, no existen drivers disponibles para el SGBD); además exigen la instalación de software en el puesto cliente.

2.3 Cómo funciona JDBC.

JDBC define varias interfaces que permiten realizar operaciones con bases de datos; a partir de ellas se derivan las clases correspondientes. Estas están definidas en el paquete **java.sql**. La siguiente tabla muestra las clases e interfaces más importantes:

CLASE - INTERFACE	Descripción
Driver	Permite conectarse a una base de datos: cada gestor de base de datos requiere un gestor distinto
DriverManager	Permite gestionar todos los drivers instalados en el sistema
DriverPropertyInfo	Proporciona diversa información acerca de un driver
Connection	Representa una conexión con una base de datos.
DatabaseMetadata	Proporciona información acerca de una base de datos
Statement	Permite ejecutar sentencias SQL sin parámetros
PreparedStatement	Permite ejecutar sentencias SQL con parámetros de entrada
CallableStatement	Permite ejecutar sentencias SQL con parámetros de entrada y salida, como llamadas a procedimientos almacenados.
ResultSet	Contiene las filas resultantes de ejecutar una SELECT
ResultSetMetadata	Permite obtener información sobre ResultSet, como número de columnas, sus nombres, etc..

Documentación oficial paquete java.sql
http://docs.oracle.com/javase/6/docs/api/java/sql/package-summary.html

El trabajo con JDBC comienza con la clase **DriverManager** que es la encargada de establecer la conexión con los orígenes de datos a través de los drivers JDBC. El funcionamiento de un programa con JDBC requiere los siguientes pasos:

- Importar las clases necesarias
- Cargar el driver JDBC
- Identificar el origen de datos
- Crear un objeto *Connection*
- Crear un objeto *Statement*
- Ejecutar una consulta con un objeto *Statement*
- Recuperar los datos del objeto *ResultSet*
- Liberar el objeto *ResultSet*
- Liberar el objeto *Statement*
- Liberar el objeto *Connection*

3. Conexión a una base de datos.

Para acceder a una base de datos y así poder 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 conector adecuado, se lanza una **SQLException**

Veamos un ejemplo con comentarios, para conectarnos a una base de datos MySQL:

```
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());
    }
}
```

3.1 Instalación del conector

Los siguientes ejemplos utilizan la base de datos relacional MySQL. Para probar los ejemplos debes tener instalado en tu equipo:

- El SGBD MySQL (es conveniente instalar las GUI Tool MySQLWorkbench para gestionar directamente la BD MySQL)
- El conector o driver para Java (JDBC): Connector/J (*mysql-connector-java-5.1.22-bin.jar*).

El anterior archivo .jar debe estar incluido en el CLASSPATH o añadirlo a nuestro IDE a los proyectos que lo vayan a usar.

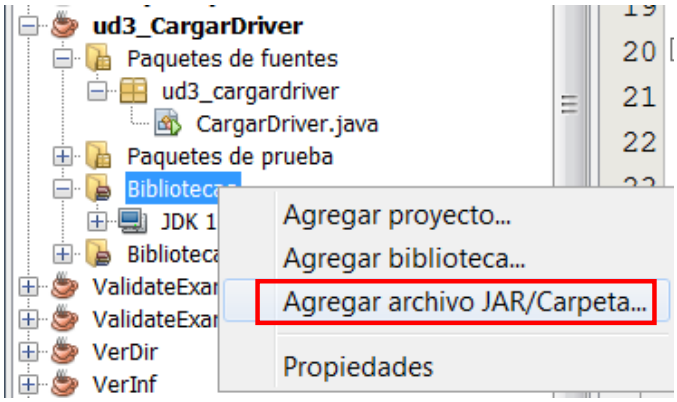
Recuerda que entre nuestra aplicación Java y el Sistema Gestor de Base de Datos (SGBD), se intercala el conector JDBC. Este conector es el que implementa la funcionalidad de las clases de acceso a datos y proporciona la comunicación entre el API JDBC y el SGBD.

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

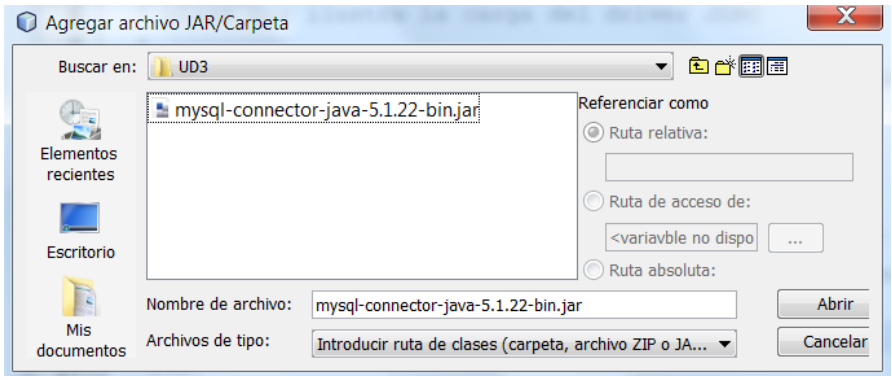
Descarga de Software MySQL
http://dev.mysql.com/downloads/

Una vez descargado el driver JDBC, para añadirlo a nuestro proyecto en NetBeans: seguimos los pasos:

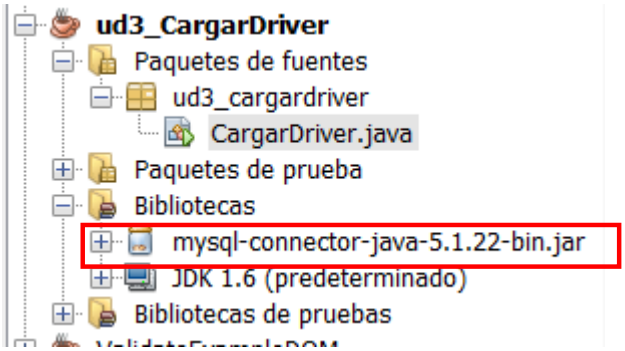
1.- Sobre Bibliotecas del proyecto, seleccionamos “Agregar archivo JAR/Carpeta…”



2. Seleccionamos archivo .jar de interés



3. Ya ha quedado integrado en nuestro proyecto.



4. Creación de la base de datos.

Normalmente es 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 conectores JDBC soportan la creación de la base de datos mediante el lenguaje de definición de datos (**DDL**). Es decir, la sentencia ***CREATE DATABASE*** no es parte del estándar SQL, sino que es dependiente del sistema gestor de la base de datos.

Así pues, mediante JDBC podemos conectarnos y manipular bases de datos: crear tablas, modificarlas, borrarlas, añadir datos en las tablas, etc. Pero la creación en sí de la base de datos la hacemos con la herramienta específica para ello.

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

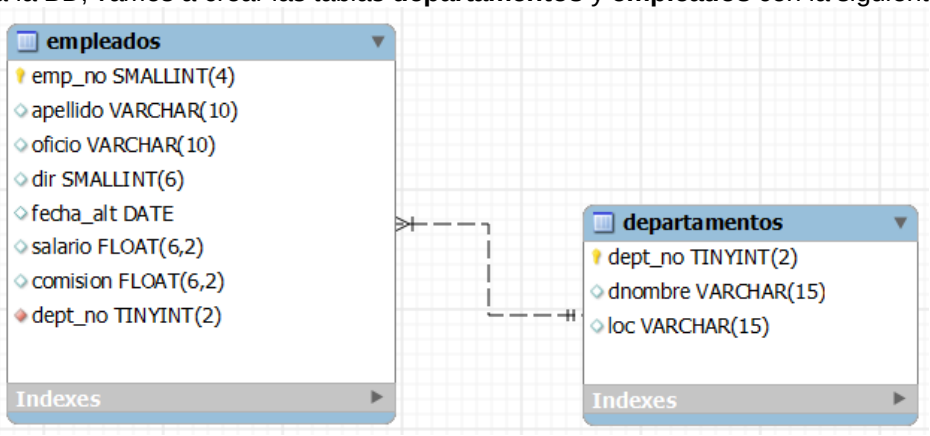
También, como en el caso de MySQL, o de Oracle, y la mayoría de sistemas gestores de bases de datos, se puede crear la base de datos, desde la línea de comandos de MySQL o de Oracle, con las sentencias SQL apropiadas.

Para el siguiente ejemplo crearemos la base de datos de nombre **empresaz** desde MySQL (mediante un cliente gráfico o bien un cliente en modo texto):

```
mysql> CREATE DATABASE IF NOT EXISTS empresaz;
```

Probaremos los ejemplos con el usuario **root** de MySQL o bien otro **usuario con permisos** sobre la base de datos en cuestión.

Una vez creada la BD, vamos a crear las tablas **departamentos** y **empleados** con la siguiente estructura:



Las tablas las podemos crear mediante un script SQL, redactando las sentencias DDL correspondientes, mediante un cliente gráfico, mediante una herramienta de modelado de datos e ingeniería directa o mediante programación.

Las clases ***CreaEmpleados.java*** y ***CreaDepartamentos.java*** permiten crear las tablas de empleados y departamentos e insertar algunos datos en cada una de ellas.

EJEMPLO 1. [CreaDepartamentos.java]

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
/* clase que ilustra la creación de la tabla 'departamentos' en la base de datos
 * 'empresaz' del servidor 'localhost', así como la iserción de unos cuantos
 * registros de ejemplo. */
public class CreaDepartamentos {
  /* parámetros de conexión con la base de datos MySql */
  private static final String url = "jdbc:mysql://localhost/empresaz";
  private static final String user = "root";
  private static final String password = "admin";
  /* sentencia de creación de la tabla 'departamentos' */
  private static final String creaDepartamentos =
          "CREATE TABLE departamentos("
          + "dept_no TINYINT(2) NOT NULL PRIMARY KEY,"
          + "dnombre VARCHAR(15),"
          + "loc VARCHAR(15)"
          + ")";
  /* sentencia de inserción de los registros de ejemplo */
  private static final String insertaDepartamentos =
          "INSERT INTO departamentos VALUES"
          + "(10,'CONTABILIDAD','SEVILLA'),"
          + "(20,'INVESTIGACIÓN','MADRID'),"
          + "(30,'VENTAS','BARCELONA'),"
          + "(40,'PRODUCCIÓN','BILBAO')";

  public static void main(String[] args) {
    /* objeto de conexión */
    Connection conexion = null;
    /* comando para enviar sentencias a través de la conexión */
    Statement sentencia = null;
    try {
      /* carga el driver previamente añadido a las 'Bibliotecas' */
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      /* establece la conexión */
      conexion = DriverManager.getConnection(url, user, password);
      /* crea el comando para enviar sentencias SQL */
      sentencia = conexion.createStatement();
      /* ejecuta la sentencia de creación de la tabla 'departamentos' */
      sentencia.execute(creaDepartamentos);
      /* mensaje de confirmación */
      System.out.println("Tabla 'departamentos' creada correctamente");
      /* ejecuta la sentencia de inserción de departamentos */
      sentencia.execute(insertaDepartamentos);
      /* mensaje de confirmación */
      System.out.println("Nuevos departamentos insertados correctamente");
    } catch (Exception ex) {
      /* mensaje de error */
      System.err.println(ex);
    } finally {
      /* si llegó a crearse el comando */
      if (sentencia != null) {
        try {
          /* trata de liberar recursos */
          sentencia.close();
        } catch (SQLException ex) {
        }
      }
      /* si llegó a establecerse la conexión */
      if (conexion != null) {
        try {
          /* trata de liberar recursos */
          conexion.close();
```

```
        } catch (SQLException ex) {
        }
      }
    }
  }
}
```

EJEMPLO 2. [CreaEmpleados.java]

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
/* clase que ilustra la creación de la tabla 'empleados' en la base de datos
 * 'empresaz' del servidor 'localhost', así como la iserción de unos cuantos
 * registros de ejemplo.*/
public class CreaEmpleados {
  /* parámetros de conexión con la base de datos MySql */
  private static final String url = "jdbc:mysql://localhost/empresaz";
  private static final String user = "root";
  private static final String password = "admin";
  /* sentencia de creación de la tabla 'empleados' */
  private static final String creaEmpleados =
          "CREATE TABLE empleados("
          + "emp_no SMALLINT(4) UNSIGNED NOT NULL PRIMARY KEY,"
          + "apellido VARCHAR(10),"
          + "oficio VARCHAR(10),"
          + "dir SMALLINT,"
          + "fecha_alt DATE,"
          + "salario FLOAT(6,2),"
          + "comision FLOAT(6,2),"
          + "dept_no TINYINT(2) NOT NULL,"
          + "FOREIGN KEY (dept_no) REFERENCES departamentos(dept_no)"
          + ")";
  /* sentencia de inserción de los registros de ejemplo */
  private static final String insertaEmpleados =
          "INSERT INTO empleados VALUES"
          + "(1000,'AMARO','AUXILIAR',1,20100102,940.00,0.00,10),"
          + "(2000,'CASTRO','AUXILIAR',2,20110102,890.00,0.00,10),"
          + "(3000,'HERNANDEZ','CONTABLE',3,20010102,1890.00,0.00,10),"
          + "(4000,'FERNANDEZ','TEC.LAB.',4,20080102,1500.00,0.00,20),"
          + "(5000,'HERMOSILLA','DIR.LAB.',5,20060102,2420.00,0.00,20),"
          + "(6000,'DE ANDRES','GERENTE',6,20010102,3450.00,20.00,40),"
          + "(7000,'GOMEZ','VENDEDOR',7,20020102,1200.00,10.00,30),"
          + "(8000,'BORREGO','VENDEDOR',8,20020102,1200.00,10.00,30)";

  public static void main(String[] args) {
    /* objeto de conexión */
    Connection conexion = null;
    /* comando para enviar sentencias a través de la conexión */
    Statement sentencia = null;
    try {
      /* carga el driver previamente añadido a las 'Bibliotecas' */
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      /* abre la conexión */
      conexion = DriverManager.getConnection(url, user, password);
      /* crea el comando para enviar sentencias SQL */
      sentencia = conexion.createStatement();
      /* ejecuta la sentencia de creación de la tabla 'empleados' */
      sentencia.execute(creaEmpleados);
      /* mensaje de confirmación */
      System.out.println("Tabla 'empleados' creada correctamente");
      /* ejecuta la sentencia de inserción de empleados */
      sentencia.execute(insertaEmpleados);
      /* mensaje de confirmación */
      System.out.println("Nuevos empleados insertados correctamente");
    } catch (Exception ex) {
      /* mensaje de error */
```

```
      System.err.println(ex);
    } finally {
      /* si llegó a crearse el comando */
      if (sentencia != null) {
        try {
          /* trata de liberar recursos */
          sentencia.close();
        } catch (SQLException ex) {
        }
      }
      /* si llegó a establecerse la conexión */
      if (conexion != null) {
        try {
          /* trata de liberar recursos */
          conexion.close();
        } catch (SQLException ex) {
        }
      }
    }
  }
}
```

4.1 Ejemplo de consulta

Recuerda que para operar con una base de datos, ejecutando las consultas necesarias, nuestra aplicación deberá hacer:

- **Cargar** el driver 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`**
- Actualizaciones: **`INSERT`**, **`UPDATE`**, **`DELETE`**, sentencias DDL

EJEMPLO 3. Consulta de la tabla departamentos **[CreaEmpleados.java]**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
/* clase que ilustra la consulta de la tabla 'departamentos' en la base de datos
 * 'empresaz' del servidor 'localhost', su recogida en un ResultSet, y su
 * posterior impresión en la Salida.*/
public class ConsultaDepartamentos {
  /* parámetros de conexión con la base de datos MySql */
  private static final String url = "jdbc:mysql://localhost/empresaz";
  private static final String user = "root";
  private static final String password = "admin";
  /* sentencia de consulta de la tabla 'departamentos' */
  private static final String consultaDepartamentos =
          "SELECT * FROM departamentos";

public static void main(String[] args) {
    /* objeto de conexión */
    Connection conexion = null;
    /* comando para enviar sentencias a través de la conexión */
    Statement sentencia = null;
```

```
 /* objeto ResultSet para recoger los registros devueltos por la consulta */
    ResultSet result = null;

 try {
      /* carga el driver previamente añadido a las 'Bibliotecas' */
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      /* abre la conexión */
      conexion = DriverManager.getConnection(url, user, password);
      /* crea el comando para enviar sentencias SQL */
      sentencia = conexion.createStatement();
      /* ejecuta la consulta y recoge el resultado */
      result = sentencia.executeQuery(consultaDepartamentos);
      /* mientras hay registros*/
      while (result.next()) {
        /* recupera los datos por su ordinal en la tabla o por el nombre
         * de la columna, y los muestra en la Salida debidamente tabulados */
        System.out.printf("%2d  %-15s  %s\n", result.getInt(1),
                result.getString("dnombre"), result.getString(3));
      }
      /* mensaje de confirmación */
      System.out.println("\nTabla 'departamentos' consultada correctamente");
    } catch (Exception ex) {
      /* mensaje de error */
      System.err.println(ex);
    } finally {
      /* si llegó a rellenarse el RecordSet */
      if (result != null) {
        try {
          /* trata de liberar recursos */
          result.close();
        } catch (SQLException ex) {
        }
      }
      /* si llegó a crearse el comando */
      if (sentencia != null) {
        try {
          /* trata de liberar recursos */
          sentencia.close();
        } catch (SQLException ex) {
        }
      }
      /* si llegó a establecerse la conexión */
      if (conexion != null) {
        try {
          /* trata de liberar recursos */
          conexion.close();
        } catch (SQLException ex) {
        }
      }
    }
  }
}
```

Observa en el ejemplo los pasos indicados anteriormente:

- **Cargar el driver:**

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

- **Establecer la conexión**:

```
 private static final String url = "jdbc:mysql://localhost/empresaz";
 private static final String user = "root";
 private static final String password = "admin";
Connection conexion = DriverManager.getConnection(url, user, password);
```

- **Ejecutar o enviar sentencias SQL**

Se necesita un objeto Statement para enviar una consulta sencilla SQL. Para crear un objeto Statement del Interfaz Statement se usa el método `createStatement() de un objeto Connection válido.`

```
Statement sentencia = conexion.createStatement();
/* ejecuta la consulta y recoge el resultado */
```

`La sentencia obtenida tiene el método executeQuery() que sirve para realizar una consulta a la base de datos, se le pasa un string donde está la consulta SQL, en el ejemplo: “"SELECT * FROM departamentos" asociada a la constante consultaDepartamentos`

```
ResultSet result = sentencia.executeQuery(consultaDepartamentos);
```

El resultado lo devuelve como un ResultSet, que es un objeto similar a una lista en el que está el resultado de la consulta.. Cada elemento de la lista, es uno de los registros de la tabla *departamentos*.

ResultSet no contiene todos los datos, sino que los va consiguiendo de la base de datos según se van pidiendo. Por ello, el método executeQuery puede tardar poco, pero recorrer todos los elementos del ResultSet puede no ser tan rápido.

ResultSet tiene internamente un puntero que apunta al primer registro de la lista. Mediante el método next() el puntero avanza al siguiente registro. Para recorrer la lista de registros usamos dicho método dentro de un bucle while que se ejecutará mientras next() devuelva trae(es decir, mientras haya registros).

```
while (result.next()) {

        System.out.printf("%2d  %-15s  %s\n", result.getInt(1),
                result.getString("dnombre"), result.getString(3));
}
```

Los métodos getInt() y getString() nos van devolviendo los valores de los campos de dicho registro. Entre paréntesis se pone la posición de la columna en la tabla, es decir, la columna que deseamos.

También se puede poner una cadena que indica el nombre de la columna:

```
System.out.printf("%2d  %-15s  %s\n", result.getInt("dep_no"),
                  result.getString("dnombre"), result.getString("loc"));
```

- **Liberar recursos**

Se liberan recursos y se cierra conexión

```
result.close();         //Cerrar ResultSet
sentencia.close();     //Cerrar Statement
conexion.close();      //Cerrar Conexión
```

5. Ejecución de sentencias de descripción de datos.

Normalmente, cuando desarrollamos una aplicación JDBC, conocemos las estructuras de las tablas y datos que estamos manejando, es decir, conocemos las columnas que tienen y cómo están relacionadas entre sí, etc.

Es posible que no conozcamos la estructura de las tablas de una base de datos, en este caso la información de la base de datos la podemos obtener a través de los ***metaobjetos***, que no son más que objetos que proporcionan información sobre la base de datos.

5.1 DatabaseMetaData

La interface **DatabaseMetaData** proporciona información sobre la base de datos a través de múltiples métodos de los cuales es posible obtener gran cantidad de información.

El método ***getMetaData()*** de la interface **Connection**, devuelve un objeto *DatabaseMetaData* con el que obtiene información de la base de datos.

EJEMPLO 4. Conectamos con la base de datos ***empresaz*** y se muestra información sobre el producto de base de datos, el driver, la URL para acceder a la base de datos, el nombre de usuario y las tablas y las vistas del esquema actual. **[ud3_DatabaseMetaData.java]**

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;

public class ud3_DataBaseMetaData {
  private static final String url = "jdbc:mysql://localhost/empresaz";
  private static final String user = "root";
  private static final String password = "";

  public static void main(String[] args) {
    /* variables locales */
    String nombre, driver, url_name, usuario, catalogo, esquema, tabla, tipo;
    /* objeto de conexiónn */
    Connection conexion = null;
    /* objeto ResultSet para recoger los registros devueltos por la consulta */
    ResultSet result = null;
    /* objeto de metadatos */
    DatabaseMetaData dbmd;
    try {
      /* carga el driver previamente añadido a las 'Bibliotecas' */
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      //establece  conexión
      conexion = DriverManager.getConnection(url, user, password);
      //crea objeto de metadatos y obitene información
      dbmd = conexion.getMetaData();
      nombre = dbmd.getDatabaseProductName();
      driver = dbmd.getDriverName();
      url_name = dbmd.getURL();
      usuario = dbmd.getUserName();
      /* imprime valores */
      System.out.println("INFORMACIÓN SOBRE LA BASE DE DATOS:");
      System.out.println("==================================");
      System.out.println("Nombre: " + nombre);
      System.out.println("Driver: " + driver);
      System.out.println("URL: " + url_name);
      System.out.println("Usuario: " + usuario + "\n");

/* obtiene la informaciónn de las tablas y vistas existentes */
      result = dbmd.getTables(null, "empresaz", null, null);
```

```
/* mientras puede avanzar, asigna valores obtenidos*/
      while (result.next()) {
        catalogo = result.getString(1);
        esquema = result.getString(2);
        tabla = result.getString(3);
        tipo = result.getString(4);
        /* imprime */
        System.out.println(tipo + " -Catálogo: " + catalogo + ", Esquema: "
                + esquema + ", Nombre: " + tabla);
      }
    } catch (Exception ex) {
       System.err.println(ex);
    } finally {
      /* si llegó a rellenarse el RecordSet */
      if (result != null) {
        try {
          /* trata de liberar recursos */
          result.close();
        } catch (SQLException ex) {
        }
      }
      /* si llegó a establecerse la conexión */
      if (conexion != null) {
        try {
          /* trata de liberar recursos */
          conexion.close();
        } catch (SQLException ex) {
        }
      }
    }
  }
}
```

El método ***getTables()*** devuelve un objeto *ResultSet* que proporciona información sobre las tablas y vistas de la base de datos. Necesita 4 parámetros que en el ejemplo anterior tenían el valor *null*:

- **Parámetro 1**: **catálogo** de la base de datos. El método obtiene las tablas del catálogo indicado, al poner *null*, indicamos todos los catálogos (o catálogo actual dependiendo del SGBD)
- **Parámtro 2**: **esquema** de la base de datos. Obtiene las tablas del esquema indicado, el valor *null* indica el esquema actual (o todos los esquemas dependiendo del SGBD).
- **Parámetro 3:** es un patrón en el que se indica el nombre de las **tablas** que queremos que obtenga el método. Se puede utilizar el carácter guión bajo o porcentaje, por ejemplo *"de%"* obtendría todas las tablas cuyo nombre empiece por *"de"*.
- **Parámetro 4:** es un *array* de *String*, en el que indicamos qué **tipos** de tablas queremos: TABLE (para tablas), VIEW (para vistas); al poner ***null,*** nos devolverá todos los tipos ya sean tablas o vistas. El siguiente ejemplo nos devolvería sólo tablas:

```
String[] tipos = {"TABLE"};
resul= dbmd.getTables(null, null, null, tipos);
```

Cada fila de ***ResultSet*** que devuelve ***getTables()*** tiene información sobre una tabla. Las columnas de *ResultSet* que devuelve el método son:

1. TABLE_CAT. El nombre del catálogo al que pertenece la tabla.
2. TABLE_SCHEM. Nombre del esquema al que pertenece la tabla
3. TABLE_NAME. El nombre de la tabla o vista.
4. TABLE_TYPE. El tipo TABLE o VIEW
5. REMARKS. Comentarios

- .....

(si consultas la referencia oficial de ***getTables()*** verás la relación numerada de todas las columnas devueltas).

Para obtener los resultados del ejemplo también podríamos haber puesto en el código anterior:

```
String catalogo = result.getString ("TABLE_CAT");  // result.getString(1);
String esquema = result.getString ("TABLE_SCHEM"); // result.getString(2);
String tabla = result.getString ("TABLE_NAME");    // result.getString(3);
String tipo = result.getString ("TABLE_TYPE");     // result.getString(4);
```

Otros métodos importantes del objeto **DatabaseMetaData** son:

- ***getColumns****(catalogo, esquema, nombre_tabla, nombre_columna*). Devuelve información sobre las columnas de una tabla o tablas. Para el nombre de la tabla y de la columna se pueden utilizar los caracteres comodín _ y **%**. El valor null en los 4 parámetros indica que obtiene información de todas las columnas y tablas del esquema actual.

El siguiente ejemplo muestra información sobre todas las columnas de la tabla departamentos:

```
System.out.println("COLUMNAS TABLA DEPARTAMENTOS:");
System.out.println("============================");
/* obtiene las columnas de la tabla 'Departamentos' */
result = dbmd.getColumns(null, "empresaz", "departamentos", null);
/* mientras puede avanzar */
while (result.next()) {
  /* asigna valores */
  nombreCol = result.getString("COLUMN_NAME"); /* o .getString(4) */
  tipoCol = result.getString("TYPE_NAME"); /* o .getString(6) */
  tamCol = result.getString("COLUMN_SIZE"); /* o .getString(7) */
  nula = result.getString("IS_NULLABLE"); /* o .getString(18) */
  /* imprime */
  System.out.println(" Columna: " + nombreCol + ", Tipo: "
          + tipoCol + ", Tamaño: " + tamCol + ", ¿Puede ser Nula?: "
          + nula);
}
```

Visualiza:

```
COLUMNAS TABLA DEPARTAMENTOS:
============================
 Columna: dept_no, Tipo: TINYINT, Tamaño: 3, ¿Puede ser Nula?: NO
 Columna: dnombre, Tipo: VARCHAR, Tamaño: 15, ¿Puede ser Nula?: YES
 Columna: loc, Tipo: VARCHAR, Tamaño: 15, ¿Puede ser Nula?: YES
```

- ***getPrimaryKeys (****catalogo, esquema, tabla).* Devuelve la lista de columnas que forman la clave primaria.

El siguiente ejemplo muestra la clave primaria de la tabla “departamentos”

```
ResultSet result = dbmd.getPrimaryKeys(null, "empresaz", "departamentos");
pkDep = "";
separador = "";
while (result.next()) {
  pkDep += separador + result.getString("COLUMN_NAME"); /* o .getString(4) */
  separador = "+";
}
System.out.println("Clave primaria: " + pkDep);
```

- ***getExportedKeys*** *(catalogo, esquema, tabla).* Devuelve la lista de todas las claves ajenas que utilizan la clave primaria de esta tabla.

El siguiente ejemplo muestra las tablas y sus claves ajenas que referencian a la tabla “departamentos”, en este caso solo la tabla “empleados”

```
result = dbmd.getExportedKeys(null, "empresaz", "departamentos");
while (result.next()) {
  fk_name = result.getString("FKCOLUMN_NAME");
  pk_name = result.getString("PKCOLUMN_NAME");
  fk_table = result.getString("FKTABLE_NAME");
  pk_table = result.getString("PKTABLE_NAME");
  System.out.println("Clave ajena: " + fk_name + " en la tabla " + fk_table);
  System.out.println("Clave primaria: " + pk_name + " en la tabla " + pk_table);
}
```

- ***getImportedKeys*** *(catalogo, esquema, tabla).* Devuelve la lista de claves ajenas existentes en la tabla. Se utiliza igual que el método anterior, en este caso *dbmd.getImportedKeys(null, "empresaz", "departamentos")* no devuelve nada ya que no hay claves ajenas en "departamentos".

- ***getProcedures*** *(catalogo, esquema, procedure)*. Devuelve la lista de procedimientos almacenados.

El siguiente ejemplo mostraría los procedimientos y funciones que tiene el esquema de nombre "empresaz"

```
result = dbmd.getProcedures(null, "empresaz", null);
while (result.next()) {
  proc_name = result.getString("PROCEDURE_NAME");
  proc_type = result.getString("PROCEDURE_TYPE");
  System.out.println("Nombre Procedimiento: " + proc_name + " -Tipo "
          + proc_type);
}
```

Referencia oficial de DatabaseMetaData
http://docs.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html

5.3 ResultSetMetaData

Se pueden obtener metadatos (datos sobre datos) a partir de un ***ResultSet*** mediante la interfaz **ResultSetMetaData**; es decir podemos obtener más información sobre los tipos y propiedades de las columnas como por ejemplo el número de columnas devueltas.

El siguiente segmento de código muestra el uso de la interfaz para conocer más información acerca de las columnas devueltas por ***ResultSet***, en este caso desconocemos el nombre de las columnas devueltas por la consulta SELECT * FROM departamentos; el método ***getMetaData()*** del objeto ***ResultSet*** devuelve una referencia a un objeto ***ResultSetMetaData*** con el que se obtendría información acerca de las columnas devueltas.

```
Statement sentencia = conexion.createStatement();
ResultSet result = sentencia.executeQuery("SELECT * FROM departamentos");
ResultSetMetaData rsmd = result.getMetaData();
int nColumnas = rsmd.getColumnCount();
System.out.println("Número de columnas devueltas por la consulta: "
        + nColumnas);
for (int i = 1; i <= nColumnas; i++) {
  System.out.println("\nColumna " + i + ":");
  System.out.println("Nombre: " + rsmd.getColumnName(i));
  System.out.println("Tipo: " + rsmd.getColumnTypeName(i));

  if (rsmd.isNullable(i) == 0)
        nula = "NO";
  else nula = "SI";

  System.out.println("Puede ser nula?: " + nula);
  System.out.println("Máximo ancho de columna: " + rsmd.getColumnDisplaySize(i));
}
```

Métodos interesantes son:

- ***getColumnCount().*** Devuelve el número de columnas devueltas por la consulta.
- ***getColumnName****(índice columna)*. Devuelve el nombre de la columna.
- ***getColumnTypeNam****e(indice)*. Devuelve el nombre del tipo de dato que contiene la columna específico del sistema de bases de datos.
- ***isNullable****(indice).* Devuelve 0 si la columna puede contener valores nulos.
- ***getColumnDisplaySiz****e(indice)*. Devuelve el máximo ancho en caracteres de la columna.

Referencia oficial de ResultSetMetaData
http://docs.oracle.com/javase/6/docs/api/java/sql/ResultSetMetaData.html

6. Ejecución de sentencias de manipulación de datos.

Ya hemos comentado anteriormente que para manipulación de datos 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.

Y que el API JDBC distingue dos tipos de consultas:

- Consultas: **`SELECT`**
- Actualizaciones: **`INSERT`**, **`UPDATE`**, **`DELETE`**, sentencias DDL

6.1 Sentencias Statement

Statement es un interfaz que proporciona métodos para ejecutar sentencias SQL y obtener resultados. Al ser un interfaz no se pueden crear objetos directamente, en su lugar los objetos se obtienen con una llamada al método ***createStatement()*** de un objeto ***Connection*** válido.

```
Statement senctencia = conexión.createStatement();
```

Al crearse un objeto ***Statement*** se crea un espacio de trabajo para crear consultas SQL, ejecutarlas y recibir los resultados. Una vez creado el objeto se pueden usar los siguientes métodos:

- **executeQuery (String).** Se utiliza para sentencias SQL que recuperan datos de un único objeto *ResultSet,* se utiliza para las sentencias SELECT.
- **executeUpdate (String).** Se utiliza para sentencias que no devuelven un *ResultSe*t, como son las sentencias de manipulación de datos (DML): INSERT, UPDATE y DELETE; y las sentencias de definición de datos (DDL): CREATE, DROP y ALTER. El método devuelve un entero indicando el número de filas que se vieron afectadas y en el caso de las sentencias DDL devuelve el valor 0.
- **execute (String).** Se utiliza para sentencias que devuelven más de un *ResultSet*, como por ejemplo para ejecutar procedimientos almacenados. También se puede utilizar como en el caso anterior.

Referencia oficial Statement
http://docs.oracle.com/javase/6/docs/api/java/sql/Statement.html

A través de un objeto ***ResultSet*** se puede acceder al valor de cualquier columna de la fila actual por nombre o posición, también se puede obtener información sobre las columnas como el número de columnas o su tipo.

Referencia oficial ResultSet
http://docs.oracle.com/javase/6/docs/api/java/sql/ResultSet.html

Ejemplo. Insertar un departamento en la tabla departamentos. Los datos del nuevo departamento se introducen desde la línea de comandos.

```
try {
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    conexion = DriverManager.getConnection(url, user, password);
    /* crea el comando para enviar sentencias SQL */
    sentencia = conexion.createStatement();
    /* ejecuta la insercción del nuevo departamento y anota el resultado */
    int filasInsertadas = sentencia.executeUpdate(
            "INSERT INTO departamentos VALUES
               (" + dep + ",'" + dnombre + "','" + loc + "')");
    System.out.println("Filas afectadas: " + filasInsertadas);
    ......................
```

Ejemplo. Incrementar el salario de los empleados de un departamento. El incremento y el departamento se reciben desde la línea de comandos.

```
/* crea el comando para enviar sentencias SQL */
    sentencia = conexion.createStatement();
/* ejecuta el incremento del salario */
int filasModificadas = sentencia.executeUpdate(
        "UPDATE empleados SET salario=salario+"+ subida + " WHERE dept_no=" + depSub);
    System.out.println("Filas afectadas: " + filasModificadas);
    ......................
```

Ejemplo. Crea una vista de nombre totales que contiene por cada departamento el número, el nombre y el número de empleados y la suma de salarios.

```
/* crea el comando para enviar sentencias SQL */
    sentencia = conexion.createStatement();
/* crea una vista de nombre 'totales' */
int resultado = sentencia.executeUpdate(
        "CREATE OR REPLACE VIEW totales(dep,dnombre,nemp,media) AS "
        + "SELECT d.dept_no,dnombre,COUNT(emp_no),AVG(salario) "
        + "FROM departamentos d LEFT JOIN empleados e "
        + "ON e.dept_no=d.dept_no "
        + "GROUP BY d.dept_no,dnombre");
/* mensaje de confirmación */
System.out.println ("vista creada con resultado " + resultado);
```

Ejemplo. Consultar la vista anterior.

```
System.out.println ("CONSULTA A VISTA: \n Departamento, Nombre, "
              + "Total empleados, Salario medio \n");
//consulta a la vista o VIEW creada anteriormente
result=sentencia.executeQuery("SELECT * FROM totales ORDER BY nemp");
//procesamos o recorremos el ResultSet devuelto
while (result.next()) {
      System.out.printf("%2d  %-15s  %3d  %6.2f\n", result.getInt(1),
      result.getString("dnombre"), result.getInt(3), result.getFloat(4));
}
```

Ejemplo. Eliminar a un empleado. El número de empleado se introduce desde teclado.

```
/* crea el comando para enviar sentencias SQL */
    sentencia = conexion.createStatement();
/* ejecuta la sentencia de borrado */
   int filasAfectadas = sentencia.executeUpdate("DELETE FROM empleados
                                                 WHERE emp_no=" + numemp);
 System.out.println("Filas afectadas: " + filasAfectadas);
```

6.2 Sentencias preparadas. PreparedStatement

Las **consultas o sentencias preparadas** están representadas por la interface **PreparedStatement**.

Las sentencias preparadas de JDBC permiten la "precompilación" del código SQL antes de ser ejecutado, permitiendo **consultas o actualizaciones más eficientes**. En el momento de compilar la sentencia SQL, se analiza cuál es la estrategia adecuada según las tablas, las columnas, los índices y las condiciones de búsqueda implicados. Este proceso, obviamente, consume tiempo de procesador, pero al realizar la compilación una sola vez, se logra mejorar el rendimiento en siguientes consultas iguales con valores diferentes.

Para las consultas que se realizan muy a menudo es aconsejable usar este tipo de consultas, de modo que el rendimiento del sistema mejorará bastante.

La interfaz **PreparedStatement** nos va a permitir construir una cadena de caracteres SQL con ***marcadores de posición* (*placeHolder*)** que representarán los datos que serán asignados más tarde. El **marcador de posición** se representa mediante el signo de interrogación (**?**).

Ejemplo. Una sentencia INSERT sobre "departamentos" podría representarse así:

```
String sql="INSERT INTO departamentos VALUES (?, ?, ?)";
                                          //  1  2  3  valor del índice marcador posición
```

Cada marcador de posición (placeHolder) tiene un índice, el 1 corresponde al primero que se encuentre en la cadena, 2 al segundo y así sucesivamente. ***Solo se pueden utilizar para ocupar el sitio de los datos en la cadena SQL***, no se pueden usar para representar una columna o un nombre de una tabla, por ejemplo FROM ? sería incorrecto.

Antes de ejecutar un *PreparedStatement* es necesario asignar los datos para que cuando se ejecute la base de datos asigne variables de unión con estos datos y ejecute la orden SQL.

Los objetos *PreparedStatement* se pueden preparar o precompilar una sola vez y ejecutar las veces que queramos asignando diferentes valores a los marcadores de posición, en cambio en los objetos *Statement,* la sentencia SQL, se suministra en el momento de ejecutar la sentencia.

Los **métodos de *PreparedStatement*** tienen los mismos nombres que en Statement:

- **executeQuery()**
- **executeUpdate()**
- **execute()**

pero no se necesita enviar la cadena de caracteres con la orden SQL en la llamada, ya que lo hace el método ***prepareStatement(String):***

```
PreparedStatement pstmt = (PreparedStatement) conexión.prepareStatement(sql);
```

Los **pasos a seguir** cuando se utilizan consultas preparadas son:

1. Asignar a una sentencia preparada la consulta SQL. Se compila**.**
2. Establecer los valores de los parámetros mediante métodos **set().**
3. Ejecutar la consulta utilizando el método apropiado **executeQuery(), executeUpdate()** o **execute(),** todos ellos sin parámetros.

Ejemplo. Para realizar una consulta de los empleados del departamento 10, haríamos la consulta siguiente:

1. Asignar una consulta a una sentencia preparada

```
PreparedStatement pstmt = (PreparedStatement)conexion.prepareStatement("SELECT * FROM empleados WHERE dept_no = ? ");
```

2. Establecemos los parámetros de una consulta utilizando métodos **set** que dependen del tipo de dato SQL de la columna.

Así, indicamos que el primer parámetro, el único que tiene esta consulta, por ejemplo es el valor 10:

```
pstmt.setInt(1, 10);
```

3. Ejecutamos executeQuery()

```
ResultSet result = pstmt.executeQuery();
```

Ejemplo. El ejemplo de inserción de una fila en la tabla Departamentos quedaría así:

```
//construir una orden INSERT indicando los marcadores
      String sql= "INSERT INTO departamentos VALUES (?, ?, ?)";
      PreparedStatement pstmt = (PreparedStatement)conexion.prepareStatement(sql);

//establecemos los parámetros mediante métodos set
       pstmt.setInt(1, dep);                        //número departamento
      pstmt.setString(2, dnombre);                 //nombre
      pstmt.setString(3, loc);                    //localidad

//ejecutamos la consulta de actualización
      int filas = pstmt.executeUpdate();  //filas afectadas
```

Se utilizan los métodos **setInt**(indice, entero), **setString**(indice, cadena) para asignar los valores a cada uno de los marcadores de posición.

Existen métodos **set()** para los diferentes tipos de datos*: setDate(), setDouble(), setLong(), setTime(),* etc.

Ejemplo. El ejemplo de la modificación del salario de los empleados de cierto departamento quedaría así:

```
//construir orden UPDATE
    String sql = "UPDATE empleados SET salario=salario + ? WHERE dept_no = ?";

//compilar sentencia preparada
    PreparedStatement pstmt = (PreparedStatement) conexion.prepareStatement(sql);

//establecer marcadores de posición
    pstmt.setInt(2, dep);
    pstmt.setFloat(1, subida);

//ejecutamos la consulta de actualización
    int filas = pstmt.executeUpdate();  //filas afectadas

    System.out.println("Filas afectadas: " + filas);
```

Ejemplo. Mostramos el apellido y salario de los empleados de un departamento y oficio concreto, el departamento y oficio se introducen desde la línea de comandos al ejecutar el programa:

```
//construir orden SELECT
    sql="SELECT apellido, salario FROM empleados  "
            + "WHERE dept_no=? AND oficio = ? ORDER BY 1";
    //sentencia preparada
    pstmt = (PreparedStatement) conexion.prepareStatement(sql);

    //establecemos marcadores
    pstmt.setInt(1, dep);
    pstmt.setString(2, ofic);

    //ejecutamos consulta
    result=pstmt.executeQuery();
    //Recorremos el resultset
System.out.println("Empleados departamento " + dep + "oficio "+ ofic + "\n");
   while (result.next()) {
      System.out.printf("%-15s  %6.2f  \n", result.getString(1),
              result.getFloat("salario"));
    }
```

Los valores NULL se insertan en la base de datos usando **setNull().**

Referencia oficial PreparedStatement
http://docs.oracle.com/javase/6/docs/api/java/sql/PreparedStatement.html

7. Gestión de errores.

Cuando se produce una excepción SQL, **SQLException,** podemos acceder a cierta información usando los siguientes métodos:

- **getMessage():** devuelve una cadena que describe el error.
- **getSQLState():** es una cadena que contiene un estado definido por el estándar X/OPEN SQL
- **getErrorCode():** es un entero que proporciona el código de error del fabricante. Normalmente este será el código de error real devuelto por la base de datos.

Utilizando estos métodos de un objeto SQLException, podemos averiguar la causa exacta del error y personalizar el mensaje de información mostrado al usuario cuando intenta realizar una operación de actualización o consulta sobre la BD y no está permitida:

- Consulta de una tabla inexistente
- Claves primarias duplicadas
- Violación con las restricciones de integridad
- Etc.

Ejemplo.

```
try {
      //código

}catch (SQLException exq) {

    System.err.println("HA OCURRIDO UNA EXCEPCIÓN \n");
    System.out.println("Mensaje:  "+ exq.getMessage());
    System.out.println("SQL estado:  "+ exq.getSQLState());
    System.out.println("Código Error:  "+ exq.getErrorCode());

} catch (Exception ex) {

    System.err.println("Por Exception: " + ex);
}
```

Mostraría por ejemplo la siguiente salida:

```
HA OCURRIDO UNA EXCEPCIÓN
Mensaje:  Duplicate entry '63' for key 'PRIMARY'
SQL estado:  23000
Código Error:  1062
```

Podemos personalizar el mensaje para las claves primarias duplicadas, por ejemplo de la siguiente forma:

```
try {

      //código

}catch (SQLException exq) {
    System.err.println("ERROR \n");
    if (exq.getErrorCode() == 1062) {
      System.err.println("Departamento existente: "+ dep +",  no se pudo insertar ");

} catch (Exception ex) {
        System.err.println("Por Exception: " + ex);
}
```

De manera que al intentar insertar un departamento existente obtendremos la siguiente información:

```
ERROR
Departamento existente: 63,  no se pudo insertar
```

Puedes consultar los **códigos, mensajes y estado de error de MySQL** en el siguiente enlace:
https://dev.mysql.com/doc/refman/5.5/en/error-messages-server.html

Referencia oficial SQLException
http://docs.oracle.com/javase/6/docs/api/java/sql/SQLException.html

8. Ejecución de procedimientos almacenados.

Un **procedimiento almacenado** (***stored prodecure***) es un ***conjunto de sentencias, almacenadas en el servidor con un nombre determinado***, ***y que los programas cliente pueden ejecutar simplemente indicando el nombre del procedimiento y pasándole opcionalmente los parámetros necesarios***. En cierto modo, crear un procedimiento almacenado es algo así como añadir a SQL nuestras propias sentencias a medida, con el objetivo de llevar a cabo tareas más complejas.

Estos procedimientos suelen ser de dos clases:

- **Procedimientos** almacenados. Realizan una tarea concreta con o sin parámetros.
- **Funciones** almacenadas**.** Realizan una tarea con o sin parámetros y devuelven un valor que se puede emplear en otras sentencias SQL

El conjunto de sentencias almacenado en el servidor se encuentra enlazado (compilado) y optimizado para su ejecución repetida; lo que brinda grandes ventajas de rendimiento y una disminución importante de carga de tráfico en la red.

Además de optimizar el rendimiento del entorno, el uso de procedimientos almacenados permite incrementar las medidas de seguridad de un entorno de bases de datos, ya que elimina la necesidad de dar permisos directos a las tablas por parte de aplicaciones y clientes, y en su lugar se asignan permisos para ejecutar los procedimientos almacenados en el servidor.

En un esquema de seguridad máxima, se podría implementar un entorno en el que las únicas instrucciones que se ejecutan sobre las bases de datos son las definidas en los procedimientos almacenados.

Dependiendo del SGBD, los procedimientos almacenados se pueden poner en ejecución desde diferentes lenguajes de programación, por ejemplo PHP, Java, C, C++, C#, etc.

El estándar SQL establece la sintaxis de las sentencias que hay que utilizar para definir procedimientos o funciones almacenados, **una sintaxis que es implementada con notables diferencias en cada SGDBR.**

Algunos de ellos, como ocurre con las versiones de MySQL previas a la 5.0 o Access, ni siquiera contemplan el concepto de procedimiento almacenado, si bien es posible escribir funciones en lenguajes externos a SQL y utilizarlas para manipular los datos.

Informix, Oracle, MySQL 5.x y SQL Server, por el contrario, se ajustan bastante al estándar, si bien cada uno de ellos cuenta con un **lenguaje específico para estas tareas:**

- **PL/SQL** en el caso de **Oracle**,
- **SQL/PSM** en el caso de **MySQL**,
- **Transact-SQL** en el caso de **SQL Server**,
- **Informix 4GL** en el caso de **Informix**, etc...

Se trata pues de ***lenguajes creados para dar nuevas posibilidades a SQL***. Esas posibilidades permiten utilizar condiciones y bucles al estilo de los lenguajes de tercera generación.

Un **procedimiento almacenado típico** tiene:
- un nombre,
- una lista de parámetros
- sentencias SQL
- sentencias condicionales y/o repetitivas extensión de SQL
- declaración de variables

En el caso de MySQL:
- los procedimientos almacenados pueden definirse con parámetros de entrada (IN), de salida (OUT), de entrada/salida (INOUT) o sin ningún parámetro.
- Las funciones almacenadas sólo admiten parámetros de entrada IN.
- Cuando se omite el tipo de parámetro, se supone IN.

DEBES CONOCER

En el capítulo 19 del manual de referencia de MySQL que puedes encontrar en el enlace siguiente, puedes familiarizarte con los comandos que puedes necesitar para realizar procedimientos almacenados y funciones en MySQL:

http://dev.mysql.com/doc/refman/5.0/es/stored-procedures.html

EJEMPLO. Procedimiento que sube el sueldo a los empleados de un departamento. El procedimiento recibe dos parámetros de entrada que son el número de departamento y el % de subida como valor entero.

```
DELIMITER //
CREATE PROCEDURE subida_salario (dep INT, subida INT)
BEGIN
UPDATE empleados
SET salario=salario + (salario*subida/100)
WHERE dept_no=dep;
END //
DELIMITER ;
```

Para invocar al procedimiento basta poner su nombre y pasar dos parámetros de entrada que representen al departamento y el % de subida en formato entero.

```
subida_salario(10, 5);  //ejecutar el procedimiento para el departamento 10 con subida del 5%
```

La invocación desde un cliente gráfico de MySQL como Workbench, sería de la siguiente forma:

```
CALL subida_salario(10,5);
```

EJEMPLO. Procedimiento almacenado que recibe el número de un departamento y muestra los datos de los empleados de ese departamento. También calcula el total de empleados y deja ese total en un parámetro de salida.

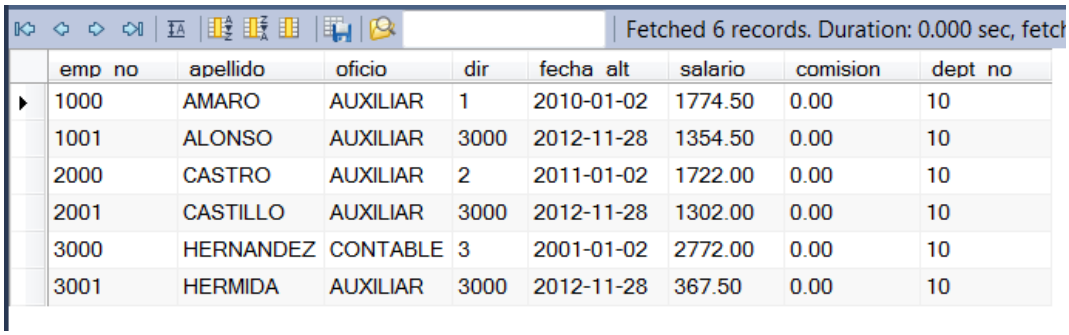
```
DELIMITER //
CREATE PROCEDURE  emple_depar (IN dep INT, OUT cuenta INT)
BEGIN
SELECT * FROM empleados
WHERE dept_no=dep;

SELECT COUNT(*) INTO cuenta
FROM empleados
WHERE dept_no=dep;
END //
DELIMITER ;
```

La invocación desde un cliente gráfico de MySQL como Workbench, sería de la siguiente forma:

```
CALL emple_depar(10, @total);  /* Ejecuta el procedimiento. @total variable para el parámetro de salida*/
```

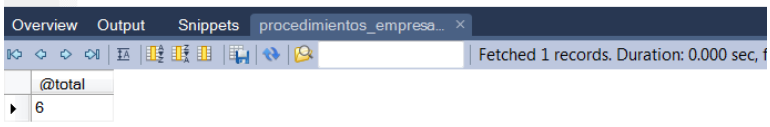
Produciendo el resultado siguiente:



emp_no	apellido	oficio	dir	fecha_alt	salario	comision	dept_no
1000	AMARO	AUXILIAR	1	2010-01-02	1774.50	0.00	10
1001	ALONSO	AUXILIAR	3000	2012-11-28	1354.50	0.00	10
2000	CASTRO	AUXILIAR	2	2011-01-02	1722.00	0.00	10
2001	CASTILLO	AUXILIAR	3000	2012-11-28	1302.00	0.00	10
3000	HERNANDEZ	CONTABLE	3	2001-01-02	2772.00	0.00	10
3001	HERMIDA	AUXILIAR	3000	2012-11-28	367.50	0.00	10

Para recuperar el valor del parámetro de salida desde un cliente de MySQL se consulta la variable @total:

```
SELECT @total;  /* Consulta el valor de la variable @total */
```

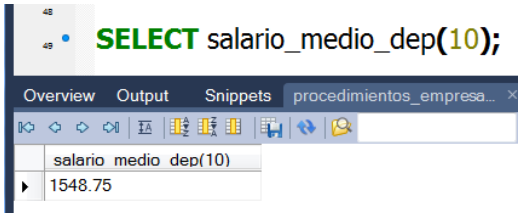


@total
6

EJEMPLO. Función que devuelve el total de empleados de un departamento. La función recibe el código o número del departamento.

```
DELIMITER //
CREATE FUNCTION salario_medio_dep (dep INT)
RETURNS DOUBLE
DETERMINISTIC
BEGIN
DECLARE salario_medio  DOUBLE;
SELECT AVG (salario) INTO salario_medio
FROM empleados
WHERE dept_no=dep;
RETURN salario_medio;
END //
DELIMITER ;
```

La ejecución de la función desde un cliente MySQL sería:



Los permisos asociados a las distintas operaciones que se pueden realizar con procedimientos almacenados son: CREATE ROUTINE (creación), EXECUTE (ejecución), ALTER ROUTINE (borrado).

8.1 Sentencias CallableStatement

La interfaz **CallableStatement,** que hereda de PreparedStatement**,** permite que se puedan llamar desde Java a los procedimientos almacenados.

Para crear un objeto de este tipo se llama al método **prepareCall(String)** de un objeto Connection.

El siguiente ejemplo declara la llamada al procedimiento subida_salario que tiene dos parámetros y para indicar los parámetros se utilizan marcadores de posición (?) numerados internamente desde 1 en adelante y que se corresponden en el mismo orden con los parámetros del procedimiento.

```
String sql= "{call subida_salario (?, ?) }";
CallableStatement procAlmacenado = conexión.prepareCall(sql);
```

Hay 4 **formas de declarar las llamadas a los procedimientos y las funciones** que dependen del uso u omisión de parámetros, y de la devolución de valores. Son las siguientes:

- **{call procedimiento}:** para un procedimiento almacenado sin parámetros.
- **{?= call función}**: para una función almacenada que devuelve un valor y no recibe parámetros, el valor se recibe a la izquierda del igual y es el primer parámetro.
- **{call procedimiento (?, ?, ….)}**: para un procedimiento almacenado que recibe parámetros.
- **{?=call función(?, ?, …..)}**: para una función almacenada que devuelve un valor (primer parámetro) y recibe varios parámetros.

Otras consideraciones a tener en cuenta al utilizar CallableStatement:

- Los **valores de los parámetros de entrada** se establecen mediante métodos **set()** heredados de *PreparedStatement.*
- Los **tipos de los parámetros de salida y valores devueltos** deben ser **registrados** antes de ejecutar el procedimiento almacenado, y después obtener su valor mediante métodos **get().**
- Una sentencia CallableStatement puede devolver uno o más objetos **ResultSet**, que habrá que procesar adecuadamente.
- Para ejecutar una sentencia CallableStatement() se usará el método **execute().**
- El método **execute**() devuelve un *booleano* para indicar la forma del primer resultado.
  - Devuelve true si el primer resultado es un objeto ResultSet, y false si el primer resultado es un recuento de actualizaciones o si no hay resultado.
- Hay que llamar **getResultSet()** o **getUpdateCount()** para recuperar el resultado. Si hay más resultados se debe llamar a **getMoreResults()** para desplazarse a cualquier resultado subsiguiente.

Referencia Oficial CallableStatement
http://docs.oracle.com/javase/6/docs/api/java/sql/CallableStatement.html

Los **pasos a seguir cuando solo hay parámetros de entrada**:

1. Asignar a una ***CallableStatement*** la llama al procedimiento /función almacenado**.**
2. Establecer los valores de los parámetros mediante métodos **set().**
3. Ejecutar la llamada con el método **execute()** sin parámetros.

EJEMPLO. Llamada al procedimiento subida_salario(). Los valores de los parámetros se introducen al ejecutar el programa desde la línea de comandos (en el ejemplo se asigna su valor dentro del programa).

Ud3_CallableStatement.java

```
import java.sql.*;
public class Ud3_CallableStatement {
  //valores de las variables con los posibles datos de entrada
  static  int depar=10, subida=5;
  /* parámetros de conexión con la base de datos MySql */
  private static final String url = "jdbc:mysql://localhost/empresaz";
  private static final String user = "root";
  private static final String password = "";
  public static void main(String[] args) {
    /* objeto de conexión */
    Connection conexion = null;
    try {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      conexion = DriverManager.getConnection(url, user, password);
      /*construir la orden de llamada a procAlmacenado */
      String sql="{call subida_salario (?,?)}";
      /*1. asignar la llamada al procAlmacenado a una CallableStatement*/
      CallableStatement procAlmacenado = conexion.prepareCall(sql);
      /*2. establecer el valor de los parámetros del procedimiento almacenado*/
      procAlmacenado.setInt(1, depar);  //primer parámetro
      procAlmacenado.setInt(2, subida);  //segundo parámetro
      /*3. ejecutar el procedimiento almacenado*/
      procAlmacenado.execute();
      System.out.println("Subida de sueldo realizada al departamento "+ depar);
           /*excepciones*/
    } catch (Exception ex) {
      ex.printStackTrace();
    }finally{
     /*librerar recursos*/
     try {
        conexion.close();
      } catch (SQLException exq) {
        exq.printStackTrace();
      }
    }
  }
}
```

Cuando un **procedimiento tiene parámetros de salida (OUT)** deben ser registrados antes de que la llamada tenga lugar, si no se registra se producirá un error.

- El método que se utilizará es**: registerOutParameter(indice, tipoJDBC),** el primer parámetro es la posición y el siguiente es una constante definida en la clase **java.sql.Types**.
- La **clase jav.sql.Types** define una constante para cada tipo genérico SQL, algunas son: TINYINT, SMALLINT, INTEGER, FLOAT, REAL, DOUBLE, NUMERIC, CHAR, VARCHAR, DATE, TIME, ARRAY, BOOLEAN, etc.

Por ejemplo, si el segundo parámetro de un procedimiento es OUT y de tipo VARCHAR, en el programa Java se pondría:

```
procAlmacenado.registerOutParameter(2, java.sql.Types.VARCHAR);
```

Uma vez ejecutada la llamada al procedimiento, los valores de los parámetros OUT e INOUT se obtienen en los métodos **getXXX(indice)** similares a los utilizados para obtener los valores de las columnas en un **Resultset**.

EJEMPLO. El siguiente ejemplo ejecuta el procedimiento emple_depar (IN dep INT, OUT cuenta INT). Recuerda que este procedimiento al ejecutarse, también devuelve un ResultSet.

Ud3_CallableStatementOUT.java

```
import java.sql.*;
public class Ud3_CallableStatementOUT {
 //valores de las variables
  static  int depar=10, totEmple=0;
  /* parámetros de conexión con la base de datos MySql */
  private static final String url = "jdbc:mysql://localhost/empresaz";
  private static final String user = "root";
  private static final String password = "";

  public static void main(String[] args) {
    /* objeto de conexión */
    Connection conexion = null;
    /* objeto ResultSet para recoger los registros devueltos */
    ResultSet result = null;
    try {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      /* abre la conexión */
      conexion = DriverManager.getConnection(url, user, password);
      /*construir la orden de llamada al procAlmacenado*/
      String sql="{call emple_depar (?,?)}";

      /* 1. asignar la llamada al procAlmacenado a una CallableStatement*/
      CallableStatement procAlmacenado = conexion.prepareCall(sql);

      /*2. establecer el valor de los parámetros del procedimiento almacenado*/
      procAlmacenado.setInt(1, depar);  //primer parámetro

      /*3. registro del parámetro de salida OUT*/
      procAlmacenado.registerOutParameter(2, Types.INTEGER);

      /*4. ejecutar el procedimiento almacenado*/
      procAlmacenado.execute();

      /*5. Obtener el valor de los parámetros de salida*/
      totEmple=procAlmacenado.getInt(2);
      System.out.println("Total de empleados departamento "+ depar +" son: " + totEmple);

      /*6 Obtener el ResultSet devuelto*/
      result=procAlmacenado.getResultSet();

      /*7. Procesar el resultset*/
       System.out.println("\nNumEmp, apellido,     oficio,          salario ");
      while (result.next()) {
        System.out.printf("%4d  %-15s %-15s %6.2f  \n", result.getInt(1),
         result.getString(2), result.getString(3) ,result.getFloat("salario"));
      }
```

```
      /*excepciones*/
    } catch (Exception ex) {
      ex.printStackTrace();
    }finally{
     /*librerar recursos*/
     try {
        conexion.close();
      } catch (SQLException exq) {
        exq.printStackTrace();
      }
    }
  }

}
```

En el siguiente enlace puedes ver otros **ejemplos de llamadas a procedimientos almacenados MySQL desde Java**.
http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=usando_CallableStatements_to_execute_stored_procedure

El usuario debe tener los permisos apropiados para poder ejecutar procedimientos almacenados.

En MySQL, la tabla **mysql.proc** contiene la información sobre todos los procedimientos almacenados en la base de datos. Al usuario que se desee que pueda ejecutar procedimientos almacenados se le debe dar permiso SELECT sobre la tabla anterior.

También es necesario incluir en la conexión el parámetro noAccessToProcedureBodies con el valor true:

"jdbc:mysql://localhost/empresaz?noAccessToProcedureBodies=true".

9. Acceso a datos mediante ODBC.

ODBC (Open Database Connectivity)es un estándar de acceso a datos desarrollado por Microsoft Corporation con el objetivo de posibilitar el acceso a cualquier dato desde cualquier aplicación, sin importar que sistema gestor de bases de datos almacene los datos.

Cada sistema de base de datos compatible con ODBC proporciona una biblioteca que se debe enlazar con el programa cliente. Cuando el programa cliente realiza una llamada a la API ODBC el código de la biblioteca se comunica con el servidor para realizar la acción solicitada y obtener los resultados.

Los **pasos para usar ODBC** son:
1. Configurar la interfaz ODBC, para ello el programa asigna en primer lugar un entorno SQL con la función SQLAllocHandle(), después un manejador (o handle) para la conexión a la base de datos basada en el entorno anterior. El propósito de este manejador es traducir las consultas de datos de la aplicación en comandos que el sistema de base de datos entienda.

ODBC define varios tipos de manejadores:

- SQLHENV: define el entorno de acceso a los datos.
- SQLHDBC: identifica el estado y configuración de la conexión (driver y origen de datos)
- SQLHSTMT: declaración SQL y cualquier conjunto de resultados
- SQLHDESC: recolección de metadatos utilizados para describir una sentencia SQL.

2. Abrir la conexión. Una vez reservados los manejadores el programa abre la conexión a la base de datos usando SQLDriverConnect() o SQLConnect().
3. Envío de órdenes SQL. Una vez realizada la conexión el programa puede enviar órdenes SQL a la base de datos usando SQLExecDirect().
4. Desconexión. Al final de la sesión el programa se desconecta de la base de datos y libera la conexión y los manejadores del entorno SQL.

La API ODBC usa una interface escrita en lenguaje C y no es apropiada para su uso directo desde Java. Las llamadas desde Java a código C nativo tiene un número de inconvenientes en la seguridad, implementación, robustez y portabilidad de las aplicaciones.

ODBC mezcla características elementales con otras más avanzadas y tiene complejas opciones incluso para las consultas más simples, por lo que se hace duro de aprender.

Algunas funciones importantes son:

- ***SQLAllocHandle, SQLDriverConnect***: necesarias para establecer la conexión con la base de datos.
- ***SQLAllocStmt, SQLExecDirect***: ejecutan sentencias SQL sobre la base de datos.
- ***SQLFetch, SQLGetData, SQLNumResultCols, SQLRowCount***: obtienen datos de la consulta SQL, como los resultados de una consulta, número de filas o de columnas.
- ***SQLDisconnect, SQLFreeHandle***: operaciones de liberación de memoria y desconexión a la base de datos.

9.1 Acceso a Datos mediante el puente JDBC-ODBC

Hay productos (aunque muy pocos) para los que no hay controlador (o driver) JDBC pero si hay un controlador ODBC. En estos casos se utiliza un puente denominado normalmente JDBC-ODBC Bridge.

El **puente JDBC-ODBC** es un controlador JDBC que implementa operaciones JDBC traduciéndolas en operaciones ODBC.

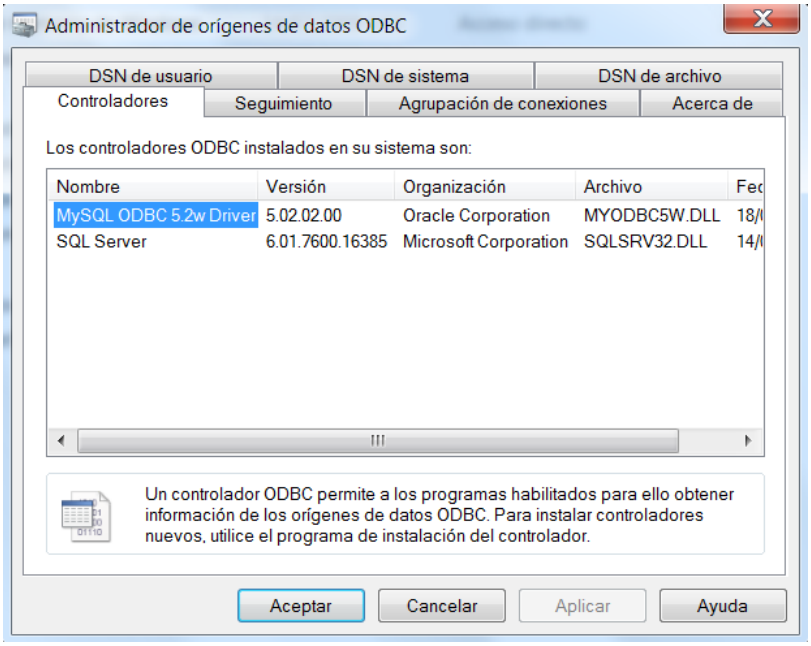
- Está implementado en Java y usa métodos nativos de Java para llamar a ODBC
- Se instala automáticamente con el JDK como el paquete sun.jdbc.odbc (por lo que no es necesario añadir ningún JAR a nuestros proyectos para trabajar con él).

Para acceder **a una base de datos MySQL usando JDBC-ODBC** necesitamos:

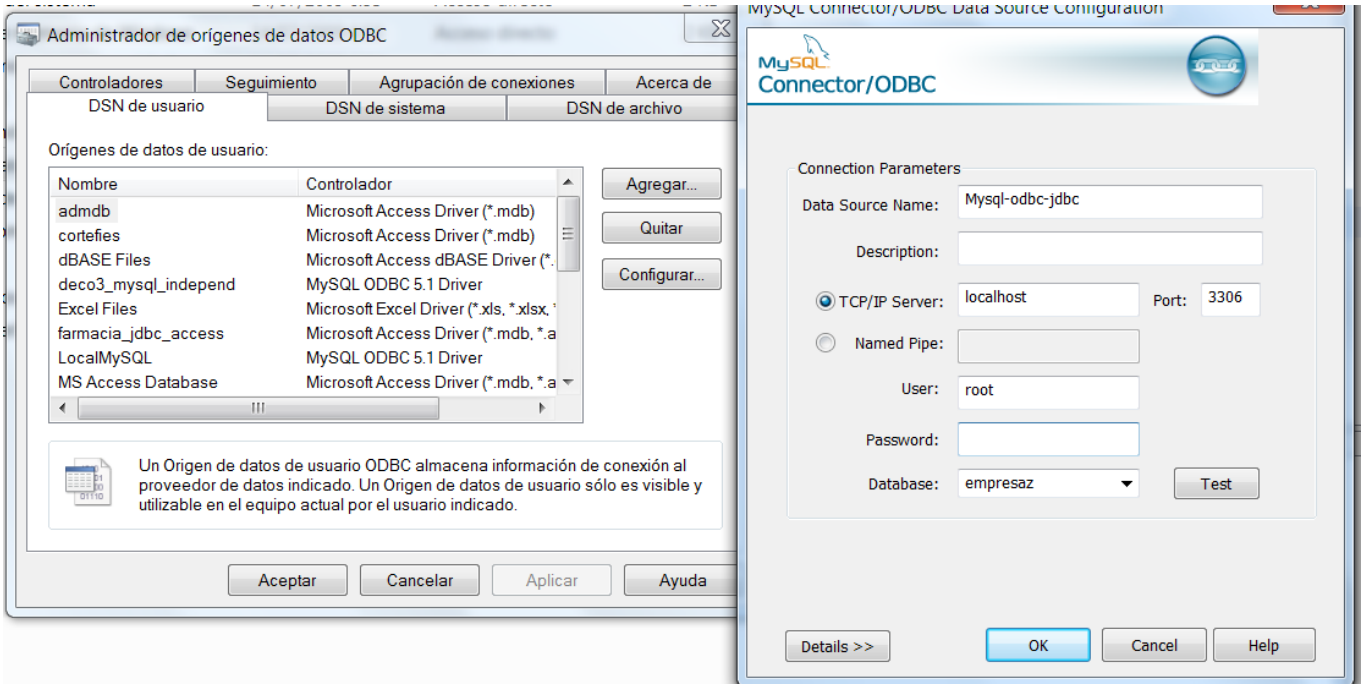
- Tener instalado el driver ODBC para conectar a la base de datos MYSQL. (Descargar *desde la web de MySQL el ODBC Driver for MySQL – Connector/ODBC e instalarlo)*

http://dev.mysql.com/downloads/connector/odbc/

- *Una vez instalado aparecerá en los orígenes de datos ODBC del sistema.(Panel de control→Herrameintas Administrativas→Orígenes de datos (ODBC)*



- Crear un origen de datos o DSN (Data Source Name). (*Desde Panel de control→Herrameintas Administrativas→Orígenes de datos (ODBC).*
    - Pulsamos el botón Agregar
    - Seleccionamos el driver ODBC para MySQL
    - Finalizar
    - En la siguiente pantalla:
        - damos un nombre la DSN: Mysql-odbc-jdbc
        - TCP/IP Server: localhost
        - Puerto: 3306
        - User: root
        - Password
        - Database: empresaz



Para cada esquema de la base de datos será necesario crear un origen de datos.

EJEMPLO. Consulta a la base de datos 'empresaz' mediante ODBC-JDBC. Consulta de los empleados con mayor salario.

Ud3_odbc_jdbc.jav

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
/**
 * conexión ODBC-JDBC(no se necesita ningua biblioteca adicional: sólo
 * hay que configurar un DSN de usuario 'Mysql-odbc-jdbc' de tipo MySql, a la
 * base de datos empresaz en el servidor localhost, para el usuario 'root' de
 * contraseña '').
 */
public class Ud3_odbc_jdbc {
  /* sentencia de consulta de los empleados en el departamento 10 */
  private static final String consultaEmpleadosDep10 =
          "SELECT apellido,oficio,salario FROM empleados WHERE dept_no=10";
  /* sentencia de consulta del salario máximo */
  private static final String consultaSalarioMaximo =
          "SELECT apellido,salario FROM empleados "
          + "WHERE salario IN (SELECT MAX(salario) FROM empleados)";
    public static void main(String[] args) {
    /* objeto de conexión */
    Connection conexion = null;
    /* comando para enviar sentencias a través de la conexión */
    Statement sentencia = null;
    /* objeto ResultSet para recoger los registros devueltos por la consulta */
    ResultSet result = null;
    try {
      /* carga el driver ODBC */
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
      /* abre la conexión hacia el DSN de usuario Mysql-odbc-jdbc de tipo MySql */
      conexion = DriverManager.getConnection("jdbc:odbc:Mysql-odbc-jdbc");
      /* crea el comando para enviar sentencias SQL */
      sentencia = conexion.createStatement();
      /* ejecuta la consulta y recoge el resultado */
      result = sentencia.executeQuery(consultaEmpleadosDep10);
      /* mientras puede avanzar*/
      while (result.next()) {
        /* recupera los datos por su ordinal en la consulta o por el nombre
         * de la columna, y los muestra en la Salida debidamente tabulados */
        System.out.printf("%-10s  %-10s  %6.2f\n", result.getString(1),
                result.getString("oficio"), result.getFloat(3));
      }
      /* ejecuta la consulta y recoge el resultado */
      result = sentencia.executeQuery(consultaSalarioMaximo);
      /* avanza al primer registro (que por como están ordenados, será¡
       * el se salario mÃ¡ximo) */
      result.next();
      /* muestra el resultado */
      System.out.printf("\nEmpleado de salario máximo:\n%s (%6.2f)\n",
              result.getString(1), result.getFloat(2));
      /* mensaje de confirmación */
      System.out.println("\nConsultas ejecutadas correctamente");
    } catch (Exception ex) {
      /* mensaje de error */
      System.err.println(ex);
    } finally {
      /* si llegó a rellenarse el RecordSet */
      if (result != null) {
        try {
          /* trata de liberar recursos */
          result.close();
        } catch (SQLException ex) {
        }
      }
```

```
      /* si llegó a crearse el comando */
      if (sentencia != null) {
        try {
          /* trata de liberar recursos */
          sentencia.close();
        } catch (SQLException ex) {
        }
      }
      /* si llegó a establecerse la conexión */
      if (conexion != null) {
        try {
          /* trata de liberar recursos */
          conexion.close();
        } catch (SQLException ex) {
        }
      }
    }
  }
}
```

10. Bibliografía.

- Acceso a Datos , Editorial Garceta
- Java, cómo programar, Edición 9, Editorial DEITEL
- http://docs.oracle.com/javase/6/docs/api/
- http://dev.mysql.com/doc/refman/5.0/es/index.html