

Unidad didáctica 17 Conector JDBC

OBJETIVOS

El alumno al término de esta unidad debe ser capaz de:

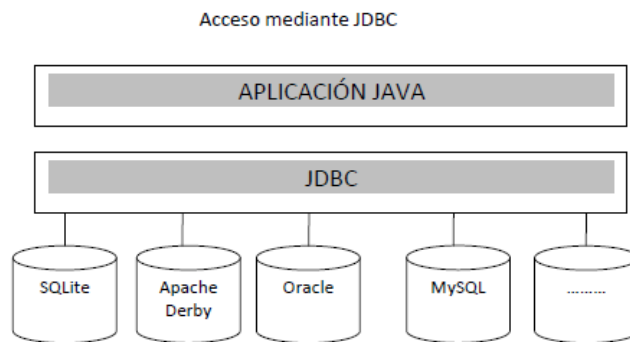
- Establecer conexiones a bases de datos MySQL
- Desarrollar aplicaciones para acceder a los datos de una base de datos
- Ejecutar procedimientos de bases de datos

Tabla de contenido

ACCESO A DATOS MEDIANTE JDBC	2
Arquitectura JDBC.....	2
Cómo Funciona JDBC	3
EJEMPLO	5
la Conexion.....	5
Cargar el driver:	6
Establecer la conexión:	7
Ejecutar sentencias SQL.....	7
Liberar recursos	9
Sentencias Preparadas. Uso de PreparedStatement con Java y MySQL	9
PreparedStatement desde java. Establecer la conexión.	11
EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS	13
EJECUCIÓN DE PROCEDIMIENTOS.....	18
EJECUCIÓN DE SENTENCIAS DE DESCRIPCIÓN DE DATOS	25
ResultSetMetaData.....	32
GESTION DE ERRORES.....	34

ACCESO A DATOS MEDIANTE JDBC

JDBC proporciona una librería estándar para acceder a fuentes de datos principalmente orientados a bases de datos relacionales que usan SQL. No solo provee una interfaz sino que también define una arquitectura estándar, para que los fabricantes puedan crear los drivers que permitan a las aplicaciones Java el acceso a los datos. **JDBC** dispone de una interfaz distinta para cada base de datos, llamado **driver** (controlador o conector). Esto permite que las llamadas a los métodos Java de las clases JDBC se correspondan con el API de la base de datos.



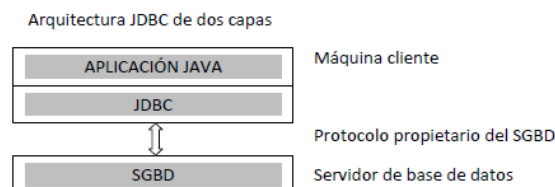
JDBC consta de un conjunto de clases e interfaces que nos permiten escribir aplicaciones Java para gestionar las siguientes tareas con una base de datos relacional:

- Conectarse a la base de datos
- Enviar consultas e instrucciones de actualización a la base de datos
- Recuperar y procesar los resultados recibidos de la base de datos en respuesta a las consultas.

Arquitectura JDBC

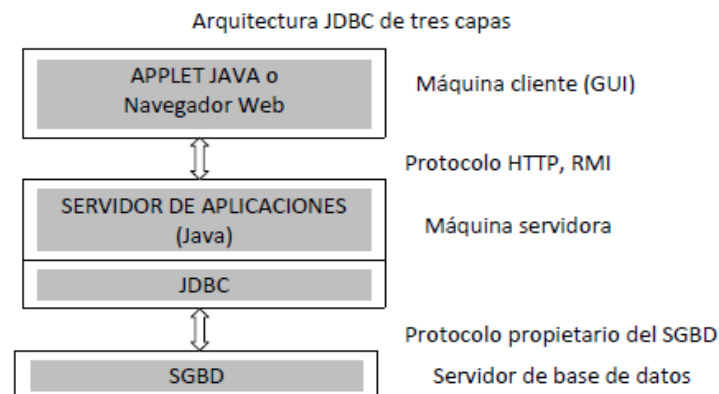
La API JDBC es compatible con los modelos tanto de dos como de tres capas para el acceso a las bases de datos.

En el **modelo de dos capas**, un applet o una aplicación Java “hablan” directamente con la base de datos, esto requiere un driver JDBC residiendo en el mismo lugar que la aplicación. Desde el programa Java se envían sentencias SQL al sistema gestor de base de datos para que los procese y los resultados se envían de vuelta al programa. La base de datos puede encontrarse en otra máquina diferente a la de la aplicación y las solicitudes se hacen a través de la red (arquitectura cliente-servidor). El driver será el encargado de manejar la comunicación a través de la red de forma transparente al programa.



En el **modelo de tres capas**, los comandos se envían a una capa intermedia que se encargará de enviar los comandos SQL a la base de datos y de recoger los resultados de la ejecución de las sentencias. Es decir, tenemos una aplicación o applet corriendo en una máquina y

accediendo a un driver de la base de datos situado en otra máquina. En este caso los drivers no tienen que residir en la máquina cliente.



Existen 4 tipos de conectores (drivers o controladores) JDBC:

1º) **JDBC-ODBC Bridge** (*JDBC-ODBC bridge plus ODBC driver*): permite el acceso a bases de datos JDBC mediante un driver ODBC. Exige la instalación y configuración de ODBC en la máquina cliente.

2º) **Native** (*Native-API partly-Java driver*): controlador escrito parcialmente en Java y en código nativo. Traduce las llamadas al API de JDBC Java en llamadas propias del motor de base de datos. Exige instalar en la máquina cliente código binario propio del cliente de base de datos y del sistema operativo.

3º) **Network** (*JDBC-Net pure Java driver*): controlador de Java puro que utiliza un protocolo de red (por ejemplo, HTTP) 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.

4º) **Thin** (*Native-protocol pure Java driver*): controlador de Java puro con protocolo nativo. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red usado por el motor de base de datos. No exige instalación en cliente.

Los tipos 3 y 4 son la mejor forma para acceder a bases de datos JDBC. 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 es ODBC (es decir, no existen drivers disponibles para el SGBD); pero exigen instalación de software en el puesto cliente. En la mayoría de los casos la opción más adecuada será el tipo 4.

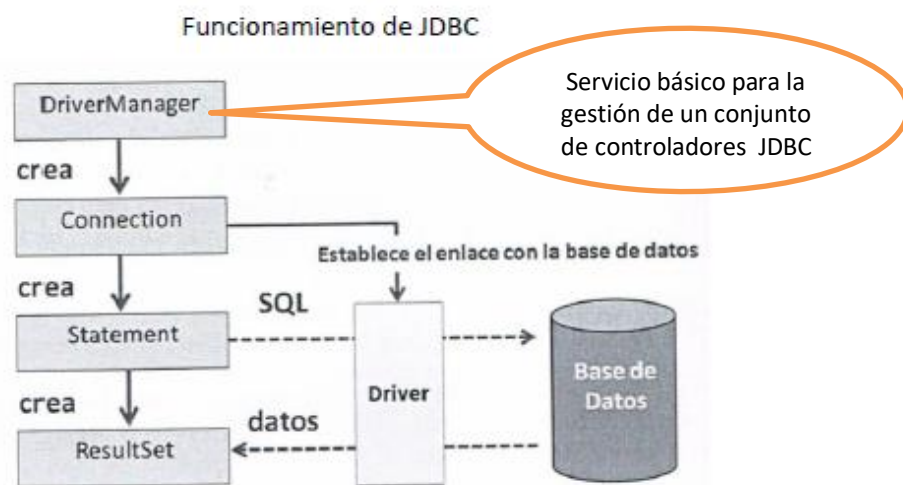
Cómo Funciona JDBC

JDBC define varias interfaces que permite 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 E INTERFACE	DESCRIPCIÓN
Driver	Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver 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. Una aplicación puede tener más de una conexión.
DatabaseMetadata	Proporciona información acerca de una Base de datos, como las tablas que contiene, etc.
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 orden SELECT.
ResultSetMetadata	Permite obtener información sobre un ResultSet, como el número de columnas, sus nombres, etc.
http://docs.oracle.com/javase/7/docs/api/java/sql/package-summary.html	

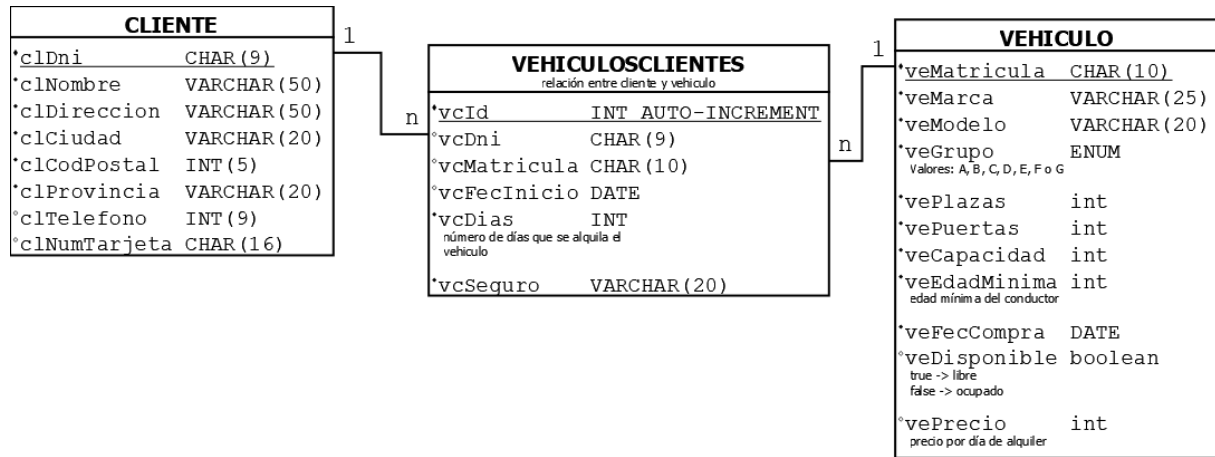
El trabajo con JDBC comienza con la clase DriverManager que es la encargada de establecer las conexiones con los orígenes de datos a través de los drivers JDBC. El funcionamiento de un programa con JDBC requiere los siguientes pasos:

- 1º) Importar las clases necesarias.
- 2º) Cargar el driver JDBC.
- 3º) Identificar el origen de datos.
- 4º) Crear un objeto Connection.
- 5º) Crear un objeto Statement.
- 6º) Ejecutar una consulta con el objeto Statement.
- 7º) Recuperar los datos del objeto ResultSet.
- 8º) Liberar el objeto ResultSet.
- 9º) Liberar el objeto Statement.
- 10º) Liberar el objeto Connection.



EJEMPLO

Para el siguiente ejemplo JAVA, creamos la base de datos MySQL **BDAquilerVehiculos** y un usuario con nombre **PrimeroDAM**, la clave de usuario es la misma. Este usuario tendrá todos los privilegios sobre esta base de datos. A continuación crearemos las tablas e insertaremos datos en ellas. El diagrama relacional es el siguiente:

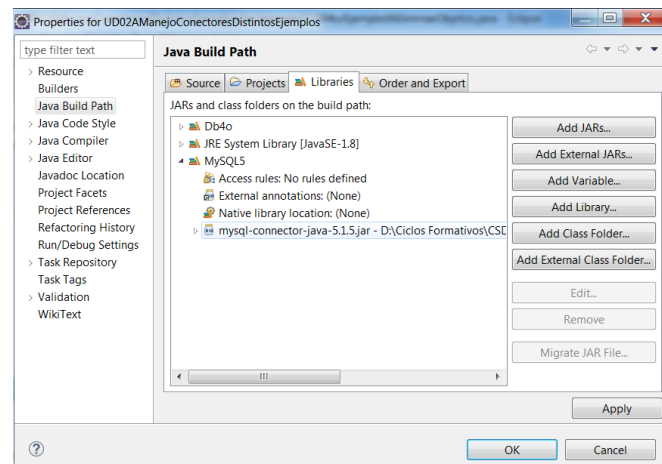


Las sentencias SQL para crear la base de datos y el usuario están en DROPBOX con el nombre BDEmpleadosUD17.sql.

El siguiente ejemplo muestra el funcionamiento de JDBC accediendo a la base de datos BDEmpleados, para que funcione necesitamos el fichero JAR que contiene el driver MySQL. Para añadirlo en el IDE de Eclipse pulsamos en el proyecto con el botón derecho del ratón seleccionamos:

Build Paths -> Add External JARs... para localizar el fichero JAR.

Desde la web <http://www.mysql.com/products/connector/> lo podemos descargar:



LA CONEXION

En Eclipse necesitamos crearnos una Conexión con nuestra Base de datos, para ello vamos a crear una clase Conexión cuyo código podría ser:

```
package conexion;
```

```
import java.sql.Connection;
import java.sql.DriverManager;
```

```

import java.sql.SQLException;

public class Conexion {

    String bd = "BDAlquilerVehiculos";
    String user = "PrimerODAM";
    String password = "PRIMERODAM";

    String url = "jdbc:mysql://127.0.0.1/"+bd+
        "?serverTimezone=Europe/Madrid&useSSL=false";

    Connection conexion = null;

    public Conexion() {
        //obtener el driver
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            //obtener la conexion
            conexion = DriverManager.getConnection(url, user,
password);
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public Connection getConexion() {
        return conexion;
    }

    public void desconectar() {
        conexion = null;
    }

}

```

Cargar el driver:

En primer lugar se carga el driver, con el método **forName()** de la clase **Class**, se le pasa un objeto **String** con el nombre de la clase del driver como argumento. En el ejemplo, como se accede a una base de datos MySQL necesitaremos cargar el driver **com.mysql.cj.jdbc.Driver**:

```

// Código para cargar el Driver de la Base de Datos Conector MySQL 5
Class.forName("com.mysql.cj.jdbc.Driver");

```

Establecer la conexión:

A continuación se establece la conexión con la base de datos, el servidor MySQL debe estar arrancado, usamos la clase **DriverManager** con el método **getConnection()** al que le tenemos que pasar 3 parámetros: la dirección de la base de datos, el usuario y el password:

```
// Configurar parametros de conexion
//Connection crea una conexión con una base de datos
String user = "PrimeroDAM";
String password = " PrimeroDAM "; // la contraseña para ejecutar
MySQL
String url = "jdbc:mysql://127.0.0.1/"+bd+
            "?serverTimezone=Europe/Madrid&useSSL=false";

//Establecemos la conexión
//DriverManger El servicio básico para la gestión de un conjunto de
controladores JDBC.
conexion = DriverManager.getConnection(url, user, password);
```

El primer parámetro del método **getConnection()** representa la URL de conexión a la base de datos:

- **jdbc:mysql** indica que estamos utilizando un driver JDBC para MySQL.
- **localhost:3306**: indica que el servidor de base de datos está en la misma máquina en la que se ejecuta el programa Java. Aquí puede poner una IP o un nombre de máquina que esté en la red.
- **bd**: es el nombre de la base de datos a la que nos vamos a conectar y que debe existir en MySQL.

El segundo parámetro es el nombre de usuario que accede a la base de datos, en este caso se llama **PrimeroDAM**.

El tercer parámetro es la contraseña del usuario, que en este caso es **PrimeroDAM**.

EJECUTAR SENTENCIAS SQL

Para realizar consultas, para ello recurrimos a la interfaz **Statement** para crear una sentencia. Para obtener un objeto **Statement** se llama al método **createStatement()** de un objeto **Connection** válido. La sentencia obtenida (o el objeto obtenido) tiene el método **executeQuery()** que sirve para realizar una consulta a la base de datos, se le pasa un **String** en el que está la consulta SQL, en el ejemplo **"SELECT * FROM Departamentos"**.

```
//Preparamos la consulta
Statement instruccionSQL = conexion.createStatement();
ResultSet result = instruccionSQL.executeQuery("SELECT DepNombre,
Localidad FROM Departamentos");
```

El resultado nos lo devuelve como un **ResultSet**, que es un objeto similar a una lista en la que está el resultado de la consulta. Cada elemento de la lista es uno de los registros de la tabla Clientes. **ResultSet** no contiene todos los datos, sino que los va cogiendo 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 usaremos dicho método dentro de un bucle **while** que se ejecutará mientras **next()** devuelva **true** (es decir, mientras haya registros).

```
//Recorremos el resultado de la consulta visualizando los registros
while(result.next()){
    System.out.println(result.getString("clNombre") + "\t"
                        + result.getInt("clTelefono"));
} // fin while
```

Los métodos **getInt()** y **getString()** nos van devolviendo los valores de los campos de dichos registros. El parámetro puede ser la posición de la columna en la tabla, también se puede poner una cadena que indica el nombre de la columna.

El código completo podría ser:

```
package ejemplos;
```

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

```
import conexion.Conexion;
```

```
/*
 * ejemplo que muestra el funcionamiento de JDBC accediendo a la
 base de datos * BDEmpleados
 * Ejemplo interfaz Statement
 * Listar todos los departamentos
 */
public class Ejemplo01Statement {

    public static void main(String[] args) {
        Conexion conexion = new Conexion();
        Statement st = null;
        ResultSet rs = null;

        String consulta = "SELECT deCodigo, deNombre, deLocalidad
FROM Departamentos";

        try {
            st = conexion.getConexion().createStatement();
            rs = st.executeQuery(consulta);

            while(rs.next()) {
                int cod = rs.getInt("deCodigo");
                String nb = rs.getString("deNombre");
                String lo = rs.getString("deLocalidad");
                System.out.println(cod + "\t" + nb + "\t" + lo);
            }
        }
```



```

        rs.close();
        st.close();
    } catch (SQLException e) {

        e.printStackTrace();
    }
    conexion.desconectar();
}
}

```

Liberar recursos

Por último se liberan todos los recursos y se cierra la conexión:

```

//Liberar recursos
result.close(); //cerrar ResultSet
instruccionSQL.close(); //cerrar Statement
conexion.close(); //cerrar conexión

```

Sentencias Preparadas. Uso de PreparedStatement con Java y MySQL

Con la interfaz **Statement** para crear las sentencias SQL lo hacíamos a partir de cadenas de caracteres en las que íbamos concatenando los datos necesarios para construir la sentencia completa.

Cuando trabajamos con una base de datos es posible que haya sentencias SQL que tengamos que ejecutar varias veces durante la sesión, aunque sea con distintos parámetros. Por ejemplo, durante una sesión con base de datos podemos querer insertar varios registros en una tabla. Cada vez los datos que insertamos serán distintos, pero la sentencia SQL será la misma: Un *INSERT* sobre determinada tabla que será siempre igual, salvo los valores concretos que queramos insertar.

Casi todas las bases de datos tienen previsto un mecanismo para que en estos casos la ejecución de esas sentencias repetidas sea más rápida.

Ejemplo, Si utilizamos la tabla Empleados con los siguientes campos: CodEmpleado (Smallint), Nombre (VARCHAR), Puesto (VARCHAR), Dir (SmallInt), FechaAlta (Date), Salario (Float), Comision (Float) y CodDepartamento (TinyInt) para insertar varios empleados tendríamos que codificar varios *INSERT* así:

```

INSERT INTO Empleados VALUES
(1, 'Alvarez Juan', 'Comercial', 5, '2000-05-01', 2000.00, 825.50,
10);
INSERT INTO Empleados VALUES
(2, 'Pérez María', 'Gerente', 5, '2010-11-05', 3000.00, NULL,20);
INSERT INTO Empleados VALUES
(3, 'Arrieta María', 'Comercial', 5, '2008-03-01', 2500.00, 1225.50,
10);
INSERT INTO Empleados VALUES
(14, 'Barcenas Mario', 'Administrativo', 5, '2010-02-05', 1200.00,
NULL, 20);

```

En cada caso la base de datos deberá analizar la sentencia SQL, comprobar que es correcta, convertir los datos al tipo adecuado (por ejemplo, los enteros a int) y ejecutar la sentencia.

El mecanismo que prevén las bases de datos para hacer más eficiente este proceso es que le indiquemos, previamente, el tipo de sentencia que vamos a usar, de forma que la base de datos la "*precompila*" y la guarda en condiciones de ser ejecutada inmediatamente, sin necesidad de analizarla en cada caso. Esto es lo que se conoce como una ***prepared statement***. En el caso de *mysql*, se haría de la siguiente forma:

```
PREPARE insertar FROM "INSERT INTO Empleados VALUES (?, ?, ?, ?, ?, ?, ?, ?)";
SET @auCod = 21;
SET @auNombre='Alvarez Ana';
SET @auPuesto='Comercial';
SET @auDir=8;
SET @auFechaAlta='2000-05-01';
SET @auSalario=2000.00;
SET @auComision=825.50;
SET @auCodDep=10;
```

Para ejecutar la inserción ejecutamos la siguiente sentencia:

```
EXECUTE insertar USING @auCod, @auNombre, @auPuesto, @auDir,
@auFechaAlta, @auSalario, @auComision, @auCodDep;
```

Para insertar otro empleado simplemente tenemos que cambiar los valores de las variables:

```
SET @auCod = 15;
SET @auNombre='Barrena Sonsoles';
SET @auPuesto='Administrativo';
SET @auDir=8;
SET @auFechaAlta='2007-12-01';
SET @auSalario=2000.00;
SET @auComision=null;
SET @auCodDep=10;

EXECUTE insertar USING @auCod, @auNombre, @auPuesto, @auDir,
@auFechaAlta, @auSalario, @auComision, @auCodDep;

DEALLOCATE PREPARE insertar;
```

En el ejemplo anterior se prepara una ***prepared statement*** de nombre ***insertar*** con la sentencia *SQL* del *INSERT* que queremos ejecutar, en ella reemplazamos los valores concretos por interrogantes. No se utilizan comillas entre los interrogantes. Para realizar las inserciones se utilizan unas variables *@XXXXXXX* a las que se les asignan los valores, y son las que se usarán en el *EXECUTE* de la ***prepared statement***. Una vez finalizadas las inserciones, avisamos a la base de datos que no vamos a usar más esta ***prepared statement*** con un ***DEALLOCATE***.

La ejecución de los *insert* realizados de esta manera es, teóricamente, más eficiente que los realizados de la forma tradicional, escribiendo el *INSERT* directamente. Otra ventaja adicional es la seguridad, que afecta más al programa *java*.

PreparedStatement desde java. Establecer la conexión.

Si la base de datos soporta **prepared statement** y el **driver/conector** que usemos para hablar con esa base de datos desde *java* los soporta también, entonces podemos usar los **prepared statement** desde *java*.

Lo primero de todo es establecer la conexión. En el caso de *MySQL* debemos además poner un pequeño parámetro adicional para configurar el driver para que use las **prepared statement** en el servidor. Este parámetro es **useServerPrepStmts=true**, por lo que la cadena de conexión completa sería:

```
package ejemplos;
```

```
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Date;
```

```
import conexion.Conexion;
import introducirDatosFechas.IntroducirDatos;
```

```
/*
 * ejemplo que muestra el funcionamiento de JDBC accediendo a la
base de datos
 * BDEmpleados
 *
 * Ejemplo interfaz PreparedStatement
 * Listar todos los empleados de una categoría que introducimos como
parametro
 */
```

```
public class Ejemplo05PreparedStatement {
    public static void main(String[] args) {
        Conexion conexion = new Conexion();

        PreparedStatement ps = null;
        ResultSet rs = null;

        String consulta = "SELECT emNombre, emFechaAlta, emSalario FROM
Empleados WHERE emPuesto = ?";
```

```
        try {
            ps = conexion.getConexion().prepareStatement(consulta);
            //pasamos el parámetro
            ps.setString(1, IntroducirDatos.introducirDatos("Puesto:
"));

            //ejecutamos la consulta
            rs = ps.executeQuery();
```

```

        //recorremos el resultSet de la misma manera que con
Statement
        while(rs.next()) {
            String nb = rs.getString("emNombre");
            Date fecha = rs.getDate("emFechaAlta");
            Double su = rs.getDouble("emSalario");
            System.out.println(nb + "\t" + fecha + "\t" + su);
        }
        rs.close();
        ps.close();
    } catch(SQLException e) {
        e.printStackTrace();
    }

    conexion.desconectar();
}
}

```

El ejemplo en el que insertamos un empleado quedaría así:

```

package ejemplos;

import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Date;

import conexion.Conexion;
import introducirDatosFechas.ConvertirFechas;
import introducirDatosFechas.IntroducirDatos;

/*
 * ejemplo que muestra el funcionamiento de JDBC accediendo a la
base de datos
 * BDEmpleados
 *
 * Ejemplo interfaz PreparedStatement
 * Insertar un empleado del que introducimos todos sus datos como
parametros
 */
public class Ejemplo06PreparedStatement {

    public static void main(String[] args) {

        Conexion conexion = new Conexion();

        PreparedStatement ps = null;
        String consulta = "INSERT INTO Empleados (emCodigo,
emNombre, emPuesto, emFechaAlta, " +

```

```

VALUES (?, ?, ?, ?, ?, ?, ? )";

        try {
            ps =
conexion.getConnection().prepareStatement(consulta);
            //pasamos los parámetros
            ps.setInt(1,
Integer.parseInt(IntroducirDatos.introducirDatos("Código: ")));
            ps.setString(2,
IntroducirDatos.introducirDatos("Nombre: "));
            ps.setString(3,
IntroducirDatos.introducirDatos("Puesto: "));
            ps.setDate(4,
ConvertirFechas.convertirJavaDateASqlDate(ConvertirFechas.convertirS
tringDate(IntroducirDatos.introducirDatos("Fecha de
alta<dd/MM/yyyy>: ")));
            ps.setDouble(5,
Double.parseDouble(IntroducirDatos.introducirDatos("Salario: ")));
            ps.setDouble(6,
Double.parseDouble(IntroducirDatos.introducirDatos("Comisión: ")));
            ps.setInt(7,
Integer.parseInt(IntroducirDatos.introducirDatos("Código
Departamento: ")));

            //ejecutamos la consulta
            int filas = ps.executeUpdate();

            if(filas != 0)
                System.out.println("Inserción correcta");
            else
                System.out.println("La inserción no se ha
podido realizar");

            ps.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        conexion.desconectar();
    }
}

```

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

EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS

Para ejecutar sentencias SQL usamos la interfaz **Statement** o la interfaz **PreparedStatement**, que nos proporcionan métodos para ejecutar sentencias SQL y obtener los resultados. Como

Statement y PreparedStatement son interfaces no se pueden crear objetos directamente, en su lugar los objetos se obtienen con una llamada al método **createStatement()** o **prepareStatement** de un objeto **Connection** válido:

```
Statement sentencia = conexion.createStatement();
```

```
PreparedStatement sentencia = conexión.prepareStatement();
```

Al crearse un objeto **Statement o PreparedStatement** se crea un espacio de trabajo para crear consultas SQL, ejecutarlas y para recibir los resultados de las consultas. 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 **ResultSet** como son las sentencias de manipulación de datos (DML): **INSERT, UPDATE, 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 0.

Execute(String): se utiliza para sentencias que devuelven más de un **ResultSet**, se suele utilizar para ejecutar procedimiento almacenados.

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

Método	Devuelve objetos tipo
getString(columna)	String
getBoolean(columna)	boolean
getByte(columna)	byte
getShort(columna)	short
getInt(columna)	int
getLong(columna)	long
getFloat(columna)	float
getDouble(columna)	double
getDate(columna)	Date
getTime(columna)	Time

El siguiente ejemplo inserta un departamento en la tabla Departamentos. Antes de ejecutar la orden **INSERT** construimos la sentencia en un String, las cadenas de caracteres deben ir encerradas entre comillas simples:

```
package ejemplos;
```

```
import java.sql.PreparedStatement;
```

```
import java.sql.SQLException;
```

```
import conexion.Conexion;
```

```
import introducirDatosFechas.ConvertirFechas;
```

```
import introducirDatosFechas.IntroducirDatos;
```

```

public class Ejemplo06PreparedInsertarDepartamentos {

    public static void main(String[] args) {
        Conexion conexion = new Conexion();

        PreparedStatement ps = null;
        String consulta = "INSERT INTO Departamentos (deCodigo,
deNombre, deLocalidad) VALUES (?, ?, ?)";

        try {
            ps =
conexion.getConexion().prepareStatement(consulta);
            //pasamos los parámetros
            ps.setInt(1,
Integer.parseInt(IntroducirDatos.introducirDatos("Código: ")));
            ps.setString(2,
IntroducirDatos.introducirDatos("Nombre: "));
            ps.setString(3,
IntroducirDatos.introducirDatos("Localidad: "));

            //ejecutamos la consulta
            int filas = ps.executeUpdate();

            if(filas != 0)
                System.out.println("Inserción correcta");
            else
                System.out.println("La inserción no se ha
podido realizar");

            ps.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        conexion.desconectar();
    }
}

```

Ejemplo en el que se incrementa el salario de los empleados de un departamento en un importe

```

package ejemplos;

import java.sql.PreparedStatement;
import java.sql.SQLException;

import conexion.Conexion;
import introducirDatosFechas.IntroducirDatos;

/*

```

```

* ejemplo que actualiza el salario de los empleados de un
departamento en la base de datos
* BDEmpleados utilizando JDBC y la clase Prepared Statement
*/
public class Ejemplo06PreparedActualizar {

    public static void main(String[] args) {
        Conexion conexion = new Conexion();

        PreparedStatement ps = null;
        String consulta = "UPDATE Empleados SET emSalario =
emSalario + ? WHERE emCodDepar = ?" ;
        try {
            ps =
conexion.getConexion().prepareStatement(consulta);
            //pasamos los parámetros
            ps.setInt(1,
Integer.parseInt(IntroducirDatos.introducirDatos("Aumento salarial:
"))));
            ps.setInt(2,
Integer.parseInt(IntroducirDatos.introducirDatos("Código
departamento: ")));

            //ejecutamos la consulta
            int filas = ps.executeUpdate();

            if(filas != 0)
                System.out.println("Actualización correcta");
            else
                System.out.println("La inserción no se ha
podido realizar");

            ps.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        conexion.desconectar();
    }
}

```

El siguiente ejemplo crea una vista (de nombre Totales) que contiene por cada departamento el código, el nombre, el número de empleados y la suma de salarios:

```

package ejemplos;

import java.sql.PreparedStatement;
import java.sql.SQLException;

import conexion.Conexion;

```



```

import introducirDatosFechas.IntroducirDatos;

public class Ejemplo07CrearVista {

    public static void main(String[] args) {
        Conexion conexion = new Conexion();
        PreparedStatement ps = null;

        //construimos la orden CREATE VIEW
        String consulta = "CREATE OR REPLACE VIEW Totales (CodDep,
NbDepar, NumEmp,TotSal) AS " +
                        "SELECT deCodigo, deNombre, COUNT(emCodigo),
SUM(emSalario) " +
                        "FROM Departamentos JOIN Empleados ON deCodigo
= emCodDepar " +
                        "GROUP BY deCodigo;";

        try {
            ps =
conexion.getConexion().prepareStatement(consulta);

            //ejecutamos la consulta
            int filas = ps.executeUpdate();

            System.out.println("OK. Vista creada");

            ps.close();
        } catch (SQLException e) {
            System.out.println("La Vista no se ha podido
crear");
            e.printStackTrace();
        }
        conexion.desconectar();
    }
}

```

El siguiente ejemplo utiliza la vista creada en el ejemplo anterior.

```

package ejemplos;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Date;

import conexion.Conexion;
import introducirDatosFechas.IntroducirDatos;

public class Ejemplo07UtilizarVista {
    /*

```

```

    * ejemplo que muestra el funcionamiento de JDBC accediendo a
la base de datos
    * BDEmpleados
    *
    * Ejemplo interfaz PreparedStatement. Lista los elementos de
una vista
    */
    public static void main(String[] args) {
        Conexion conexion = new Conexion();

        PreparedStatement ps = null;
        ResultSet rs = null;

        String consulta = "SELECT codDep, nbDepar, numemp, totalsal
FROM totales ORDER BY codDep";

        try {
            ps =
conexion.getConexion().prepareStatement(consulta);

            //ejecutamos la consulta
            rs = ps.executeQuery();
            //recorremos el resultSet de la misma manera que con
Statement
            while(rs.next()) {
                System.out.println(rs.getInt("codDep") + "\t"
+rs.getString("nbDepar") + "\t"
+rs.getInt("numEmp") + "\t" +rs.getDouble("totalsal"));
            }
            rs.close();
            ps.close();
        }catch(SQLException e) {
            e.printStackTrace();
        }
        conexion.desconectar();
    }
}

```

EJECUCIÓN DE PROCEDIMIENTOS

Los procedimientos almacenados en la base de datos son un conjunto de sentencias SQL y del lenguaje procedural utilizado por el SGBD que se pueden llamar por su nombre para llevar a cabo alguna tarea en la base de datos. Pueden definirse con parámetros de entrada (**IN**), de salida (**OUT**), de entrada/salida (**INOUT**) o sin ningún parámetro. También pueden devolver un valor, en este caso sería una función. Las técnicas para desarrollar procedimientos y funciones almacenadas depende del SGBD, en MySQL, por ejemplo las funciones no admiten parámetros OUT e INOUT, solo admiten parámetros IN.

La interfaz **CallableStatement** permite que se pueda llamar desde Java a los procedimientos almacenados, para crear un objeto se llama al método **prepareCall(String)** del objeto **Connection**.

```
//construimos la orden de llamada
String sql = "{CALL PrSubidaSal (?, ?)}";
```

```
//Preparamos la llamada al procedimiento
CallableStatement llamada = conexion.prepareCall(sql);
```

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

Llamada	Significado
{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.

El siguiente ejemplo muestra un procedimiento de nombre *PrSubidaSal* que sube el sueldo a los empleados de un departamento, el procedimiento recibe dos parámetros de entrada que son el código del departamento y la subida de salario:

```
DELIMITER $
CREATE PROCEDURE prSubidaSal(dep INT, incre INT)
BEGIN
    UPDATE Empleados SET emSalario = emSalario + incre
    WHERE emCodDepar = dep;
END $
DELIMITER ;
```

El siguiente ejemplo declara la llamada al procedimiento *PrSubidaSal* que tiene dos parámetros y para darles valor se utilizan los marcadores de posición (?):

```
package ejemplos;
```

```
import java.sql.CallableStatement;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import conexion.Conexion;
import introducirDatosFechas.IntroducirDatos;
```

```
public class Ejemplo08Procedimientos {
/*
 * ejemplo que ejecuta el procedimiento PrSubidaSal que tiene dos
 * parámetros el código del departamento y el incremento del sueldo
en la base de datos
 * BDEmpleados utilizando JDBC y la clase Prepared Statement
```

```

*/
public static void main(String[] args) {
    Conexion conexion = new Conexion();

    //Preparamos la llamada al procedimiento
    CallableStatement cs = null;

    //construimos la orden de llamada
    String consulta = "{CALL PrSubidaSal (?, ?)}";

    try {
        cs = conexion.getConnection().prepareCall(consulta);
        //pasamos los parametros
        cs.setInt(1,
Integer.parseInt(IntroducirDatos.introducirDatos("Departamento:
")));
        cs.setInt(2,
Integer.parseInt(IntroducirDatos.introducirDatos("Incremento: ")));

        //ejecutamos el procedimiento
        cs.execute();
        System.out.println("Subida de salario
realizada.....");
        // liberar recursos
        cs.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    conexion.desconectar();
}
}

```

Puede ocurrir que el usuario no tenga permiso para ejecutar procedimientos, en cuyo caso habría que otorgárselos con **GRAN OPTION**.

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

```
String url =
"jdbc:mysql://localhost:3306/UD02BD01Empleados?noAccessToProcedureBo
dies=true";
```

Cuando un procedimiento o función 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 **Types** define una constante para cada tipo genérico SQL, algunas son: **TINYINT**, **SMALLINT**, **INTEGER**, **FLOAT**, **REAL**, **DOUBLE**, **NUMERIC**, **DECIMAL**, **CHAR**, **VARCHAR**, **DATE**, **TIME**, **TIMESTAMP**, **JAVA_OBJECT**, **ARRAY**, **BOOLEAN**, etc. Por ejemplo, si el segundo parámetro es **OUT** y de tipo **VARCHAR** en la base de datos, el código sería:

```
llamadaFuncion.registerOutParameter(1, Types.INTEGER);
```

Una vez ejecutada la llamada al procedimiento, los valores de los parámetros *OUT* o *INOUT* se obtienen con los métodos **getXXX(índice)** similares a los utilizados para obtener valores de las columnas de un **ResultSet**.

```
System.out.println("El empleado: " +auxInt +" Años: " +llamadaFuncion.getInt(1));
```

El siguiente ejemplo crea, un procedimiento de nombre *prAnhosTrabajados* con dos parámetros el primero de entrada que recibe el código de un empleado y, el segundo de salida que devuelve los años que lleva trabajando en la empresa dicho empleado:

```
DELIMITER $
CREATE PROCEDURE prAnhosTrabajados(cod INT, OUT anhos INT )
BEGIN
    SELECT TRUNCATE(datediff(curdate(), emFechaAlta)/365, 0)
    INTO anhos from empleados
    WHERE emCodigo = cod;
END $
DELIMITER ;
```

El código para ejecutarlo sería:

```
package ejemplos;
import java.sql.CallableStatement;
import java.sql.SQLException;
import java.sql.Types;

import conexion.Conexion;
import introducirDatosFechas.IntroducirDatos;

/*
 * Ejemplo ejecución de un procedimiento de usuario mysql desde un
programa en Java
 * Vamos a ejecutar la el procedimiento prAnhosTrabajados de la base
de datos BDEmpleados
 * El procedimiento calcula el número de años trabajados en la
empresa de un determinado
 * trabajador, cuyo código se lo pasamos como parámetro y lo
almacenamos en una variable
 */
public class Ejemplo08ProcedimientosSalida {

    public static void main(String[] args) {
        Conexion conexion = new Conexion();

        //Preparamos la llamada al procedimiento
        CallableStatement cs = null;

        //construimos la orden de llamada
        String consulta = "{CALL prAnhosTrabajados (?, ?)}";
        try {
            //Preparamos la llamada al procedimiento
```

```

        cs = conexion.getConnection().prepareCall(consulta);

        //Damos valor a los argumentos valor de retorno
        cs.registerOutParameter(2, Types.INTEGER);
//parametro de salida
        // parametro de entrada

        int emple =
Integer.parseInt(IntroducirDatos.introducirDatos("Código Empleado:
"));
        cs.setInt(1, emple );

        //ejecutamos el procedimiento
        cs.execute();
        int anhos = cs.getInt(2);
        System.out.println("El empleado: " +emple +" Años: "
+anhos);

        // Otra forma de visualizar la informacion
        System.out.println("El empleado: " +emple +" Años: "
+cs.getInt(2));
        cs.close();

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

```

El siguiente ejemplo crea, una función de nombre *FuAnhoTrabajo* con un parámetro de entrada que recibe el código de un empleado, después devuelve los años que lleva trabajando en la empresa dicho empleado:

```

delimiter $
CREATE FUNCTION FuAnhoTrabajo (codEmp INT) RETURNS INT
BEGIN
DECLARE anhos INT DEFAULT 0;
SELECT YEAR(CURDATE()) - YEAR(emFechaAlta) INTO anhos
FROM empleados WHERE emCodigo = codEmp;
RETURN anhos;
END $
delimiter ;

```

En el siguiente ejemplo llamamos a la función *FuAnhoTrabajo* que tiene un parámetro de entrada y devuelve un valor.

```

package ejemplos;
/*
 * Ejemplo ejecución de una función de usuario mysql desde un
programa en Java

```

```

* Vamos a ejecutar la funcion FuAnhoTrabajo de la base de datos
BDEmpleados
* La función calcula el número de años trabajados en la empresa de
un determinado
* trabajador, cuyo código se lo pasamos como parámetro
*
* Ejecuta la función como si fuese un procedimiento
*/

```

```

import java.sql.CallableStatement;
import java.sql.SQLException;
import java.sql.Types;

import conexion.Conexion;
import introducirDatosFechas.IntroducirDatos;

public class Ejemplo09Funciones {

    public static void main(String[] args) {
        Conexion conexion = new Conexion();
        CallableStatement cs = null;

        //construimos la orden de llamada
        String consulta = "{? = CALL fuAnhoTrabajo (?)}";

        try{
            cs = conexion.getConexion().prepareCall(consulta);

            //Damos valor a los argumentos valor de retorno
            cs.registerOutParameter(1, Types.INTEGER);
            // parametro de entrada
            int emple =
Integer.parseInt(IntroducirDatos.introducirDatos("Código Empleado:
"));

            cs.setInt(2, emple);

            //ejecutamos el procedimiento
            cs.execute();
            int anhos = cs.getInt(1);
            System.out.println("El empleado: " +emple + " Años: "
+anhos);

            // Otra forma de visualizar la informacion
            System.out.println("El empleado: " +emple + " Años: "
+cs.getInt(1));

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

```

```

    }

}

```

El siguiente ejemplo muestra otra forma de ejecutar una función en una sentencia SELECT, en este caso el resultado simplemente se muestra:

```

package ejemplos;
/*
 * Ejemplo ejecución de una función de usuario mysql desde un
programa en Java
 * Vamos a ejecutar la función fuAnhoTrabajo de la base de datos
BDEmpleados
 * La función calcula el número de años trabajados en la empresa de
un determinado
 * trabajador, cuyo código se lo pasamos como parámetro la función
se ejecuta en la sentencia SELECT
 */

```

```

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import conexion.Conexion;

public class Ejemplo09Funciones02 {
    public static void main(String[] args) {
        Conexion conexion = new Conexion();
        PreparedStatement ps = null;
        ResultSet rs = null;

        String consulta = "SELECT emNombre,
FuAnhoTrabajo(emCodigo) FROM empleados";

        try {
            ps =
conexion.getConnection().prepareStatement(consulta);

            rs = ps.executeQuery();
            while(rs.next()) {
                //
                System.out.println(rs.getString("emNombre")+"\t\t"
+rs.getInt(2));

                System.out.println(rs.getString("emNombre")+"\t\t"
+rs.getInt("FuAnhoTrabajo(emCodigo)"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        conexion.desconectar();
    }
}

```



```
}
}
```

EJECUCIÓN DE SENTENCIAS DE DESCRIPCIÓN DE DATOS

Normalmente, cuando desarrollamos una aplicación JDBC conocemos la estructura de las tablas y datos que estamos manejando, es decir, conocemos las columnas que tienen y cómo están relacionadas entre sí. También 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 se puede obtener a través de los **metaobjetos**, que no son más que objetos que proporcionan información sobre la base de datos.

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

El siguiente ejemplo conecta con la base de datos MySQL de nombre **BDEmpleados** y muestra información sobre el producto de base de datos, la URL para acceder a la base de datos, el nombre de usuario y las tablas y vistas del esquema actual (o de todos los esquemas dependiendo del sistema gestor de la base de datos), un esquema se corresponde generalmente con un usuario de la base de datos; el método **getMetaData()** de la clase **Connection** devuelve un objeto **DatabaseMetaData** con el que se obtendrá la información sobre la base de datos:

```
package ejemplosDescripcionDatos;
/*
 * El siguiente ejemplo conecta con la base de datos MySQL de nombre
 * ejemplo
 * y muestra información sobre el producto de base de datos, la URL
 * para acceder
 * a la base de datos, el nombre de usuario y las tablas y vistas
 * del esquema actual
 */
import java.sql.*;

import conexion.Conexion;

public class Ejemplo01DatabaseMetaData {

    public static void main(String[] args) {
        Conexion conexion = new Conexion();
        DatabaseMetaData dbmd = null;
        ResultSet rs = null;

        try {
            dbmd = conexion.getConexion().getMetaData();
            //Recuperamos información de la base de datos
            System.out.println("Información sobre la Base de
Datos");
            System.out.println("-----");
        }
    }
}
```

```

        System.out.println("Nombre: "
+dbmd.getDatabaseProductName());
        System.out.println("Driver: "
+dbmd.getDriverName());
        System.out.println("URL: " +dbmd.getURL());
        System.out.println("Usuario: " +dbmd.getUserName());

        //Obtener información de las tablas y vistas de la
base de datos
        System.out.println("\nInformación sobre las tablas y
Vistas de la Base de Datos");
        System.out.println("-----");
        rs = dbmd.getTables(null, "BDEmpleados", null,
null);
        while(rs.next()) {
            System.out.println("Catálogo: "
+rs.getString(1)+"\t" +"Esquema: " +rs.getString(1)
            +"Tabla: " +rs.getString(3) +"\t" +"Tipo: "
+rs.getString(4));
        }
        //Obtener información de la tabla cuyo nombre se
indica como parámetro
        System.out.println("\nInformación de la tabla
Empleados");
        System.out.println("-----");
        rs = dbmd.getTables(null, "BDEmpleados",
"Empleados", null);
        while(rs.next()) {
            System.out.println("Catálogo: "
+rs.getString(1)+"\t" +"Esquema: " +rs.getString(1)
            +"Tabla: " +rs.getString(3) +"\t" +"Tipo: "
+rs.getString(4));
        }
        }catch(SQLException e) {
            e.printStackTrace();
        }
        conexion.desconectar();
    }
}

```

La ejecución muestra la siguiente información:

```

Problems @ Javadoc Declaration Console
<terminated> Ejemplo01DatabaseMetaData [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (16 may. 2019 11:07:23)
Información sobre la Base de Datos
-----
Nombre: MySQL
Driver: MySQL Connector Java
URL: jdbc:mysql://localhost/BDEmpleados?noAccessToProcedureBodies=true
Usuario: PrimeroDAM@localhost

Información sobre las tablas y Vistas de la Base de Datos
-----
Catálogo: BDEmpleados Esquema: BDEmpleadosTabla: departamentos Tipo: TABLE
Catálogo: BDEmpleados Esquema: BDEmpleadosTabla: empleados Tipo: TABLE
Catálogo: BDEmpleados Esquema: BDEmpleadosTabla: totales Tipo: VIEW

Información de la tabla Empleados
-----
Catálogo: BDEmpleados Esquema: BDEmpleadosTabla: empleados Tipo: TABLE

```

El método **getTables()** devuelve un objeto **ResultSet** que proporciona información sobre las tablas y vistas de la base de datos. Necesita 4 parámetros:

- *Primer parámetro:* catálogo de la base de datos. El método obtiene las tablas del catálogo indicado, al poner **null**, indicamos todos los catálogos.
- *Segundo parámetro:* 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).
- *Tercer parámetro:* 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".
- *El cuarto parámetro:* 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 devolverá solo las tablas:

```

//Obtener información solo de las tablas
String[] tipos = {"TABLE"};
result = dbmd.getTables(null, "ejemplo", null, tipos);

```

Cada fila de **ResultSet** que devuelve **getTables()** tiene información sobre una tabla. Las columnas de **ResultSet** que devuelve el método **TABLE_CAT** (el nombre del catálogo al que pertenece la tabla), **TABLE_SCHEM** (el nombre del esquema al que pertenece la tabla), **TABLE_NAME** (el nombre de la tabla o vista), **TABLE_TYPE** (el tipo **TABLE** o **VIEW**), **REMARKS** (comentarios). Para obtener estos resultados también podríamos haber puesto en el ejemplo el siguiente código:

```

// obtiene la misma información indicando el nombre de la columna en
// lugar de la posición
String catalogo1 = result.getString("TABLE_CAT"); // columna 1 que
// devuelve el ResultSet
String esquema1 = result.getString("TABLE_SCHEM"); // columna 2
String tabla1 = result.getString("TABLE_NAME"); // columna 3
String tipo1 = result.getString("TABLE_TYPE"); // columna 4

```

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

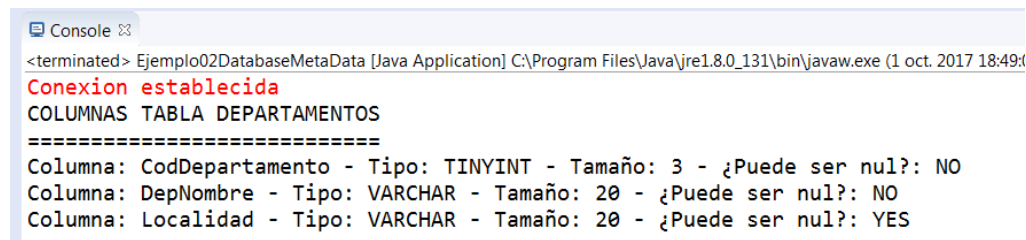
- **getColumns(catálogo, esquema, nombreTabla, nombreColumna)** devuelve información sobre las columnas de una tabla o tablas. Para el nombre de la tabla y la columna se

puede utilizar el carácter guión bajo o porcentaje. Por ejemplo, ***getColumns(null, "BDEmpleados", "departamentos", "%d")***, obtiene todos los nombre de columna que empiezan por la letra d en la tabla departamentos y en el esquema BDEmpleados. El valor ***null*** en los 4 parámetros indica que obtiene información sobre todas las columnas y tablas del esquema actual.

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

```
//Obtener información de las columnas de la tabla Departamentos
System.out.println("COLUMNAS TABLA DEPARTAMENTOS");
System.out.println("=====");
ResultSet columnas = null;
columnas = dbmd.getColumns(null, " BDEmpleados ", "departamentos",
null);
while(columnas.next()){
    String nombreCol = columnas.getString("COLUMN_NAME"); //
getString(4)
    String tipoCol = columnas.getString("TYPE_NAME"); //
getString(6)
    String tamCol = columnas.getString("COLUMN_SIZE"); //
getString(7)
    String tipo = columnas.getString("IS_NULLABLE"); //
getString(18)
    System.out.println("Columna: " +nombreCol +" - Tipo: " +tipoCol
+" - Tamaño: " +tamCol +" - ¿Puede ser nul?: " +tipo);
} // fin while
```

Visualiza:



```
Console
<terminated> Ejemplo02DatabaseMetaData [Java Application] C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe (1 oct. 2017 18:49:
Conexion establecida
COLUMNAS TABLA DEPARTAMENTOS
=====
Columna: CodDepartamento - Tipo: TINYINT - Tamaño: 3 - ¿Puede ser nul?: NO
Columna: DepNombre - Tipo: VARCHAR - Tamaño: 20 - ¿Puede ser nul?: NO
Columna: Localidad - Tipo: VARCHAR - Tamaño: 20 - ¿Puede ser nul?: YES
```

- ***getPrimaryKeys(catálogo, esquema, tabla)*** devuelve una lista de columnas que forman la clave primaria.

El siguiente ejemplo muestra la clave primaria de la tabla departamentos:

```
//obtener información sobre la clave primaria de la tabla
departamentos
ResultSet pk = dbmd.getPrimaryKeys(null, " BDEmpleados ",
"departamentos");
String pkDep = "", separador = "";
while (pk.next()){
    pkDep = pkDep + separador + pk.getString("COLUMN_NAME");
//getString(4)
    separador = "+";
} //fin while
```

```
System.out.println("Clave primaria: "+pkDep);
```

- **getExportedKeys(catálogo, 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:

```
//obtener información sobre las claves ajenas que hacen
// referencia a la tabla departamentos
ResultSet fk = dbmd.getExportedKeys(null, " BDEmpleados ",
"departamentos");
while (fk.next()){
    String fkNbColumna = fk.getString("FKCOLUMN_NAME");
    String pkNbColumna = fk.getString("PKCOLUMN_NAME");
    String pkNbTabla = fk.getString("PKTABLE_NAME");
    String fkNbTabla = fk.getString("FKTABLE_NAME");
    System.out.println("Tabla PK: "+pkNbTabla +" Clave Primaria: "
+pkNbColumna);
    System.out.println("Tabla FK: "+fkNbTabla +" Clave Ajena: "
+fkNbColumna);
} //fin while
```

Visualiza

```
Tabla PK: departamentos Clave Primaria: CodDepartamento
Tabla FK: empleados Clave Ajena: CodDepartamento
```

El método no devuelve nada si escribimos:

```
ResultSet fk = dbmd.getExportedKeys(null, " BDEmpleados ",
"empleados");
```

Ya que la tabla Empleados no es referenciada por ninguna clave ajena.

- **getImportedKeys(catálogo, esquema, tabla)** devuelve la lista de claves ajenas existentes en la tabla.

Se utiliza igual que el método anterior, en este caso

```
//obtener la lista de claves ajenas de una tabla
ResultSet fk1 = dbmd.getImportedKeys(null, " BDEmpleados ",
"empleados");
```

Devuelve la salida anterior y el método

```
ResultSet fk1 = dbmd.getImportedKeys(null, " BDEmpleados ",
"departamentos");
```

No devuelve nada ya que no tiene claves ajenas.

- **getProcedures(catálogo, esquema, procedure)** devuelve la lista procedimientos almacenados.

El siguiente ejemplo muestra los procedimientos y funciones que tiene el esquema de nombre ejemplo.

```
//obtener información sobre los procedimientos almacenados
ResultSet proc = dbmd.getProcedures(null, " BEmpleados ", null);
while (proc.next()){
    String procNb = proc.getString("PROCEDURE_NAME");
    String procTipo = proc.getString("PROCEDURE_TYPE");
    System.out.println("Nombre Procedimiento: "+procNb +" Tipo: "
+procTipo);
} //fin while
```

El código completo:

```
package ejemplosDescripcionDatos;

import java.sql.*;

import conexion.Conexion;

/*
 * ejemplo que devuelve información de las tablas de la BD
 * utilizando metodos de la clase
 * DatabaseMetaData
 */
public class Ejemplo02DatabaseMetaData {
    public static void main(String[] args) {

        // Crear la conexion conexion
        Conexion conexion = new Conexion();
        DatabaseMetaData dbmd = null;
        ResultSet rs = null;

        try{

            dbmd = conexion.getConexion().getMetaData();

            //Obtener información de las columnas de la tabla
            Departamentos
            System.out.println("COLUMNAS TABLA DEPARTAMENTOS");
            System.out.println("=====");
            ResultSet columnas = null;
            columnas = dbmd.getColumns(null, "BEmpleados",
"departamentos", null);
            while(columnas.next()){
                String nombreCol =
columnas.getString("COLUMN_NAME"); // getString(4)
                String tipoCol =
columnas.getString("TYPE_NAME"); // getString(6)
                String tamCol =
columnas.getString("COLUMN_SIZE"); // getString(7)
                String tipo =
columnas.getString("IS_NULLABLE"); // getString(18)
```

```

        System.out.println("Columna: " + nombreCol + " -
Tipo: " + tipoCol + " - Tamaño: " + tamCol
        + " - ¿Puede ser nul?: " + tipo);
    }// fin while

    //obtener información sobre la clave primaria de la
tabla departamentos
    System.out.println("\nPRIMARY KEY TABLA
DEPARTAMENTOS");
    System.out.println("=====");
    ResultSet pk = dbmd.getPrimaryKeys(null,
"BDEmpleados", "departamentos");
    String pkDep = "", separador = "";
    while (pk.next()){
        pkDep = pkDep + separador +
pk.getString("COLUMN_NAME"); //getString(4)
        separador = "+";
    }//fin while
    System.out.println("Clave primaria: "+pkDep);

    //obtener información sobre las claves ajenas que
hacen

    // referencia a la tabla departamentos
    System.out.println("\nFOREIGN KEY TABLA
DEPARTAMENTOS");
    System.out.println("=====");
    ResultSet fk = dbmd.getExportedKeys(null,
"BDEmpleados", "departamentos");
    // ResultSet fk = dbmd.getExportedKeys(null,
"BDEmpleados", "empleados");
    // no devuelve nada la tabla empleados no esta
referenciada en ninguna tabla
    while (fk.next()){
        String fkNbColumna =
fk.getString("FKCOLUMN_NAME");
        String pkNbColumna =
fk.getString("PKCOLUMN_NAME");
        String pkNbTabla =
fk.getString("PKTABLE_NAME");
        String fkNbTabla =
fk.getString("FKTABLE_NAME");
        System.out.println("Tabla PK: "+pkNbTabla +
Clave Primaria: " +pkNbColumna);
        System.out.println("Tabla FK: "+fkNbTabla +
Clave Ajena: " +fkNbColumna);
    }//fin while

    //obtener la lista de claves ajenas de una tabla
    System.out.println("\nFOREIGN KEY TABLA EMPLEADOS");

```

```

        System.out.println("=====");
        ResultSet fk1 = dbmd.getImportedKeys(null,
"BDEmpleados", "empleados");
        while (fk1.next()){
            String fkNbColumna =
fk1.getString("FKCOLUMN_NAME");
            String pkNbColumna =
fk1.getString("PKCOLUMN_NAME");
            String pkNbTabla =
fk1.getString("PKTABLE_NAME");
            String fkNbTabla =
fk1.getString("FKTABLE_NAME");
            System.out.println("Tabla PK: "+pkNbTabla +"
Clave Primaria: " +pkNbColumna);
            System.out.println("Tabla FK: "+fkNbTabla +"
Clave Ajena: " +fkNbColumna);
        }//fin while

        //obtener información sobre los procedimientos
almacenados
        System.out.println("\nPROCEDIMIENTOS DE LA BASE DE
DATOS");
        System.out.println("=====");
        ResultSet proc = dbmd.getProcedures(null, "
BDEmpleados ", null);
        while (proc.next()){
            String procNb =
proc.getString("PROCEDURE_NAME");
            String procTipo =
proc.getString("PROCEDURE_TYPE");
            System.out.println("Nombre Procedimiento:
"+procNb +" Tipo: " +procTipo);
        }//fin while
    }catch(SQLException sqle){
        sqle.printStackTrace();
    }
    conexion.desconectar();
}
}

```

ResultSetMetaData

Se pueden obtener metadatos (datos sobre los datos) a partir de **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 ejemplo muestra el uso de la interfaz **ResultSetMetaData** para conocer más información acerca de las columnas devueltas por **ResultSet**, en el ejemplo 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 obtendrá la información acerca de las columnas devueltas:

package ejemplosDescripcionDatos;

import java.sql.*;

import conexion.Conexion;

/*

* El siguiente ejemplo conecta con la base de datos MySQL de nombre BDEmpleados

* ejecuta una sentencia SELECT y nos devuelve información sobre las columnas devueltas

*/

public class Ejemplo03ResultSetMetaData {

public static void main(String[] args) {

 // Crear la conexion conexion

 Conexion conexion = **new** Conexion();

 Statement st = **null**;

 ResultSet rs = **null**;

 ResultSetMetaData rsmd = **null**;

try{

 //Preparamos la consulta

 st = conexion.getConnection().createStatement();

 rs = st.executeQuery("SELECT * FROM Departamentos");

 rsmd = rs.getMetaData();

int numCol = rsmd.getColumnCount();

 String nula;

 System.out.println("Número de columnas recuperadas:

" +numCol);

for(**int** i= 1; i<= numCol; i++){

 System.out.println("Columna: " +i);

 System.out.println("Nombre: "

+rsmd.getColumnName(i));

 System.out.println("Tipo: "

+rsmd.getColumnType(i));

if(rsmd.isNullable(i) == 0)

 nula = "No";

else

 nula = "Si";

 System.out.println("¿Puede ser nula?: " +nula);

 System.out.println("Máximo ancho de la columna:

" +rsmd.getColumnDisplaySize(i));

 }

 }**catch**(SQLException sqle){

 sqle.printStackTrace();

```

    }
    conexion.desconectar();
}
}

```

Visualiza la siguiente información:

```

<terminated> Ejemplo03ResultSetMetaData [Java Application] C:\Program Files\
Número de columnas recuperadas: 3
Columna: 1
Nombre: deCodigo
Tipo: 4
¿Puede ser nula?: No
Máximo ancho de la columna: 2
Columna: 2
Nombre: deNombre
Tipo: 12
¿Puede ser nula?: No
Máximo ancho de la columna: 40
Columna: 3
Nombre: deLocalidad
Tipo: 12
¿Puede ser nula?: No
Máximo ancho de la columna: 20

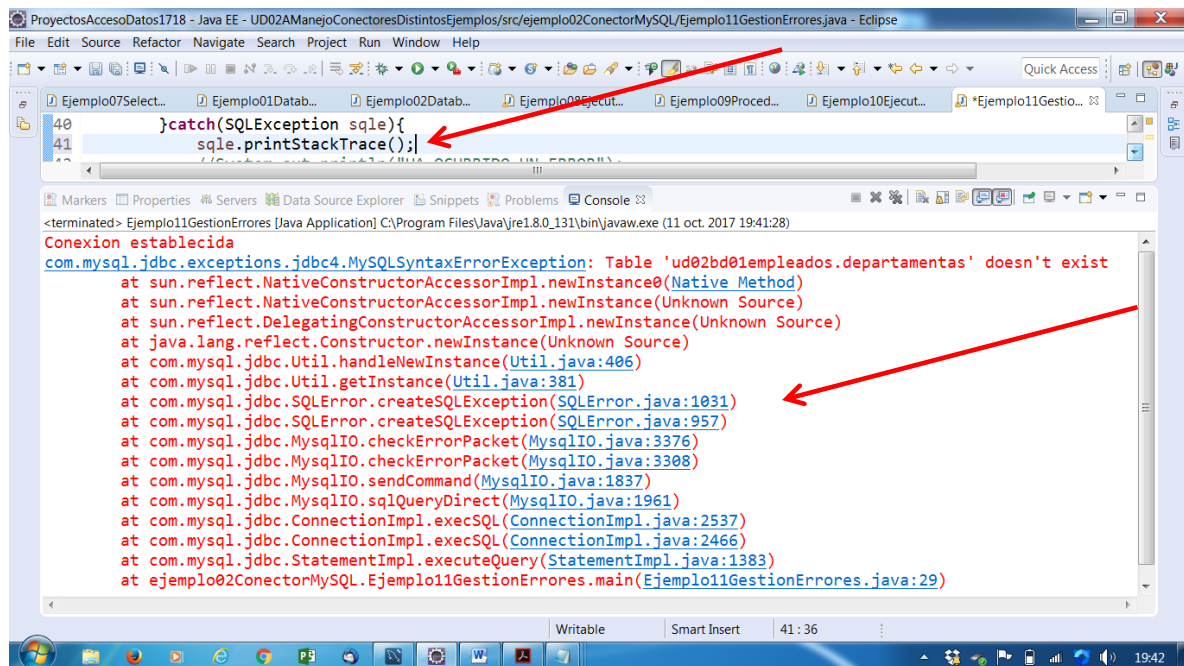
```

Los métodos utilizados son:

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

GESTION DE ERRORES

Si cuando se produce un error se visualiza con ***printStackTrace()*** obtenemos una salida como la siguiente:



Cuando se produce un error **SQLException** podemos acceder a cierta información usando los siguientes métodos:

Método	Función
<code>getMessage()</code>	Devuelve una cadena que describe el error
<code>getSQLState</code>	Es una cadena que contiene un estado definido por el estándar X/OPEN SQL
<code>getErrorCode()</code>	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.

```

}catch(SQLException sqle){
    sqle.printStackTrace();
    System.out.println("HA OCURRIDO UN ERROR");
    System.out.println("Mensaje: " +sqle.getMessage());
    System.out.println("SQL estado: " +sqle.getSQLState());
    System.out.println("Cod. error: " +sqle.getErrorCode());

```

El siguiente ejemplo produce un error porque la tabla no existe en la base de datos:

```
package ejemplos;
```

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

```
import conexion.Conexion;
```

```
public class Ejemplo10GestionErrores {

    public static void main(String[] args) {
        Conexion conexion = new Conexion();
        Statement st = null;
        ResultSet rs = null;

```

```

String consulta = "SELECT deCodigo, deNombre, deLocalidad
FROM Departamentass";

try {
    st = conexion.getConnection().createStatement();
    rs = st.executeQuery(consulta);

    while(rs.next()) {
        int cod = rs.getInt("deCodigo");
        String nb = rs.getString("deNombre");
        String lo = rs.getString("deLocalidad");
        System.out.println(cod + "\t" + nb + "\t" + lo);
    }
    rs.close();
    st.close();
} catch (SQLException e) {
    e.printStackTrace();
}
conexion.desconectar();
}
}

```

El resultado que produce es el siguiente:

```

<terminated> Ejemplo10GestionErrores [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (16 may. 2019 11:41:56)
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Table 'bdempleados.departamentass' doesn't exist
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Source)
    at java.lang.reflect.Constructor.newInstance(Unknown Source)
    at com.mysql.jdbc.Util.handleNewInstance(Util.java:425)
    at com.mysql.jdbc.Util.getInstance(Util.java:408)
    at com.mysql.jdbc.SQLException.createSQLException(SQLException.java:943)
    at com.mysql.jdbc.MySQLIO.checkErrorPacket(MySQLIO.java:3970)
    at com.mysql.jdbc.MySQLIO.checkErrorPacket(MySQLIO.java:3906)

```