1. INTRODUCCIÓN

En general, el término de **acceso a datos** significa el proceso de recuperación o manipulación de datos extraídos de un origen de datos local o remoto. El origen de datos puede provenir de diversas fuentes diferentes: una base de datos relacional remota en un servidor, una base de datos relacional local, una hoja de cálculo, un fichero de texto, un servicio de información online, etc.

Los **sistemas gestores de bases de datos (DBMS)** contienen lenguajes propios (como SQL) para gestionar los datos mediante operaciones de consulta y manipulaciones de datos. Sin embargo, cuando se quiere acceder a los datos desde un lenguaje de programación de una misma manera con independencia de sistema gestor, entonces es necesario utilizar conectores. Los conectores proporcionan al programador una forma homogénea de acceder a cualquier DBMS (preferiblemente relacional u objeto-relacional).

Para acceder a una base de datos y realizar consultas o manipulaciones de datos, una aplicación siempre necesita tener un conector asociado. Se llama **conector** al conjunto de clases encargadas de implementar la interfaz de programación de aplicaciones (API) y facilitar el acceso a una base de datos.

2. EL DESFASE OBJETO-RELACIONAL

Actualmente las bases de datos orientadas a objetos están ganando terreno frente a las bases de datos relacionales, junto con el auge de la programación orientada a objetos.

El problema del desfase objeto-relacional consiste en la diferencia de aspectos que existen entre la **programación orientada a objetos**, con la que se desarrollan aplicaciones, y la **base de datos**, con la que se almacena la información. Estos aspectos se pueden presentar relacionados cuando:

- Se realizan **actividades de programación**, donde el programador debe conocer el lenguaje de programación orientado a objetos y el lenguaje de acceso a datos.
- Se realiza una **especificación de los tipos de datos**. En las bases de datos relacionales siempre hay restricciones en cuanto a los tipos de datos que se pueden usar, mientras que en la programación orientada a objetos se utilizan tipos de datos complejos.
- Se realiza una **traducción del modelo orientado a objetos al modelo entidad-relación** en el proceso de elaboración del software. El primer modelo maneja objetos y el segundo maneja tablas y tuplas (filas), lo que implica que el desarrollador tiene que diseñar dos diagramas diferentes para el diseño de la aplicación.

La discrepancia objeto-relacional surge porque en el modelo relacional se trata con relaciones y conjuntos debido a su naturaleza matemática. Sin embargo, en el modelo de programación orientada a objetos se trabaja con objetos y asociaciones entre ellos. A este problema se le denomina **desfase objeto-relacional** y se define como el conjunto de dificultades técnicas que aparecen cuando una base de datos relacional se usa conjuntamente con un programa escrito con un lenguaje de programación orientado a objetos.

No obstante, estos problemas tienen solución. Cada vez que los objetos deben extraerse o almacenarse en una base de datos relacional, se requiere un **mapeo objeto-relacional**: desde las estructuras provistas en el modelo de datos a las provistas por el entorno de programación.

3. PROTOCOLOS DE ACCESO A BASES DE DATOS

Muchos servidores de bases de datos utilizan protocolos de comunicación específicos que facilitan el acceso a los mismos, lo que obliga a aprender un lenguaje nuevo para trabajar con cada uno de ellos. Para reducir esta diversidad de protocolos, se puede utilizar varias **interfaces de alto nivel** que ofrezcan al programador una serie de métodos de acceso a la base de datos. Dichas interfaces ofrecen facilidades para:

- Establecer una conexión a una base de datos.
- Ejecutar consultas sobre una base de datos.
- Procesar los resultados de las consultas realizadas.

Las tecnologías disponibles abstraen la complejidad subyacente y proporcionan una interfaz común, basada en el lenguaje de consulta estructurado (SQL), para el acceso homogéneo a los datos.

Se denomina **conector** o **driver** al conjunto de clases encargado de implementar la interfaz de programación de aplicaciones (API) y facilitar el acceso a una base de datos. Para poder conectarse a una base de datos y lanzar consultas, una aplicación siempre necesita tener un conector asociado, que oculta los detalles específicos de la base de datos.

Existen dos normas de conexión a una base de datos relacional SQL:

1) **ODBC (Open DataBase Connectivity)** de Microsoft. Define una API (escrita en C) que las aplicaciones pueden usar para abrir una conexión con una base de datos, enviar consultas, actualizaciones y obtener resultados.
2) **JDBC (Java DataBase Connectivity)** de Sun. Define una API que los programas Java pueden usar para conectarse a una base de datos relacional. Es independiente de la plataforma y del gestor de bases de datos utilizado.

Además, existen otros protocolos que se utilizan cuando el origen de datos no es una base de datos relacional, por ejemplo, en el caso de los ficheros de texto planos o los almacenes de correo electrónico:

- **OLE-DB (Object Linking and Embedding for DataBases)** de Microsoft. Es una API (escrita en C++) con objetivos parecidos a los de ODBC pero para orígenes de datos que no son bases de datos. Proporciona estructuras para la conexión con orígenes de datos, ejecución de comandos y devolución de resultados en forma de conjunto de filas. Los programas OLE-DB pueden negociar con el origen de datos para averiguar las interfaces que soporta y los comandos pueden estar en cualquier lenguaje soportado por el origen de datos (SQL, subconjunto limitado de SQL, sin capacidad de consultas, etc.).
- **ADO (Active Data Objects)** de Microsoft. Es una API que ofrece una interfaz sencilla de utilizar con la funcionalidad OLE-DB y que se puede llamar desde los lenguajes de guiones como VBScript y Jscript.

3.1. ACCESO A DATOS MEDIANTE ODBC

ODBC (Open DataBase Connectivity) es un estándar de acceso a bases de datos desarrollado por Microsoft con el objetivo de posibilitar el acceso a cualquier dato desde cualquier aplicación, sin importar qué sistema gestor de bases de datos se utilice. Cada sistema gestor de bases de datos (DBMS) compatible con ODBC proporciona una biblioteca que se debe enlazar con el programa. Cuando el programa realiza una llamada a la API de ODBC, dicha biblioteca se comunica con el DBMS para realizar la acción solicitada y obtener resultados.

Para utilizar ODBC en un programa, se siguen estos pasos:

1) **Configurar la interfaz ODBC.** En primer lugar, el programa asigna un entorno SQL con la función *SQLAllocHandle()*, y después un manejador (o *handle*) para la conexión a la base de datos basada en el entorno anterior. Este manejador traduce las consultas de datos de la aplicación en comandos que el sistema gestor de base de datos entienda. ODBC define varios tipos de manejadores:
   - SQLHENV: define el entorno de acceso a los datos.
   - SQLHDBC: identifica el estado y configuración de la conexión (driver y origen de datos).
   - SQLHSTMT: declara sentencias SQL y cualquier conjunto de resultados asociados.
   - SQLHDESC: recolecta metadatos utilizados para describir sentencias SQL.
2) **Abrir una conexión con la base de datos**, usando las funciones *SQLDriverConnect()* o *SQLConnect().*
3) **Enviar órdenes SQL al sistema gestor de la base de datos**, usando la función *SQLExecDirect().*
4) **Desconectarse de la base de datos y liberar la conexión y los manejadores del entorno SQL**, usando las funciones *SQLFreeHandle()* y *SQLDisconnect().*

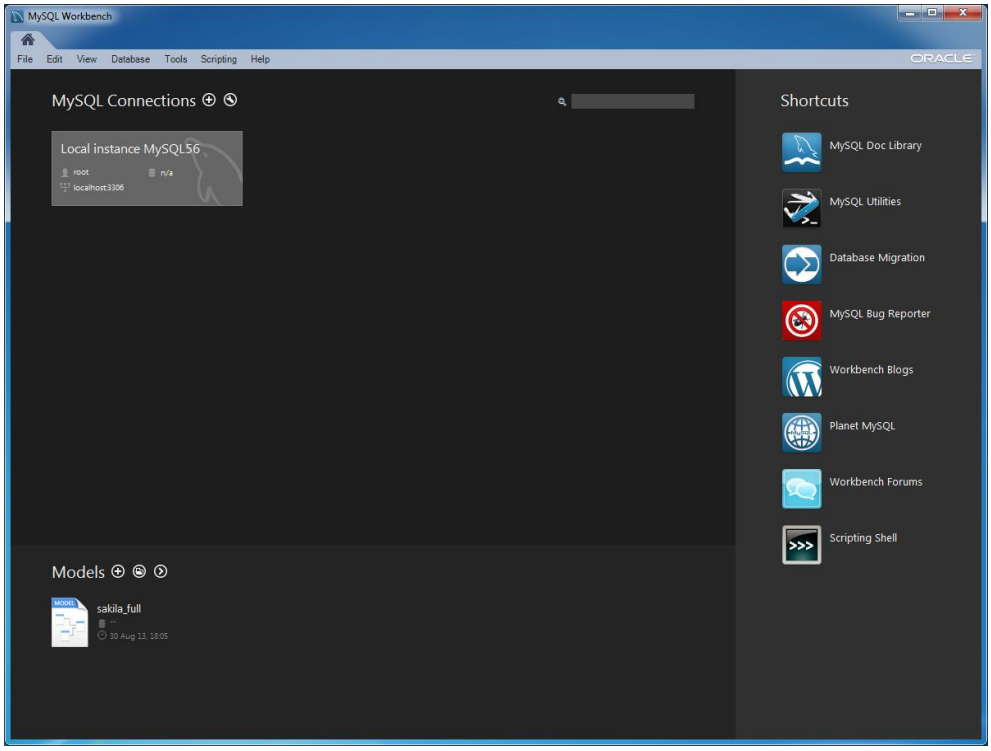
La API ODBC usa una interfaz escrita en el lenguaje C y no es apropiada para su uso directo con Java. Las llamadas desde Java a código C nativo tienen inconvenientes respecto a la seguridad, la implementación, la robustez y la portabilidad de las aplicaciones.

FUNCIÓN	EXPLICACIÓN
`SQLAllocStmt()` `SQLExecDirect()`	Ejecutan sentencias SQL sobre la base de datos.
`SQLAllocHandle()`	Asigna descriptores de contexto de entorno, de conexión y de sentencia.
`SQLDriverConnect()`	Soporta orígenes de datos que requieren más información de la conexión que los 3 argumentos de la función **SQLConnect()**.
`SQLFreeHandle()`	Libera recursos asociados a un entorno, una conexión, una sentencia o un manejador.
`SQLDisconnect()`	Cierra la conexión asociada a un manejador específico.
`SQLFetch()`	Obtiene el siguiente conjunto de filas del "resulset" y devuelve los datos de todas las columnas asociadas.
`SQLGetData()`	Obtiene datos de una sola columna del "resultset" o de un solo parámetro después de que **SQLParamData()** devuelva SQL_PARAM_DATA_AVAILABLE.
`SQLNumResultCols()`	Devuelve el número de columnas de un "resulset".
`SQLRowCount()`	Devuelve el número de filas afectadas por una sentencia de actualización, inserción o eliminación. Una operación SQL_ADD, SQL_UPDATE_BY_BOOKMARK, o SQL_DELETE_BY_BOOKMARK en **SQLBulkOperations()**; o una operación SQL_UPDATE o SQL_DELETE en **SQLSetPos()**.

El siguiente programa C accede mediante ODBC a una base de datos MySQL llamada `ejemplo` que tiene creadas las tablas `empleado` y `departamento`.

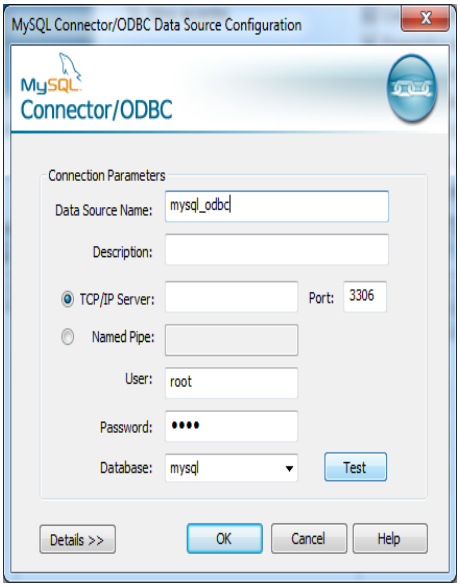
```c
#include <windows.h>
#include <odbcinst.h>
#include <stdio.h>
#include <sql.h>
#include <sqlext.h>
#include <sqltypes.h>

int main() {
    // definir el entorno de acceso
    SQLHENV env;
    // identificar el estado y la configuración de la conexión
    SQLHDBC dbc;
    // sentencia SQL y resultados asociados
    SQLHSTMT stmt;
    SQLRETURN ret;

    // definir el entorno de la conexión
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
    // definir los atributos del entorno de la conexión
    SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*) SQL_OV_ODBC3, 0);
    SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);

    // conectarse al origen de datos MySQL con ODBC
    ret = SQLDriverConnect(dbc, NULL, (SQLCHAR*) "FILEDSN=odbc;PWD={root};",
        SQL_NTS, NULL, 0, NULL, SQL_DRIVER_COMPLETE);

    // crear una sentencia SQL de consulta
    char* consulta = "SELECT * FROM departamento";
    SQLSMALLINT nColumnas = 0;
    SQLINTEGER nFilas = 0, indicador = 0;
    SQLSCHAR buf[1024] = {0};

    int i;
    if (SQL_SUCCEEDED(ret)) {
        printf("\nConectado.");
        // ejecutar la sentencia SQL de consulta y obtener resultados
        SQLAllocStmt(dbc, &stmt);
        ret = SQLExecDirect(stmt, consulta, SQL_NTS);

        // obtener número de columnas y número de filas de los resultados
        SQLNumResultCols(stmt, &nColumnas);
        SQLRowCount(stmt, &nFilas);
        printf("\nNúmero de Columnas: %d",nColumnas);
        printf("\nNúmero de Filas: %d", nFilas);

        // recorrer el resultado de la consulta
        while (SQL_SUCCEEDED(ret = SQLFetch(stmt))) {
            printf("\n");
            for (i = 1 ; i <= nColumnas ; i++) {
                ret = SQLGetData(stmt, i, SQL_C_CHAR, buf, 1024, &indicador);
                if (SQL_SUCCEEDED(ret)) {
                    printf(" * %s", buf);
                }
            } //fin de for
        } //fin de while

        // liberar recursos asociados con la sentencia SQL
        SQLFreeHandle(SQL_HANDLE_STMT, stmt);
        // cerrar la conexión OBDC
        SQLDisconnect(dbc);
    }
    else {
        printf("\nNo ha sido posible realizar la conexión.");
    }
    return 1;
} //fin de main
```

Previamente se debe instalar MySQL (servidor, workbench y otros) en Windows.



Después se necesita descargar el conector ODBC para MySQL desde el sitio web de MySQL: http://dev.mysql.com/downloads/connector/odbc/

Por último, hay que crear el DSN de acceso a la base de datos MySQL. Para ello, hay que ir al *Administrador de Orígenes de Datos de ODBC* (en *Herramientas Administrativas*) y tendrá que estar abierta la conexión con MySQL.



Una vez compilado este programa C, el resultado de su ejecución es el siguiente:

```
Conectado.
Número de Columnas: 3
Número de Filas: 4
 * 10 * Contabilidad * Sevilla
 * 20 * Investigacion * Madrid
 * 30 * Ventas * Barcelona
 * 40 * Produccion * Bilbao
```

Para más información:

- Guía de referencia de ODBC para el programador: https://msdn.microsoft.com/en-us/library/windows/desktop/ms714177(v=vs.85).aspx
- Tutoriales y ejemplos escritos en C de ODBC: http://www.easysoft.com/developer/languages/c/index.html

3.2. ACCESO A DATOS MEDIANTE JDBC

JDBC (Java DataBase Connectivity) proporciona una librería estándar para acceder a datos, principalmente provenientes de bases de datos relacionales que usan SQL. Además de proporcionar una interfaz, define una arquitectura estándar para que los fabricantes puedan crear drivers que permitan a las aplicaciones Java el acceso a los datos. JDBC dispone de una interfaz distinta para cada base de datos (se denomina driver, controlador o conector). Esto permite que las llamadas a los métodos Java de las clases JDBC se correspondan con la API de la base de datos.

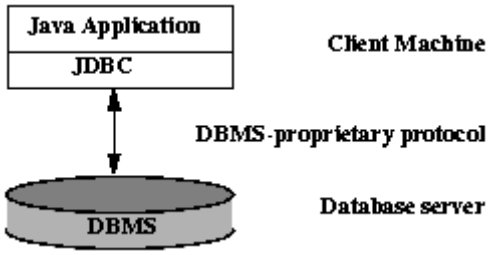
JDBC consta de un conjunto de clases e interfaces que permite escribir aplicaciones Java para gestionar las siguientes tareas con una base de datos relacional:

- Conectarse a la base de datos.
- Enviar sentencias de consulta y modificación a la base de datos.
- Recuperar y procesar los resultados recibidos de la base de datos.

3.2.1. Arquitecturas

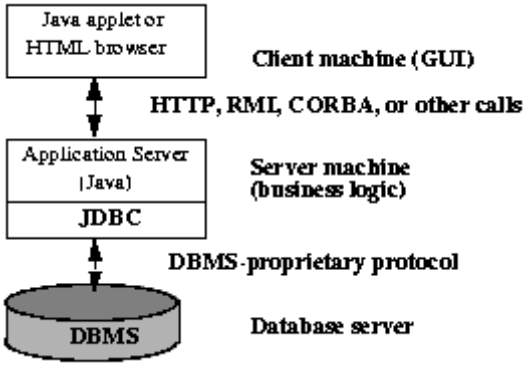
A la hora de establecer el canal de comunicación entre una aplicación Java y una base de datos se pueden identificar dos modelos distintos de acceso. Estos modelos dependen del número de capas que se contemple.

En el **modelo de dos capas**, la aplicación Java habla directamente con la fuente de datos. Esto requiere un driver JDBC residiendo en el mismo lugar que la aplicación. Desde la aplicación Java se envían sentencias SQL al sistema gestor de base de datos para que las procese y los resultados se devuelven al programa.



La base de datos puede encontrarse en otra máquina diferente a la de la aplicación Java, y en este caso, las solicitudes se hacen a través de la red. Esta configuración se denomina arquitectura cliente-servidor, y en ella, el driver JDBC se encarga de manejar la comunicación a través de la red de forma transparente a la aplicación Java.

En el **modelo de tres capas**, la aplicación Java envía los comandos SQL a una capa intermedia de servicios, que a su vez los reenvía a la fuente de datos. Entonces la base de datos procesa dichas sentencias SQL y envía los resultados de la ejecución a la capa intermedia, para que después los reenvíe a la aplicación Java.



En este modelo, la aplicación Java se ejecuta en una máquina y accede al driver JDBC de base de datos situado en otra máquina. Varios ejemplos de puesta en práctica son:

- Cuando se tiene una aplicación Java accediendo al driver JDBC a través de un servidor web.
- Cuando una aplicación Java accede a un servidor remoto que comunica localmente con el driver JDBC.
- Cuando una aplicación Java, que está en comunicación con un servidor de aplicaciones, accede a la base de datos.

3.2.2. Componentes y Tipos de Conectores

El conector JDBC incluye cuatro componentes:

1) **La propia API de JDBC.** Facilita el acceso desde el lenguaje de programación Java a bases de datos relacionales y permite que se puedan ejecutar sentencias SQL, recuperar resultados y realizar cambios en la base de datos. Dicha API está disponible en los paquetes *java.sql* y *javax.sql*, ambos incluidos en las plataformas Java SE y EE.

2) **El gestor del conector JDBC.** Conecta una aplicación Java con el driver correcto de JDBC. Se puede realizar por conexión directa (*DriverManager*) o a través de un pool de conexiones (*DataSource*).

3) **La suite de pruebas JDBC.** Se encarga de comprobar si un conector (*driver*) cumple con los requisitos de JDBC.

4) **El puente JDBC-OBDC.** Proporciona acceso a través de un driver ODBC. Se necesita cargar el código binario ODBC en la máquina cliente que usa este conector.

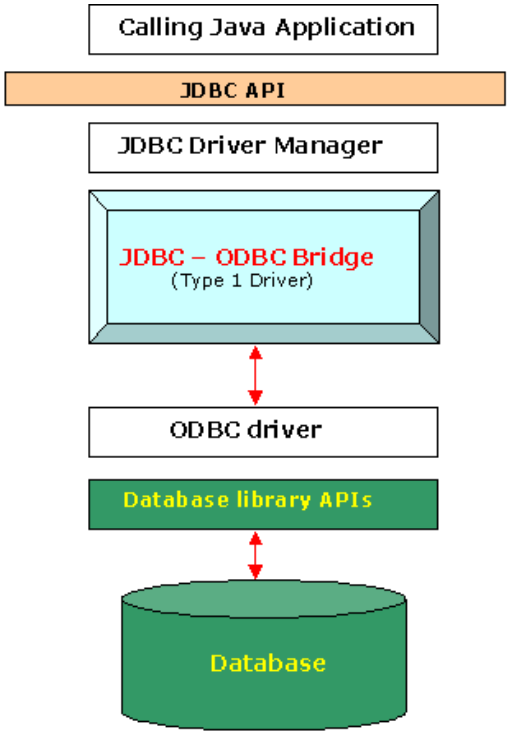
En función de los componentes anteriores, existen cuatro tipos de conectores JDBC. La denominación de estos controladores viene determinada por el grado de independencia respecto de la plataforma y las prestaciones:

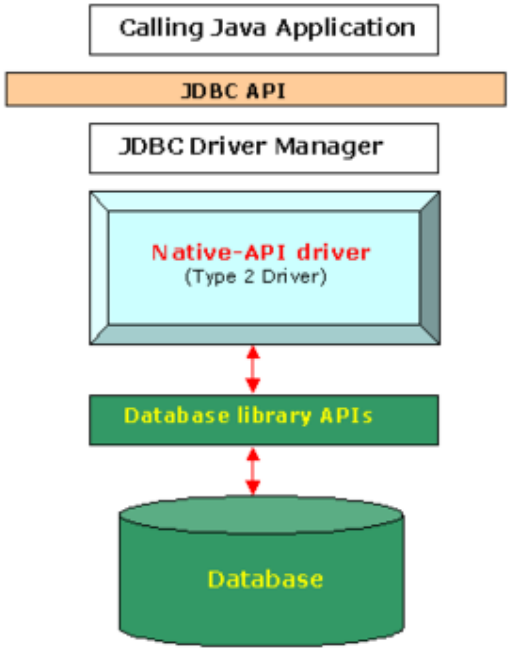
1) **Tipo 1: JDBC-ODBC bridge.** Permite el acceso a bases de datos JDBC mediante un driver ODBC. Exige la instalación y configuración de ODBC en la máquina cliente. Utiliza una API nativa estándar, que traduce las llamadas de JDBC a invocaciones ODBC a través de librerías ODBC del sistema operativo.

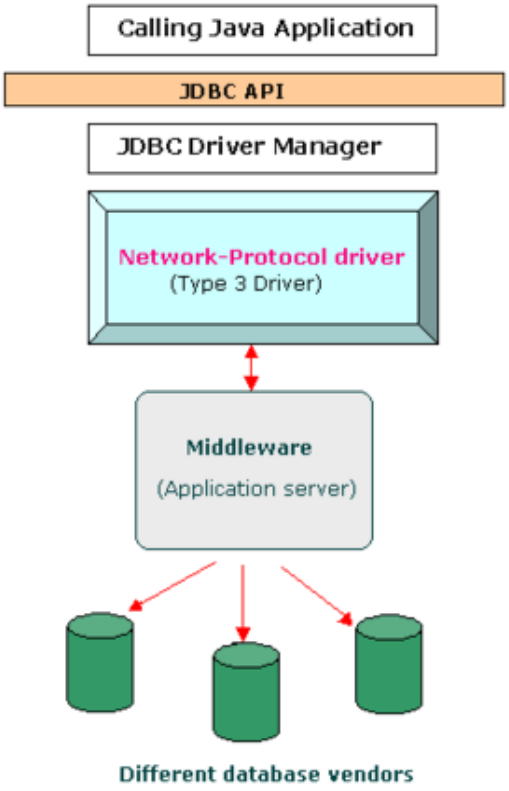
2) **Tipo 2: Native-API driver.** Es un controlador escrito parcialmente en Java y en código nativo, es decir, utiliza una API nativa de la base de datos. Traduce las llamadas al API de JDBC Java en llamadas propias del motor de la base de datos. Exige la instalación de código binario propio del cliente de base de datos y del sistema operativo en la máquina cliente.

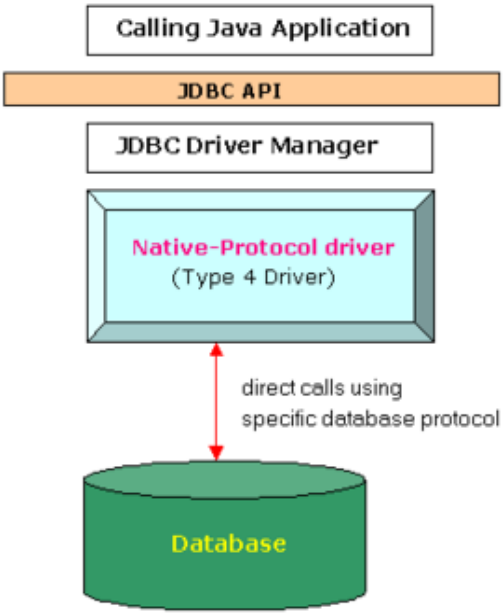
3) **Tipo 3: Network-Protocol driver.** Es un controlador de Java puro que utiliza un protocolo de red (HTTP, TCP/IP), que es independiente de la base de datos, para comunicarse con un servidor remoto de base de datos (middleware). Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red. No exige una instalación en el cliente.

4) **Tipo 4: Database-Protocol driver.** Es un controlador de Java puro con protocolo nativo que es suministrado por el fabricante de la base de datos. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red usado por el motor de la base de datos. No exige una instalación en el cliente. Éste es el método más eficiente de acceso a la base de datos.


Calling Java Application
JDBC API
JDBC Driver Manager
JDBC – ODBC Bridge
(Type 1 Driver)
ODBC driver
Database library APIs
Database


Calling Java Application
JDBC API
JDBC Driver Manager
Native-API driver
(Type 2 Driver)
Database library APIs
Database


Calling Java Application
JDBC API
JDBC Driver Manager
Network-Protocol driver
(Type 3 Driver)
Middleware
(Application server)
Different database vendors


Calling Java Application
JDBC API
JDBC Driver Manager
Native-Protocol driver
(Type 4 Driver)
direct calls using
specific database protocol
Database

3.2.3. Funcionamiento

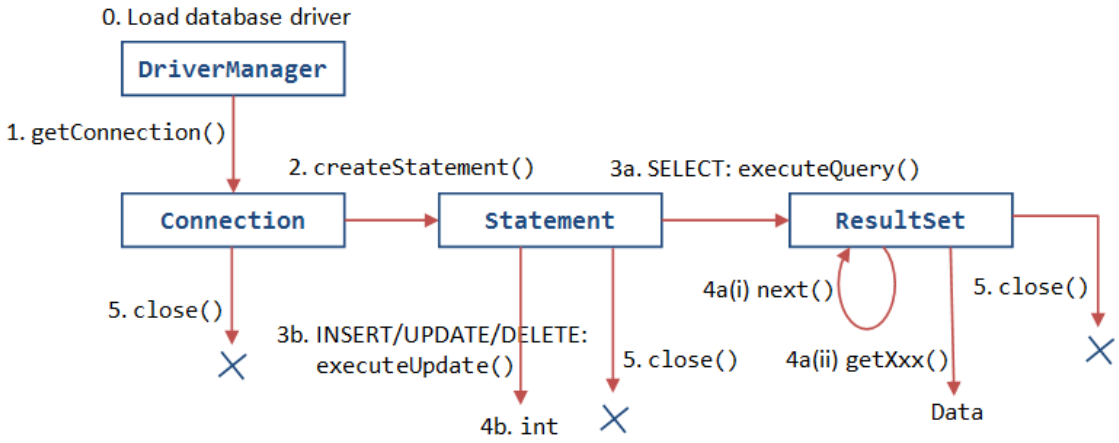
JDBC define varias clases e interfaces que permiten realizar operaciones con bases de datos. En Java, estas clases e interfaces vienen definidas en el paquete **java.sql**:

CLASE/INTERFAZ	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.
`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 y sus nombres.

La **documentación** del paquete java.sql está en:
http://docs.oracle.com/javase/7/docs/api/java/sql/package-summary.html

El funcionamiento de un programa con JDBC requiere los siguientes pasos:

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



El siguiente programa Java accede mediante JDBC a una base de datos MySQL llamada `ejemplo` que tiene creadas las tablas `empleado` y `departamento`.

```java
import java.sql.*;

public class Main {
    public static void main(String[] args) {
        try {
            // cargar el conector JDBC de la base de datos MySQL
            Class.forName("com.mysql.jdbc.Driver");

            // establecer una conexión con la base de datos:
            // parámetro 1 -> conexión a la base de datos:
            //     indica un driver JDBC para MySQL
            //     indica la ruta y el nombre de la base de datos
            // parámetro 2 -> nombre del usuario que accede a la base de datos
            // parámetro 3 -> contraseña del usuario
            Connection conexion = DriveManager.getConnection
                ("jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");

            // preparar la sentencia SQL
            Statement sentencia = conexion.createStatement();
            ResultSet resul = sentencia.executeQuery
                ("SELECT * FROM departamento");

            // recorrer el resultado para visualizar cada fila
            while (resul.next()) {
                System.out.println(resul.getInt(1) + " * " +
                    resul.getString(2) + " * " + resul.getString(3));
            }

            // liberar el objeto ResultSet
            resul.close();
            // liberar el objeto Statement
            sentencia.close();
            // liberar el objeto Connection
            conexion.close();
        }
        catch (ClassNotFoundException cnfe) { cnfe.printStackTrace(); }
        catch (SQLException sqle) { sqle.printStackTrace(); }
    } // fin de main
} // fin de la clase
```

En este programa Java se realizan los siguientes pasos:

1) **Cargar el driver**. Se usa el método *forName()* de la clase *Class*. En este caso, se accede a una base de datos MySQL ("com.mysql.jdbc.Driver"). Esta llamada puede lanzar la excepción *ClassNotFoundException* si no se encuentra la base de datos.

2) **Establecer la conexión**. El servidor MySQL debe estar previamente arrancado y se usa el método *getConnection()* de la clase *DriverManager*. El primer parámetro representa la URL de conexión a la base de datos, el segundo es el nombre del usuario que accede a la base de datos y el tercero es la contraseña del usuario.

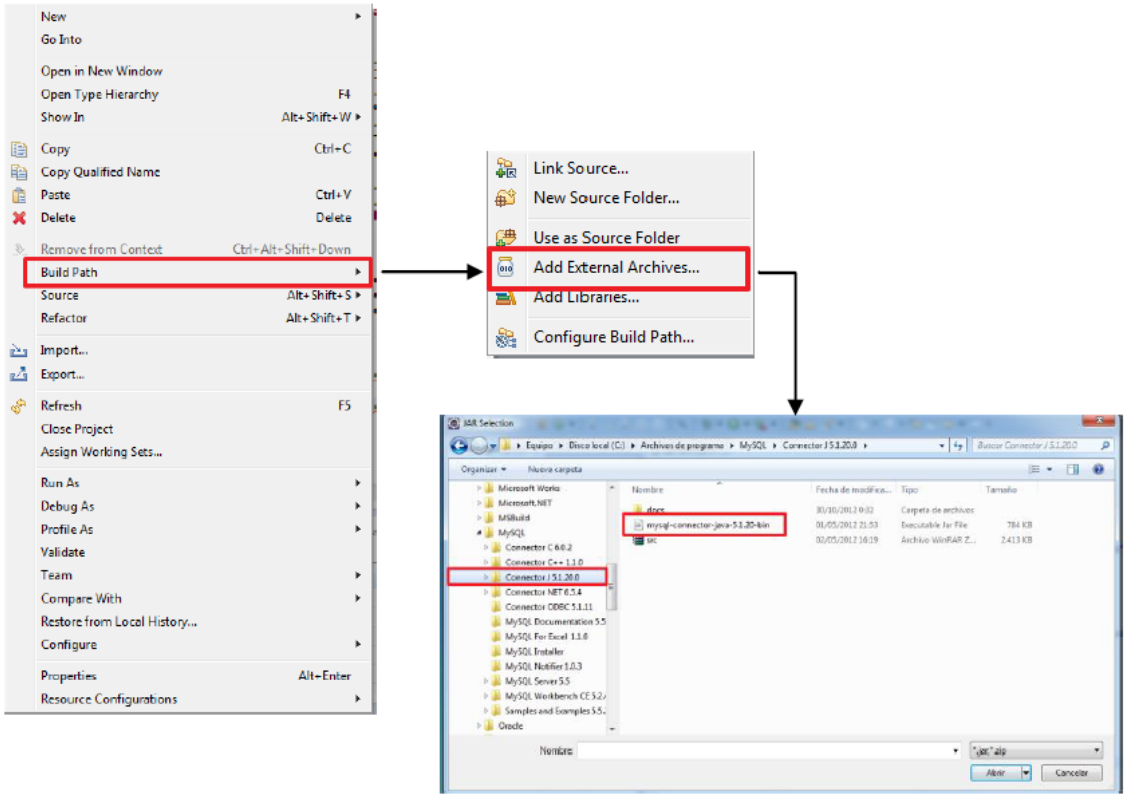
3) **Ejecutar sentencias SQL.** Se usa la interfaz *Statement* para crear una sentencia con el método *createStatement()* de un objeto *Connection* y luego se utiliza el método *executeQuery()* para realizar una consulta a la base de datos. El resultado se devuelve en un objeto ResultSet, que es similar a una lista de registros y tiene internamente un puntero que inicialmente apunta al comienzo de dicha lista. Con el método *next()* se avanza al siguiente registro y los métodos *getInt()* y *getString()* devuelven los valores de los campos de cada registro.

4) **Liberar recursos.** Se liberan todos los recursos utilizados y se cierra la conexión con la base de datos.

Previamente, se debe tener instalado MySQL (servidor, workbench y otros) en Windows. Después se necesita descargar el conector JDBC para MySQL desde el sitio web de MySQL: http://dev.mysql.com/downloads/connector/j/

Dentro de este fichero ZIP se encuentra el driver **mysql-connector-java-5.1.40-bin.jar** para MySQL. Para poder probar el anterior programa Java, hay dos opciones:

- Incluir este fichero JAR en la variable de entorno CLASSPATH. Esto es necesario si se ejecuta el programa Java desde la línea de comandos.
- Añadir este fichero JAR al proyecto del programa Java en Eclipse: botón derecho del ratón sobre el proyecto, Build Paths, Add External Archives y aquí se localiza el fichero JAR.



4. ACCESO A BASES DE DATOS CON JDBC

A continuación, se van a desglosar con más detalles los pasos necesarios para acceder a una base de datos relacional (como MySQL) desde una aplicación Java mediante el conector JDBC.

4.1. ESTABLECIMIENTO DE CONEXIONES

Para realizar la conexión a la base de datos MySQL, se necesita la librería **mysql-connector-java-5.1.40-bin.jar**, que está ubicada dentro del fichero *mysql-connector-java-5.1.40.zip*.

El primer paso es **cargar el controlador de la base de datos**. Esto se realiza con método estático *forName()* de la clase *Class*. Para evitar problemas con la máquina virtual, es aconsejable utilizar *newInstance()*. Por ejemplo:

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

La ejecución de esta instrucción puede provocar las siguientes excepciones:

- **ClassNorFoundException:** si no se ha encontrado la clase en el driver JDBC.
- **InstantiationException:** si no se ha podido crear el driver para la base de datos.
- **IllegalAccessException:** si el acceso a la base de datos no ha sido correcto.

El segundo paso es **realizar la conexión a la base de datos** con el método estático *getConnection()* de la clase *DriverManager*, indicándole la URL de la base de datos, el usuario y la contraseña. Por ejemplo:

```
Connection conexion = DriverManager.getConnection
      ("jdbc:mysql://localhost/sakila", "root", "root");
```

Al intentar la conexión puede surgir una excepción SQLException. Este error puede ocurrir debido a que la URL de la base de datos está mal construida, que el servidor de la base de datos no está en ejecución, que el usuario y/o la contraseña no son correctos, etc.

4.2. EJECUCIÓN DE SENTENCIAS DE DESCRIPCIÓN DE DATOS

El **Lenguaje de Descripción de Datos** (Data Description Language, DDL) es la parte de SQL dedicada a la definición de una base de datos. Consta de sentencias para definir la estructura de la base de datos. Normalmente, estas sentencias son utilizadas por el administrador de la base de datos.

Las principales sentencias de DDL son:

- **CREATE.** Sirve para crear bases de datos y tablas de bases de datos.
- **ALTER.** Sirve para modificar la estructura de una base de datos o de una tabla de la base de datos.
- **DROP.** Permite eliminar una base de datos o una tabla de una base de datos.

Cuando el programador desconoce la estructura de las tablas de la base de datos, puede obtener esta información a través de *metaobjetos*, que son objetos que proporcionan información sobre la base de datos.

4.2.1. La Interfaz DatabaseMetaData

La interfaz **DatabaseMetaData** proporciona una gran cantidad de información sobre la base de datos a través de múltiples métodos. La documentación de esta interfaz se encuentra disponible en: http://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html

Un objeto de *DatabaseMetaData* se crea de la siguiente forma:

```
Class.forName("com.mysql.jdbc.Driver").newInstance;
Connection conexion = DriverManager.getConnection
      ("jdbc:mysql://localhost/sakila", "root", "root");
DatabaseMetaData dbmd = conexion.getMetaData();
```

Los principales métodos de un objeto *DatabaseMetaData* son:

- **getDatabaseProductName().** Devuelve el nombre comercial de la base de datos.
- **getDatabaseProductVersion().** Devuelve la versión de producto del sistema de la base de datos.
- **getDriverName().** Devuelve el nombre del driver JDBC en uso.
- **getURL().** Devuelve la URL del sistema gestor de bases de datos.
- **getUserName().** Devuelve el nombre del usuario actual del gestor de base de datos.
- **getTables(catálogo, esquema, tabla, tipos).** Devuelve un objeto *ResultSet* que proporciona información sobre las tablas y vistas de la base de datos. Tiene cuatro parámetros:
    - *Catálogo de la base de datos.* Se obtiene las tablas del catálogo indicado. Si se pone *null*, se indican todos los catálogos.
    - *Esquema de la base de datos.* Se obtiene las tablas del esquema indicado. Un valor *null* indica el esquema actual o todos los esquemas, en función del SGBD.
    - *Patrón de nombre de tabla.* Es un patrón para indicar los nombres de las tablas a obtener. Un valor *null* indica todas las tablas.
    - *Tipos de tablas.* Indica los tipos de tablas a obtener (TABLE para tablas, VIEW para vistas). Un valor *null* indica todos los tipos.

  Cada fila de *ResultSet* tiene información sobre una tabla: TABLE_CAT (nombre del catálogo al que pertenece la tabla), TABLE_SCHEM (nombre del esquema al que pertenece la tabla), TABLE_NAME (nombre de la tabla o la vista), TABLE_TYPE (tipo de tabla TABLE o VIEW) y REMARKS (comentarios).

```
String[] tipos = {"TABLE"};
ResultSet tablas = dbmd.getTables(null, null, null, tipos);
while (tablas.next()) {
      String catalogo = tablas.getString("TABLE_CAT");
      String esquema = tablas.getString("TABLE_SCHEM");
      String nombre = tablas.getString("TABLE_NAME");
      String tipo = tablas.getString("TABLE_TYPE");
}
```

- **gerColumns(catálogo, esquema, nombre_tabla, nombre_columna).** Devuelve un objeto ResultSet en el que cada fila tiene información sobre las columnas de una tabla o tablas. Para el nombre de la tabla y de la columna se puede utilizar el carácter _ o %. Un valor *null* en los cuatro parámetros indica que obtiene información de todas las columnas y tablas del esquema actual.

  Cada fila de *ResultSet* tiene información sobre una columna: COLUMN_NAME (nombre de la columna), TYPE_NAME (nombre del tipo de datos de la columna), COLUMN_SIZE (tamaño de la columna), DECIMAL_DIGITS (número de dígitos decimales) e IS_NULLABLE (indica si el campo puede contener valores nulos).

```
ResulSet columnas = dbmd.getColumns(null,"ejemplo",
      "departamento", null);
while (columnas.next()) {
      String nombre = columnas.getString("COLUMN_NAME");
      String tipo = columnas.getString("TYPE_NAME");
      String tamano = columnas.getString("COLUMN_SIZE");
      String nula = columnas.getString("IS_NULLABLE");
}
```

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

```
ResultSet claves = dbmd.getPrimaryKeys(null,"ejemplo",
      "departamento");
while (claves.next()) {
      String nombre = claves.getString("COLUMN_NAME");
}
```

- **getExportedKeys(catálogo, esquema, tabla).** Devuelve la lista de todas las claves ajenas que utilizan la clave primaria de la tabla.
- **getImportedKeys(catálogo, esquema, tabla).** Devuelve la lista de claves ajenas existentes en la tabla.
- **getProcedures(catálogo, esquema, procedure).** Devuelve la lista de procedimientos almacenados.

4.2.2. La Interfaz ResultSetMetaData

La interfaz **ResultSetMetaData** proporciona información sobre el conjunto de resultados devueltos (ResultSet) a través de diversos métodos (por ejemplo, los tipos y las propiedades de las columnas). La documentación de esta interfaz se encuentra disponible en: http://docs.oracle.com/javase/7/docs/api/java/sql/ResultSetMetaData.html

Un objeto de *ResultSetMetaData* se crea y se usa de la siguiente forma:

```
Statement sentencia = conexion.createStatement();
ResultSet rs = sentencia.executeQuery
      ("SELECT * FROM departamento");
ResultSetMetaData rsmd = rs.getMetaData();
int nColumnas = rsmd.getColumnCount();
for (int i = 1; i <= nColumnas; i++) {
      String nombre = rsmd.getColumnName(i);
      String tipo = rsmd.getColumnTypeName(i);
      int tamano = rsmd.getColumnDisplaySize(i);
      int nula = rsmd.isNullable(i);
}
```

Los principales métodos de un objeto *ResultSetMetaData* son:

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

4.3. EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS

El **Lenguaje de Manipulación de Datos** (Data Manipulation Language, DML) es la parte de SQL dedicada a la manipulación de una base de datos. Consta de sentencias de inserción, modificación, eliminación y consulta de datos.

Las principales sentencias de DML son:

- **INSERT.** Permite añadir una fila a una tabla.
- **UPDATE.** Permite modificar la información de una o más filas de una tabla.
- **DELETE.** Permite eliminar una o más filas de una tabla.
- **SELECT.** Sirve para recuperar información de una base de datos, mediante la selección de una o más filas y columnas de una o varias tablas.

La interfaz **Statement** proporciona métodos para ejecutar sentencias SQL en una base de datos y obtener los resultados correspondientes. Como *Statement* es una interfaz, no se pueden crear objetos directamente, sino que se realiza con el método *createStatement()* de un objeto *Connection* válido:

```
Connection conexion = DriverManager.getConnection
      ("jdbc:mysql://localhost/sakila", "root", "root");
Statement sentencia = conexion.createStatement();
ResultSet resul = sentencia.executeQuery
      ("SELECT * FROM sakila.actor");
```

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

- **executeQuery(consulta).** Se utiliza para sentencias SQL que recuperan datos en un único objeto *ResultSet*. Normalmente, se usa para las sentencias SELECT. Devuelve siempre un *ResultSet* no nulo, tanto si devuelve una fila como si no.
- **executeUpdate(consulta).** Se utiliza para sentencias SQL que no devuelven un objeto *ResultSet*, como son las sentencias de definición de datos (CREATE, ALTER y DROP) y las sentencias de manipulación de datos (INSERT, UPDATE y DELETE). Si la sentencia es de definición de datos (DDL), este método devuelve cero. Si la sentencia es de manipulación de datos (DML), este método devuelve un entero que indica el número de filas que se han visto afectadas por la sentencia SQL.
- **execute(consulta).** Se utiliza para sentencias SQL que devuelven datos en más de un objeto *ResultSet*. Normalmente, se usa para ejecutar procedimientos almacenados, o cuando no se sabe si la sentencia SQL es una consulta o una actualización.

4.3.1. La Interfaz ResultSet

La interfaz **ResultSet** encapsula los datos devueltos por una consulta SQL, y a través de sus métodos, se puede acceder a la información almacenada en las diferentes columnas de la consulta y se puede obtener información sobre ellas. La documentación de esta interfaz se encuentra disponible en: http://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html

Un objeto *ResultSet* mantiene un **cursor** que apunta a la fila de datos actual. Al comienzo, el cursor se posiciona antes de la primera fila. Por defecto, el cursor de un objeto *ResultSet* solo se puede mover hacia adelante, desde la primera fila hasta la última.

La interfaz *ResultSet* proporciona **métodos de gestión del cursor** para desplazarlo dentro de los resultados devueltos. Los principales métodos de manipulación del cursor son:

- **previous().** Mueve el cursor a la fila anterior de su posición actual.
- **next().** Mueve el cursor a la fila siguiente de su posición actual.

- **absolute(int fila).** Mueve el cursor a la fila indicada según su índice.
- **relative(int filas).** Mueve el cursor un número relativo de filas (positivo o negativo) respecto de la fila actual.
- **beforeFirst().** Coloca el cursor antes de la primera fila.
- **first().** Coloca el cursor en la primera fila.
- **last().** Coloca el cursor en la última fila.
- **afterLast().** Coloca el cursor después de la última fila.
- **isBeforeFirst().** Devuelve *true* si el cursor está antes de la primera fila.
- **isFirst().** Devuelve *true* si el cursor está en la primera fila.
- **isLast().** Devuelve *true* si el cursor está en la última fila.
- **isAfterLast().** Devuelve *true* si el cursor está después de la última fila.
- **getRow().** Devuelve el número de la fila actual, donde el cursor se encuentra.

Además, la interfaz *ResultSet* proporciona **métodos de acceso** para recuperar los valores de las columnas de la fila actual. Estos valores se pueden obtener usando el número de índice de la columna (empezando en 1) o el nombre de la columna.

MÉTODO	MÉTODO	TIPO DE DATO DEVUELTO
getString(int)	**getString(String)**	String
getBoolean(int)	**getBoolean(String)**	boolean
getByte(int)	**getByte(String)**	byte
getShort(int)	**getShort(String)**	short
getInt(int)	**getInt(String)**	int
getLong(int)	**getLong(String)**	long
getFloat(int)	**getFloat(String)**	float
getDouble(int)	**getDouble(String)**	double
getDate(int)	**getDate(String)**	Date
getTime(int)	**getTime(String)**	Time

Otro método interesante es **wasNull()**, que devuelve *true* si el último valor leído de una columna es un valor SQL *null*.

El siguiente ejemplo Java inserta un departamento en la tabla `departamento`, ubicada en una base de datos relacional como MySQL. Estos datos se introducen al programa desde la línea de comandos.

```
// recuperar argumentos desde la línea de comandos
String codigo = args[0];    // codigo del departamento
String nombre = args[1];    // nombre del departamento
String localidad = args[2]; // localidad del departamento

// construir la sentencia INSERT
String sql = "INSERT INTO departamento VALUES (" +
    codigo + ", '" + nombre + "', '" + localidad + "')";
System.out.println(sql);

// ejecutar la sentencia INSERT
Statement sentencia = conexion.createStatement();
int filas = sentencia.executeUpdate(sql);
System.out.println("Filas afectadas: " + filas);
```

El siguiente ejemplo Java modifica varios empleados de la tabla `empleado`, subiéndoles el salario una cantidad determinada a los empleados de un determinado departamento. Estos datos se introducen al programa desde la línea de comandos.

```
// recuperar argumentos desde la línea de comandos
String codigo = args[0];  // codigo del departamento
String subida = args[1];  // subida de salario

// construir la sentencia UPDATE
String sql = "UPDATE empleado SET salario = salario + " +
    subida + " WHERE codigo_dept = " + codigo;
System.out.println(sql);

// ejecutar la sentencia UPDATE
Statement sentencia = conexion.createStatement();
int filas = sentencia.executeUpdate(sql);
System.out.println("Filas modificadas: " + filas);
```

4.3.2. La Interfaz PreparedStatement

En los ejemplos anteriores se han creado las sentencias SQL a partir de cadenas de caracteres concatenadas con los datos necesarios. La interfaz **PreparedStatement** permite construir una cadena de caracteres que representa una sentencia SQL mediante marcadores de posición (o *placeholders*).

Cada **marcador de posición** representa a un dato que será asignado después, se denota con el símbolo de interrogación (?) y tiene un índice que comienza en 1. El marcador de posición solo se puede utilizar para ocupar el sitio de un dato en la cadena SQL, no se puede usar para representar un nombre de una columna o de una tabla.

Antes de ejecutar un *PreparedStatement*, es necesario asignar los datos para que cuando se ejecute, la base de datos asigne variables de unión con estos datos y ejecute la sentencia SQL. Un objeto *PreparedStatement* se puede preparar o precompilar una sola vez, pero se puede ejecutar múltiples veces asignando diferentes valores a los marcadores de posición. Por el contrario, en un objeto *Statement*, la sentencia SQL se suministra en el momento de su ejecución.

La interfaz *PreparedStatement* tiene los mismos métodos (*executeQuery(), executeUpdate()* y *execute()*) que la interfaz *Statement*, pero no se necesita enviar la cadena de caracteres con la sentencia SQL en la llamada, ya que lo hace el método *prepareStatement(String).*

El ejemplo Java anterior, de inserción de un departamento (código, nombre y localidad), queda así:

```
// recuperar argumentos desde la línea de comandos
String codigo = args[0];    // codigo del departamento
String nombre = args[1];    // nombre del departamento
String localidad = args[2]; // localidad del departamento

// construir y ejecutar la sentencia INSERT
String sql = "INSERT INTO departamento VALUES (?, ?, ?)";
PreparedStatement sentencia = conexion.prepareStatement(sql);
sentencia.setInt(1, Integer.parseInt(codigo)); // codigo del departamento
sentencia.setString(2, nombre);                // nombre del departamento
sentencia.setString(3, localidad);             // localidad del departamento
int filas = sentencia.executeUpdate();
System.out.println("Filas insertadas: " + filas);
```

El ejemplo Java anterior, de modificación del salario de los empleados de un departamento concreto, queda así:

```
// recuperar argumentos desde la línea de comandos
String codigo = args[0]; // codigo del departamento
String subida = args[1]; // subida de salario

// construir y ejecutar la sentencia UPDATE
String sql = "UPDATE empleado SET salario = salario + ? WHERE codigo_dept = ?";
PreparedStatement sentencia = conexion.prepareStatement(sql);
sentencia.setFloat(1, Float.parseFloat(subida)); // subida de salario
sentencia.setInt(2, Integer.parseInt(codigo));   // nombre del departamento
int filas = sentencia.executeUpdate();
System.out.println("Filas modificadas: " + filas);
```

La interfaz *PreparedStatement* proporciona **métodos de asignación de valores** para los marcadores de posición definidos en una sentencia SQL:

MÉTODO
void setString(índice, String)
void setBoolean(índice, boolean)
void setByte(índice, byte)
void setShort(índice, short)
void setInt(índice, int)
void setLong(índice, long)
void setFloat(índice, float)
void setDouble(índice, double)
void setDate(índice, Date)
void setTime(índice, Time)
void setNull(índice, tipoSQL)