

1.	INTRODUCCIÓN.....	2
2.	EL DESFASE OBJETO-RELACIONAL.....	2
3.	BASES DE DATOS EMBEBIDAS	3
3.1.	SQLITE.....	3
3.2.	APACHE DERBY.....	5
3.3.	HSQldb.....	7
3.4.	H2.....	10
3.5.	DB4O.....	12
3.6.	OTRAS	15
4.	PROTOCOLOS DE ACCESO A BASES DE DATOS	16
4.1.	ACCESO A DATOS MEDIANTE JDBC	17
4.1.1.	Arquitecturas: modelos de acceso a bases de datos.....	17
4.1.2.	Componentes y Tipos de Conectores	18
4.1.3.	Funcionamiento	21
4.1.4.	Recursos	22
5.	ACCESO A BASES DE DATOS CON JDBC	29
5.1.	ESTABLECIMIENTO DE CONEXIONES.....	29
5.1.1.	Conexión a SQLite	29
5.1.2.	Conexión a Apache Derby	29
5.1.3.	Conexión a HSQldb.....	30
5.1.4.	Conexión a H2.....	30
5.1.5.	Conexión a MySQL	30
5.1.6.	Conexión a Access.....	30
5.1.7.	Conexión a Oracle.....	30
5.2.	EJECUCIÓN DE SENTENCIAS DE DESCRIPCIÓN DE DATOS	31
5.2.1.	La Interfaz DatabaseMetaData	31
5.2.2.	La Interfaz ResultSetMetaData	36
5.3.	EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS.....	37
5.3.1.	La Interfaz ResultSet	39
5.3.2.	Ejecución de Scripts	41
5.3.3.	La Interfaz PreparedStatement	42
6.	GESTIÓN DE ERRORES	45
7.	ACCESO A DATOS MEDIANTE ODBC	47

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 (SGBD - 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**, que no son más que el software que se necesita para realizar las conexiones desde nuestro programa Java con una base de datos relacional, que es el origen de datos en el que nos centraremos. Un conector nos podrá a disposición un 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 que solucionan las necesidades de aplicaciones más sofisticadas que requieren el tratamiento de elementos más complejos, junto con el auge de la programación orientada a objetos.

En este sentido las bases de datos relacionales no están diseñadas para almacenar estos objetos, ya que existe un desfase entre las construcciones típicas que proporciona el modelo de datos relacional y las proporcionadas por los ambientes de POO; es decir; al guardar los datos de un programa bajo el enfoque orientado a objetos se incrementa la complejidad del programa, dando lugar a más código a más esfuerzo de programación debido a la diferencia entre de esquemas entre los elementos a almacenar (objetos) y las características de repositorio de la BBDD (tablas). Este problema se denomina, 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. BASES DE DATOS EMBEBIDAS

Cuando se desarrollan pequeñas aplicaciones en las que no se va a almacenar grandes cantidades de información, no es necesario utilizar un sistema gestor de bases de datos como Oracle o MySQL. En su lugar, se puede usar una **base de datos embebida**, donde el motor esté incrustado en la aplicación y sea exclusivo para ella. La base de datos se inicia cuando se abre la aplicación y termina cuando se cierra la aplicación.

Por lo general, este tipo de bases de datos provienen del movimiento *Open Source*, aunque también existen algunas de origen propietario.

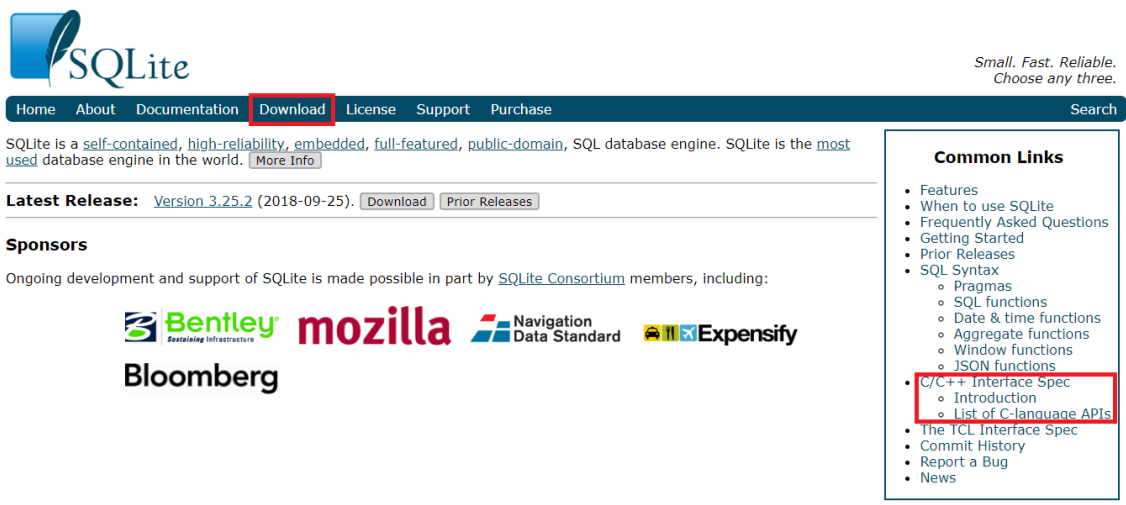
3.1. SQLITE

SQLite es un sistema gestor de bases de datos multiplataforma escrito en C que proporciona un motor muy ligero. La base de datos se guarda en forma de fichero, por lo que es fácil trasladar la base de datos con la aplicación que la usa. Cuenta con una utilidad que permite ejecutar comandos SQL contra una base de datos SQLite en modo consola. Es un proyecto de dominio público, y por tanto, su uso es libre para cualquier propósito comercial o privado.

La biblioteca implementa la mayor parte del estándar SQL-92, incluyendo transacciones de bases de datos atómicas, consistencia de bases de datos, aislamiento y durabilidad, *triggers* (o disparadores), y la mayor parte de las consultas complejas. Los programas pueden usar la funcionalidad de SQLite a través de llamadas simples a subrutinas y funciones. SQLite se puede utilizar desde programas escritos en C/C++, PHP, Visual Basic, Perl, Delphi, Java, etc.

El **sitio web** oficial de SQLite es: <https://sqlite.org/index.html>

La instalación de SQLite es sencilla. Accediendo a su página web oficial, en la sección *Download*, se puede descargar para sistemas operativos de escritorio (Windows, Linux, Mac OS X) y para sistemas operativos móviles (Windows Phone 8, Android):



Small. Fast. Reliable.
Choose any three.






Home About Documentation **Download** License Support Purchase Search

SQLite is a [self-contained](#), [high-reliability](#), [embedded](#), [full-featured](#), [public-domain](#), SQL database engine. SQLite is the [most used](#) database engine in the world. [More Info](#)

Latest Release: [Version 3.25.2](#) (2018-09-25). [Download](#) [Prior Releases](#)

Sponsors

Ongoing development and support of SQLite is made possible in part by [SQLite Consortium](#) members, including:

Common Links

- Features
- When to use SQLite
- Frequently Asked Questions
- Getting Started
- Prior Releases
- SQL Syntax
 - Pragmas
 - SQL functions
 - Date & time functions
 - Aggregate functions
 - Window functions
 - JSON functions
- [C/C++ Interface Spec](#)
 - Introduction
 - [List of C-language APIs](#)
- The TCL Interface Spec
- Commit History
- Report a Bug
- News

Para sistemas Windows, se puede descargar el fichero ZIP **sqlite-tools-win32-x86-3250200.zip**, que contiene una serie de herramientas ejecutables desde la línea de comandos para la gestión de ficheros de bases de datos SQLite.

sqlite-android-3250200.aar (2.94 MiB)	A precompiled Android library containing the core SQLite together with appropriate Java bindings, ready to drop into any Android Studio project. (sha1: df168493edf3f65244d7e3aac716d53908a61def)
Precompiled Binaries for Linux	
sqlite-tools-linux-x86-3250200.zip (1.88 MiB)	A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff program, and the sqlite3_analyzer program. (sha1: 92ea7ea5f4726657b1bd1b204a9441ae60cef0e4)
Precompiled Binaries for Mac OS X (x86)	
sqlite-tools-osx-x86-3250200.zip (1.23 MiB)	A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff program, and the sqlite3_analyzer program. (sha1: 2f4b997c9cf0b1860ebf1b70213f3322f80b8451)
Precompiled Binaries for Windows	
sqlite-dll-win32-x86-3250200.zip (465.00 KiB)	32-bit DLL (x86) for SQLite version 3.25.2. (sha1: 461aeb0e6d738fe66229dd93d94850346894b35)
sqlite-dll-win64-x64-3250200.zip (776.42 KiB)	64-bit DLL (x64) for SQLite version 3.25.2. (sha1: 9fad624c1aa1c0c9ba20230f2e176fac305edac5)
sqlite-tools-win32-x86-3250200.zip (1.68 MiB)	A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff.exe program, and the sqlite3_analyzer.exe program. (sha1: 39ef9709972af563768a121c2372f0cd868364cf)
Universal Windows Platform	
sqlite-uwp-3250200.vsix (6.78 MiB)	VSIX package for Universal Windows Platform development using Visual Studio 2015. (sha1: 51150a65ff65b57ae2b465a63ff4f81f3ca1fa36)

Este fichero ZIP incluye la utilidad **sqlite3.exe**, que permite al usuario introducir y ejecutar manualmente sentencias SQL contra una base de datos SQLite.

Para ejecutar la utilidad **sqlite3.exe**, simplemente se escribe “sqlite3” en la línea de comandos. Este comando “sqlite3” puede ir seguido opcionalmente del nombre del fichero que contiene la base de datos SQLite. Si el fichero no existe, se creará automáticamente un nuevo fichero de base de datos con el nombre dado. Si no se especifica un fichero de base de datos en la línea de comandos, se creará una base de datos temporal y ésta se eliminará cuando el usuario salga del programa “sqlite3”.

```

SQLite version 3.25.2 2018-09-25 19:08:10
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open ejemplo.db
sqlite> CREATE TABLE departamentos(
...> dept_no TINYINT(2) NOT NULL PRIMARY KEY,
...> dnombre VARCHAR(15),
...> loc VARCHAR(15)
...> );
sqlite> INSERT INTO departamentos VALUES (10,'CONTABILIDAD','SEVILLA');
sqlite> INSERT INTO departamentos VALUES (20,'INVESTIGACIÓN','MADRID');
sqlite> INSERT INTO departamentos VALUES (30,'VENTAS','BARCELONA');
sqlite> INSERT INTO departamentos VALUES (40,'PRODUCCIÓN','BILBAO');
sqlite> .tables
departamentos
sqlite> SELECT * FROM departamentos;
10|CONTABILIDAD|SEVILLA
20|INVESTIGACIÓN|MADRID
30|VENTAS|BARCELONA
40|PRODUCCIÓN|BILBAO
sqlite>

```

La **documentación** de la utilidad de línea de comandos **sqlite3.exe** para SQLite se encuentra disponible en: <https://sqlite.org/cli.html>

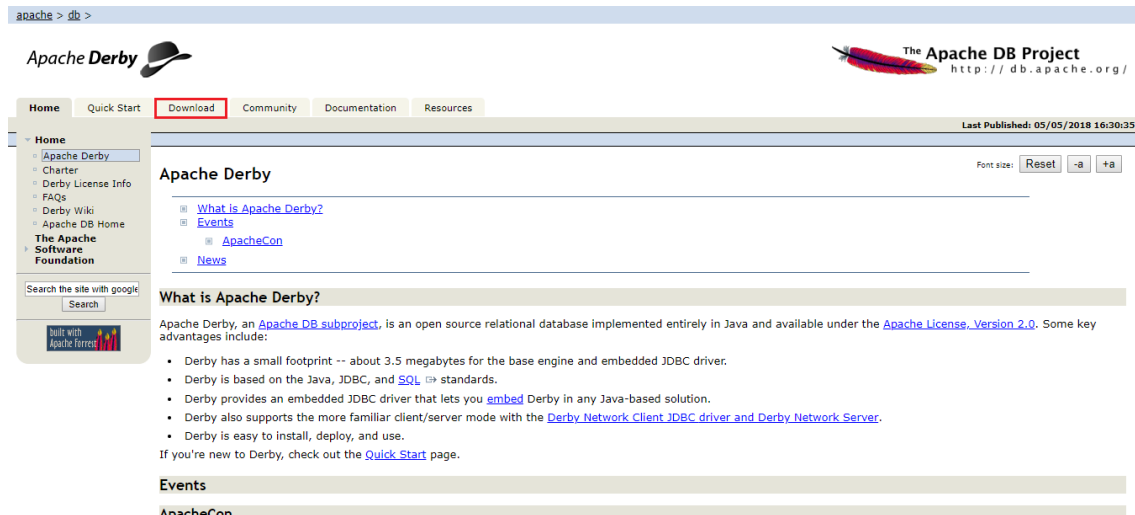
3.2. APACHE DERBY

Apache Derby es una base de datos relacional de código abierto, implementada en su totalidad en Java que forma parte del **Apache DB Project**, y está disponible bajo la licencia Apache (versión 2.0). Algunas ventajas de esta base de datos son:

- Tiene un tamaño reducido (unos 2,6 MB para el motor de la base de datos y el controlador JDBC integrado).
- Está basada en Java y soporta los estándares de SQL.
- Ofrece un controlador JDBC integrado que permite incrustar Derby en cualquier solución basada en Java.
- Soporta el tradicional paradigma cliente-servidor, utilizando un framework denominado *Derby Network Server*, que incluye a Derby y maneja accesos a la base de datos desde aplicaciones. Estas aplicaciones se pueden ejecutar en diferentes JVM en la misma máquina o en máquinas remotas.
- Es fácil de instalar, implementar y utilizar.

El **sitio web** oficial de Apache Derby es: <https://db.apache.org/derby/>

Accediendo a su página web oficial, en la sección *Download*, se puede descargar la última versión oficial de Apache Derby para Java.



Hay cuatro distribuciones diferentes de Apache Derby:

- **Distribución bin.** Contiene la documentación, javadoc, y los ficheros JAR para Derby.
- **Distribución lib.** Contiene solo los ficheros JAR para Derby.
- **Distribución lib-debug.** Contiene los ficheros JAR para Derby con números de líneas de código fuente.
- **Distribución src.** Contiene la estructura en árbol del código fuente de Derby en el momento de creación de los ficheros binarios.

Para utilizar Derby en un sistema Windows con Java, se puede descargar el fichero **db-derby-10.13.1.1-bin.zip**, correspondiente a la distribución bin.

[db-derby-10.14.2.0-bin.zip](#) [PGP] [MD5]
[db-derby-10.14.2.0-bin.tar.gz](#) [PGP] [MD5]

[db-derby-10.14.2.0-lib.zip](#) [PGP] [MD5]
[db-derby-10.14.2.0-lib.tar.gz](#) [PGP] [MD5]

[db-derby-10.14.2.0-lib-debug.zip](#) [PGP] [MD5]
[db-derby-10.14.2.0-lib-debug.tar.gz](#) [PGP] [MD5]

[db-derby-10.14.2.0-src.zip](#) [PGP] [MD5]
[db-derby-10.14.2.0-src.tar.gz](#) [PGP] [MD5] (Note that, due to long filenames, you will need gnu tar to unravel this tarball.)

Este fichero ZIP incluye las librerías derby.jar y derbytools.jar, las cuales deberán estar accesibles en la variable de entorno CLASSPATH, para poder utilizar Derby en un programa Java.

CONFIGURACIÓN DE LA VARIABLE CLASSPATH EN WINDOWS:

```
C:\> set DERBY_INSTALL=D:\db-derby-10.13.1.1-bin
C:\> set CLASSPATH=DERBY_INSTALL\lib\derby.jar;DERBY_INSTALL\lib\derbytools.jar;.
```

VISUALIZACIÓN DE LA VARIABLE CLASSPATH EN WINDOWS:

```
C:\> echo %CLASSPATH%
```

Además, Apache Derby trae una serie de ficheros BAT, como la utilidad **ij.bat**, que permite lanzar órdenes por consola para crear bases de datos y ejecutar sentencias de definición y de manipulación de datos. Para crear una base de datos de nombre *ejemplo* se escribe desde la línea de comandos de ij la siguiente orden:

```
ij> connect 'jdbc:derby:ejemplo.db;create=true';
```

Donde:

- *connect* es el comando para establecer la conexión.
- *jdbc:derby* es el protocolo JDBC especificado por Derby.
- *ejemplo.db* es el nombre de la base de datos creada (se crea una carpeta con dicho nombre y dentro de ella hay una serie de ficheros).
- *create=true* es el atributo utilizado para crear la base de datos.

```
C:\ApacheDerby\bin>ij
Versión de ij 10.13
ij> connect 'jdbc:derby:ejemplo.db;create=true';
ij> CREATE TABLE departamento (
> codigo INT NOT NULL PRIMARY KEY,
> nombre VARCHAR(15),
> localidad VARCHAR(15)
> );
0 filas insertadas/actualizadas/suprimidas
ij> INSERT INTO departamento VALUES (10, 'Contabilidad', 'Sevilla');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamento VALUES (20, 'Investigacion', 'Madrid');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamento VALUES (30, 'Ventas', 'Barcelona');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamento VALUES (40, 'Produccion', 'Bilbao');
1 fila insertada/actualizada/suprimida
ij> SELECT * FROM departamento;
CÓDIGO      |NOMBRE      |LOCALIDAD
-----
10          |Contabilidad|Sevilla
20          |Investigacion|Madrid
30          |Ventas      |Barcelona
40          |Produccion  |Bilbao
4 filas seleccionadas
ij> exit;
```

El comando show tables; muestra las tablas existentes en la base de datos. Para obtener ayuda podemos escribir el comando help;

Para volver a usar la base de datos creada, se escribe desde la línea de comandos de `ij` la siguiente orden:

```
ij> connect 'jdbc:derby:ejemplo.db';.
```

La **documentación** de la utilidad de línea de comandos `ij.bat` para Apache Derby se encuentra disponible en: https://db.apache.org/derby/papers/DerbyTut/ij_intro.html

3.3. HSQLDB

HSQLDB (Hyperthreaded Structured Query Language DataBase) es un sistema gestor de bases de datos relacional escrito en Java. La suite ofimática *OpenOffice* lo incluye desde su versión 2.0 para dar soporte a la aplicación *Base*. Sus principales características son:

- Ofrece un motor de bases de datos completamente multihilo y transaccional con la posibilidad de mantener la base de datos en memoria o en ficheros en disco.
- Incluye una herramienta SQL de línea de comandos potente y otras herramientas de consulta con una interfaz de usuario sencilla.
- Es compatible con los estándares SQL ANSI-92 y SQL:2011.
- Permite integridad referencial (claves ajenas), procedimientos almacenados en Java, disparadores (*triggers*) y tablas en disco de hasta 8 GB (hasta 270 millones de folios de datos en una sola base de datos y una capacidad de copia de seguridad en caliente).
- Se utiliza como motor de persistencia y de bases de datos en más de 1700 proyectos de software libre y en muchos productos comerciales.
- Se distribuye con licencia BSD (Berkeley Software Distribution), que es una licencia muy cercana al dominio público.

El **sitio web** oficial de HSQLDB es: <http://hsqldb.org/>



HSQLDB - 100% Java Database

- [Download](#) • [Support](#) • [License](#)
- [Features](#) • [FAQ](#) • [Documentation](#) • [How To](#)
- [Developers](#) • [Software using HSQLDB](#)
- [SourceForge Project Page](#) • [OpenOffice.org Integration](#)

latest official release

20 May 2018

[Download latest version 2.4.1](#)

Latest version 2.4.1 works with JDK 6, 9 and 10. Version 2.3.6 for JDK 6 is also available.

The [How To](#) pages are regularly updated and include a list of useful links.

commercial support:

HyperXtremeSQL

Commercial support for business users of HSQLDB is available from the [HyperXtremeSQL](#) web site. A higher-performance database engine based on HSQLDB with several additional features such as extended OLAP, the PL/HSQL procedural language and COMBINECT memory tables is also available from that site.

2.4.1 Released

Version 2.4.1 for Java 8 supports java.time classes in JDBC and adds many enhancements. These include table spaces for disk-based tables, more compatibility functions and improved SQL routine support.

Version 2.3.4 added the UUID type for columns, SYNONYM for tables and functions, PERIOD predicates, and auto-updated TIMESTAMP columns on row updates. Other new features included the ability to cancel long-running statements from JDBC as well as from admin sessions, and UTF-16 file support for text table sources, in addition to 8-bit text files. MySQL compatibility for REPLACE, INSERT IGNORE and ON DUPLICATE KEY UPDATE statements.

Each release incorporates extensive code reviews, enhancements and bug fixes.

HSQLDB (HyperSQL DataBase) is the leading SQL relational database software written in Java. It offers a small, fast multithreaded and transactional database engine with in-memory and disk-based tables and supports embedded and server modes. It includes a powerful command line [SQL tool](#) and simple GUI query tools.

HSQLDB supports the widest range of SQL Standard features seen in any open source database engine: SQL:2011 core language features and an extensive list of SQL:2011 optional features. It supports full Advanced [ANSI-92 SQL](#) with only two exceptions. Many extensions to the Standard, including syntax compatibility modes and features of other popular database engines, are also supported.

Version 2.4.1 is **fully multithreaded** and supports high performance 2PL and MVCC (multiversion concurrency control) transaction control mode. See the [list of features](#) in

[Follow @hyperSQL](#)

follow HyperSQL on Twitter

HSQLDB in 2018

HSQLDB is a mature product. Versions released in recent years have enhanced reliability and performance.

The new 2.4.x releases have improvements in all areas and support Java 8's JDBC updates and new date/time classes.

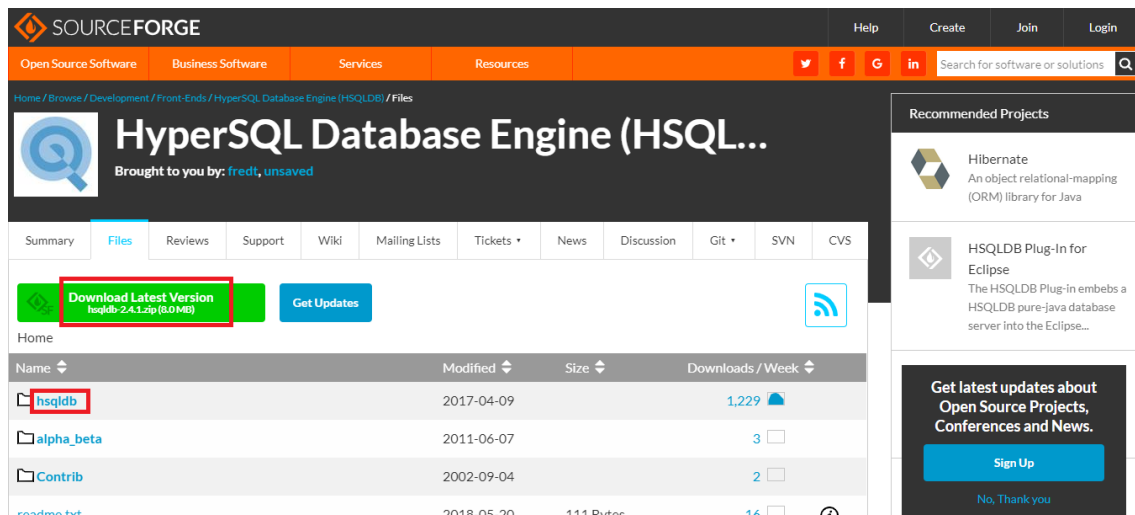
HSQLDB SupportWare

HSQLDB is Java developers' best choice for development, testing and deployment of database applications.

HSQLDB SupportWare allows organizations and individual developers to support the development and maintenance of HSQLDB.

Participation in the program is by annual subscription or sponsorship.

Accediendo a su página web oficial, la sección *Download* redirige hacia el repositorio *SourceForge* (<https://sourceforge.net/projects/hsqldb/files/>), que contiene los ficheros de las distintas versiones del proyecto. Desde allí se puede descargar el fichero **hsqldb-2.4.1.zip**, la última versión estable 2.4.1 de HSQLDB.

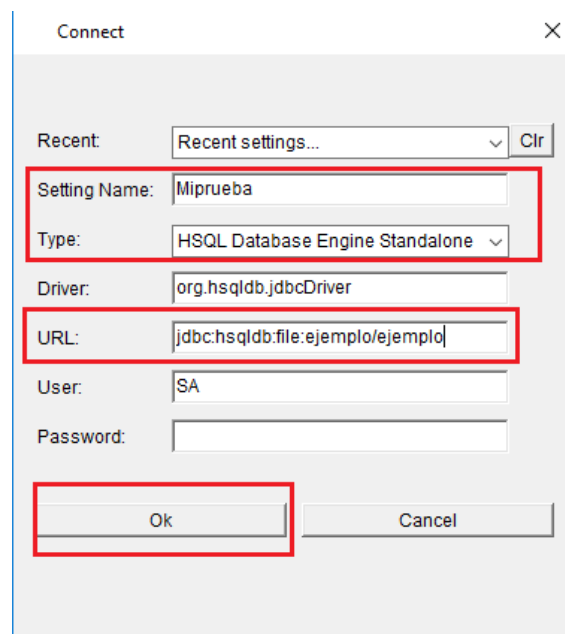


Al descomprimir se crea una carpeta con el nombre del fichero ZIP, dentro de esa carpeta hay una con el nombre *hsqldb*, la llevamos a la unidad para que nos quede instalado. Creamos una carpeta ejemplo dentro de data (para no mezclar las bases de datos que creamos).

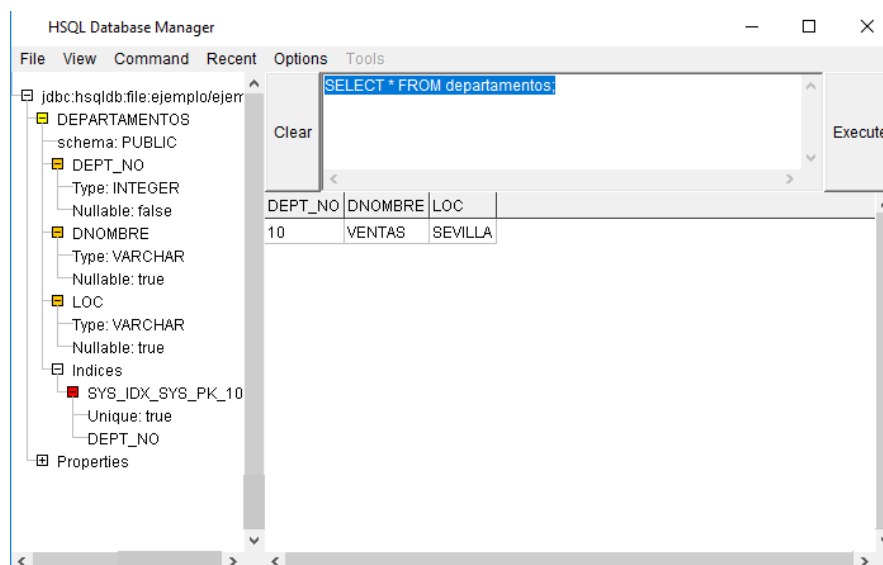
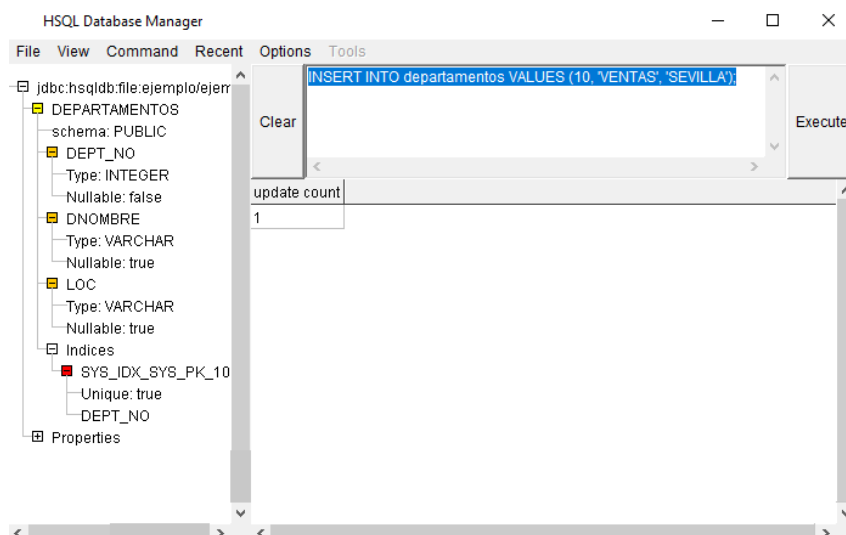
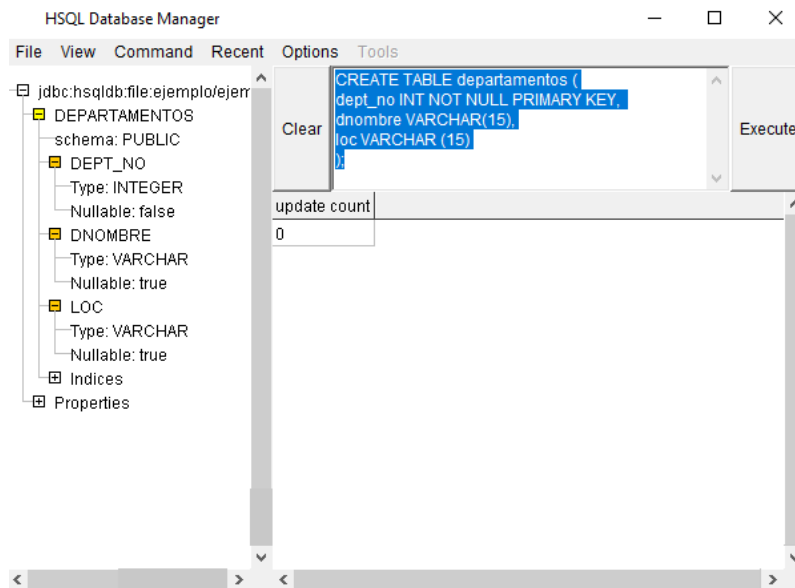
Para realizar la conexión a la base de datos y ejecutar la interfaz gráfica de HSQLDB, se abre la línea de comandos de DOS y se ejecuta el fichero **runUtil.bat** de esta forma:

```
D:\HSQLDB\bin> runUtil DatabaseManager
```

A continuación se abre una ventana para configurar la conexión. Aquí hay que introducir el tipo de base de datos (en memoria, en fichero, en servidor) y la URL donde se almacenarán los ficheros de la base de datos.



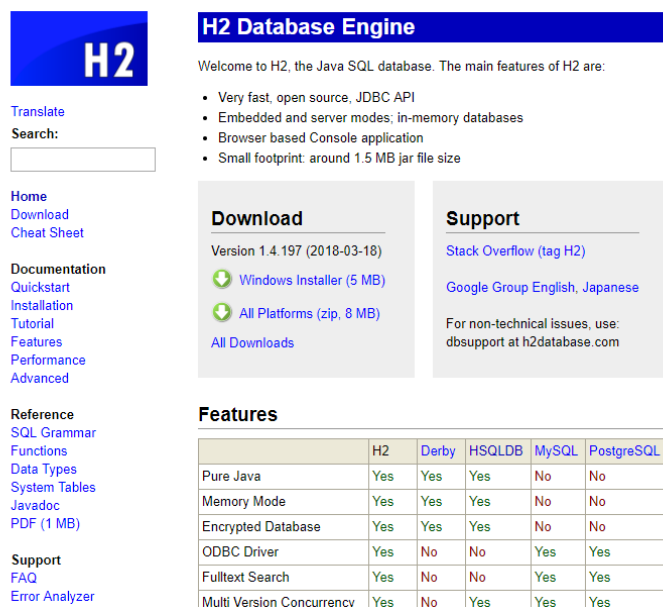
A continuación, se abre la ventana **HSQL Database Manager**, desde la cual se pueden ejecutar comandos de definición de datos (CREATE, ALTER, DROP) y comandos de manipulación de datos (INSERT, UPDATE, DELETE, SELECT). Para ejecutar una sentencia SQL pulsamos el botón *Execute*. Desde la opción de menú **View->Refresh Tree** podemos actualizar el árbol de objetos.



3.4. H2

H2 es un sistema gestor de bases de datos relacional programado íntegramente en Java. Se puede incorporar en aplicaciones Java o ejecutarse de modo cliente-servidor. Permite su integración total en aplicaciones Java y el acceso a la base de datos lanzando SQL directamente, sin tener que pasar por una conexión a través de sockets. Está disponible como software de código libre bajo la Licencia Pública de Mozilla o la Licencia Pública Eclipse.

El **sitio web** oficial de H2 es: <http://www.h2database.com/html/main.html>



H2 Database Engine

Welcome to H2, the Java SQL database. The main features of H2 are:

- Very fast, open source, JDBC API
- Embedded and server modes; in-memory databases
- Browser based Console application
- Small footprint: around 1.5 MB jar file size

Download

Version 1.4.197 (2018-03-18)

- Windows Installer (5 MB)
- All Platforms (zip, 8 MB)
- All Downloads

Support

Stack Overflow (tag H2)

Google Group English, Japanese

For non-technical issues, use: dbsupport@h2database.com

Features

	H2	Derby	HSQLDB	MySQL	PostgreSQL
Pure Java	Yes	Yes	Yes	No	No
Memory Mode	Yes	Yes	Yes	No	No
Encrypted Database	Yes	Yes	Yes	No	No
ODBC Driver	Yes	No	No	Yes	Yes
Fulltext Search	Yes	No	No	Yes	Yes
Multi Version Concurrency	Yes	No	Yes	Yes	Yes

Para utilizar H2, se puede descargar de la sección *Download* el fichero multi-plataforma **h2-2018-03-18.zip**, que contiene

Version 1.4.197 (2018-03-18)

[Windows Installer](#) (SHA1 checksum: 4aa6d13109cc6f6b211feecbae60bdc08082424c)

[Platform-Independent Zip](#) (SHA1 checksum: 3048c3c33fbb6b1b699e49432bde9ad635760ffe)

Version 1.4.196 (2017-06-10), Last Stable

[Windows Installer](#)

[Platform-Independent Zip](#)

Para realizar la conexión a la base de datos y ejecutar la interfaz gráfica de H2, se abre la línea de comandos de DOS y se ejecuta el fichero **h2.bat** de esta forma:

```
D:\H2\bin> h2
```

A continuación se abre el navegador web con la consola de administración de H2. Aquí hay que introducir un nombre para la configuración de la base de datos y la URL para la conexión a la base de datos (donde se almacenarán los ficheros de la base de datos). Probamos la conexión y finalmente conectamos.

English ▼ [Preferencias](#) [Tools](#) [Ayuda](#)

Registrar

Configuraciones guardadas: Generic H2 (Embedded) ▼

Nombre de la configuración: Generic H2 (Embedded) Guardar Eliminar

Controlador: org.h2.Driver

URL JDBC: jdbc:h2:C:\EJERCICIOS\UNIZ\pruebaH2

Nombre de usuario: mp

Contraseña:

Conectar Probar la conexión

Prueba correcta

El enlace **Preferences** permite configurar varios aspectos como: los clientes permitidos (locales y remotos), conexión segura (uso de SSL), puerto del servidor web y notificaciones de sesiones activas. El enlace **Tools** presenta una serie de herramientas que se pueden utilizar sobre la base de datos: copia de seguridad (backup), restauración, ejecución de scripts, conversión de la base de datos en un script y encriptación.

Una vez realizada la conexión con éxito, el navegador web visualiza la pestaña **H2 Console**, desde la cual se puede realizar operaciones sobre la base de datos. Ésta muestra al inicio los comandos más importantes y un ejemplo de script SQL. En la zona de instrucciones SQL se pueden escribir sentencias para crear tablas, insertar filas y realizar consultas.

☒ Auto commit Número máximo de filas: 1000 Auto completado Desactivado Auto select On

jdbc:h2:D:/basedatos/ejemplo

- INFORMATION_SCHEMA
 - CATALOGS
 - COLLATIONS
 - COLUMNS
 - COLUMN_PRIVILEGES
 - CONSTANTS
 - CONSTRAINTS
 - CROSS_REFERENCES
 - DOMAINS
 - FUNCTION_ALIASES
 - FUNCTION_COLUMNS
 - HELP
 - INDEXES
 - IN_DOUBT
 - LOCKS
 - QUERY_STATISTICS
 - RIGHTS
 - ROLES
 - SCHEMATA
 - SEQUENCES
 - SESSIONS
 - SESSION_STATE
 - SETTINGS
 - TABLES
 - TABLE_PRIVILEGES
 - TABLE_TYPES
 - TRIGGERS
 - TYPE_INFO
 - USERS
 - VIEWS
- Usuarios
 - SA
 - Administrador

H2 1.4.193 (2016-10-31)

Ejecutar Run Selected Auto completado Eliminar Instrucción SQL:

Comandos importantes

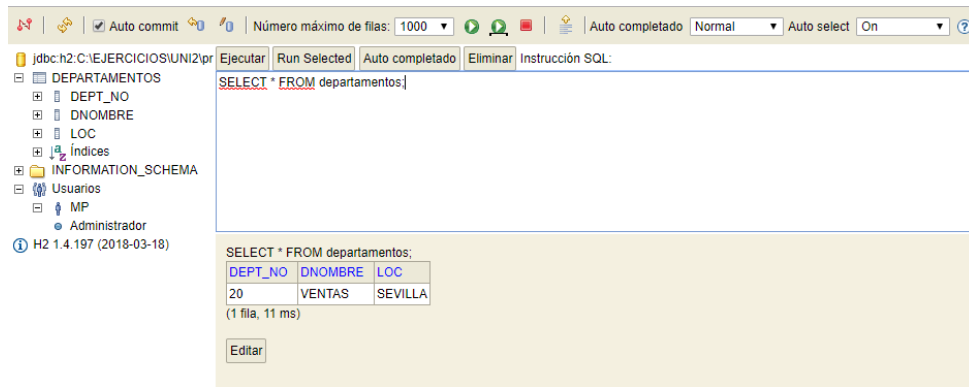
	Visualizar esta página de ayuda.
	Ver histórico de comandos
	Ctrl+Enter Ejecuta la actual sentencia SQL
	Shift+Enter Executes the SQL statement defined by the text selection
	Ctrl+Space Auto completado
	Desconectar de la base de datos.

Ejemplo SQL Script

Borrar la tabla si existe	DROP TABLE IF EXISTS TEST;
Crear una tabla nueva con las columnas ID y NAME	CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));
Añadir una fila nueva	INSERT INTO TEST VALUES(1, 'Hello');
Añadir otra fila	INSERT INTO TEST VALUES(2, 'World');
Consulta la tabla	SELECT * FROM TEST ORDER BY ID;
Modificar datos en una fila	UPDATE TEST SET NAME='Hi' WHERE ID=1;
Borrar una fila	DELETE FROM TEST WHERE ID=2;
Ayuda	HELP ...

Añadiendo drivers de base de datos

Se pueden registrar otros drivers añadiendo el archivo Jar del driver a la variable de entorno H2DRIVERS o CLASSPATH. Por ejemplo (Windows): Para añadir la librería del driver de base de datos C:/Programs/hsqldb/lib/hsqldb.jar, hay que establecer la variable de entorno H2DRIVERS a C:/Programs/hsqldb/lib/hsqldb.jar.



3.5. DB4O

DB4O (DataBase for Objects) es un motor de bases de datos orientado a objetos. Se puede utilizar de forma embebida o en aplicaciones cliente-servidor. Tiene un alto rendimiento (sobre todo en modo embebido) y propone un abandono completo del paradigma relacional de las bases de datos tradicionales, en favor de un modelo de desarrollo orientado a objetos.

Actualmente está disponible para entornos Java y .NET. Dispone de una licencia dual GPL/comercial. Es decir, si se desea desarrollar software libre con DB4O, su uso no conlleva ningún coste por licencia, pero si se quiere aplicar DB4O a un software privativo, se usa otro modelo de licenciamiento concreto.

Las principales características de DB4O son:

- Se evita el problema del desfase objeto-relacional.
- No existe un lenguaje SQL para la manipulación de datos. En su lugar, existen métodos delegados y procesos automáticos mediante código compilable.
- Se elimina la representación del modelo de datos en dos tipos de esquemas: modelo de objetos y modelo relacional. Ahora el esquema de datos del dominio viene definido a partir del diagrama de clases.
- Se instala añadiendo un único fichero de librería (JAR par Java o DLL para .NET).
- Se crea un único fichero de base de datos con la extensión YAP.

El **sitio web** oficial de DB4O es: <http://www.db4o.com/> [enlace caído], podemos descargar la última versión desde la URL <http://supportservices.actian.com/versant/default.html>

En eclipse para usar el JAR seleccionamos nuestro proyecto y añadimos como archivo externo localizando el fichero JAR a incluir en el proyecto.

Veamos un ejemplo que crea una base de datos llamada DBPersonas.yap y almacena objetos Persona:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

public class Main {
    final static String BDPer = "D:/eclipse/bd_db4o/DBPersonas.yap";

    public static void main(String[] args) {
        ObjectContainer db= Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),BDPer);

        // Creamos Personas
        Persona p1 = new Persona("Juan", "Guadalajara");
        Persona p2 = new Persona("Ana", "Madrid");
        Persona p3 = new Persona("Luis", "Granada");
        Persona p4 = new Persona("Pedro", "Asturias");

        //Almacenar objetos Persona en la base de datos
        db.store(p1);
        db.store(p2);
        db.store(p3);
        db.store(p4);

        db.close(); //Cerrar base de datos

    } //fin de main
} //fin de la clase Main
```

Hemos definido la clase Persona formada por los atributo de nombre y ciudad con los métodos get y set para obtener y almacenar los valores de un objeto Persona.

```
public class Persona {
    private String nombre;
    private String ciudad;

    public Persona(String nombre,String ciudad)    {
        this.nombre=nombre;
        this.ciudad=ciudad;
    }
    public Persona() {
        this.nombre=null;
        this.ciudad=null;
    }
    public String getNombre(){return nombre;}
    public void setNombre(String nom){nombre=nom; }

    public String getCiudad(){return ciudad; }
    public void setCiudad(String dir){ciudad=dir;}
} //fin Persona
```

El paquete Java com.db4o contiene casi toda la funcionalidad de la base de datos. Para este ejemplo hemos necesitado db4o.Db4oEmbedded y com.db4o.ObjectContainer.

Para realizar cualquier acceso ya sea insertar, modificar o realizar consultas debemos manipular una instancia de ObjectContainer donde se define el fichero de base de datos, nuestro ejemplo DBPersonas.yap y el nombre se almacena en una variable BDPer donde será necesario incluir el trayecto donde se encuentra el fichero.

Métodos importantes a tener en cuenta son:

- Para abrir la base de datos **openFile()**:

```
ObjectContainer db= Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),BDPer);
```

- Para cerrarla llamamos a **close()**:

```
db.close();
```

- Para almacenar un objeto **store()**:

```
db.store(p1);
```

- Para recuperar objetos podemos utilizar el sistema de consultas QBE (Query-By-Ejemplo) mediante el método **queryByExample()**, vamos a mostrar las personas que hay en la base de datos obteniendo los resultados en forma de ObjectSet. Si no existe ningún objeto el método **size()** sobre el objeto ObjectSet devolverá 0.

```
// Mostrar todas las personas
Persona per=new Persona(null,null); //para recuperar todas las personas nos
//valem de un objeto persona vacío

ObjectSet result=db.queryByExample(per); //usamos
listResult(result);

db.close(); //cerrar base de datos

} //fin de main

public static void listResult(ObjectSet result) {
    System.out.println(result.size());
    while(result.hasNext()) {
        System.out.println(result.next());
    }
}
```

- Para modificar un objeto, primero hay que localizarlo y después se modifica con el método `store()`. El siguiente ejemplo modifica la ciudad de Juan a Toledo y luego visualiza sus datos, si hay varios objetos *Persona* con nombre Juan solo se modifica el primero que encuentre, para modificar todos tendríamos que hacer un bucle:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

public class Modificar {
    static String BDPer = "DBPersonas.yap";

    public static void main(String[] args) {
        ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), BDPer);

        ObjectSet<Persona> result =
            db.queryByExample(new Persona("Juan", null));
        if (result.size() == 0)
            System.out.println("No existe Juan...");
        else {
            Persona existe = (Persona) result.next();
            existe.setCiudad("Toledo");
            db.store(existe); //ciudad modificada
            //consultar los datos
            result = db.queryByExample(new Persona("Juan", null));
            existe = (Persona) result.next();
            System.out.printf("Nombre: %s, Nueva Ciudad: %s %n",
                existe.getNombre(), existe.getCiudad());
        }

        db.close();
    }
}
```

- Para eliminar objetos utilizaremos el método `delete()`, ante evidentemente será necesario localizar el objeto que queremos eliminar. A continuación eliminamos todos los objetos cuyo nombre sea Juan:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

public class Eliminar {

    final static String BDPer = "DBPersonas.yap";

    public static void main(String[] args) {

        ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), BDPer);

        ObjectSet<Persona> result = db.queryByExample(new Persona("Juan", null));

        if (result.size() == 0)
            System.out.println("No existe Juan...");
        else {
            System.out.printf("Registros a borrar: %d %n", result.size());

            while (result.hasNext()) {
                Persona p = result.next();
                db.delete(p);
                System.out.println("Borrado...");
            }

            db.close();
        }
    }
}
```

**En el tema de Bases de datos objeto relacionales y orientado a objetos profundizaremos más en estos conceptos.

3.6. OTRAS

Existen más sistemas de bases de datos embebidos tanto en software libre como o el sistemas propietarios. Algunos ejemplos son:

- **Firebird que se deriva del código fuente de Interbase 6.0 de Borland** es un sistema gestor de bases de datos relacional de código abierto que no tiene licencia duales, por lo que es totalmente libre y se puede usar tanto en aplicaciones comerciales como de código abierto. Se representa en tres versiones del servidor: *SuperServer*, *Classic* y *Embedded*. La edición embebida (*Embedded*) es un completo servidor Firebird empackado en unos cuantos ficheros. Es fácil distribuir aplicaciones, puesto que no requiere de instalación, para crear catálogos en CDROM, versiones mono usuario, de evaluación o portátiles de las aplicaciones. Algunas de sus características son:
 - Completo soporte para procedimientos almacenados y disparadores.
 - Integridad referencial.
 - Bajo consumo de recursos.
 - Lenguaje interno para procedimientos almacenados y disparadores (PLSQL).
 - Soporte para funciones externas.
 - Poca o ninguna necesidad de administradores especializados.
 - Múltiples formas de acceder a la base de datos: nativo/API, drivers dbExpress, ODBC, OLEDB, proveedor .NET, driver JDBC nativo tipo 4, módulo Python, PHP, Perl, etc.
 - Etc.
- **Microsoft SQL Server Compact (SQL Server CE)**: es una base de datos compacta y con una gran variedad de funciones diseñada para entornos móviles. Un producto de Microsoft que incluye varias características de las bases de datos relacionales a la vez que ocupa poco espacio. Algunas características son:
 - Posee un motor de base de datos así como un procesador y uno optimizador de consultas especialmente diseñado para entornos móviles.
 - Soporta un subconjunto de tipos de datos y de sentencias *Transact-SQL* y *SQL Server*.
 - En cuanto a los datos de tipo texto, únicamente soporta tipos de datos de cadena compatibles con Unicode (nchar, nvarchar, ntext).
 - Se integra desde la versión 3.0, con Microsoft Visual Studio (incluyendo la edición *Express* desde la versión 3.5) y SQL Server Management Studio.
 - A nivel de seguridad que ofrece la posibilidad de cifrado del fichero de base de datos con una contraseña de acceso restringida.
 - *SQL Server Compact 4.0* era utilizado y ajustado para usarse con aplicaciones web ASP.NET.

4. 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:

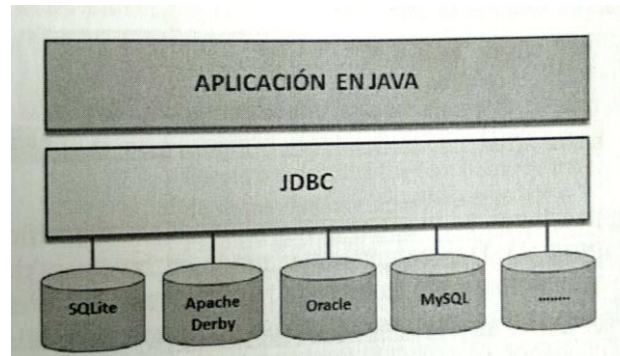
- 1) **ODBC (Open DataBase Connectivity)** de Microsoft. Define una API (escrita en C) que las aplicaciones pueden usar para abrir una conexión con una base de datos, enviar consultas, actualizaciones y obtener resultados. Podremos usar esta API para conectarnos a cualquier servidor de base de datos que sea compatible con ODBC.
- 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.

4.1. 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.



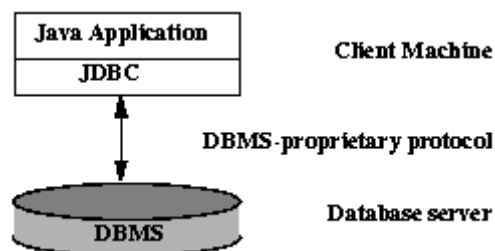
JDBC consta de un conjunto de clases e interfaces que permite escribir aplicaciones Java para gestionar las siguientes tareas con una base de datos relacional:

- 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.

4.1.1. Arquitecturas: modelos de acceso a bases de datos

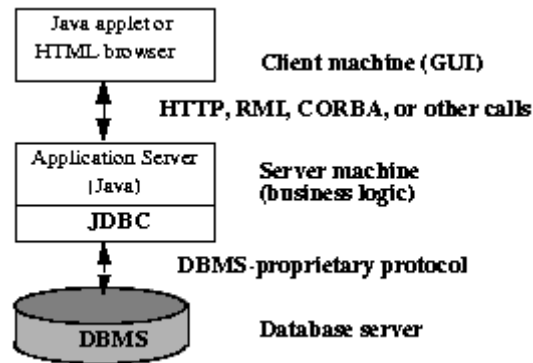
A la hora de establecer el canal de comunicación entre una aplicación Java y una base de datos se pueden identificar dos modelos distintos de acceso. Estos modelos dependen del número de capas que se contemple. La API JDBC es compatible tanto de dos como de tres capas para el acceso a la base de datos.

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.

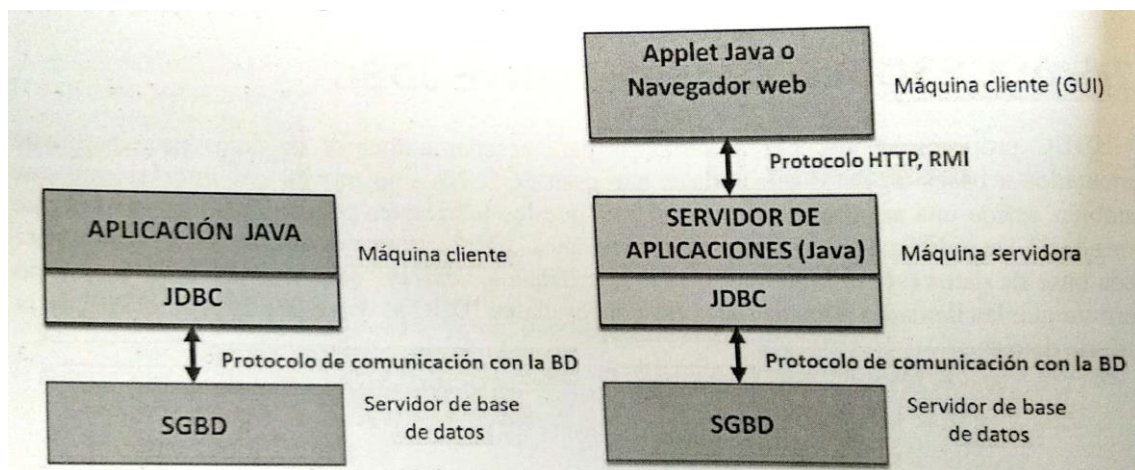


Ilustración 1 Modelo dos capas y Modelo de tres capas

Un **servidor de aplicaciones** es una implementación de la especificación J2EE que es un entorno centrado en Java para desarrollar, construir y desplegar aplicaciones empresariales multicapa basadas en la Web. Existen diversas implementaciones cada una con sus propias características.

4.1.2. Componentes y Tipos de Conectores

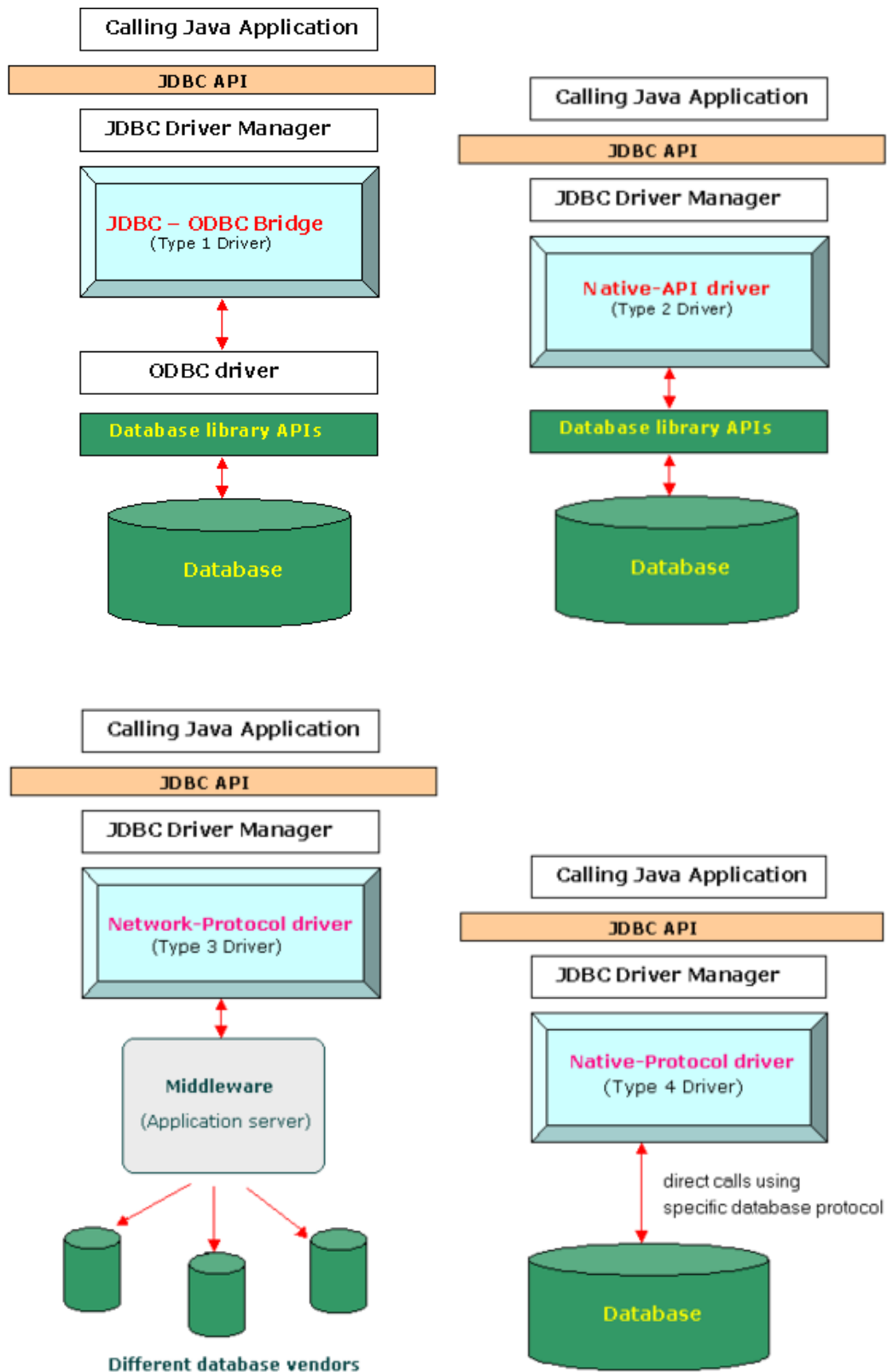
El conector JDBC incluye cuatro componentes:

- 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-ODBC.** 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), que es independiente de la base de datos, para comunicarse con un servidor remoto de base de datos (Middleware). El driver JDBC no comunica directamente con la base de datos si no con el software intermedio que a su vez se comunica con la base de datos. 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: Thin-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. Esta será la opción más adecuada en la mayoría de los casos.



4.1.3. 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**:

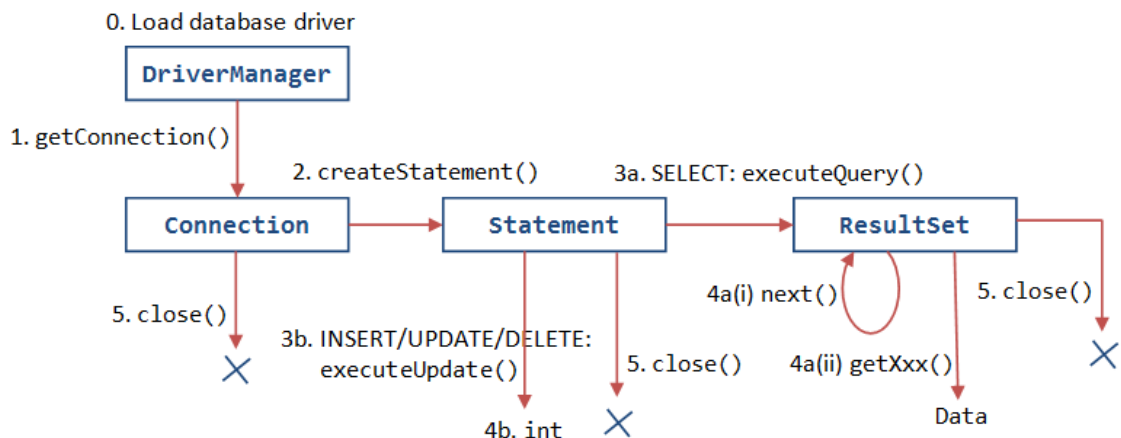
CLASE/INTERFAZ	DESCRIPCIÓN
Driver	Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto.
DriverManager	Permite gestionar todos los drivers instalados en el sistema.
DriverPropertyInfo	Proporciona diversa información acerca de un driver.
Connection	Representa 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 <i>ResultSet</i> , como el número de columnas y sus nombres.

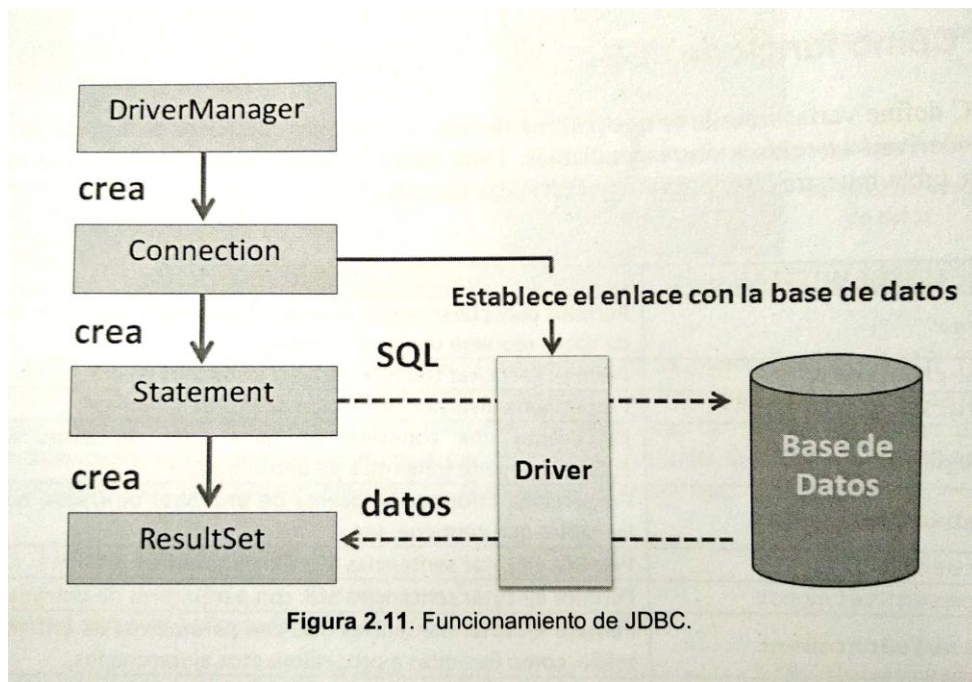
La **documentación** del paquete java.sql está en:

<http://docs.oracle.com/javase/7/docs/api/java/sql/package-summary.html>

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

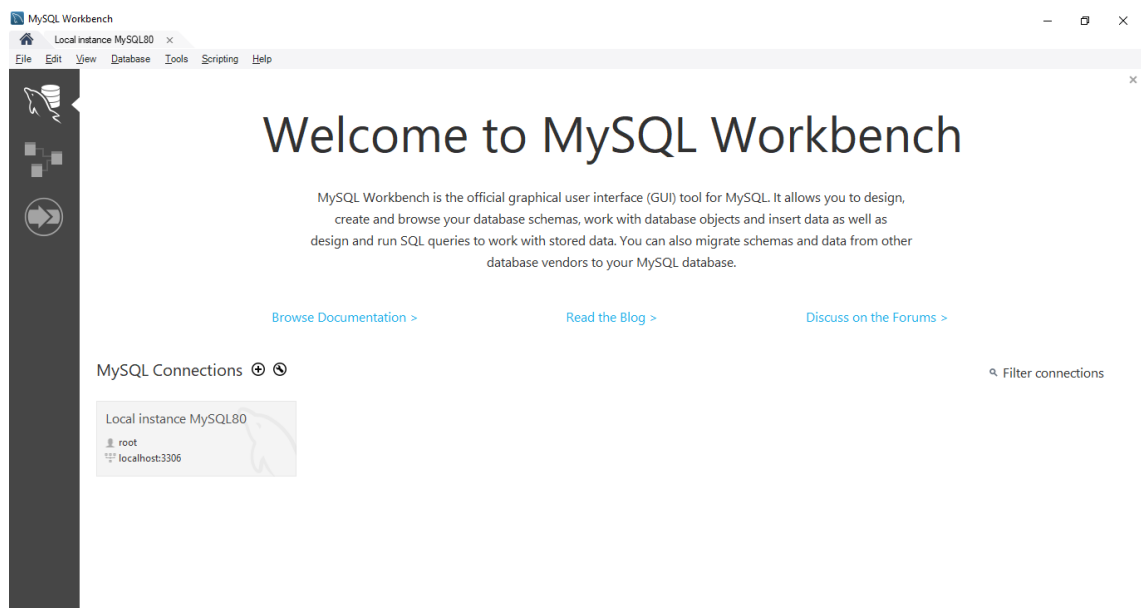
- 1) Importar las clases necesarias.
- 2) Cargar el driver JDBC.
- 3) Identificar el origen de datos.
- 4) Crear un objeto **Connection**.
- 5) Crear un objeto **Statement**.
- 6) Ejecutar una consulta con el objeto **Statement**.
- 7) Recuperar los datos del objeto **ResultSet**.
- 8) Liberar el objeto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





4.1.4. Recursos

- El primer paso para empezar a trabajar con acceso a datos usando mysql es [descargar](#) e instalar en nuestro equipo MySQL (servidor, workbench y otros) en Windows.



Manual MySQL Workbench: <https://dev.mysql.com/doc/workbench/en/wb-intro.html>

- El segundo paso para trabajar desde java con mysql es descargar el driver o el conector JDBC con la base de datos. Ese Driver es la clase que, de alguna forma, sabe cómo hablar con la base de datos. Desgraciadamente (y hasta cierto punto es lógico), java no viene con todos los Drivers de todas las posibles bases de datos del mercado. Debemos descargar el conector JDBC para MySQL desde el sitio web de MySQL: <http://dev.mysql.com/downloads/connector/j/> (para cada uno normalmente en la página de nuestra base de datos):



Generally Available (GA) Releases

Connector/J 8.0.12

Select Operating System:
Platform Independent

Looking for previous GA versions?

Platform Independent (Architecture Independent), Compressed TAR Archive (mysql-connector-java-8.0.12.tar.gz)	8.0.12	4.8M	Download
Platform Independent (Architecture Independent), ZIP Archive (mysql-connector-java-8.0.12.zip)	8.0.12	5.5M	Download

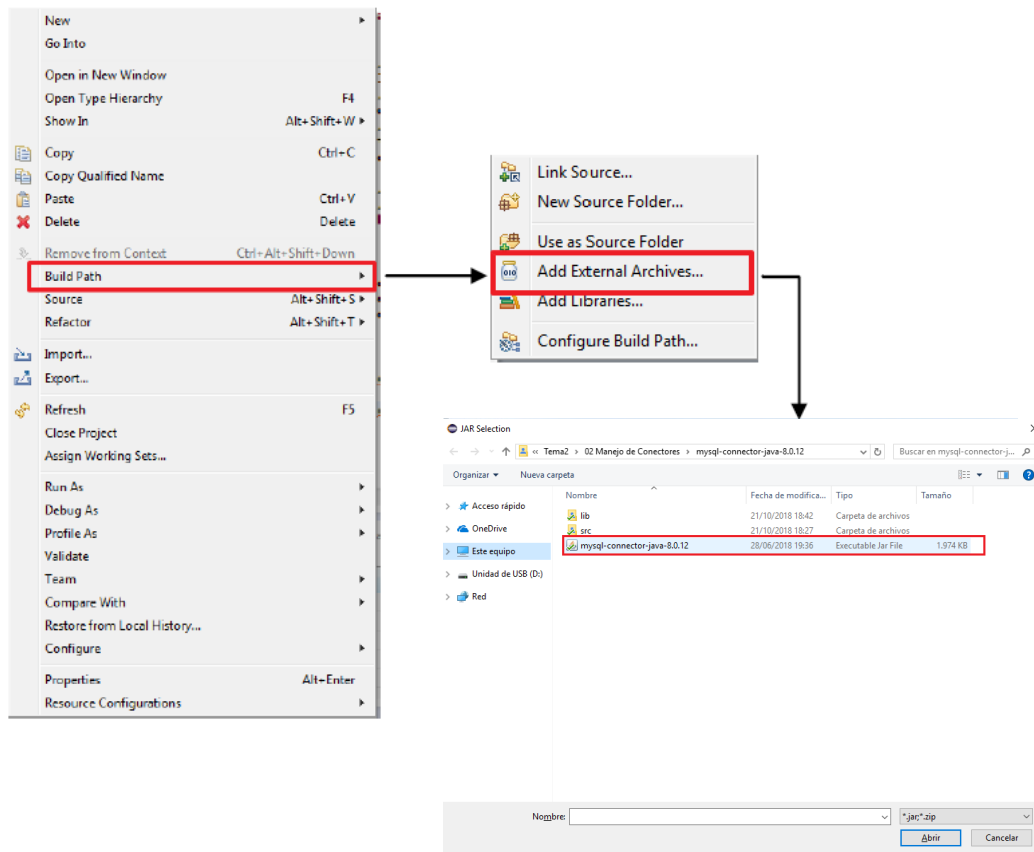
MD5: 368c686aec816ca7d2ad9c98ac0e739f | Signature

MD5: d7f2ef57f603d245dc7b86db2941ccd6 | Signature

We suggest that you use the MD5 checksums and GnuPG signatures to verify the integrity of the packages you download.

Dentro de este fichero ZIP se encuentra el driver **mysql-connector-java-8.0.12-bin.jar** para MySQL. Tendremos que poner ese jar accesible en nuestro proyecto para no obtener el error [java.lang.ClassNotFoundException](#): **com.mysql.jdbc.Driver**. 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.



- El tercer paso es crear la base de datos con la que vamos a trabajar. Para ello los principales comandos para la consola de mysql son:

Arrancar

mysql

Conectarse al programa:

mysql -h localhost -u root -p pas

Mostrar bases de datos

mysql> show databases;

Crear bases de datos

mysql> create database datos;

Borrar una base de datos

mysql> drop database datos;

Seleccionar una base de datos

mysql> use datos;

Crear una tabla

mysql> create table agenda(nombre text, edad int);

Borrar una tabla

mysql> drop table agenda;

Mostrar la estructura de una tabla

mysql> describe agenda;

Insertar un registro en una tabla

mysql> insert into agenda values("Ana",15);

Mostrar el contenido de una tabla

mysql> select * from agenda;

Modificar registros de una tabla

mysql> update agenda set edad=20 where nombre="Ana";

Borrar registros de una tabla

mysql> delete from agenda where nombre="Ana";

Crear un usuario de MySQL

mysql> create user 'adriana'@'localhost' IDENTIFIED BY '123456';

Borrar un usuario de MySQL

```
mysql> drop user 'adriana'@'localhost';
```

Dar TODOS los privilegios a un usuario de MySQL

```
mysql> use tiendita;
```

```
Database changed;
```

```
mysql> grant all privileges on clientes to adriana;
```

Los privilegios asignados quedan registrados en la tabla "tables_priv" de la base de datos "mysql".

Para quitar TODOS los privilegios a un usuario de MySQL

Para quitarle todos los privilegios sobre la tabla "clientes" de la base de datos "tiendita" al usuario "adriana" tecleamos:

```
mysql> use tiendita;
```

```
Database changed;
```

```
mysql> revoke all on clientes from adriana;
```

Para asignar ALGUNOS privilegios a un usuario de MySQL

```
mysql> use tiendita;
```

```
mysql> grant insert on clientes to adriana;
```

all privileges - todos los privilegios

alter - permite alterar la estructura de una tabla ya creada

create - permite crear nuevas bases y/o tablas

delete - permite borrar datos de una tabla

drop - permite borrar bases y/o tablas

index - permite usar, crear, alterar y borrar índices de una tabla

insert - permite insertar datos en una tabla

select - permite leer datos de una tabla

shutdown - permite detener el demonio de MySQL

update - permite actualizar datos de una tabla

El siguiente programa Java accede mediante JDBC a una base de datos MySQL y un usuario con el nombre *ejemplo*, la clave del usuario es la misma. Dicho usuario va a tener todos los privilegios sobre la base de datos creada. Crearemos las tablas EMPLEADO y DEPARTAMENTO.

```
--
-- TABLA DEPARTAMENTOS MySQL
--
CREATE TABLE departamentos (
  dept_no TINYINT(2) NOT NULL PRIMARY KEY,
  dnombre VARCHAR(15),
  loc VARCHAR(15)
) ENGINE=INNODB;

-----
-- TABLA EMPLEADOS - MySQL
-----
CREATE TABLE empleados (
  emp_no SMALLINT(4) NOT NULL PRIMARY KEY,
  apellido VARCHAR(10),
  oficio VARCHAR(10),
  dir SMALLINT,
  fecha_alt DATE,
  salario FLOAT(6,2),
  comision FLOAT(6,2),
  dept_no TINYINT(2) NOT NULL,
  FOREIGN KEY(dept_no) REFERENCES departamentos(dept_no)
) ENGINE=INNODB;
```

Para ello tenemos que tener previamente instalado MySQL (servidor, workbench y otros) en Windows.

```

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 = DriverManager.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` pasando como String el nombre de la clase del driver. 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.

```

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

```

```

Connection conexion = DriverManager.getConnection(url,user,pass);

```

```

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

```

El primer parámetro del método `getConnection()` representa la URL de la conexión a la base de datos. Tiene el siguiente formato:

```
jdbc:mysql://nombrehost:puerto/nombre_basededatos
```

- **jdbc:mysql:** indica que usamos el driver JDBC para MySQL
- **nombre_host:** indica el nombre del servidor donde está la base de datos. Puede ponerse IP o nombre de la máquina que esté en la red. Si ponemos *localhost* como nombre del servidor, estamos indicando que la base de datos está en nuestra máquina donde ejecutamos el programa Java.
- **puerto:** es el puerto predeterminado para la base de datos MySQL, por defecto es 3306. Si no se pone lo interpreta por defecto.
- **nombre_basedatos:** es el nombre de la base de datos que vamos a conectar y que previamente ha sido creada en MySQL. En nuestro caso ejemplo.

- 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 le pasamos un String donde se especifica la consulta a realizar a la base de datos.

```
// Preparamos la sentencia
Statement sentencia = conexion.createStatement();
String sql = "SELECT * FROM departamentos";
ResultSet resul = sentencia.executeQuery(sql);
```

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. Cada elemento de la lista es uno de los registros de la tabla departamentos. Con el método `next()` se avanza al siguiente registro y los métodos `getInt()` y `getString()` devuelven los valores de los campos de cada registro. En nuestro caso usamos un bucle *while* que se ejecutará mientras `next()` devuelva true (mientras haya registros).

```
while (resul.next()) {
    System.out.printf("%d, %s, %s %n", resul.getInt(1),
        resul.getString(2), resul.getString(3));
}
```

La clase *ResultSet* proporciona varios métodos `getXxx` que toman como parámetro el número de la columna o el nombre de la columna y devuelven los datos.

Para modificar la BD se utiliza `executeUpdate` pasándole una cadena con la operación: UPDATE, DELETE o INSERT.

```
int nfilas = sentencia.executeUpdate("DELETE FROM tabla1 WHERE ...");
```

ResultSet dispone de varios métodos para mover el puntero del objeto ResultSet:

Método	Función
<code>boolean next()</code>	Mueve el puntero del objeto ResultSet una fila hacia adelante a partir de la posición actual. Devuelve <i>true</i> si el puntero se posiciona correctamente y <i>false</i> si no hay registros en el ResultSet
<code>boolean first()</code>	Mueve el puntero del objeto ResultSet al primer registro de la lista. Devuelve <i>true</i> si el puntero se posiciona correctamente y <i>false</i> si no hay registros
<code>boolean last()</code>	Mueve el puntero del objeto ResultSet al último registro de la lista. Devuelve <i>true</i> si el puntero se posiciona correctamente y <i>false</i> si no hay registros
<code>boolean previous()</code>	Mueve el puntero del objeto ResultSet al registro anterior de la lista. Devuelve <i>true</i> si el puntero se posiciona correctamente y <i>false</i> si se coloca antes del primer registro
<code>void beforeFirst()</code>	Mueve el puntero del objeto ResultSet justo antes del primer registro
<code>int getRow()</code>	Devuelve el número de registro actual. Para el primer registro del objeto ResultSet devuelve 1, para el segundo 2 y así sucesivamente

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

5. 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 y embebida (como SQLite, Apache Derby, HSQLDB, H2 o MySQL) desde una aplicación Java mediante el conector JDBC.

5.1. ESTABLECIMIENTO DE CONEXIONES

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

La ejecución de esta instrucción puede provocar las siguientes excepciones:

- **ClassNotFoundException:** 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.

5.1.1. Conexión a SQLite

Para realizar la conexión a la base de datos SQLite, se necesita la librería **sqlite-jdbc-3.23.1.jar.jar**, que se puede descargar desde la siguiente URL: <https://bitbucket.org/xerial/sqlite-jdbc/downloads>

Se debe ubicar este fichero JAR en la misma carpeta donde se encuentra el fichero fuente de la aplicación Java y se debe indicar su localización en las propiedades del proyecto en Eclipse. Es decir, en el entorno gráfico que usaremos para ejecutar el programa incluimos el fichero JAR o en el CLASSPATH si lo ejecutamos desde línea de comando.

De acuerdo a lo visto anteriormente para MySQL en este caso cambiaremos: la carga del driver JDBC y la conexión a la base de datos SQLite quedando así:

```
Class.forName("org.sqlite.JDBC");  
Connection conexion = DriverManager.getConnection  
("jdbc:sqlite:D:/SQLite/ejemplo.db");
```

5.1.2. Conexión a Apache Derby

Para realizar la conexión a la base de datos Apache Derby, se necesita la librería **derby.jar**, que está ubicada dentro de la carpeta *lib* (generada al descomprimir el fichero **db-derby-10.13.1.1-bin.zip**).

La carga del driver JDBC y la conexión a la base de datos Apache Derby quedan así:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");  
Connection conexion = DriverManager.getConnection  
("jdbc:derby:D:/ApacheDerby/ejemplo.db");
```


5.1.3. Conexión a HSQLDB

Para realizar la conexión a la base de datos HSQLDB, se necesita la librería **hsqldb.jar**, que está ubicada dentro de la carpeta *lib* (generada al descomprimir el fichero *hsqldb-2.4.1*).

La carga del driver JDBC y la conexión a la base de datos HSQLDB quedan así:

```
Class.forName("org.hsqldb.jdbcDriver");
Connection conexion = DriverManager.getConnection
    ("jdbc:hsqldb:file:D:/HSQLDB/ejemplo");
```

5.1.4. Conexión a H2

Para realizar la conexión a la base de datos H2, se necesita la librería **h2-1.4.197.jar**, que está ubicada dentro de la carpeta *bin* (generada al descomprimir el fichero *h2-2018-03-18.zip*).

La carga del driver JDBC y la conexión a la base de datos H2 quedan así:

```
Class.forName("org.h2.Driver");
Connection conexion = DriverManager.getConnection
    ("jdbc:h2:D:/H2/ejemplo", "sa", "");
```

Para esta conexión debemos incluir en nombre de usuario y la contraseña de la conexión. Por ejemplo "sa" y la clave no estableció cuando la base de datos fue creada.

5.1.5. Conexión a MySQL

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*.

La carga del driver JDBC y la conexión a la base de datos MySQL quedan así:

```
Class.forName("com.mysql.jdbc.Driver");
Connection conexion = DriverManager.getConnection
    ("jdbc:mysql://localhost/sakila", "ejemplo", "ejemplo");
```

5.1.6. Conexión a Access

Para realizar la conexión a la base de datos Access, se necesita las siguientes librerías **mysql-connector-java-5.1.40-bin.jar**, que está ubicada dentro del fichero *mysql-connector-java-5.1.40.zip*. Pueden descargarse de la URL: <http://ucanaccess.sourceforge.net/site.html>. Al acceder al sitio podemos descargar un fichero similar a UCanAccess-4.0.4-bin.zip con los JAR o UCanAccess-4.0.4-src.zip con ejemplos.

La carga del driver JDBC y la conexión a la base de datos Access quedan así:

```
Class.forName("net.Ucanaccess.jdbc.UcanaccessDriver");
Connection conexion = DriverManager.getConnection
    ("jdbc:ucanaccess://mibasededatosaccess);
```

5.1.7. Conexión a Oracle

Para realizar la conexión a la base de datos Oracle, se necesita el driver JDBC Thin. Se puede descargar de la web oficial de Oracle (comprobando previamente la versión de la base de datos instalada) en <https://www.oracle.com/technetwork/apps-tech/jdbc-112010-090769.html>. Para el ejemplo se ha descargado el driver ojdbc6.jar. Necesitamos saber el nombre del servicio que usa la base de datos para incluirlo en la URL de la conexión. Normalmente para la versión Express Edition el nombre es XE.

Para la conexión especificando la URL de la base de datos, nombre de usuario y contraseña necesitamos:

La URL de conexión de manera genérica es `jdbc:oracle:<drivertype>:@<database>`, en nuestro caso: `jdbc:oracle:thin:@localhost:1521:XE`.

El driver se llama **oracle.jdbc.driver.OracleDriver**. El siguiente ejemplo conecta al usuario ejemplo con la contraseña ejemplo a una base de datos a través del puerto 1521 localhost, utilizando el controlador Thin. La conexión a la base de datos Oracle quedaría así:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection conexion = DriverManager.getConnection
    ("jdbc:oracle:thin:@localhost:1521:XE", "ejemplo", "ejemplo");
```

5.2. 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.

En generar al desarrollar una aplicación JDBC, el programador conoce la estructura de las tablas y datos que está manejando, es decir, conoce, unas que tienen y como están relacionadas entre sí. Pero también es posible que se desconozca parte de la estructura de las tablas de la base de datos, por ello puede obtener esta información a través de *metaobjetos*, que son objetos que proporcionan información sobre la base de datos.

5.2.1. La Interfaz DatabaseMetaData

La interfaz **DatabaseMetaData** proporciona información sobre la base de datos a través de múltiples métodos, muchos de estos métodos devuelven un ResultSet.

La documentación de esta interfaz se encuentra disponible en:

<https://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html>

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

```
Class.forName("com.mysql.jdbc.Driver");
Connection conexion = DriverManager.getConnection
    ("jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");
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.

getTables(String catalogo, String esquema, String patronDeTabla, String[] tipos) throws SQLException

- 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, como en Oracle).
 - Patrón de nombre de tabla. Es un patrón para indicar los nombres de las tablas a obtener. Un valor *null* indica todas las tablas. Podemos usar el carácter %, por ejemplo, "de%" obtiene todas las tablas cuyo nombre empieza por "de".
 - Tipos de tablas. Es un array de String en el que se indica los tipos de tablas a obtener (TABLE para tablas, VIEW para vistas). Un valor *null* indica todos los tipos, obtiene tanto tablas como vistas. Tipos validos son: "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM".

Cada fila de *ResultSet* tiene información sobre una tabla: TABLE_CAT (columna1, nombre del catálogo al que pertenece la tabla), TABLE_SCHEM (columna 2, nombre del esquema al que pertenece la tabla), TABLE_NAME (columna3, nombre de la tabla o la vista), TABLE_TYPE (columna 4, tipo de tabla TABLE o VIEW), REMARKS (columna 5 ,comentarios), TYPE_CAT, TYPE_SCHEMA, TYPE_NAME, SELF_REFERENCING_COL_NAME, REF_GENERATION.

```
String[] tipos = {"TABLE"};
ResultSet tablas = dbmd.getTables(null, "ejemplo", "%", 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");
}
```

Más información:

<https://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html#getTables-java.lang.String-java.lang.String-java.lang.String-java.lang.String:A->

A continuación tenemos un ejemplo en el que se conecta con una base de datos MySQL con el nombre *ejemplo* y que va mostrar información sobre el producto de base de datos, el drive, la URL para el acceso, el nombre de usuario y las tablas y vistas del esquema actual (o todos los esquemas dependiendo del SGBD). El método es **getMetaData()** de la interfaz *Connection* que devuelve un objeto *DataBaseMetaData* con la información sobre la base de datos representada por el objeto *Connection*:

```

import java.sql.*;
public class EjemploDatabaseMetadata {
    public static void main(String[] args) {
        try
        {
            Class.forName("com.mysql.jdbc.Driver"); //Cargar el driver
            //Establecemos la conexion con la BD
            Connection conexion = DriverManager.getConnection
                ("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");

            DatabaseMetaData dbmd = conexion.getMetaData(); //Creamos
                //objeto DatabaseMetaData
            ResultSet resul = null;

            String nombre = dbmd.getDatabaseProductName();
            String driver = dbmd.getDriverName();
            String url = dbmd.getURL();
            String usuario = dbmd.getUserName();

            System.out.println("INFORMACIÓN SOBRE LA BASE DE DATOS:");
            System.out.println("=====");
            System.out.println("Nombre : " + nombre);
            System.out.println("Driver : " + driver);
            System.out.println("URL : " + url);
            System.out.println("Usuario : " + usuario);

            //Obtener información de las tablas y vistas que hay
            resul = dbmd.getTables(null, "ejemplo", "%", null);

            while (resul.next()) {
                String catalogo = resul.getString(1); //columna 1
                String esquema = resul.getString(2); //columna 2
                String tabla = resul.getString(3); //columna 3
                String tipo = resul.getString(4); //columna 4
                System.out.println("Tipo: " + tipo + "Catalogo: " + catalogo +
                    " Esquema: " + esquema + " Tabla: " + tabla);
            }
            conexion.close(); //Cerrar conexion
        }
        catch (ClassNotFoundException cn) {cn.printStackTrace();}
        catch (SQLException e) {e.printStackTrace();}
    } //fin de main
} //fin de la clase

```

- **getColumns(catalogo, esquema, nombre_tabla, nombre_columna).** Devuelve un objeto `ResultSet` en el que cada fila tiene información sobre las columnas de una tabla o tablas.

```

ResultSet getColumns(String catalogo, String esquema,
                    String patronNombreDeTabla, String patronNombreDeColumna)
throws SQLException

```

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)... Más información: <https://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html#getColumns-java.lang.String-java.lang.String-java.lang.String-java.lang.String->

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. En el siguiente ejemplo se obtendrán todos los nombres de columna que empiezan por la letra "d" en la tabla departamentos y en el esquema de nombre ejemplo. Con valor null para todos los campos indica que obtendría información de todas las columnas de todas las tablas del esquema actual.

```
ResultSet columnas = dbmd.getColumns(null, "ejemplo",
                                     "departamento", "d%");

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");
}
```

En el siguiente ejemplo vemos la información sobre todas las columnas de la tabla departamentos.

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

DatabaseMetaData dbmd = conexion.getMetaData(); // Creamos
// objeto DatabaseMetaData
ResultSet resul = null;

System.out.println("COLUMNAS TABLA DEPARTAMENTOS:");
System.out.println("=====");
ResultSet columnas = null;
columnas = dbmd.getColumns(null, "ejemplo", "departamentos", null);
while (columnas.next()) {
    String nombCol = columnas.getString("COLUMN_NAME"); //getString(4)
    String tipoCol = columnas.getString("TYPE_NAME"); //getString(6)
    String tamCol = columnas.getString("COLUMN_SIZE"); //getString(7)
    String nula = columnas.getString("IS_NULLABLE"); //getString(18)

    System.out.printf("Columna: %s, Tipo: %s, Tamaño: %s, ¿Puede ser Nula? %s %n", nombCol, tipoCol, tamCol, nula);
}
```

- **getPrimaryKeys(catálogo, esquema, tabla).** Devuelve la lista de columnas que forman la clave primaria. La descripción de cada columna tiene las siguientes columnas: TABLE_CAT, TABLE_SCHEM, TABLE_NAME, COLUMN_NAME y KEY_SEQ.

```
ResultSet getPrimaryKeys(String catalogo, String esquema, String tabla)
throws SQLException
```

En el siguiente ejemplo se muestra la clave primaria de la tabla *departamentos*:

```
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. La descripción de cada columna de clave ajena tiene muchas [columnas](#).

ResultSet getExportedKeys (String catalogo, String esquema, String tabla)
throws SQLException

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

```
DatabaseMetaData dbmd = conexion.getMetaData(); // Creamos
// objeto DatabaseMetaData
ResultSet resul = null;

System.out.println("CLAVES ajenas que referencian a DEPARTAMENTOS:");
System.out.println("=====");

ResultSet fk = dbmd.getExportedKeys(null, "EJEMPLO", "DEPARTAMENTOS");

while (fk.next()) {
    String fk_name = fk.getString("FKCOLUMN_NAME");
    String pk_name = fk.getString("PKCOLUMN_NAME");
    String pk_tablename = fk.getString("PKTABLE_NAME");
    String fk_tablename = fk.getString("FKTABLE_NAME");
    System.out.printf("Tabla PK: %s, Clave Primaria: %s %n", pk_tablename, pk_name);
    System.out.printf("Tabla FK: %s, Clave Ajena: %s %n", fk_tablename, fk_name);
}
```

El método no devolverá nada si queremos ver las claves ajenas que referencian a la tabla empleados ya que la tabla empleados no es referenciada por ninguna clave ajena.

Para que este método nos devuelva la información deseada, debemos haber definido las restricciones de clave ajena y asignarles un nombre, usando la cláusula CONSTRAINT nombre FOREIGN KEY (col1, col2, ...) REFERENCES tabla (col1, col2, ...).

- **getImportedKeys(catálogo, esquema, tabla).** Devuelve la lista de claves ajenas existentes en la tabla. Se utiliza igual que el método anterior.

ResultSet getImportedKeys (String catalogo, String esquema, String tabla)
throws SQLException

- **getProcedures(catálogo, esquema, procedure).** Devuelve la lista de procedimientos almacenados. Cada descripción de procedimiento tiene varias [columnas](#). La sintaxis es:

ResultSet getProcedures (String catalogo, String esquema, String procedure)
throws SQLException

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

```
Class.forName("com.mysql.jdbc.Driver"); // Cargar el driver
Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");

DatabaseMetaData dbmd = conexion.getMetaData(); // Creamos objeto
// DatabaseMetaData

ResultSet proc = dbmd.getProcedures(null, "ejemplo", null);
while (proc.next()) {
    String proc_name = proc.getString("PROCEDURE_NAME");
    String proc_type = proc.getString("PROCEDURE_TYPE");
    System.out.printf("Nombre Procedimiento: %s - Tipo: %s %n", proc_name, proc_type);
}
```

5.2.2. La Interfaz ResultSetMetaData

La interfaz **ResultSetMetaData** proporciona metadatos (datos sobre los datos) sobre el conjunto de resultados devueltos (ResultSet) a través de diversos métodos (por ejemplo, los tipos y las propiedades de las columnas). Permite obtener información sobre los tipos y propiedades de las columnas de los objetos ResultSet. La documentación de esta interfaz se encuentra disponible en: <https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSetMetaData.html>.

Vamos a obtener información sobre las columnas devueltas por esta consulta “*SELECT * from departamentos*”, al usar * desconocemos el nombre de las columnas devueltas. Usaremos el método *getMetadata()* del objeto ResultSet que devuelve una referencia a ResultSetMetaData con el que se obtendrá la información acerca de las columnas devueltas. Veamos cómo se crea y se usa un objeto *ResultSetMetaData*:

```
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 tamaño = 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 cuya posición se indica en *índice*.
- **getColumnTypeName(índice)**. Devuelve el nombre del tipo de dato, específico del sistema gestor de base de datos, que contiene la columna indicada en *índice*.
- **getColumnDisplaySize(índice)**. Devuelve el máximo ancho en caracteres de la columna indicada en *índice*.
- **isNullable(índice)**. Devuelve 0 si la columna puede contener valores nulos.

5.3. 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:

- **ResultSet executeQuery(String 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.
- **int executeUpdate(String consulta).** Se utiliza para sentencias SQL que no devuelven un objeto *ResultSet*, como son las sentencias de definición de datos DDL (CREATE, ALTER y DROP) y las sentencias de manipulación de datos DML (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.
- **boolean execute(String consulta).** Se utiliza para ejecutar cualquier sentencia SQL. Tanto para las que devuelven un *ResultSet* (por ejemplo, SELECT) como para las que devuelven el número de filas afectadas (por ejemplo, INSERT, UPDATE, DELETE) y para las de definición de datos como por ejemplo CREATE.

El método devuelve true si devuelve un *ResultSet* (para recuperar las filas será necesario llamar al método *getResultSet()*) y false si se trata de un recuento de actualizaciones o no hay resultados; en este caso se usará el método *getUpdateCount()* para recuperar el valor devuelto.

Vamos a ver un ejemplo *execute()* donde se ejecuta una sentencia SELECT, devuelve true; por tanto, es necesario recuperar las filas devueltas usando el método *getResultSet()*:

```
import java.sql.*;
public class EjemploExecute {

    public static void main(String[] args) throws
        ClassNotFoundException, SQLException {

        //CONEXION A MYSQL
        Class.forName("com.mysql.jdbc.Driver");
        Connection conexion = DriverManager.getConnection
            ("jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");

        String sql="SELECT * FROM departamentos";
        Statement sentencia = conexion.createStatement();
        boolean valor = sentencia.execute(sql);

        if(valor){
            ResultSet rs = sentencia.getResultSet();
            while (rs.next())
                System.out.printf("%d, %s, %s %n",
                    rs.getInt(1), rs.getString(2), rs.getString(3));
            rs.close();
        } else {
            int f = sentencia.getUpdateCount();
            System.out.printf("Filas afectadas:%d %n", f);
        }

        sentencia.close();
        conexion.close();
    } //main
} //
```

Si cambiamos la sentencia SQL por *"UPDATE departamentos SET dnombre=LOWER(dnombre)"*, entonces la variable *valor* será igual a *false* y la salida del programa cambiará.

Normalmente, se usa para ejecutar procedimientos almacenados, o cuando no se sabe si la sentencia SQL es una consulta o una actualización.

5.3.1. 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: <https://docs.oracle.com/javase/8/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, esta interfaz a través de un objeto *ResultSet*, proporciona **métodos de acceso** para recuperar los valores de las columnas de la fila actual. Para acceder a estos valores se pueden obtener usando el número de índice de la columna (empezando en 1) o el nombre de la columna. También se puede obtener información sobre las columnas como el número o su tipo, como vimos anteriormente usando el método *getMetadata()* de un objeto *ResultSet*.

Algunos métodos *getXXX()* para obtener los valores son los siguientes:

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 SQLite o MySQL. Estos datos se introducen al programa desde la línea de comandos.

```
//datos a insertar
int dep = 15; // num. departamento
String dnombre = "INFORMATICA"; // nombre
String loc = "MADRID"; // localidad

//construir orden INSERT
String sql = String.format("INSERT INTO departamentos VALUES (%s, '%s', '%s')", dep, dnombre, loc);

System.out.println(sql);

//https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html

System.out.println(sql);
Statement sentencia = conexion.createStatement();
int filas=0;
try {
    filas = sentencia.executeUpdate(sql.toString());
    System.out.println("Filas afectadas: " + filas);
} catch (SQLException e) {
    //e.printStackTrace();
    System.out.printf("HA OCURRIDO UNA EXCEPCIÓN:\n");
    System.out.printf("Mensaje : %s \n", e.getMessage());
    System.out.printf("SQL estado: %s \n", e.getSQLState());
    System.out.printf("Cód error : %s \n", e.getErrorCode());
}
```

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.

```
String dep = "10", subida = "100";
try {
    Class.forName("com.mysql.jdbc.Driver");// Cargar el driver
    // Establecemos la conexión con la BD
    Connection conexion = DriverManager.getConnection(
        "jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");

    String sql = String.format("UPDATE empleados SET salario = salario + %s WHERE dept_no = %s",
        subida, dep);

    System.out.println(sql);

    Statement sentencia = conexion.createStatement();
    int filas = sentencia.executeUpdate(sql);
    System.out.printf("Empleados modificados: %d \n", filas);

    sentencia.close(); // Cerrar Statement
    conexion.close(); // Cerrar conexión
} catch (ClassNotFoundException cn) {
    cn.printStackTrace();
} catch (SQLException e) {
    if (e.getErrorCode() == 1062)
        System.out.println("CLAVE PRIMARIA DUPLICADA");
    else
        if (e.getErrorCode() == 1452)
            System.out.println("CLAVE AJENA "+ dep + " NO EXISTE");
        else {
            System.out.println("HA OCURRIDO UNA EXCEPCIÓN:");
            System.out.println("Mensaje: " + e.getMessage());
            System.out.println("SQL estado: " + e.getSQLState());
            System.out.println("Cód error: " + e.getErrorCode());
        }
}
```

El siguiente ejemplo crea una vista (de nombre *totales*) que contiene por cada departamento el número de departamento, el nombre, el número de empleados que tiene y el salario medio:

```
StringBuilder sql = new StringBuilder();
sql.append("CREATE OR REPLACE VIEW totales ");
sql.append("(dep, dnombre, nemp, media) AS ");
sql.append("SELECT d.dept_no, dnombre, COUNT(emp_no), AVG(salario) ");
sql.append("FROM departamentos d LEFT JOIN empleados e " );
sql.append("ON e.dept_no = d.dept_no ");
sql.append("GROUP BY d.dept_no, dnombre ");
System.out.println(sql);

Statement sentencia = conexion.createStatement();
int filas = sentencia.executeUpdate(sql.toString());
System.out.printf("Resultado de la ejecución: %d %n", filas);

sentencia.close(); // Cerrar Statement
conexion.close(); // Cerrar conexión
```

5.3.2. Ejecución de Scripts

Algunas bases de datos admiten la ejecución de varias sentencias DDL y DML en una misma cadena. Podemos cargar un script con la creación de tablas y los INSERT de las tablas desde un fichero a un String y hacer el *executeUpdate()* de ese String. Para ello, es necesario indicarlo en la conexión añadiendo la propiedad *allowMultiQueries=true* de la siguiente manera:

```
Connection connmysql =
    (Connection) DriverManager.getConnection("jdbc:mysql://localhost/
    ejemplo?allowMultiQueries=true", "ejemplo", "ejemplo");
```

No todas las bases de datos admiten la ejecución de múltiples sentencias SQL, entre las que lo admiten esta MySQL. Tendríamos un script para ejecutar almacenado en un fichero *scriptmysql.sql* dentro de nuestro Proyecto Java.

Posteriormente deberemos leer el fichero de texto que contiene el script línea a línea, y lo amancamos en un *StringBuilder*. Posteriormente lo ocnvertimos a String para ejecutarlo finalmente con un *executeUpdate()*:

```
public static void ejecutarScriptMySQL() {
    File scriptFile = new File("./script/scriptmysql.sql");
    System.out.println("-----");
    System.out.println("\n\nFichero de consulta : " + scriptFile.getName());
    System.out.println("Convirtiendo el fichero a cadena...");

    BufferedReader entrada = null;
    try {
        entrada = new BufferedReader(new FileReader(scriptFile));
    } catch (FileNotFoundException e) {
        System.out.println("ERROR NO ENCUENTRA el fichero: " + e.getMessage());
        e.printStackTrace();
    }

    String linea = null;
    StringBuilder stringBuilder = new StringBuilder();
    String salto = System.getProperty("line.separator");
    try {
        while ((linea = entrada.readLine()) != null) {
            stringBuilder.append(linea);
            stringBuilder.append(salto);
        }
    } catch (IOException e) {
        System.out.println("ERROR de E/S, al operar con el fichero: " + e.getMessage());
        e.printStackTrace();
    }
}
```

```

String consulta = stringBuilder.toString();

System.out.println(consulta);

try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
    System.out.println("ERROR En el Driver: " + e.getMessage());
    e.printStackTrace();
}

try {
    Connection connmysql = (Connection) DriverManager
        .getConnection("jdbc:mysql://localhost/ejemplo?allowMultiQueries=true", "usuario", "password");
    Statement sents = connmysql.createStatement();
    int res = sents.executeUpdate(consulta);
    System.out.println("Script creado con éxito, res = " + res);
    connmysql.close();
    sents.close();
} catch (SQLException e) {
    System.out.println("ERROR AL EJECUTAR EL SCRIPT: " + e.getMessage());
    e.printStackTrace();
}
}
}

```

5.3.3. 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.

Por ejemplo, la orden INSERT para insertar a un departamento se representa así:

```
String sql = "INSERT INTO departamentos VALUES (?, ?, ?)";

//1 2 3 valor del índice
```

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

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);
```

Para asignar valor a cada uno de los marcadores o placeholder de posición se utilizan los métodos `setXXX()`, el valor indicado en *tipoJava* al *parámetro* cuyo índice coincide con *indiceDelParametro*, que es transformado por el controlador JDNC en un tipo SQL correspondiente para pasarlo a la base de datos.

```
public abstract void setXXX(int indiceDelParametro, tipoJava valor)
```

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);
```

*`Integer.parseInt(dep)` y `Float.parseFloat(subida)`, convierte la cadena *dep* a un tipo entero y la cadena *subida* a un `Float`.

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

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

Más información: <https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html>

También podríamos usar esta interfaz con la sentencia SELECT. Por ejemplo recuperamos el apellido y el salario de los empleados de un departamento y un oficio en concreto obteniendo desde teclado los datos:

```
//recuperar parámetros de main
String dep = args[0];    //departamento
String oficio = args[1]; //oficio

//construimos la orden SELECT
String sql= "SELECT apellido, salario FROM empleados

                WHERE dept_no = ? AND oficio = ? ORDER BY 1";

// Preparamos la sentencia
PreparedStatement sentencia = conexion.prepareStatement(sql);

sentencia.setInt(1,Integer.parseInt(dep));
sentencia.setString(2,oficio);

ResultSet rs = sentencia.executeQuery();
while (rs.next())
    System.out.printf("%s => %.2f %n", rs.getString("apellido"),
                        rs.getFloat("salario"));

rs.close();// liberar recursos
sentencia.close();
conexion.close();
```

La ejecución de salida mostraría los vendedores del departamento 30:

```
java VerEmpleado 30 VENDEDOR
ARROYO => 1500,00
MARTÍN => 1600,00
SALA => 1625,00
TOVAR => 1350,00
```

6. GESTIÓN DE ERRORES

Hasta ahora todos los ejemplos cuando se producía un error se visualizaba con el método **printStackTrace()** la secuencia de llamadas al método que ha producido la excepción y la línea de código donde se produce el error.

Por ejemplo, se muestra el siguiente error cuando se intenta insertar alguna final en una tabla inexistente:

```
java.sql.SQLException: ORA-00942: la tabla o vista no existe

    at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:439)
    at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:395)
    at oracle.jdbc.driver.T4C8Oall.processError(T4C8Oall.java:802)
    at oracle.jdbc.driver.T4CTTIfun.receive(T4CTTIfun.java:436)
    at oracle.jdbc.driver.T4CTTIfun.doRPC(T4CTTIfun.java:186)
    at oracle.jdbc.driver.T4C8Oall.doOALL(T4C8Oall.java:521)
    at oracle.jdbc.driver.T4CStatement.doOall8(T4CStatement.java:194)
    at
    oracle.jdbc.driver.T4CStatement.executeForRows(T4CStatement.java:1000)
    at
    oracle.jdbc.driver.OracleStatement.doExecuteWithTimeout(OracleStatement
.java:1307)
    at
    oracle.jdbc.driver.OracleStatement.executeUpdateInternal(OracleStatemen
t.java:1814)
    at
    oracle.jdbc.driver.OracleStatement.executeUpdate(OracleStatement.java:1
779)
    at
    oracle.jdbc.driver.OracleStatementWrapper.executeUpdate(OracleStatement
Wrapper.java:277)
    at InsertarDep.main(InsertarDep.java:38)
```

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

Método	Función
<code>int getErrorCode()</code>	Devuelve un entero que proporciona el código de error del fabricante. Normalmente, será el código de error real devuelto por la base de datos.
<code>String getSQLState()</code>	Devuelve una cadena que contiene un estado definido por el estándar X/OPEN SQL.
<code>String getMessage()</code>	Devuelve una cadena que describe el error. Es un método heredado de la clase java.lang.Throwable .

A continuación se utilizan estos métodos para visualizar el mensaje de error:

```
try {
    //Codigo
} catch (ClassNotFoundException cn) {cn.printStackTrace();}
} catch (SQLException e) {
    System.out.printf("HA OCURRIDO UNA EXCEPCIÓN:\n");
    System.out.printf("Mensaje : %s \n", e.getMessage());
    System.out.printf("SQL estado: %s \n", e.getSQLState());
    System.out.printf("Cód error : %s \n", e.getErrorCode());
}
```

La siguiente salida muestra un ejemplo que se produce cuando se intenta hacer un SELECT de una tabla que no existe (en MySQL):

```
HA OCURRIDO UNA EXCEPCIÓN:  
Mensaje      : Table 'ejemplo.departamento' doesn't exist  
SQL estado:  42S02  
Cód error   : 1146
```

En Oracle se visualizaría la información de distinta forma:

```
HA OCURRIDO UNA EXCEPCIÓN:  
Mensaje      : ORA-00942: la tabla o vista no existe  
  
SQL estado:  42000  
Cód error   : 942
```

Cuando intentamos insertar una fila en una tabla cuya clave primaria ya existe en MySQL se muestra la siguiente información:

```
Mensaje      : Duplicate entry '10' for key 'PRIMARY'  
SQL estado:  23000  
Cód error   : 1062
```

En el caso de ORACLE:

```
Mensaje      : ORA-00001: restricción única (EJEMPLO.PK_DEP) violada  
SQL estado:  23000  
Cód error   : 1
```

Cuando se inserta una clave ajena en una tabla y no existe su correspondiente clave primaria en otra tabla, en MySQL nos indica lo siguiente:

```
Mensaje      : Cannot add or update a child row: a foreign key constraint  
fails (`ejemplo`.`empleados`, CONSTRAINT `FK_DEP` FOREIGN KEY  
(`dept_no`) REFERENCES `departamentos` (`dept_no`))  
SQL estado:  23000  
Cód error   : 1452
```

Para ORACLE:

```
Mensaje      : ORA-02291: restricción de integridad (EJEMPLO.FK_EMP)  
violada - clave principal no encontrada  
SQL estado:  23000  
Cód error   : 2291
```


7. 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 "resultset" 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 "resultset".
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.

```
#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 nColumns = 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);

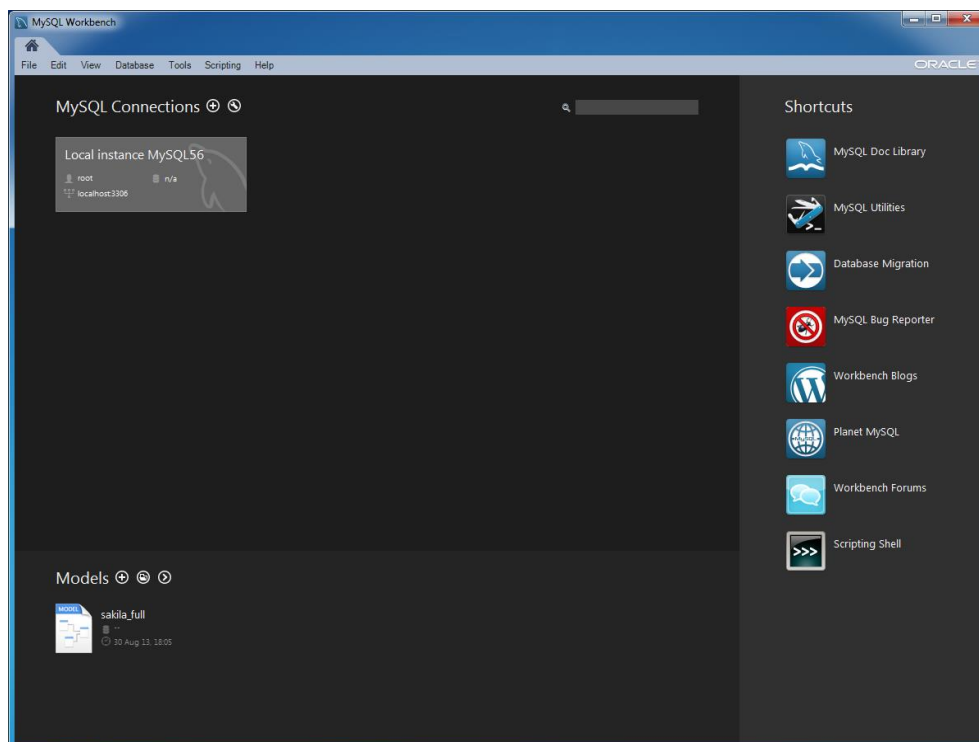
    // obtener número de columnas y número de filas de los resultados
    SQLNumResultCols(stmt, &nColumnas);
    SQLRowCount(stmt, &nFilas);
    printf("\nNúmero de Columnas: %d", nColumnas);
    printf("\nNúmero de Filas: %d", nFilas);

    // 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 ODBC
    SQLDisconnect(dbc);
}
else {
    printf("\nNo ha sido posible realizar la conexión.");
}
return 1;
} //fin de main

```

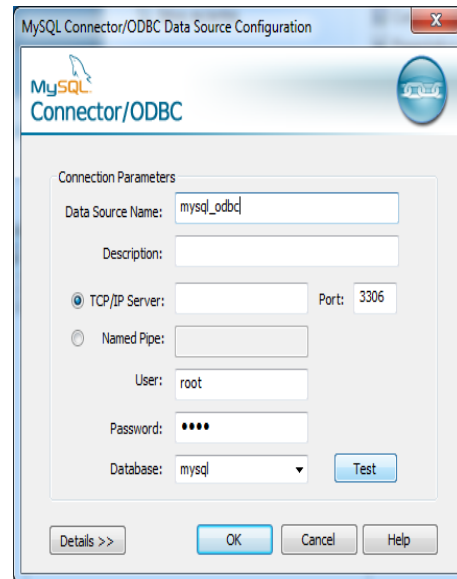
Previamente se debe instalar MySQL (servidor, workbench y otros) en Windows.



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](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>