



Acceso a datos con JDBC

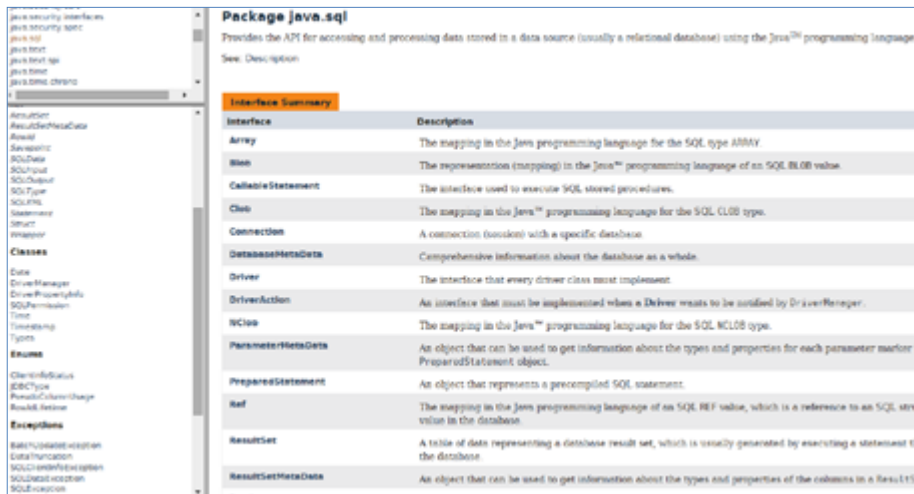
Índice



1 Definición JDBC	3
2 Arquitectura y API de JDBC	5
3 Drivers JDBC	7
4 La excepción SQLException	9
5 Cargar el controlador	10
6 Obtener la conexión	13
7 Obtener el objeto Statement	15
8 Obtener el ResultSet	19
9 Navegar por el ResultSet	21
10 Obtener información de las columnas del ResultSet	23
11 Tipos de datos y conversiones	24
12 Actualizar, borrar e insertar filas en ResultSet	25
13 Actualizar, borrar e insertar registros con sentencias SQL	29
14 Ejecutar SQL con PreparedStatement	31
15 Acceso a los metadatos	34

1. Definición JDBC

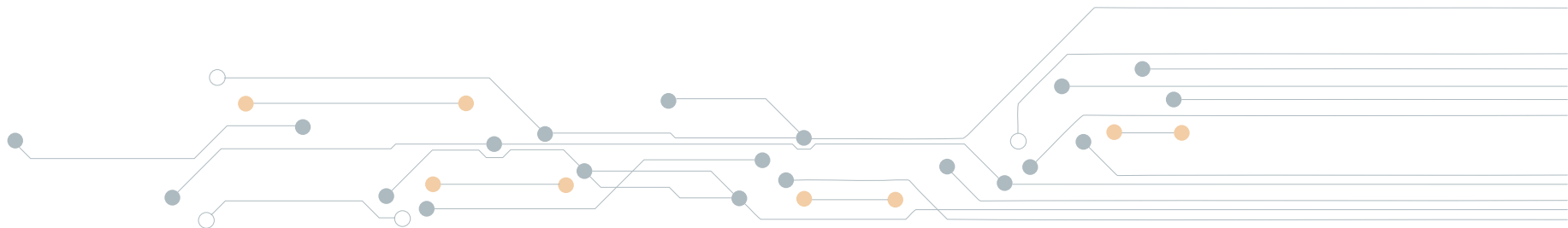
JDBC (Java Database Connectivity) es un conjunto de clases, interfaces, enumeraciones y excepciones (API) que permite el acceso a bases de datos relacionales, la ejecución de sentencias SQL y la manipulación del resultado de las mismas. El paquete principal de JDBC es **java.sql**, también existe el paquete **javax.sql**.



Interface	Description
Array	The mapping in the Java programming language for the SQL type ARRAY.
Blob	The representation (mapping) in the Java™ programming language of an SQL BLOB value.
CallableStatement	The interface used to execute SQL stored procedures.
Clob	The mapping in the Java™ programming language for the SQL CLOB type.
Connection	A connection (session) with a specific database.
DatabaseMetaData	Comprehensive information about the database as a whole.
Driver	The interface that every driver class must implement.
DriverAction	An interface that must be implemented when a Driver wants to be notified by DriverManager.
NClob	The mapping in the Java™ programming language for the SQL NCLOB type.
ParameterMetaData	An object that can be used to get information about the types and properties for each parameter marker in a PreparedStatement object.
PreparedStatement	An object that represents a precompiled SQL statement.
Ref	The mapping in the Java programming language of an SQL REF value, which is a reference to an SQL stored value in the database.
ResultSet	A table of data representing a database result set, which is usually generated by executing a statement in the database.
ResultSetMetaData	An object that can be used to get information about the types and properties of the columns in a ResultSet.

JDBC proporciona un mecanismo estándar e independiente de acceso a la mayoría de los gestores de bases de datos relacionales. Cada fabricante de base de datos debe proporcionar el driver JDBC específico para el acceso y trabajo con su base de datos.

IMAGEN 14.1: API JDBC, PAQUETE JAVA.SQL



Con JDBC no es necesario escribir distintos programas para distintas bases de datos, sino que un único programa puede servir para acceder a distintas bases de datos de distinta naturaleza. Incluso, se puede acceder a más de una base de datos de distinta fuente (MySQL, Oracle, Derby, Postgress, Access, etc.) en la misma aplicación.

JDBC es como la pasarela entre una base de datos y un programa Java.

El esquema del proceso a seguir para trabajar con JDBC es el siguiente:

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

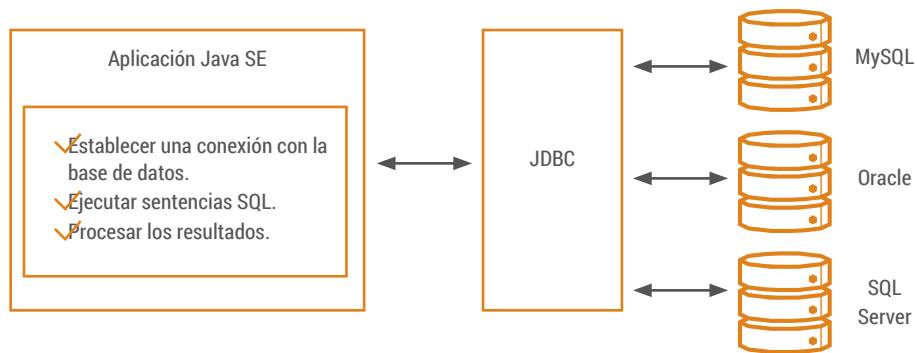
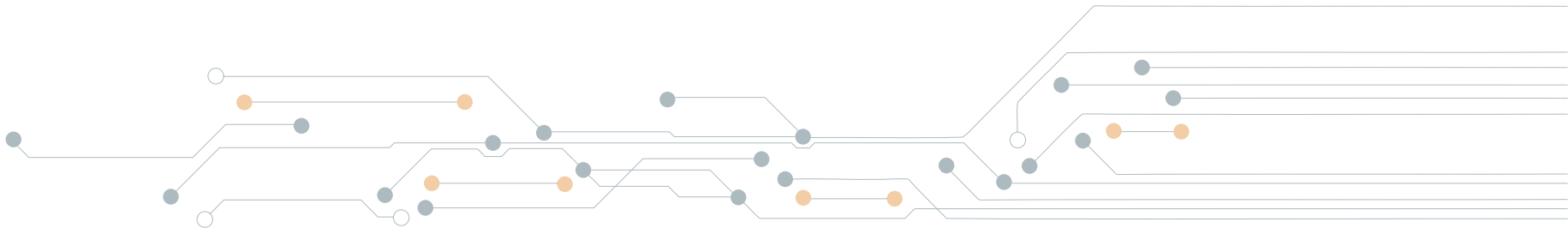


IMAGEN 14.2: JDBC (JAVA DATABASE CONNECTIVITY).



2. Arquitectura y API de JDBC

Desde la aplicación Java, para implementar los tres puntos del proceso esquemático, para trabajar con JDBC se utilizan fundamentalmente la interfaces:

- **Connection**, se utiliza para establecer la conexión con el origen de datos.
- **Statement**, es el que contiene la sentencia SQL, a través de la aplicación de diferentes métodos se ejecutarán las sentencias.
- **ResultSet**, almacena el resultado de la ejecución de sentencias SELECT. Iterando el objeto, se obtienen cada uno de los registros obtenidos en la ejecución.

La conexión, la referencia al objeto **Connection**, la proporciona la clase **DriverManager**, mediante alguno de sus métodos **getConnection**, al que habrá que proporcionar información sobre el driver a utilizar, la base de datos con la que conectar, así como el usuario y password con el que se accede.



Previamente habrá que haber registrado o cargado la clase de Java que encapsula el driver en concreto que se va a utilizar para hacer la conexión, proporcionada por el fabricante. Esta operación se realiza normalmente mediante la función static **Class.forName**.

La representación de esta arquitectura de utilización de JDBC es como se muestra en la imagen siguiente:

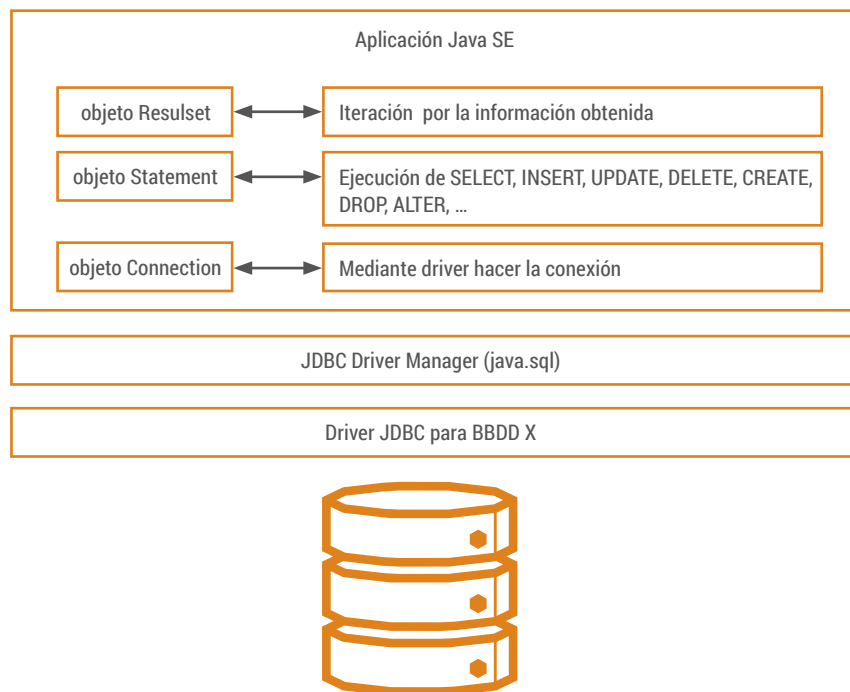


IMAGEN 14.2: JDBC (JAVA DATABASE CONNECTIVITY).

Las clases e interfaces más significativas del API De JDBC del paquete java.sql son:

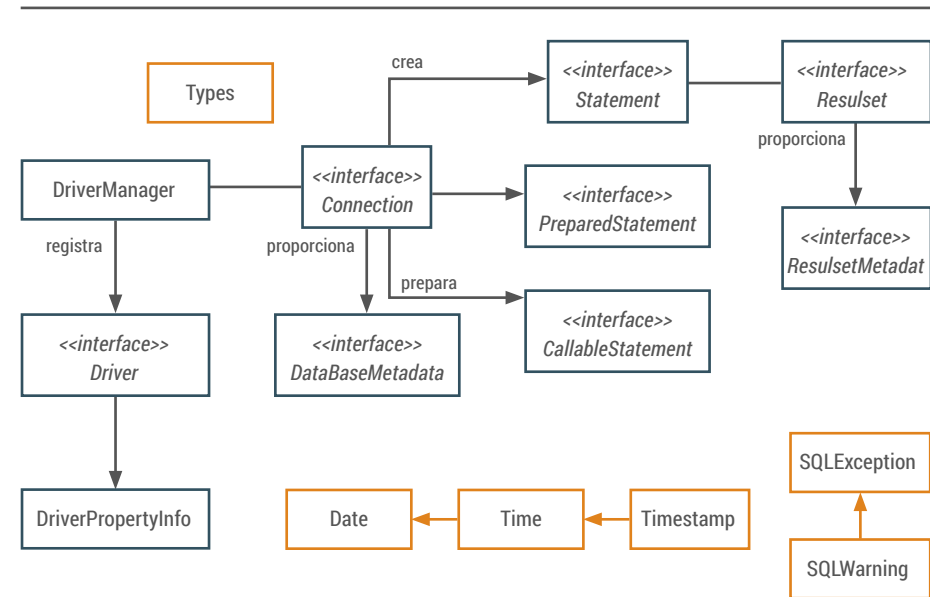


IMAGEN 14.4: DIAGRAMA CLASES E INTERFACES DEL API JDBC.

3. Drivers JDBC

Un driver o controlador JDBC es la implementación del API JDBC para un gestor de base de datos concreto Oracle, MySQL, Derby, ...etc. Existen 4 tipos:

■ Tipo1: JDBC-ODBC:

Microsoft estableció una norma común para comunicarse con las bases de datos llamada **Open DataBase Connectivity (ODBC)**. Hasta que esta norma no apareció, los clientes eran específicos del servidor. Utilizando la **API ODBC** se consigue desarrollar software independiente del servidor.

El utilizar **ODBC** requiere configurar en el sistema un **Data Source Name (DSN)**, que representa a la base de datos, por tanto cada cliente tiene que tener instalado el driver.

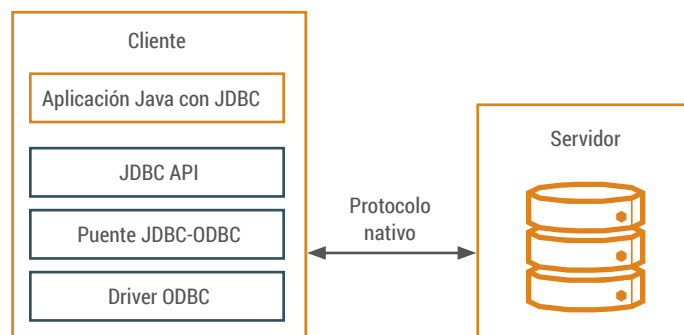
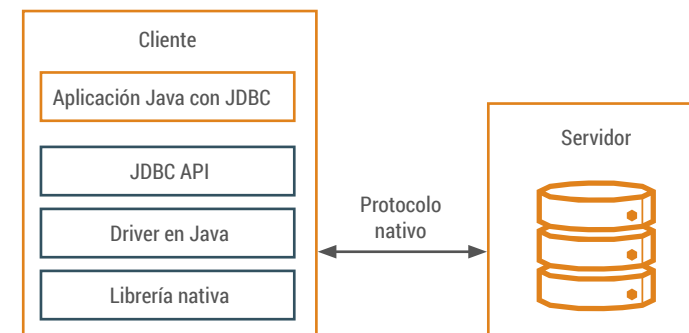


IMAGEN 14.5: TIPO 1. DRIVER JDBC-ODBC.

■ Tipo2: JDBC-API Nativo:

- Es una combinación de implementación **Java** y **API nativo** para el acceso a la base de datos. Este tipo de driver es más rápido que el anterior, puesto que no se realiza el paso por la capa ODBC. Las llamadas **JDBC** se traducen en llamadas específicas del **API de la base de datos**. Cada cliente debe tener instalado el driver. Tiene menor rendimiento que los dos siguientes y no se pueden usar en Internet, ya que necesita el API de forma local.



MAGEN 14.6: TIPO2. DRIVER JDBC-API NATIVO.

■ Tipo3: JDBC-100% Java en red.

En este tercer tipo los clientes JDBC utilizan "socket" de red para comunicarse con un servidor de aplicaciones. Los clientes utilizan un protocolo de red independiente del gestor de base de datos, utilizan el protocolo para comunicarse con el servidor de aplicaciones. La información recogida por el "socket" del servidor se traduce al formato requerido por la base de datos, y la reenvía al servidor de base de datos.

Es más flexible que los dos anteriores porque no necesita tener instalado el driver en el cliente y un único driver instalado en el servidor de aplicaciones puede permitir el acceso a varias bases de datos.

El servidor de aplicaciones, utiliza un driver JDBC de tipo 1, 2 o 4 para conectarse con la base de datos.

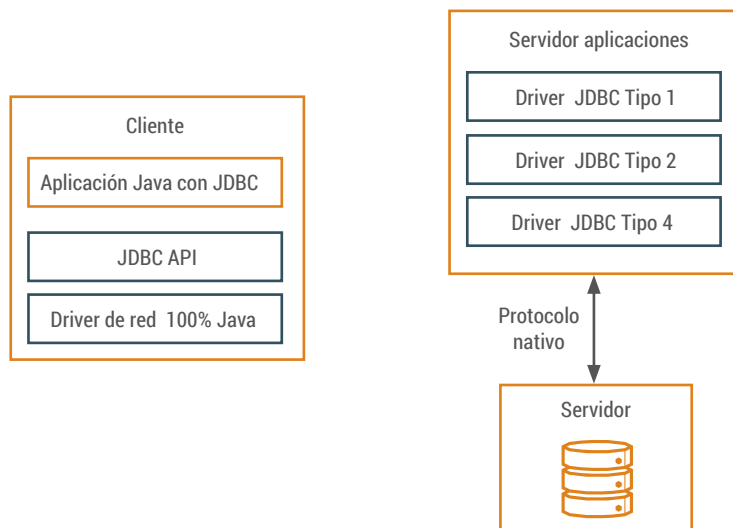


IMAGEN 14.7: TIPO3. JDBC-100% JAVA EN RED.

■ Tipo 4: 100% Java.

La llamada JDBC se traduce directamente en una llamada de red a la base de datos, sin intermediarios. Proporcionan mejor rendimiento. La mayoría de SGBD proporcionan drivers de este tipo.

Este tipo de driver es extremadamente flexible, no es necesario instalar un software especial en el cliente o servidor. Además, se pueden descargar de forma dinámica.

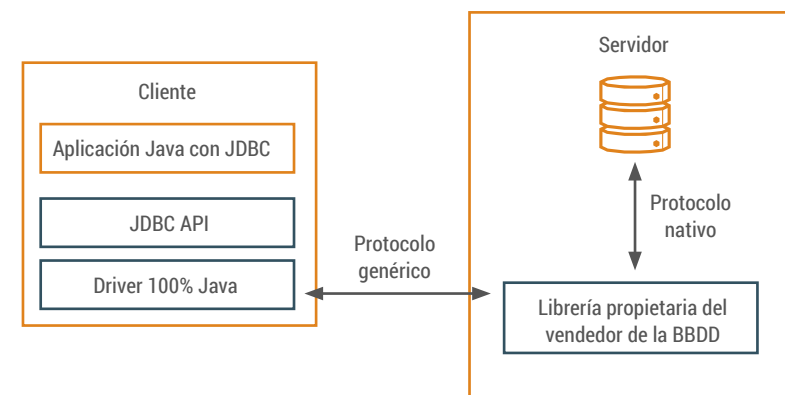


IMAGEN 14.7: TIPO3. JDBC-100% JAVA EN RED.

En la aplicación Java se necesitan las clases del driver JDBC que se vaya a utilizar. El archivo JAR con estas clases tiene que estar en el CLASSPATH.

Por ejemplo si se utiliza MySQL será como el siguiente:

```
CLASSPATH=$CLASSPATH:
/itinerario_donde_esto/mysql-connector-java-5.1.37-bin.jar
```

Para descargar el jar de un driver JDBC habrá que acceder al sitio del proveedor, por ejemplo “**MySQL Connector/J**” es el JDBC oficial para MySQL:

<http://dev.mysql.com/downloads/connector/j/>

4. La excepción SQLException

En casi todas las operaciones relacionadas con el acceso a base de datos con JDBC se deberá capturar o propagar **SQLException**.

Los objetos de la conexión a la base de datos y los objetos que preparan las sentencias SQL, son recursos que se abren y se deben de cerrar. Por esta razón las sentencias que crean estos objetos pueden estar definidas en un **try con recursos**, para así evitar escribir el código de la función **close()**, que a su vez lanza también **SQLException**.

En el ejemplo siguiente una función **lanza SQLException**. Tiene un **try con recursos** con la creación del objeto que crea la sentencia SQL que ejecuta la base de datos. Dentro del bloque **try** las sentencias que manejan la información devuelta por la sentencia SQL utilizan funciones que lanzan **SQLException**, por tanto **try** tiene su correspondiente **catch** para dicha excepción. Esta forma de código es muy típica en los códigos que utilizan el acceso a datos con JDBC:

```
public static void verTabla(Connection con) throws
SQLException {
    String sql = "select idproducto,
        nombre, precio from productos";
    try (Statement stmt = con.createStatement()) {
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            int id = rs.getInt("id");
            String nombre =
rs.getString("nombre");
            float precio = rs.getFloat("precio");
            System.out.println("Producto: [ ID: "+ id +
                ", NOMBRE: + nombre + ", PRECIO: " + precio +
```

```

    “]”);
    }
    } catch (SQLException exc) {
        Sytem.out.println(exc.getMessage());
    }
}

```

5. Cargar el controlador

Los pasos a tener en cuenta para establecer una conexión a base de datos en JBDC son estos cuatro:

- **Importar los paquetes de JDBC:**

El paquete en el que están la mayoría de las clases, interfaces, enumeraciones y excepciones a utilizar es **java.sql**.

JDBC tiene también el paquete **javax.sql**, con clases e interfaces para otro tipo de objetos de conexión como son los DataSource, para colecciones de conexiones y sentencias (pooling), transacciones distribuidas y objetos Rowsets (conjuntos de filas).

Y en determinados tipos de aplicaciones que manejan números para hacerlo con mayor precisión y exactitud se utilizan las clases BigInteger y BigDecimal del paquete **java.math**.

```

import java.sql.*    //para operaciones estándar con
JDBC
import javax.sql.*   //para operaciones con DataSource,
pooling, transacciones    //distribuidas y rowset
import java.math.*    //para la utilización de BigInteger y
BigDecimal

```

■ Cargar o registrar la clase del driver JDBC:

Cargar o registrar el driver es el proceso por el cual el **archivo de la clase del driver se carga en la memoria** y así poder utilizar las implementaciones de las interfaces de dicho driver.

Esta operación sólo se tienen que hacer una vez en la aplicación, y se puede. Puede realizar de dos maneras:

1. Utilizando Class.forName():

```
class.forName(<cadena_paquete_clase_driver>);
```

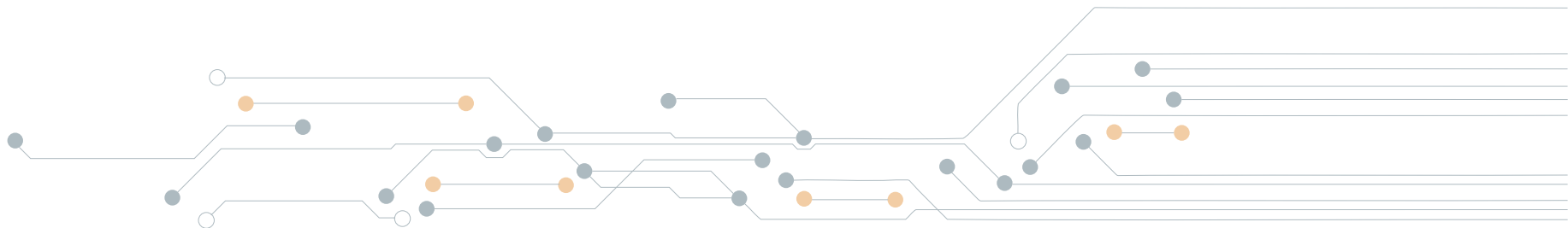
Ejemplos:

```
class.forName("com.mysql.jdbc.Driver");
class.forName("org.apache.derby.jdbc.
ClientDriver");
class.forName("oracle.jdbc.driver.OracleDriver");
```

Esta función lanza la excepción
ClassNotFoundException

En el código siguiente se carga el driver para acceso a base de datos MySQL:

```
public class PruebaDriver {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.
Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("Error: No se puede
cargar el driver: "
                + ex.getMessage());
            System.exit(1);
        }
        System.out.println("Driver cargado con
éxito");
    }
}
```



2. Utilizando registerDriver:

Utilizar el método static `DriverManager.registerDriver()`, se emplea cuando el JDK no es compatible con la JVM, como por ejemplo el proporcionado por Microsoft. Al utilizar esta forma hay que capturar la excepción `SQLException`, que lanza la función `registerDriver()`.

```
public class PruebaDriver {
    public static void main(String[] args) {
        try {
            Driver driver = new com.mysql.jdbc.Driver();
            DriverManager.registerDriver( driver );
        } catch (SQLException ex){
            System.out.println("Error: No se puede
cargar driver: "
                + ex.getMessage());
            System.exit(1);
        }
        System.out.println("Driver cargado con
éxito");
    }
}
```

■ Crear una cadena URL para la conexión a la base de datos:

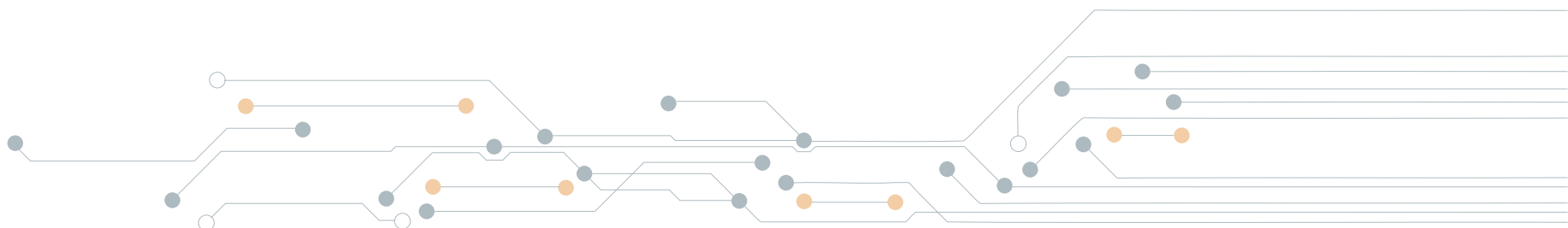
El método **`DriverManager.getConnection`**, tiene como primer parámetro en sus tres formas la cadena que define la URL De la conexión a la base de datos, cuyo formato es:

`jdbc:< subprotocolo>:<nombre_base_datos>`

Ejemplo: `"jdbc:mysql://localhost:3306/cursojdbc"`

■ Hacer la conexión:

Como se estudia en apartado siguiente la conexión se realiza utilizando una de las tres formas del método **`DriverManager.getConnection()`**.



6. Obtener la conexión

Las tres sobrecargas del método static **getConnection** de la clase **DriverManager** son los siguientes:

- **static Connection getConnection(String url) throws SQLException**

En la cadena url, se incluye además de la url de la conexión, los parámetros **user** y **password** de acceso.

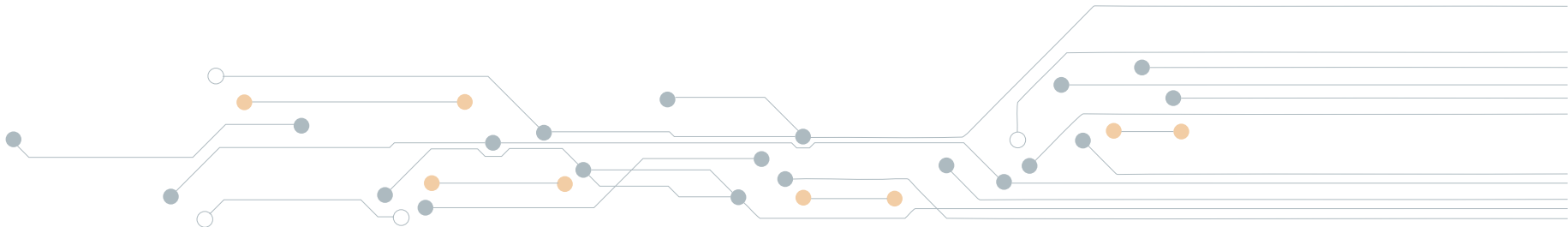
Ejemplo:

```
conexion = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/cursojdbc?
    user=root&password=juan");
```

- **static Connection getConnection(String url, String user, String password) throws SQLException**

Con los tres parámetros necesarios para la conexión en forma de string, la url a la base de datos y el user y password para hacer la conexión.

```
conexion = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/cursojdbc", "root",
    "juan");
```



- **static Connection getConnection(String url, Properties info) throws SQLException**

Además de la cadena **url** para la conexión, el segundo parámetro es una referencia a un objeto **Properties**, que define un conjunto de pares de valores, en el que al menos estar el par formado para los valores de **user** y **password**.

```
Properties connectionProps = new Properties();
connectionProps.put("user", "root");
connectionProps.put("password", "juan");
conexion = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/cursojdbc",
    connectionProps);
```

Al final del programa JDBC, se requiere de forma explícita cerrar todas las conexiones a la base de datos. con el método **close()** del objeto de la conexión. También existe el método **isClosed()** que devuelve un valor true si ya esta cerrada la conexión.

void close() throws SQLException

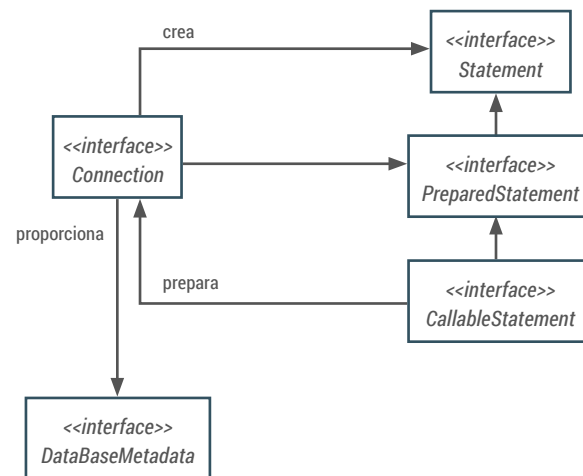
boolean isClosed() throws SQLException

Cómo **close()** lanza la excepción **SQLException**, para no tener que llamarla explícitamente en un bloque try con su catch correspondiente, se puede usar **try con recursos** para la ejecución de **getConnection**, en cualquiera de sus tres formas.



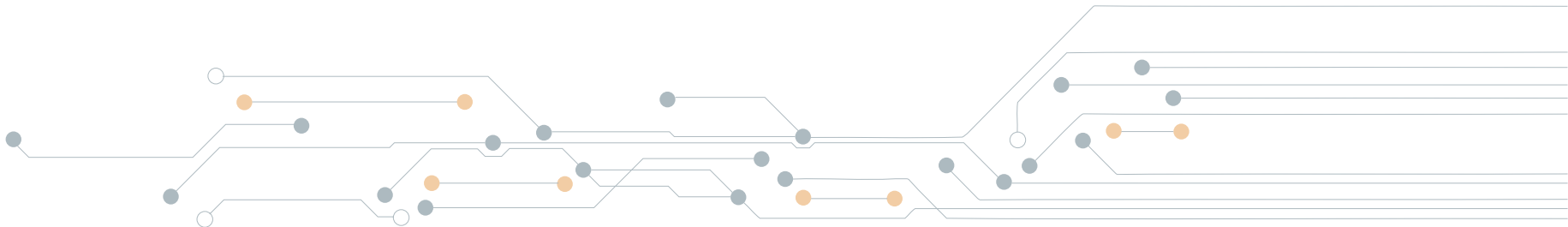
7. Obtener el objeto Statement

Una vez obtenida la conexión a la base de datos, esta se utilizara para realizar operaciones de consulta, inserción y borrado de datos, además de operaciones de creación, modificación y borrado de tablas mediante lenguaje SQL. del diagrama correspondiente a estas.



El interface **Statement** es la que permite realizar todas estas operaciones, junto con sus dos subinterfaces **PreparedStatement** y **CallableStatement**. La relación de estas interfaces es la que se muestra en la imagen anterior.

La instanciación de esta clase se realiza haciendo uso de las diferentes sobrecargas de los tres métodos que proporciona el objeto **Connection**:



Método	Valor devuelto	Funcionalidad
createStatement() createStatement(int resultSetType, int resultSetConcurrency) createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)	Statement	Se utiliza para ejecutar sentencias estáticas de SQL, formadas en Java en cadenas de texto.
prepareStatement(String sql) prepareStatement(String sql, int resultSetType, int resultSetConcurrency) prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	PreparedStatement	Se utiliza para ejecutar sentencias dinámicas o precompiladas de SQL, formadas en Java en cadenas de texto, utilizando el carácter ? para indicar los parámetros que en cada ejecución del SQL pueden tomar un valor diferente.
prepareCall(String sql) prepareCall(String sql, int resultSetType, int resultSetConcurrency) prepareCall(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	CallableStatement	Se utiliza para ejecutar procedimientos almacenados en la base de datos.

Todas estas funciones lanzan la excepción `SQLException`.

Cuando la sentencia SQL se ejecuta y produce un conjunto de filas, lo que se llama **ResultSet** puede definir tres tipos de atributos: tipos, concurrencia y "holdability". En la tabla siguiente se muestran los valores que puede tomar cada uno de estos atributos, cuyos valores son las constantes static de la interface **ResultSet**.

Tabla 14.1: Formas de obtener el Statement con el objeto conexión.

ATRIBUTO	VALOR	Funcionamiento del ResultSet generado
Tipos de ResultSet	TYPE_FORWARD_ONLY	<ul style="list-style-type: none"> • Sólo puede recorrerse hacia adelante
	TYPE_SCROLL_INSENSITIVE	<ul style="list-style-type: none"> • Puede ser recorrido hacia adelante y hacia atrás. • Se puede ir a una posición relativa a partir de la posición actual o a una posición absoluta. • Si los registros del ResultSet se cambian en la base de datos por otro thread o proceso, no se ven reflejados en los ResultSet de este tipo.
	TYPE_SCROLL_SENSITIVE	<ul style="list-style-type: none"> • La navegación y el posicionamiento en los registros es como en el tipo anterior. • Si los registros del ResultSet se cambian en la base de datos por otro thread o proceso, si se ven reflejados en los ResultSet de este tipo.
Concurrency	CONCUR_READ_ONLY	<ul style="list-style-type: none"> • El ResultSet sólo es de lectura.
	CONCUR_UPDATABLE	<ul style="list-style-type: none"> • El ResultSet es de lectura y actualizable.
Holdability	CLOSE_CURSORS_OVER_COMMIT	<ul style="list-style-type: none"> • Las instancias del ResultSet se cierran cuando se ejecuta el commit() de la transacción. Es eficiente para la navegación.
	HOLD_CURSORS_OVER_COMMIT	<ul style="list-style-type: none"> • Las instancias del ResultSet no se cierran cuando se ejecuta el commit() de la transacción. Es eficiente para la actualización de datos.

No todos los ResultSet generados por el Statement tienen todas esas capacidades, se pueden consultar con las funcionalidades de la interface **DatabaseMetaData** proporcionada por **Connection**.

Por omisión los ResultSet son **TYPE_FORWARD_ONLY** y **CONCUR_READ_ONLY**. La "holdability" del ResultSet puede ser determinada llamando a **getHoldability()**.

Tabla 14.2: Definiciones para los diferentes tipos de ResultSet generados por Statement.

En el código siguiente se crea un Statement y un PreparedStatement:

```
public static void main(String[] args) {
    Connection conexion = null;
    Statement statement = null;
    PreparedStatement preparedstatement = null;
    try {
        Class.forName("com.mysql.jdbc.Driver");
        conexion = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/cursojdbc",
            "root", "juan");
        statement = conexion.createStatement();
        System.out.println("Creado Statement");
        preparedstatement = conexion.prepareStatement(
            "SELECT * FROM cursojdbc.producto
WHERE precio > ? ");
        System.out.println("Creado PreparedStatement");
    } catch (ClassNotFoundException ex) {
        System.out.println("Error: No se puede cargar
driver: "
            + ex.getMessage());
        System.exit(1);
    } catch (SQLException ex) {
        System.out.println("Error: Problemas:" +
ex.getMessage());
        System.exit(1);
    } finally {
        if (statement != null){
```

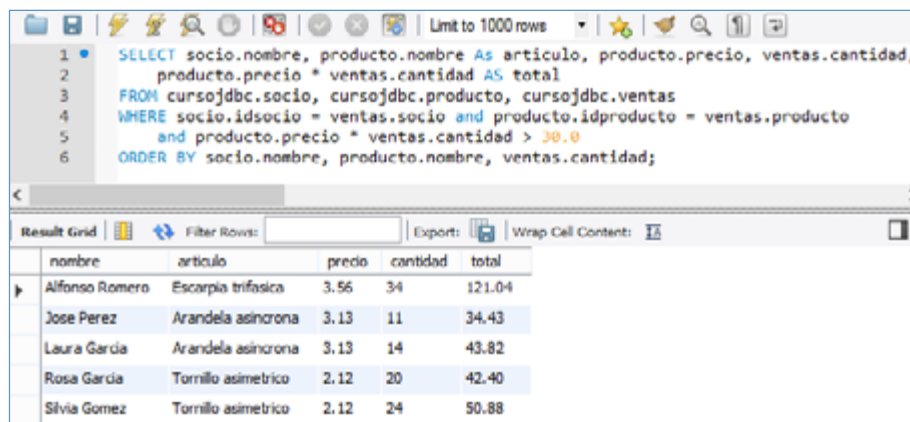
```
        try { statement.close();
        } catch (SQLException ex) {
            System.out
                .println("Error: No se puede cerrar
statement: "
                    + ex.getMessage());
        }
    }
    if (conexion != null) {
        try { conexion.close();
        } catch (SQLException ex) {
            System.out.println("Error: Al cerrar
conexión: "
                + ex.getMessage());
        }
    }
}
```

8. Obtener el ResultSet

El resultado de ejecutar una sentencia SELECT de SQL por medio de un **Statement**, proporciona un **ResultSet**, que se puede comprender mejor si se imagina como una tabla que tiene tantas columnas como la cláusula SELECT y tantas filas como las recuperadas con la cláusula WHERE, de la tabla o tablas especificadas en la cláusula FROM.

Por tanto se pueden obtener ResultSet que tengan cero, una o más filas, pudiendo tener valores nulos en algunas de sus columnas.

En la imagen se muestra el resultado de la ejecución de una sentencia SQL que genera varias filas. El **ResultSet** generado en JDBC tendría esas columnas y filas.



The screenshot shows a SQL IDE with a query editor and a results grid. The query is as follows:

```

1 SELECT socio.nombre, producto.nombre AS articulo, producto.precio, ventas.cantidad,
2    producto.precio * ventas.cantidad AS total
3 FROM cursojdbc.socio, cursojdbc.producto, cursojdbc.ventas
4 WHERE socio.id socio = ventas.socio and producto.id producto = ventas.producto
5    and producto.precio * ventas.cantidad > 30.0
6 ORDER BY socio.nombre, producto.nombre, ventas.cantidad;
  
```

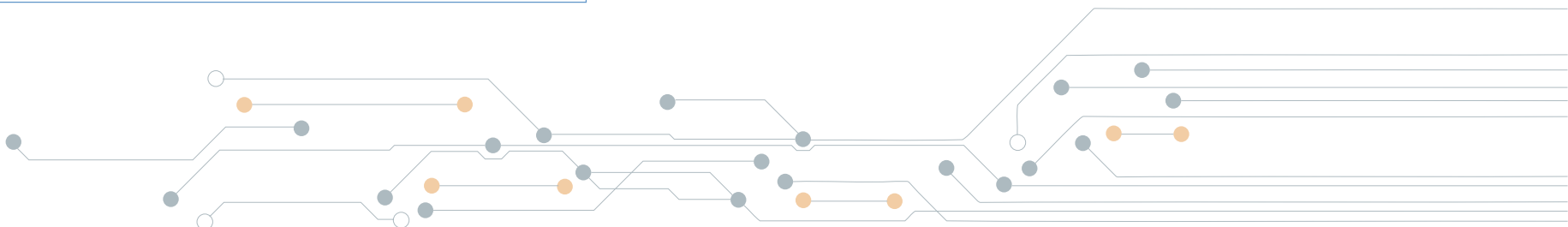
The results grid displays the following data:

nombre	artículo	precio	cantidad	total
Alfonso Romero	Escarpa trifásica	3.56	34	121.04
Jose Perez	Arandela asincrona	3.13	11	34.43
Laura Garcia	Arandela asincrona	3.13	14	43.82
Rosa Garcia	Tornillo asimétrico	2.12	20	42.40
Silvia Gomez	Tornillo asimétrico	2.12	24	50.88

Para obtener un ResultSet hay que ejecutar la función de la interface Statement:

ResultSet executeQuery (String SQL) throws SQLException

En el ejemplo se crea un ResultSet a partir de una sentencia SELECT que genera varias filas en 5 columnas. La información de las columnas se obtiene con la función **getColumnCount()** de la interfaz **ResultSetMetaData**



```

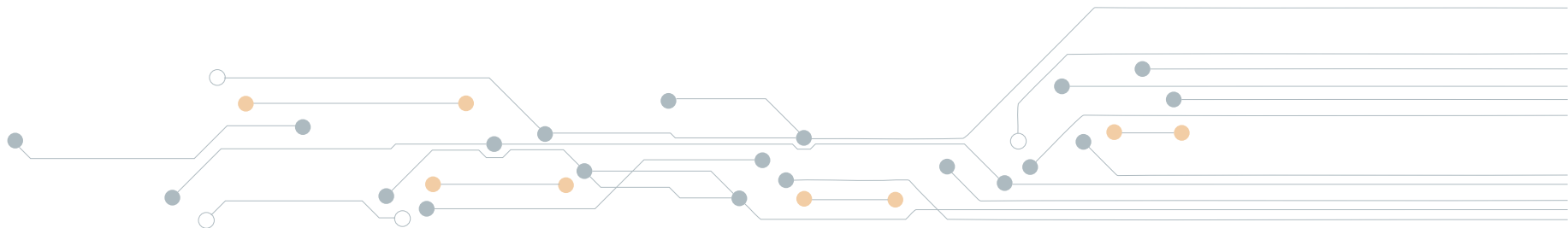
Connection conexion = null;
Statement statement = null;
ResultSet resultset = null;
String sqlSelect="SELECT socio.nombre, producto.
nombre As articulo, "+
        "producto.precio, ventas.cantidad, " +
        "producto.precio * ventas.cantidad AS
total";
String sqlFrom="FROM cursojdbc.socio, cursojdbc.
producto,"+
        " cursojdbc.ventas";
String sqlWhere="WHERE socio.idsocio = ventas.socio
and " +
        "producto.idproducto = ventas.producto and " +
        "producto.precio * ventas.cantidad > 30.0";
String sqlOrder="ORDER BY socio.nombre, producto.
nombre";
String sql = sqlSelect+" "+ sqlFrom+" "+sqlWhere+" "+
sqlOrder;
    try {
        Class.forName("com.mysql.jdbc.
Driver");

```

```

        conexion = DriverManager.
getConnection(
        "jdbc:mysql://localhost:3306/
cursojdbc",
        "root", "juan");
        statement = conexion.
createStatement();
        System.out.println("Creado
Statement");
        resultset = statement.
executeQuery(sql);
        System.out.println("Columnas del
ResultSet: "
        + resultset.getMetaData().
getColumnCount());

```



9. Navegar por el ResultSet

Para recorrer el **ResultSet** se utiliza el método **next()**. Este método devuelve **true** si existe una fila siguiente, y mueve el ResultSet a la siguiente. Si no hay más filas devuelve **false**. LA forma típica de iterar por un ResultSet es la siguiente:

```
while (result.next ()) {  
    // Obtener valores de las columnas de esta fila  
}
```

El **ResultSet** obtenido comienza apuntando antes del primer registro. Una vez se ejecuta **next()** se apunta al primer registro. Cuando **next()** devuelve **false**, el **ResultSet** apunta después del último registro. Todo ResultSet tiene un curso que apunta la fila o registro actual, incluso cuando esta antes que el primero o después del último.

Para obtener el número de filas en un **ResultSet** hay que recorrer todas las filas contándolas. La precaución que hay que tener es que si el **ResultSet** es **TYPE_FORWARD_ONLY**, no se podrá volver hacia atrás.



Además de **next()** para navegar por el **ResultSet** existen las siguientes funciones:

Método	Funcionalidad
boolean absolute(int row)	Mueve el cursor a la fila indicada en el parámetro.
boolean relative(int rows)	Mueve el cursor un número de filas especificado por el parámetro, pudiendo ser el número positivo o negativo.
boolean previous()	Mueve el cursor a la fila anterior, es como next() pero en sentido contrario.
void beforeFirst()	Mueve el puntero a la posición anterior del primer registro.
void afterLast()	Mueve el puntero a la posición después del último registro.
boolean first()	Mueve el puntero al primer registro.
boolean last()	Mueve el puntero al último registro.

Tabla 14.3: Funciones para navegar por el **ResultSet.**

La interface **ResultSet** también tiene funcionalidades para obtener información sobre la posición actual:

Método	Funcionalidad
int getRow()	Devuelve el número de la fila actual.
boolean isAfterLast()	Devuelve true si el cursor esta después de la última.
boolean isBeforeFirst()	Devuelve true si el cursor esta antes que la primera.
boolean isFirst()	Devuelve true si el cursor esta en la primera.
boolean isLast()	Devuelve true si el cursor esta en la última.
int getType()	El número que devuelve es una de las constantes static de ResultSet que definen el tipo de ResultSet : ResultSet.TYPE_FORWARD_ONLY ResultSet.TYPE_SCROLL_INSENSITIVE ResultSet.TYPE_SCROLL_SENSITIVE

Tabla 14.4: Funciones para obtener información de la fila actual del **ResultSet.**

Finalmente la interface **ResultSet** tiene un método para actualizar los valores de las fila actual en la que esta posicionado el cursor con los cambios que se hayan producido en la base de datos, si es **TYPE_SCROLL_SENSITIVE**:

```
void refreshRow()
```

10. Obtener información de las columnas del ResultSet

Para obtener la información de las columnas del ResultSet se utilizan las funciones **getXXX**, donde **XXX** denota un tipo de dato. Con estas funciones se puede acceder a las columnas de dos formas diferentes:

- **Por posición:** útil para acceso a columnas calculadas o cuando hay columnas con el mismo nombre, por ejemplo en join. Su forma es:

```
resultSet.getXXX(numero) //numero int con posición  
de la columna, 1 la primera
```

- **Por nombre:** se accede pasando como parámetro el nombre de la columna. Su forma es:

```
resultSet.getXXX(nombre) //nombre es un String  
con el nombre de la columna
```

Un ejemplo de recuperación de información con función **getString** del **ResultSet** es el siguiente:

```
while(resultSet.next()){  
    nfila++;  
    System.out.print("{Registro "+ nfila+": ";  
    for(int i=1;  
        i<=resultSet.getMetaData().getColumnCount();i++)  
        System.out.print("[ "+resultSet.getString(i)+""]  
    );  
    System.out.println("}");  
}
```

11. Tipos de datos y conversiones

Cuando se ejecuta una función **getXXX** para un determinado ResultSet, el driver JDBC convierte el dato que se va a recuperar al tipo de Java especificado, devolviendo el valor correspondiente. Esta conversión de tipos se realiza por la clase **java.sql.Types**, en la que están definidos los tipos de datos denominados JDBC, en correspondencia con sus equivalentes de datos SQL estándar.

La tabla siguiente muestra la equivalencia de tipos SQL a tipos JDBC, así como las funciones de ResultSet que obtienen valores (getXXX), o modifican el valor (updateXXX), así como las funciones setXXX de PreparedStatement que definen el valor de los parámetros de la consulta.

SQL	JDBC	getXXX	setXXX	updateXXX
VARCHAR	java.lang.String	getString	setString	updateString
CHAR	java.lang.String	getString	setString	updateString
LONGVARCHAR	java.lang.String	getString	setString	updateString
BIT	boolean	getBoolean	setBoolean	updateBoolean
NUMERIC, DECIMAL	java.math.BigDecimal	getBigDecimal	setBigDecimal	updateBigDecimal
TINYINT	byte	getByte	setByte	updateByte
SMALLINT	short	getShort	setShort	updateShort
INTEGER	int	getInt	setInt	updateInt
BIGINT	long	getLong	setLong	updateLong
REAL	float	getFloat	setFloat	updateFloat
FLOAT	float	getFloat	setFloat	updateFloat
DOUBLE	double	getDouble	setDouble	updateDouble
VARBINARY	byte[]	getBytes	setBytes	updateBytes
BINARY	byte[]	getBytes	setBytes	updateBytes
DATE	java.sql.Date	getDate	setDate	updateDate
TIME	java.sql.Time	getTime	setTime	updateTime
TIMESTAMP	java.sql.Timestamp	getTimestamp	setTimestamp	updateTimestamp
CLOB	java.sql.Clob	getClob	setClob	updateClob
BLOB	java.sql.Blob	getBlob	setBlob	updateBlob
ARRAY	java.sql.Array	getArray	setArray	updateArray
REF	java.sql.Ref	getRef	setRef	updateRef
STRUCT	java.sql.Struct	getStruct	setStruct	updateStruct

Tabla 14.5: Equivalencias de tipos SQL a JDBC y sus funciones.

12. Actualizar, borrar e insertar filas en ResultSet

Para actualizar, insertar o borrar filas del ResultSet este tiene que ser de tipo **CONCUR_UPDATABLE**, si no es así estas operaciones se tendrán que realizar con las funciones de la interface **Statement execute** y **executeUpdate**, que reciben como parámetro la sentencia SQL correspondiente, como se estudiara en siguiente.

- **Actualización de los valores de columnas de las filas del ResultSet:**

Cambios en ResultSet: Se ejecutan las funciones **updateXXX** para cada columna a actualizar. Las funciones **updateXXX**, tienen las mismas dos formas que las funciones **getXXX**, por posición o por nombre.

Cambios en la base de datos: Se ejecutan la función **updateRow()** o **refreshRow()**.

Deshacer cambios: Se ejecuta la función **cancelRowUpdates()**

En el código siguiente se muestra como se actualiza el campo "precio" de la tabla "producto" utilizando la función **updateBigDecimal** de un **ResultSet CONCUR_UPDATABLE**

```
statement = conexion.createStatement
                (ResultSet.TYPE_SCROLL_INSENSITIVE,
                 ResultSet.CONCUR_UPDATABLE);
//otras sentencias de formación de String sql
resultset = statement.executeQuery(sql);
//otras sentencias de utilización de resultset

resultset2 = statement.executeQuery("SELECT * FROM producto");
while (resultset2.next()) {
    java.math.BigDecimal precio = resultset2.getBigDecimal("precio");
    java.math.BigDecimal incto = new java.math.BigDecimal(1.5);
    resultset2.updateBigDecimal("precio", precio.multiply(incto));
    resultset2.updateRow();
}
System.out.println("====Actualizados registros====");
if (resultset.isClosed()) resultset =
    statement.executeQuery(sql);
while (resultset.next()) {
    nfila++;
    System.out.print("{Registro " + nfila + ": ");
```

```

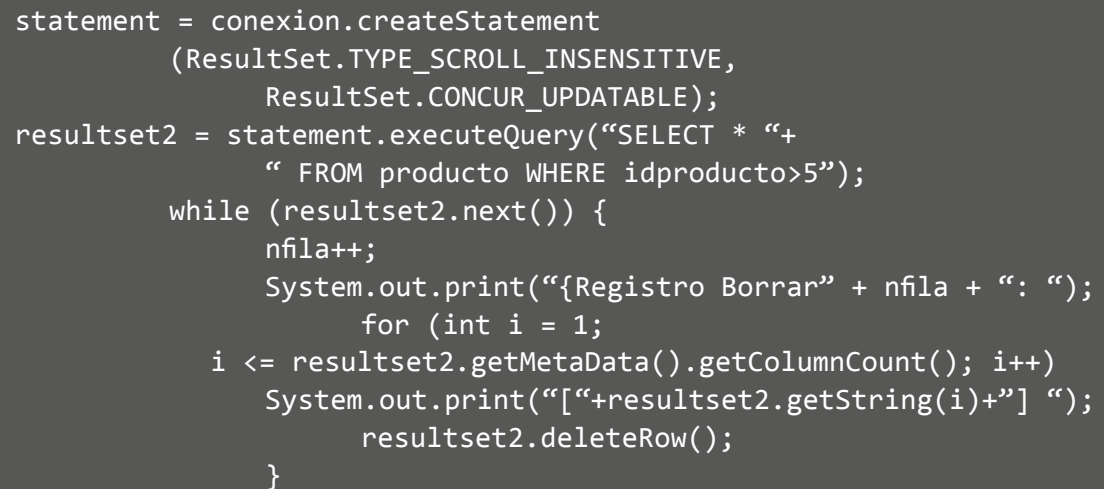
        for (int i = 1; i <=
            resultset.getMetaData().
getColumnCount(); i++)
            System.out.print("[ " + resultset.
getString(i) + " ] ");
            System.out.println("}");
    }

```

■ Borrar filas del ResultSet:

Para borrar una fila del **ResultSet**, primero hay que posicionar el cursor en dicha fila, con las funciones de navegación que se necesiten, después ejecutar la función **deleteRow()** y por último posicionarse en una fila válida del ResultSet.

En el código siguiente se muestra como se borran todos los registros de un ResultSet que cumplen la condición de la sentencia SQL que genera el **ResultSet CONCUR_UPDATABLE**.



```

statement = conexion.createStatement
    (ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
resultset2 = statement.executeQuery("SELECT * "+
    " FROM producto WHERE idproducto>5");
while (resultset2.next()) {
    nfila++;
    System.out.print("{Registro Borrar" + nfila + ": ");
        for (int i = 1;
            i <= resultset2.getMetaData().getColumnCount(); i++)
                System.out.print("[ "+resultset2.getString(i)+" ] ");
                resultset2.deleteRow();
    }

```

■ Insertar filas en ResultSet:

Para insertar una fila en el **ResultSet**, primero hay que posicionar el cursor la nueva fila que todavía no es del **ResultSet** con la función **moveToInsertRow()**, a continuación introducir los valores en las columnas de esa fila con **updateXXX**, ejecutar la función **insertRow()** y para finalizar posicionar el cursor en una fila valida del **ResultSet**.

En el código siguiente se muestra como se inserta una fila en un **ResultSet CONCUR_UPDATABLE**.

```
String sql = "SELECT * FROM cursojdbc.producto";
//sentencias dentro de bloque try
statement = conexion.createStatement
    (ResultSet.TYPE_SCROLL_INSENSITIVE,
     ResultSet.CONCUR_UPDATABLE);
//otras sentencias
resultset.moveToInsertRow();
resultset.updateString("nombre", "Pieza "+
    new Random().nextInt(1000));
resultset.updateBigDecimal("precio",
    new java.math.BigDecimal(new
Random().nextFloat()*10));
resultset.insertRow();
resultset.beforeFirst();
```

```
System.out.println("====Registro insertado
====");
while (resultset.next()) {
    nfila++;
    System.out.print("{Registro " + nfila +
": ";
    for (int i = 1;
        i <= resultset.
getMetaData().getColumnCount(); i++)
        System.out.
print("[ "+resultset.getString(i)+" "]");
    System.out.println("}");
}
```

En la imagen siguiente se hace un repaso por las operaciones principales sobre un ResultSet:

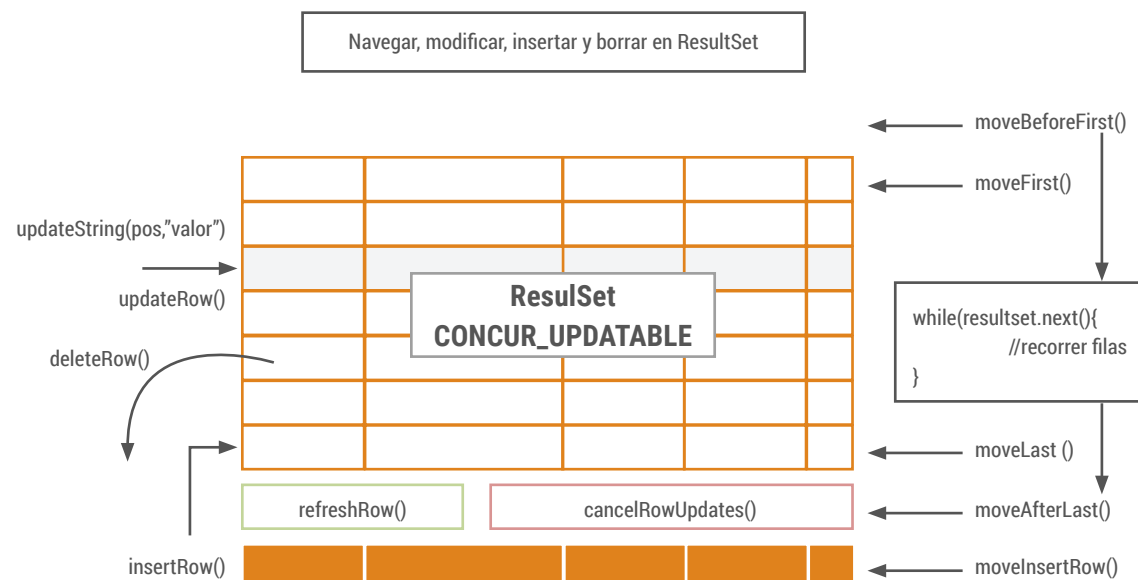


IMAGEN 14.11: OPERACIONES DE NAVEGACIÓN, ACTUALIZACIÓN, INSERCIÓN Y BORRADO EN UN RESULTSET.

13. Actualizar, borrar e insertar registros con sentencias SQL

Para actualizar, insertar o borrar filas del `ResultSet` este tiene que ser de tipo **CONCUR_UPDATABLE**, si no es así estas operaciones se tendrán que realizar con las funciones de la interface **Statement** **execute** y **executeUpdate**, que reciben como parámetro la sentencia SQL correspondiente, como se estudiara en siguiente.

Método	Funcionalidad
boolean execute(String sql)	<p>Ejecuta cualquier tipo de sentencia SQL. Devuelve true si la sentencia SQL (SELECT) genera un ResultSet y false si no se genera ResultSet (INSERT; UPDATE, DELETE; CREATE, ALTER, DROP). Funciones asociadas con el tipo de resultado generado son:</p> <ul style="list-style-type: none"> • ResultSet getResultSet() se obtiene el <code>ResultSet</code> generado. • int getUpdateCount() se obtiene el número de filas actualizadas (0, 1 o más) y -1 si el resultado es un <code>ResultSet</code> o la sentencia SQL no genera resultado (CREATE, ALTER, DROP). • boolean getMoreResults() si devuelve true, significa que se generaron más de un <code>ResultSet</code>, y por tanto con getResultSet() se obtiene el siguiente <code>ResultSet</code>. Esta situación se puede plantear al utilizar callableStatement que ejecutan un procedimiento almacenado.
ResultSet executeQuery(String sql)	Es la función que se utiliza para generar los <code>ResultSet</code> como consecuencia de la ejecución de la sentencia de SQL SELECT. Es la utilizada en los apartados anteriores.
int executeUpdate(String sql)	<p>Ejecuta la sentencia SQL de manipulación de datos (DML INSERT, UPDATE o DELETE) o de definición de datos (DDL).</p> <p>Devuelve el número de filas afectadas por la actualización, devuelve cero si no hay filas afectadas.</p>

Tabla 14.6: Funciones execute, excuteQuery y executeUpdate.

En el código siguiente se muestra la utilización de la función **execute**:

```
statement = conexion.createStatement();
//otras sentencias
for (int i = 1; i <= new Random().nextInt(3) + 1; i++) {
    sql = "INSERT INTO cursojdbc.producto (nombre, precio) "+
        "VALUES ('PRODUCTO_' + new Random().nextInt(1000) +
        '", " + new Random().nextFloat() * 10 +
        "')";
    result = statement.execute(sql);
    if (!result) n+=statement.getUpdateCount();
}
```

Y en código siguiente se muestra la utilización de **executeUpdate**:

```
statement = conexion.createStatement();
//otras sentencias
sql = "DELETE FROM cursojdbc.producto WHERE nombre like 'PRODUCTO%';";
int n = statement.executeUpdate(sql);
System.out.println("--Registros borrados: " + n);
```



14. Ejecutar SQL con PreparedStatement

La interface **PreparedStatement** representa una sentencia SQL precompilada, estas sentencias tienen parámetros que en el momento de la ejecución de la consulta son sustituidos por valores, de tal manera que cada vez que se ejecuta la sentencia SQL, si los valores son distintos el resultado de la ejecución es distinto.

En el ejemplo una sentencia SQL INSERT parametrizada. Cada ? es un parámetro, que será sustituido en tiempo de ejecución por el valor definido con una **SET**, primera interrogación con primera **SET** y así sucesivamente. El primer **null** equivale a que no se introduce valor, porque el campo es una clave autoincrementada generada por el motor de base de datos. EXECUTE ejecuta la sentencia con los valores ya definidos, en este caso se ejecuta dos veces con valores distintos. DEALLOCATE dice al gestor de base de datos que ya no se va usar más esta sentencia PREPARE.

```
PREPARE insertar FROM
```

```
    "INSERT INTO cursojdbc.ventas VALUES (null, ?,  
    ?, ?)";
```

```
SET @socio=1; SET @producto=2; SET @cantidad=5;  
EXECUTE insertar USING @socio,@producto,@cantidad;  
SET @socio=1; SET @producto=4; SET @cantidad=10;  
EXECUTE insertar USING @socio,@producto,@cantidad;  
DEALLOCATE PREPARE insertar;
```

El proceso para ejecutar sentencias SQL parametrizadas con la interface **PreparedStatement** es el siguiente:

- Se forma la cadena con la sentencia SQL, incluyendo tantos caracteres ? como parámetros deben ser sustituidos por un valor en la ejecución.
- Se obtiene la referencia a **PreparedStatement** a través del objeto **Connection** de la forma siguiente:

```
Connection con = DriverManager.getConnection(strUrl,  
strUser, strPassword);  
PreparedStatement pre =con.prepareStatement(java.  
lang.String);
```

- Se definen los valores para cada parámetro (?) con las funciones **setXXX**, que se correspondan para cada tipo de datos de cada parámetro:

```
pre.setXXX(posicion_parametro, valor_tipo_XXX);  
//La primera interrogación tiene la posición 1  
Ejemplo: pre.setInt(2, 154) //Para segunda ? columna int  
un valor int;
```

- Por último se ejecuta la sentencia SQL parametrizada con las funciones de la interface PreparedStatement: **execute()**, **executeQuery()** para consultas, **executeUpdate()** para actualizaciones.
- En el código siguiente se muestra la utilización de **PreparedStatement** con **executeUpdate** con sentencia SQL INSERT:

```
sql = "INSERT INTO cursojdbc.producto VALUES (null,  
?, ?)";  
pre = conexion.prepareStatement(sql);  
for (int i = 0; i <= new Random().nextInt(5); i++) {  
    pre.setString(1, "COSA " + new Random().  
nextInt());  
    pre.setFloat(2, new Random().nextFloat() *  
10);  
    n += pre.executeUpdate();  
}  
System.out.println("--Registros añadidos: " + n);
```



En el código siguiente se muestra la utilización de **PreparedStatement** con **executeQuery** con sentencia SQL SELECT que proporciona un **ResultSet**:

```
sql = "SELECT * FROM cursojdbc.producto WHERE precio > ?";
pre = conexion.prepareStatement(sql);
float precio=new Random().nextFloat()*100;
pre.setFloat(1, precio);
System.out.println("===REGISTROS CON PRECIO MAYOR QUE "+precio);
resultset = pre.executeQuery();
while (resultset.next()) {
    nfila++;
    System.out.print("{Registro " + nfila + ": ");
    for (int i = 1;
        i <= resultset.getMetaData().getColumnCount(); i++)
        System.out.print("[ " + resultset.getString(i) + " ] ");
    System.out.println("}");
}
```



15. Acceso a los metadatos

Los metadatos son datos o información que explican la naturaleza, característica o propiedad de otros datos. Con el acceso a los metadatos poder conocer la estructura y algunas característica de la base de datos y de los resultset, permitiendo que el código sea más independiente del conocimiento del esquema de la base de datos.

En ejemplos anteriores se ha estado obteniendo el metadato de las columnas del ResultSet, sin tener que conocer ni su nombre, ni su número.

Las interfaces de JDBC que dan información de los metadatos son:

- **DatabaseMetaData**, tiene más de 150 funciones para recuperar información de la base de datos, información de catálogos, esquemas, tablas, tipo de las tablas, columnas de las tablas, procedimientos almacenados, vistas. Además de información de algunas de las características del driver JDBC utilizado.

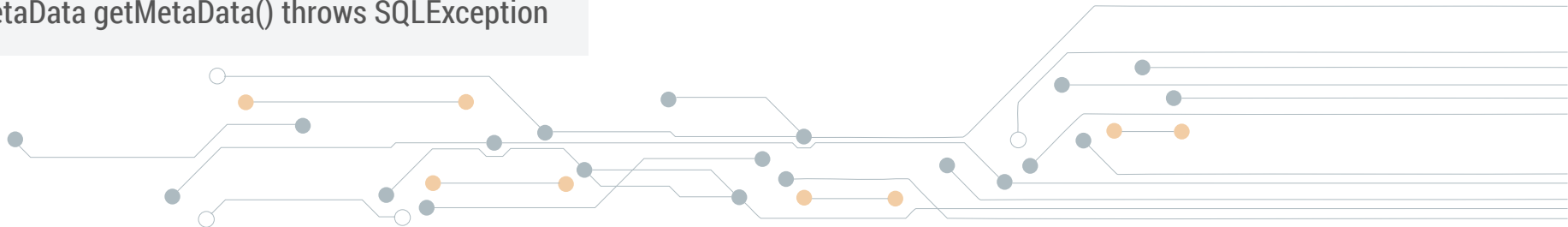
Se obtiene por medio del objeto **Connection** y su función:

```
DatabaseMetaData getMetaData() throws SQLException
```

- **ResultSetMetaData**, permiten determinar las características de un objeto **ResultSet**, como son el número de columnas, información de cada columna, como su tipo, tamaño, precisión, si puede contener nulos, etc.

Se obtiene por medio del **ResultSet** y su función:

```
ResultSetMetaData getMetaData()throws SQLException
```



- **ParameterMetaData**, permite obtener información sobre el número, tipos y propiedades de los parámetros de un **PreparedStatement**.

Se obtiene por medio del PreparedStatement y su función:

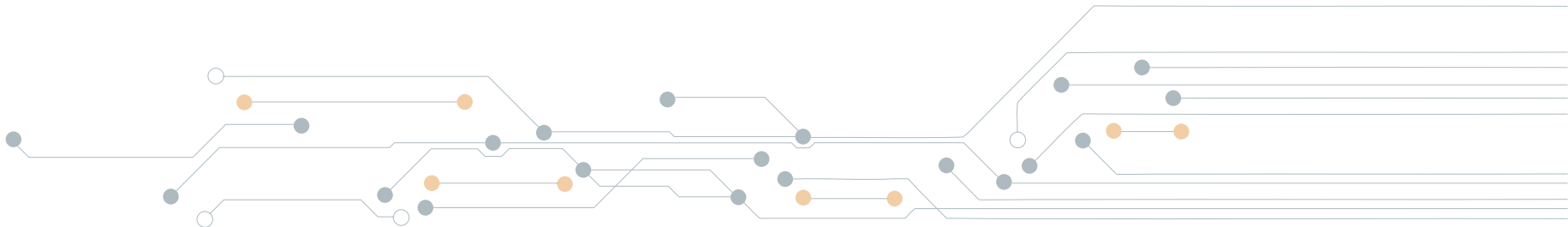
ParameterMetaData getParameterMetaData() throws
SQLException

En el ejemplo siguiente se obtiene información de DatabaseMetaData.

```
DatabaseMetaData dbmt = conexion.getMetaData();

System.out.print("Producto: " + dbmt.
getDatabaseProductName());
System.out.println(" - Version: "
+ dbmt.getDatabaseProductVersion());
System.out.println("Driver: " + dbmt.
getDriverName());
System.out.println("URL: " + dbmt.getURL());
System.out.println("User: " + dbmt.
getUserName());
```

```
ResultSet resultcatalogos = dbmt.getCatalogs();
System.out.println("====CATALOGOS====");
while (resultcatalogos.next()) {
    System.out.println("    "
+ resultcatalogos.getString("TABLE_CAT"));
}
System.out.println("====TABLAS====");
ResultSet resulttablas = dbmt.getTables(null,
null, null, null);
while (resulttablas.next()) {
    System.out.println("    " + resulttablas.
getString(3));
}
```



Telefonica

EDUCACIÓN DIGITAL