

CAPÍTULO 2

MANEJO DE CONECTORES

OBJETIVOS

- Instalar y utilizar bases de datos embebidas.
- Utilizar conectores para acceder a bases de datos.
- Establecer conexiones a bases de datos.
- Desarrollar aplicaciones para acceder a los datos de la base de datos.
- Ejecutar procedimientos de bases de datos.
- Crear informes con datos almacenados en bases de datos.

CONTENIDOS

- Desfase objeto-relacional.
- Protocolos de acceso a bases de datos. Conectores.
- Bases de datos embebidas.
- Establecimiento de conexiones.
- Ejecución de sentencias de descripción de datos.
- Ejecución de sentencias de modificación de datos.
- Ejecución de consultas.
- Ejecución de procedimientos.
- Generación de informes.

RESUMEN

En este capítulo aprenderemos a acceder a los datos almacenados en distintas bases de datos relacionales utilizando el lenguaje de programación Java. Realizaremos programas para acceder a las distintas bases de datos, para ello, usaremos diferentes conectores, cada base de datos necesitará su conector. Aprenderemos a generar informes con JasperReports.

2.1. 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. Los orígenes de datos no tienen por qué ser relacionales y pueden provenir de muchas fuentes distintas. Algunos de los orígenes de datos con los que podemos encontrarnos son: una base de datos relacional remota en un servidor, una base de datos relacional local, una hoja de cálculo, un fichero de texto en nuestro ordenador, un servicio de información online, etc.

En esta unidad nos centraremos en los orígenes de datos relacionales, aprenderemos a realizar programas Java para acceder a una base de datos relacional. Para ello necesitaremos los **conectores**, que no son más que el software que se necesita para realizar las conexiones desde nuestro programa Java con una base de datos relacional.

2.2. EL DESFASE OBJETO-RELACIONAL

Actualmente las bases de datos orientadas a objetos están ganando cada vez más aceptación frente a las bases de datos relacionales, ya que solucionan las necesidades de aplicaciones más sofisticadas que requieren el tratamiento de elementos más complejos. Un ejemplo de estas son las aplicaciones para diseño y fabricación en ingeniería, los sistemas de información geográfica (GIS), experimentos científicos, aplicaciones multimedia, etc. Dentro de estas nuevas aplicaciones se definen las orientadas a objetos (OO) y, en general, el **Paradigma de Programación Orientada a Objetos (POO)** cuyos elementos complejos (nombrados anteriormente) son los **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 programación basados en objetos; 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 y más esfuerzo de programación debido a la diferencia de esquemas entre los elementos a almacenar (objetos) y las características del repositorio de la base de datos (tablas). A esto es a lo que se denomina **desfase objeto-relacional** (o *desajuste de la impedancia*) y se refiere a los problemas que ocurren debido a las diferencias entre el modelo de datos de la base de datos y el del lenguaje de programación orientado a objetos.

Sin embargo, el paradigma relacional y el paradigma orientado a objetos pueden ser “*amigos*”. Cada vez que los objetos deben extraerse o almacenarse en una base de datos relacional se requiere un mapeo desde las estructuras provistas en el modelo de datos a las provistas por el entorno de programación. Este tema se trata más ampliamente en la Capítulo 3.

2.3. BASES DE DATOS EMBEBIDAS

Cuando desarrollamos pequeñas aplicaciones en las que no vamos a almacenar grandes cantidades de información no es necesario que utilicemos un sistema gestor de base de datos como Oracle o MySQL. En su lugar podemos utilizar 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 ejecuta la aplicación, y termina cuando se cierra la aplicación.

Por lo general, este tipo de bases de datos vienen del movimiento *Open Source*, aunque también hay algunas de origen propietario. Veamos algunas de ellas.

2.3.1. SQLite

SQLite es un sistema gestor de base de datos multiplataforma escrito en C que proporciona un motor muy ligero. Las bases de datos se guardan en forma de ficheros por lo que es fácil trasladar la base de datos con la aplicación que la usa. Cuenta con una utilidad que nos permitirá ejecutar comandos SQL en modo consola. Es un proyecto de dominio público.

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

Su instalación es sencilla. Desde la página <http://www.sqlite.org/download.html> se puede descargar. Para sistemas Windows podemos descargar el fichero ZIP **sqlite-tools-win32-x86-3110100.zip**, al descomprimirlo obtenemos varios ficheros ejecutables, entre ellos **sqlite3.exe**. Al ejecutarlo desde la línea de comandos escribimos el nombre del fichero que contendrá la base de datos, si el fichero no existe se creará, si existe cargará la base de datos. El siguiente ejemplo crea la base de datos *ejemplo.db* (en la carpeta *D:\DB\SQLITE*), todas las tablas que creamos en esta sesión se almacenarán en este fichero, para finalizar la sesión se escribe el comando **.quit**, el comando **.tables** muestra las tablas creadas:

```
D:\>sqlite3 D:\DB\SQLITE\ejemplo.db
SQLite version 3.11.1 2016-03-03 16:17:53
Enter ".help" for usage hints.
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> .quit

D:\>sqlite3 D:\DB\SQLITE\ejemplo.db
```

Para instalar SQLite en Linux escribimos desde la línea de comandos:

```
$sudo apt-get install sqlite3
```

Y para ejecutar SQLite escribimos lo siguiente para crear la base de datos en la carpeta */home/usuario/DB/SQLITE* (que tiene que existir):

```
$ sqlite3 /home/usuario/DB/SQLITE/ejemplo.db
```

ACTIVIDAD 2.1

Crea las tablas EMPLEADOS y DEPARTAMENTOS en SQLite e inserta filas en ellas. La descripción de las tablas es la siguiente:

DEPARTAMENTOS:

DEPT_NO numérico clave primaria, DNOMBRE VARCHAR(15), LOC VARCHAR(15).

EMPLEADOS:

EMP_NO numérico clave primaria, APELLIDO VARCHAR(10), OFICIO VARCHAR(10), DIR numérico, FECHA_ALT DATE, SALARIO numérico, COMISION numérico, DEPT_NO numérico, es clave ajena y referencia a la tabla DEPARTAMENTOS.

2.3.2. 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: su tamaño reducido, está basada en Java y soporta los estándares SQL, ofrece un controlador integrado JDBC que permite incrustar Derby en cualquier solución basada en Java, soporta el tradicional paradigma cliente-servidor utilizando el servidor de red Derby. Es fácil de instalar, implementar y utilizar.

Para realizar la instalación descargamos la última versión desde la página Web: http://db.apache.org/derby/derby_downloads.html. En Windows descargamos el fichero: **db-derby-10.12.1.1-bin.zip**, y lo descomprimimos por ejemplo en *D:\db-derby-10.12.1.1-bin*. A partir de ahora, para poder utilizar Derby en nuestros programas Java, solo será necesario tener accesible la librería **derby.jar** en el CLASSPATH de nuestro programa o en nuestro proyecto Eclipse o Netbeans.

Apache Derby trae una serie de ficheros .BAT que nos permitirán ejecutar por consola órdenes para crear nuestras bases de datos y ejecutar sentencias DDL y DML. El fichero es **ij.bat** y se encuentra en la carpeta *bin* (*D:\db-derby-10.12.1.1-bin\bin*). Desde la línea de comandos del DOS nos dirigimos a dicha carpeta y ejecutamos el fichero **ij.bat**:

```
D:\db-derby-10.12.1.1-bin\bin>ij
Versión de ij 10.12
ij>
```

Para crear una base de datos de nombre *ejemplo* en la carpeta *D:\DB\DERBY* escribimos desde el indicador **ij>** la siguiente orden:

```
ij> connect 'jdbc:derby:D:\DB\DERBY\ejemplo;create=true';
```

Donde:

- **connect**, es el comando para establecer la conexión.
- **jdbc:derby**, es el protocolo JDBC especificado por DERBY.
- **ejemplo**, es el nombre de la base de datos que voy a crear (se crea una carpeta con dicho nombre y dentro una serie de ficheros).
- **create=true**, atributo usado para crear la base de datos.

Para salir de la línea de comandos de **ij**, escribimos **exit**;

El siguiente ejemplo muestra la creación de la base de datos *ejemplo*, la creación de la tabla *DEPARTAMENTOS*, la inserción de filas en la tabla y la ejecución del script *Empleados.sql* (que se encuentra en la carpeta *bin*) que crea la tabla *EMPLEADOS* e inserta filas en ella, al final se ejecuta el comando **exit**; para salir:

```
D:\db-derby-10.12.1.1-bin\bin>ij
Versión de ij 10.12
ij> connect 'jdbc:derby:D:\DB\DERBY\ejemplo;create=true';
ij> CREATE TABLE departamentos (
  dept_no INT NOT NULL PRIMARY KEY,
  dnombre VARCHAR(15),
  loc      VARCHAR(15)
);
0 filas insertadas/actualizadas/suprimidas
ij> INSERT INTO departamentos VALUES (10, 'CONTABILIDAD', 'SEVILLA');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamentos VALUES (20, 'INVESTIGACIÓN', 'MADRID');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamentos VALUES (30, 'VENTAS', 'BARCELONA');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamentos VALUES (40, 'PRODUCCIÓN', 'BILBAO');
1 fila insertada/actualizada/suprimida
ij> run 'Empleados.sql';
ij> CREATE TABLE empleados (
  emp_no    INT NOT NULL PRIMARY KEY,
  apellido  VARCHAR(10),
  oficio    VARCHAR(10),
  dir       INT,
  fecha_alt DATE      ,
  salario   FLOAT,
  comision  FLOAT,
  dept_no   INT NOT NULL REFERENCES departamentos(dept_no)
);
0 filas insertadas/actualizadas/suprimidas
ij> INSERT INTO empleados VALUES (7369, 'SANCHEZ', 'EMPLEADO', 7902, '1990-12-17', 1040, NULL, 20);
1 fila insertada/actualizada/suprimida
ij> INSERT INTO empleados VALUES (7499, 'ARROYO', 'VENDEDOR', 7698, '1990-02-20', 1500, 390, 30);
1 fila insertada/actualizada/suprimida
ij> INSERT INTO empleados VALUES (7521, 'SALA', 'VENDEDOR', 7698, '1991-02-22', 1625, 650, 30);
1 fila insertada/actualizada/suprimida
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 escribimos la siguiente orden desde la línea de comandos de **ij**: *connect 'jdbc:derby:D:\DB\DERBY\ejemplo';*

Para utilizar la base de datos en Linux (Ubuntu) tenemos que tener la máquina virtual de Java instalada. Seguimos los siguientes pasos para instalar **Oracle Java 8** que incluye los paquetes JDK8 y JRE8:

```
sudo add-apt-repository ppa:webupd8team/java  
sudo apt-get update  
sudo apt-get install oracle-java8-installer
```

A continuación se ejecuta la siguiente orden que instala el paquete para establecer las variables de entorno con la instalación de Java (antes ejecutamos *sudo apt-get update*):

```
sudo apt-get install oracle-java8-set-default
```

A continuación descargamos la versión para Linux y la descomprimimos en la carpeta */opt*, en este caso se ha descargado la versión: **db-derby-10.12.1.1-bin.tar.gz**. Se creará la carpeta */opt/db-derby-10.12.1.1-bin*. Configuramos la variable **DERBY_HOME** con el nombre de carpeta donde se ha descargado, ejecutando desde la línea de comandos:

```
$ export DERBY_HOME=/opt/db-derby-10.12.1.1-bin
```

Para usar Derby necesitamos incluir en el CLASSPATH los ficheros jar: **derby.jar** y **derbytools.jar**, escribimos las siguientes órdenes:

```
$ export CLASSPATH=$DERBY_HOME/lib/derby.jar:$DERBY_HOME/lib/derbytools.jar
```

A continuación nos dirigimos a la carpeta donde está instalada Apache Derby y ejecutamos **setEmbeddedCP**:

```
$ cd $DERBY_HOME/bin  
$ ./setEmbeddedCP
```

Desde aquí ya podemos utilizar la utilidad **ij** para crear nuestra base de datos escribiendo desde la línea de comandos: **java org.apache.derby.tools.ij**. Lo primero que se visualiza es la versión. El siguiente ejemplo muestra la creación de la base de datos *ejemplo* en la carpeta */home/usuario/DB/DERBY/*, a continuación se crea una tabla y después se finaliza la conexión ejecutando la orden **exit**:

```
$ java org.apache.derby.tools.ij  
ij version 10.12  
ij> connect 'jdbc:derby:/home/usuario/DB/DERBY/ejemplo;create=true';  
Wed Mar 23 17:31:12 PDT 2016 Thread[main,5,main]  
java.io.FileNotFoundException: derby.log (Permission denied)  
-----  
Wed Mar 23 17:31:14 PDT 2016:  
Booting Derby version The Apache Software Foundation - Apache Derby -  
10.12.1.1 - (1704137): instance a816c00e-0153-a608-42cc-000002f09800  
on database directory /home/usuario/DB/DERBY/ejemplo with class loader  
sun.misc.Launcher$AppClassLoader@610455d6  
Loaded from file:/opt/db-derby-10.12.1.1-bin/lib/derby.jar  
java.vendor=Oracle Corporation  
java.runtime.version=1.8.0_74-b02  
user.dir=/opt/db-derby-10.12.1.1-bin/bin  
os.name=Linux
```

```

os.arch=amd64
os.version=4.2.0-16-generic
derby.system.home=null
Database Class Loader started - derby.database.classpath=''
ij> CREATE TABLE EJEMPLO1 (
> NOMBRE VARCHAR(15)
> );
0 rows inserted/updated/deleted
ij> exit;
-----
Wed Mar 23 17:33:04 PDT 2016: Shutting down Derby engine
-----
Wed Mar 23 17:33:05 PDT 2016:
Shutting down instance a816c00e-0153-a608-42cc-000002f09800 on database
directory /home/usuario/DB/DERBY/ejemplo with class loader
sun.misc.Launcher$AppClassLoader@610455d6
-----
$
```

Para volver a usar la base de datos escribimos la siguiente orden desde la línea de comandos de **ij**: `connect 'jdbc:derby:/home/usuario/DB/DERBY/ejemplo';`. El resto de comandos SQL se usan igual que se usaron en el ejemplo para Windows.

ACTIVIDAD 2.2

Crea las tablas EMPLEADOS y DEPARTAMENTOS en Apache Derby e inserta filas en ellas.

2.3.3. 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. Soporta la mayor parte de las características y funciones incluidas en el estándar SQL:2011. Puede mantener la base de datos en memoria o en ficheros en disco. La última versión 2.3.4 mejora en el acceso y la gestión de grandes conjuntos de datos. Es compatible con hasta 270 mil millones de filas de datos en una sola base de datos y una capacidad de copia de seguridad en caliente.

Desde la URL <https://sourceforge.net/projects/hsqldb/files/> podemos descargarnos la última versión estable. En este caso, descargamos el fichero **hsqlDb-2.3.3.zip**. Lo descomprimimos, por ejemplo, en la unidad D se creará una carpeta con el nombre del fichero ZIP (*D:\hsqlDb-2.3.3\hsqlDb*), dentro de esa carpeta hay una con el nombre *hsqlDb*, la llevamos a la unidad D para que nos quede instalado en *D:\hsqlDb*. A continuación creamos una carpeta en *D:\hsqlDb\data* para guardar los datos de la base de datos que vamos a crear, la llamamos *ejemplo*, nos debe quedar: *D:\hsqlDb\data\ejemplo* (si no se crea la carpeta para la base de datos se mezclarán en *data* todas las bases de datos).

Abrimos la línea de comandos del DOS (como usuario administrador) y nos dirigimos a la carpeta *D:\hsqlDb\bin*, ejecutamos el fichero BAT de nombre **runUtil** con el parámetro **DatabaseManager**, para conectarnos a la base de datos y que se ejecute la interfaz gráfica de este sistema gestor de base datos:

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

Se abre una ventana desde la que tenemos que configurar la conexión, escribimos en el campo **Setting Name** un nombre para la conexión, de la lista **Type** seleccionamos la opción *HSQL Database Engine Standalone*, para que la base de datos la tome de un fichero si existe y si no existe, la cree; y en la casilla de **URL**, escribimos el nombre de la carpeta donde se almacenará la base de datos y el de la base de datos: *ejemplo/ejemplo*. Pulsamos el botón *OK*, véase Figura 2.1.

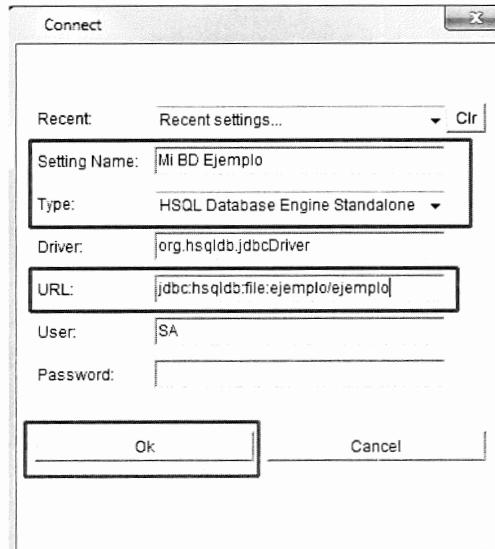


Figura 2.1. Configurar conexión en HSQLDB.

A continuación se abre una nueva ventana desde la que podemos ejecutar comandos DDL y DML para crear y manipular objetos de nuestra base de datos, véase Figura 2.2. Para ejecutar una sentencia SQL pulsamos el botón *Execute*. Desde la opción de menú *View->Refresh Tree* podemos actualizar el árbol de objetos.

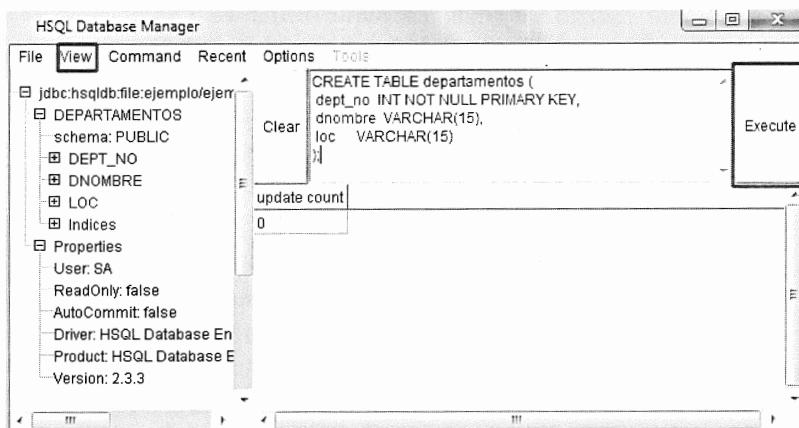


Figura 2.2. Ejecución de sentencias SQL en HSQLDB.

Para instalar la base de datos en Ubuntu primero guardamos el fichero **hsqldb-2.3.3.zip** en la carpeta */opt*. Después desde la línea de comandos nos vamos a dicha carpeta y lo descomprimimos ejecutando la siguiente orden:

```
$ sudo unzip hsqldb-2.3.3.zip
```

Se creará en `/opt` una carpeta con nombre `hsqldb-2.3.3`, dentro hay otra carpeta que se llama `hsqldb`, la movemos a `/opt`. Al final nos debe quedar `/opt/hsqldb`. A continuación escribo desde la línea de comandos las siguientes órdenes para establecer el PATH con las librerías de HSQLDB en la variable CLASSPATH:

```
$ CLASSPATH = "$CLASSPATH:/opt/hsqldb/lib/hsqldb.jar"
$ export CLASSPATH
```

Creamos en la carpeta `home/usuario/DB/HSQLDB` la carpeta `ejemplo` (nos debe quedar `/home/usuario/DB/HSQLDB/ejemplo`) para almacenar todos los datos de la base de datos de nombre `ejemplo` que crearemos a continuación. Seguidamente, ejecutamos desde la línea de comandos la siguiente orden:

```
$ java org.hsqldb.util.DatabaseManager
```

Se abre la ventana de conexión, similar a la mostrada en la Figura 2.1. Rellenamos los campos como se hizo anteriormente. En el campo **URL** escribimos la carpeta donde se almacenará la base de datos y su nombre, nos debe quedar de la siguiente manera: `jdbc:hsqldb:file:/home/usuario/DB/HSQLDB/ejemplo`. El resto de operaciones son similares.

ACTIVIDAD 2.3

Crea las tablas EMPLEADOS y DEPARTAMENTOS en HSQLDB inserta filas en ellas.

2.3.4. H2

H2 es un sistema gestor de base de datos relacional programado íntegramente en Java. Está disponible como software de código libre bajo la Licencia Pública de Mozilla o la Eclipse Public License. Desde la web <http://www.h2database.com/html/main.html> podemos descargarnos la última versión. Nos descargamos la versión ZIP para todas las plataformas **h2-2016-01-21.zip**. La descomprimimos por ejemplo en la unidad D, se creará una carpeta con el nombre `D:/h2`. Desde la línea de comandos del DOS nos dirigimos a la carpeta `D:\h2\bin` y ejecutamos el fichero **h2.bat** para arrancar la consola (también podemos hacer doble clic en el fichero para ejecutarlo):

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

Se abre el navegador Web con la consola de administración de H2, véase Figura 2.3. Escribimos un nombre para la configuración de la base de datos, en el campo **URL JDBC** escribimos la URL para la conexión a nuestra base de datos: `jdbc:h2:D:/DB/H2/ejemplo/ejemplo` (si las carpetas no existen se crean automáticamente), pulsamos el botón *Guardar* para que guarde la configuración y cada vez que queramos conectarnos a nuestra base de datos usemos ese nombre y pulsamos el botón *Conectar*. Podemos pulsar el botón *Probar conexión* antes de conectar para ver si todo ha ido bien, entonces se mostrará el mensaje: *Prueba correcta*. Se creará la carpeta `ejemplo` en H2 y dentro los ficheros de nuestra base de datos.

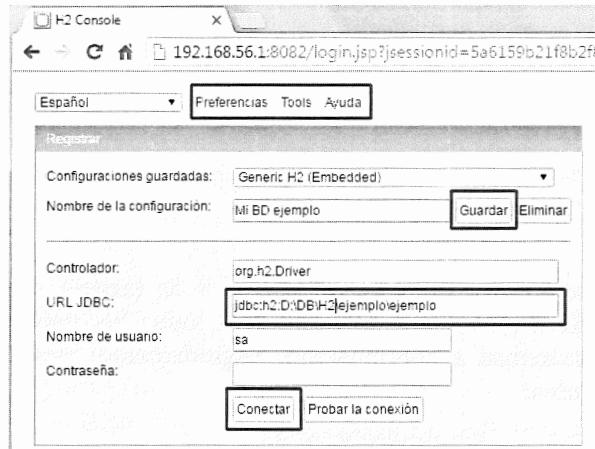


Figura 2.3. Establecer conexión en H2.

Una vez conectados se visualiza una nueva pantalla, véase Figura 2.4, desde la que podremos realizar las operaciones sobre la base de datos. Se muestran los comandos más importantes y un script de ejemplo.

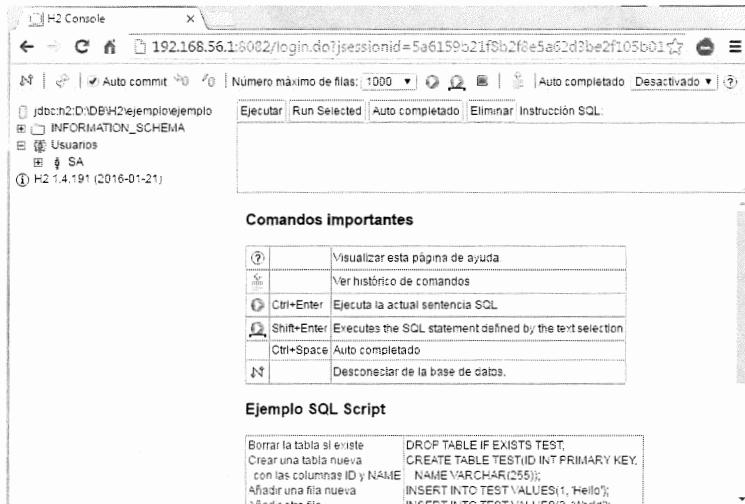


Figura 2.4. Pantalla de manejo de la base de datos en H2.

Desde la zona de instrucciones SQL podremos escribir las sentencias para crear tablas, insertar filas, etc., véase Figura 2.5. Los botones **Ejecutar** y **Esc** nos permitirán ejecutar las sentencia SQL que escribamos en el área de instrucción SQL. El botón *Desconectar* nos lleva a la pantalla inicial donde elegimos la conexión.

Desde el enlace **Preferencias** de la ventana inicial se pueden configurar diversos aspectos como: los clientes permitidos (locales/remotos), conexión segura (uso de SSL), puerto del servidor Web o notificar las sesiones activas. La opción **Tools** presenta una serie de herramientas que se pueden utilizar sobre la base de datos: backup, restaurar base de datos, ejecutar scripts, convertir la base de datos en un script, encriptación, etc.

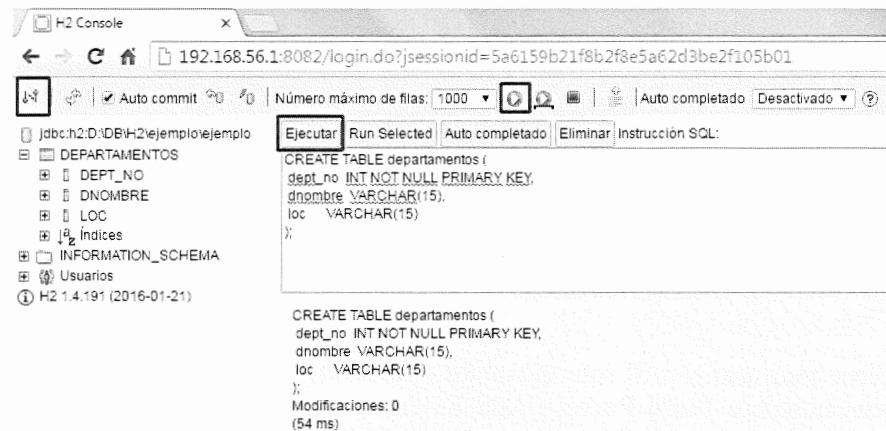


Figura 2.5. Ejecución de sentencias en H2.

En Ubuntu extraemos el fichero **h2-2016-01-21.zip** en la carpeta *opt*, de esta manera tendremos */opt/h2*. Nos dirigimos a la carpeta */opt/h2/bin* y ejecutamos el fichero **h2.sh** para arrancar la consola (también se puede ejecutar desde el entorno gráfico haciendo doble clic en el fichero), desde la línea de comandos escribimos:

```
/opt/h2/bin$ ./h2.sh
```

Si se visualiza el mensaje de permiso denegado ejecutamos las siguientes órdenes para dar permiso de ejecución y después para ejecutar el fichero **h2.sh**:

```
/opt/h2/bin$ sudo chmod +x h2.sh
/opt/h2/bin$ ./h2.sh
```

El resto de pasos son similares a los vistos anteriormente, salvo la escritura de la URL. En el campo **JDBC URL** escribimos lo siguiente *jdbc:h2:/home/usuario/DB/H2/ejemplo/ejemplo*, para que nos guarde la base de datos en la carpeta */home/usuario/DB/H2/ejemplo*. Si las carpetas no existen se crean automáticamente, y dentro los ficheros para la base de datos de nombre *ejemplo*.

ACTIVIDAD 2.4

Crea las tablas EMPLEADOS y DEPARTAMENTOS en H2 e inserta filas en ellas.

2.3.5. Db4o

Db4o (*DataBase 4 (for) Objects*) es un motor de base de datos orientado a objetos. Se puede utilizar de forma embebida o en aplicaciones cliente-servidor. Está disponible para entornos Java y .Net. Dispone de licencia dual GPL/comercial. Proporciona algunas características interesantes:

- Se evita el problema del desfase objeto-relacional.
- No existe un lenguaje SQL para la manipulación de datos, en su lugar existen métodos delegados.

- Se instala añadiendo un único fichero de librería (JAR para Java o DLL para .NET).
- Se crea un único fichero de base de datos con la extensión .YAP (tamaño de 2GB a 264GB).

Desde la URL <http://supportservices.actian.com/versant/default.html> podemos descargarnos la última versión. Para el ejemplo se ha descargado la versión **db4o-8.0.276.16149-java.zip**. Al descomprimirla hay una carpeta de nombre *lib* donde se encuentra el fichero JAR **db4o-8.0.276.16149-all-java5.jar** que necesitamos para utilizar el motor de la base de datos. Desde la URL <https://sourceforge.net/projects/db4o/files/db4o/> podemos descargar versiones anteriores de Db4o.

En Eclipse para usar el JAR seleccionamos nuestro proyecto, pulsamos el botón derecho del ratón y seleccionamos **Build Paths-> Add External Archives**, véase Figura 2.6. Se visualiza una ventana desde la que localizaremos el fichero JAR a incluir en el proyecto (conviene crear una carpeta e ir incluyendo los JAR que después usaremos en los ejercicios). Se mostrará en nuestro proyecto un elemento más, **Referenced Libraries**, con el fichero JAR añadido.

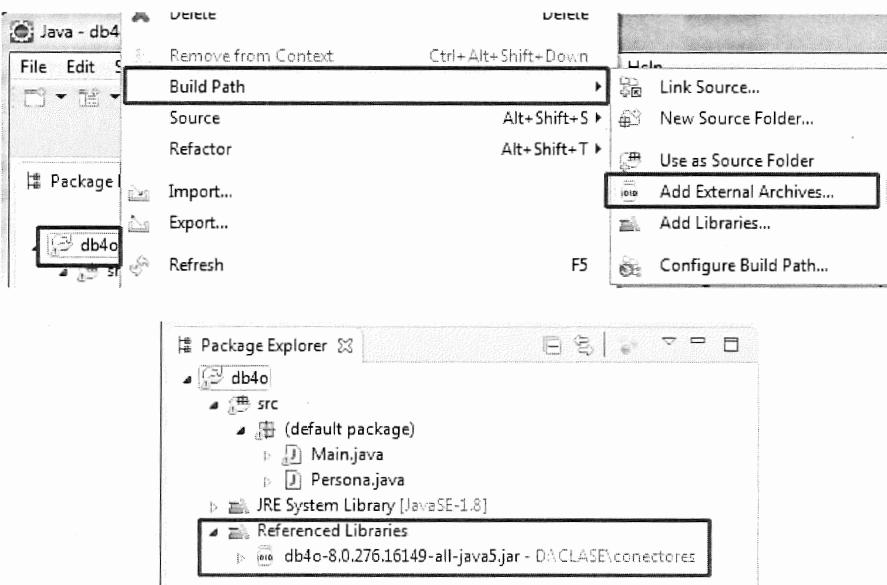


Figura 2.6. Añadir el JAR al proyecto Eclipse.

También podemos añadir los JAR desde la opción **Build Path -> Configure Build Path**. La instalación en Linux es similar. Si no usamos entorno gráfico para nuestros programas Java hemos de incluir en la variable CLASSPATH los JAR necesarios.

A continuación se muestra la clase *Main.java* que crea una base de datos (si no existe) de nombre *DBPersonas.yap* y almacena objetos *Persona* en ella:

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

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

    public static void main(String[] args) {
        ObjectContainer db =

```

```

Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), BDPer);

// Se crean objetos Persona
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 metodo main
}// fin de la clase Main

```

Se ha definido la clase *Persona* formada por los atributos *nombre* y *ciudad* y 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 nombre) {
        this.nombre = nombre;
    }
    public String getCiudad() {
        return ciudad;
    }
    public void setCiudad(String ciudad) {
        this.ciudad = ciudad;
    }
}

//fin Persona

```

Desde Eclipse para generar automáticamente los getters y los setters de los atributos pulsamos con el botón derecho del ratón en el código de la clase, seleccionamos la opción **Source** y a continuación **Generate Getters and Setters**, véase Figura 2.7. A continuación hemos de seleccionar los campos y pulsar el botón *OK*. Para generar los constructores pulsamos sobre la

opción **Generate Constructor using Fields** para generar el constructor con parámetros, o **Generate Constructors from Superclass** para generar el constructor sin parámetros.

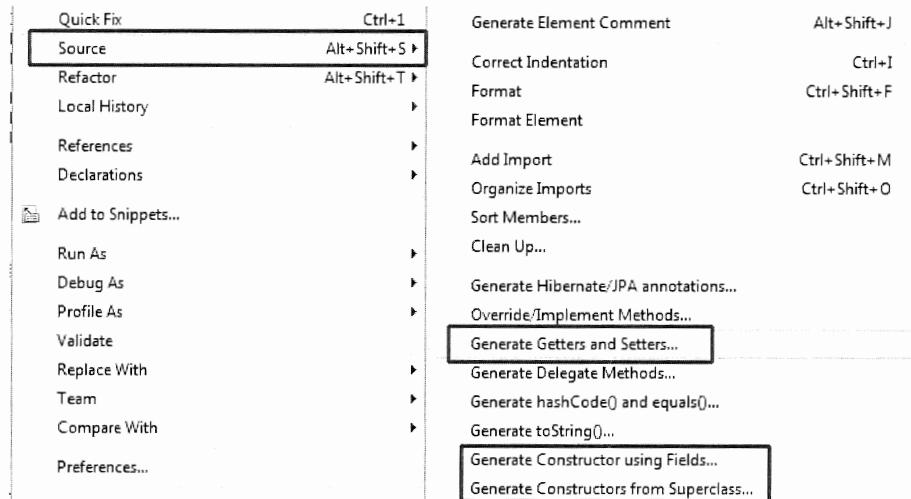


Figura 2.7. Generar getters, setters y constructores.

El paquete Java **com.db4o** contiene casi toda la funcionalidad de la base de datos. Para este ejemplo necesitamos importar la clase **com.db4o.Db4oEmbedded**, y la interfaz **com.db4o.ObjectContainer**.

Para realizar cualquier acción, ya sea insertar, modificar o realizar consultas debemos manipular una instancia de **ObjectContainer** donde se define el fichero de base de datos, en el ejemplo el fichero se llama *DBPersonas.yap* y el nombre se almacena en la variable *BDPer* donde será necesario incluir el trayecto donde se encuentra el fichero. Algunos de los métodos más importantes son:

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

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

- Para cerrarla llamamos al método ***close()***:

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

- Para almacenar un objeto utilizamos el método ***store()***:

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

- Para recuperar objetos podemos utilizar el sistema de consultas QBE (*Query-By-Example*) mediante el método ***queryByExample()***. El siguiente ejemplo muestra todos los objetos *Persona* existentes en la base de datos, los resultados se proporcionan en forma de **ObjectSet**. Si no existe ningún objeto el método ***size()*** sobre el objeto **ObjectSet** devolverá 0:

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

```

public class Consulta1 {
    static String BDPer = "DBPersonas.yap";
    public static void main(String[] args) {
        ObjectContainer db =
            Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), BDPer);

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

        if (result.size() == 0)
            System.out.println("No existen Registros de Personas..");
        else {
            System.out.printf("Número de registros: %d %n",
                result.size());
            while (result.hasNext()) {
                Persona p = result.next();
                System.out.printf("Nombre: %s, Ciudad: %s %n",
                    p.getNombre(), p.getCiudad());
            }
        }
        db.close(); // cerrar base de datos
    }//main
}//fin Consulta1

```

La siguiente consulta obtiene los objetos *Persona* cuyo nombre es Juan:

```

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

```

La siguiente consulta obtiene los objetos *Persona* cuya ciudad es Guadalajara:

```

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

```

- 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 modificarlos todos habría que hacer un bucle):

```

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());  
    }  
  
    ▪ Para eliminar objetos utilizamos el método delete(), antes será necesario localizar el  
    objeto a eliminar. El siguiente ejemplo elimina todos los objetos cuyo nombre sea  
    Juan:
```

```
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....");  
    }  
}  
}
```

En el Capítulo 4 se profundizará más acerca de las bases de datos orientadas a objetos.

ACTIVIDAD 2.5

Crea una base de datos Db4o de nombre EMPLEDEP.YAP e inserta objetos EMPLEADOS y DEPARTAMENTOS en ella. Después obtén todos los objetos empleado de un departamento concreto. Visualiza también el nombre de dicho departamento.

2.3.6. Otras

Existen más sistemas de bases de datos embebidos tanto en software libre como en 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 licencias duales, por lo que es totalmente libre y se puede usar tanto en aplicaciones comerciales como de código abierto. Se presenta en tres versiones del servidor: *SuperServer*, *Classic* y *Embedded*. La edición embebida (*Embedded*) es un completo servidor Firebird empacado en unos cuantos ficheros. Es fácil distribuir aplicaciones, puesto que no requiere instalación, ideal 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 (PSQL).
 - 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. Es 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 un optimizador de consultas especialmente diseñado para entornos móviles.
 - Soporta un subconjunto de tipos de datos y de sentencias *Transact-SQL* de *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 ofrece la posibilidad de cifrado del fichero de base de datos con una contraseña de acceso restringida.
 - *SQL Server Compact 4.0* se ha optimizado y ajustado para usarse con aplicaciones Web *ASP.NET*.

2.4. PROTOCOLOS DE ACCESO A BASES DE DATOS

En tecnologías de base de datos podemos encontrarnos con dos normas de conexión a una base de datos SQL:

- **ODBC (Open Database Connectivity)** define una API (*Application Program Interface - Interfaz para Programas de Aplicación*) que pueden usar las aplicaciones para abrir una conexión con una base de datos, enviar consultas, actualizaciones y obtener los resultados. Las aplicaciones pueden usar esta API para conectarse a cualquier servidor de base de datos compatible con ODBC. Está escrito en C.
- **JDBC (Java Database Connectivity - Conectividad de base de Datos con Java)** define una API que pueden usar los programas Java para conectarse a los servidores de bases de datos relacionales.

Hay muchos orígenes de datos que no son bases de datos relacionales, algunos puede que ni si quiera sean bases de datos, tal es el caso de los ficheros planos y los almacenes de correo electrónico.

OLE-DB (Object Linking and Embedding for Databases - Enlace e incrustación de objetos para bases de datos) de Microsoft es una API de C++ con objetivos parecidos a los de ODBC, pero para orígenes de datos que no son bases de datos. OLE-DB proporciona estructuras para la conexión con orígenes de datos, ejecución de comandos y devolución de resultados en forma de

conjunto de filas. Sin embargo, se diferencia de ODBC en algunos aspectos. Los programas OLE-DB pueden negociar con los orígenes de datos para averiguar las interfaces que soportan. En ODBC los comandos siempre están en SQL, en OLE-DB pueden estar en cualquier lenguaje soportado por el origen de datos, puede que algunos orígenes soporten SQL o un subconjunto limitado de SQL y otros ofrezcan el acceso a los datos de los ficheros planos sin ninguna capacidad de consulta.

La API ADO (*Active Data Objects – Objetos activos de datos*) creada por Microsoft ofrece una interfaz sencilla de utilizar con la funcionalidad OLE-DB, que puede llamarse desde los lenguajes de guiones como VBScript y JScript.

2.5. ACCESO A DATOS MEDIANTE ODBC

ODBC es un estándar de acceso a bases de datos desarrollado por *Microsoft Corporation* con el objetivo de posibilitar el acceso a cualquier dato desde cualquier aplicación, sin importar qué sistema gestor de bases de datos almacene los datos. Cada sistema de base de datos compatible con ODBC proporciona una biblioteca que se debe enlazar con el programa cliente. Cuando el programa cliente realiza una llamada a la API ODBC, el código de la biblioteca se comunica con el servidor para realizar la acción solicitada y obtener los resultados.

Los pasos para usar ODBC son:

1. El primer paso es configurar la interfaz ODBC, para ello el programa asigna en primer lugar un entorno SQL con la función *SQLAllocHandle()*, después un manejador (o handle) para la conexión a la base de datos basada en el entorno anterior, el propósito de este manejador es traducir las consultas de datos de la aplicación en comandos que el sistema de base de datos entienda. ODBC define varios tipos de manejadores:
 - **SQLHENV**: define el entorno de acceso a los datos.
 - **SQLHDBC**: identifica el estado y configuración de la conexión (driver y origen de datos).
 - **SQLHSTMT**: declaración SQL y cualquier conjunto de resultados asociados.
 - **SQLHDESC**: recolección de metadatos utilizados para describir una sentencia SQL.
2. Una vez reservados los manejadores el programa abre la conexión a la base de datos usando *SQLDriverConnect()* o *SQLConnect()*.
3. Una vez realizada la conexión el programa puede enviar órdenes SQL a la base de datos usando *SQLExecDirect()*.
4. Al final de la sesión el programa se desconecta de la base de datos y libera la conexión y los manejadores del entorno SQL.

La API ODBC usa una interfaz escrita en el lenguaje C y no es apropiada para su uso directo desde Java. Las llamadas desde Java a código C nativo tienen un número de inconvenientes en la seguridad, implementación, robustez y portabilidad de las aplicaciones. ODBC es duro de aprender. Mezcla características elementales con otras más avanzadas y tiene complejas opciones incluso para las consultas más simples.

Algunas funciones importantes son:

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

El siguiente programa C accede mediante ODBC a una base de datos MySQL llamada *ejemplo* que tiene creadas las tablas EMPLEADOS y DEPARTAMENTOS; el usuario y la clave tienen el mismo nombre que la base de datos (en el ejemplo el servidor de base de datos MySQL está instalado mediante el paquete *XAMPP para Linux 1.8.2-2*). Para poder acceder necesitamos crear un origen de datos ODBC. Desde Linux (Ubuntu 12) hemos de instalar los paquetes *unixodbc*, *unixodbc-dev* y *libmyodbc* y crear el origen de datos, por ejemplo *Mysql-odbc*. Al instalarse los paquetes la librería *libmyodbc.so* se instala en la carpeta */usr/lib/i386-linux-gnu/odbc*:

```
# apt-get install unixodbc unixodbc-dev libmyodbc
```

Editamos el fichero **odbcinst.ini** de la carpeta */etc/*, si no existe lo creamos y escribimos las siguientes líneas donde indicamos dónde se encuentra el driver ODBC:

```
# gedit /etc/odbcinst.ini
[MySQL]
Description      = Mysql Connector 3.51
Driver          = /usr/lib/i386-linux-gnu/odbc/libmyodbc.so
UsageCount      = 1
CPTimeout       =
CPReuse         =
```

Editamos el fichero **odbc.ini** de la carpeta */etc/*, si no existe lo creamos y escribimos las siguientes líneas para crear el origen de datos ODBC de nombre **Mysql-odbc** en el que definimos el driver, la base de datos y el usuario con su clave para acceder a dicha base de datos:

```
# gedit /etc/odbc.ini
[ODBC Data Sources]
Mysql-odbc      = MyODBC 3.51 Driver DSN

[MySQL-odbc]
Driver          = MySQL
Description     = Base de datos MySQL ejemplo
SERVER          = 127.0.0.1
PORT            = 3306
USER            = ejemplo
Password        = ejemplo
Database        = ejemplo
OPTION          = 3
SOCKET          =
```

El código del programa C (de nombre *conexión.c*) es el siguiente, necesitamos las librerías *<sql.h>* y *<sqlext.h>*:

```
#include <stdio.h>
#include <sql.h>
#include <sqlext.h>

int main() {
    SQLHENV env;
    SQLHDBC dbc;
    SQLHSTMT stmt;
    SQLRETURN ret;

    //Definir el entorno
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
    SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void *) SQL_OV_ODBC3, 0);
    SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);

    //conectarse al origen de datos Mysql-odbc
    ret = SQLDriverConnect(dbc, NULL, "DSN=Mysql-odbc;", SQL_NTS,
                           NULL, 0, NULL, SQL_DRIVER_COMPLETE);

    char *consulta ="SELECT * FROM departamentos";
    SQLSMALLINT nCols = 0;
    SQLINTEGER nFilas = 0;
    SQLINTEGER nIndicator = 0;
    SQLCHAR buf[1024] = {0};

    if (SQL_SUCCEEDED(ret)) {
        printf("Conectado\n");
        SQLAllocStmt(dbc,&stmt );
        ret =SQLExecDirect( stmt,consulta, SQL_NTS );
        SQLNumResultCols( stmt, &nCols );
        SQLRowCount( stmt, &nFilas );
        printf("* Número de Columnas: %u\n", nCols );
        printf("* Número de Filas: %u \n", nFilas );

        while( SQL_SUCCEEDED( ret = SQLFetch( stmt ) ) )
        {
            printf("\n");
            for( int i=1; i <= nCols; ++i )
            {
                ret = SQLGetData( stmt,i,SQL_C_CHAR,buf,1024,&nIndicator );
                if( SQL_SUCCEEDED( ret ) )
                {
                    printf("* Columna %s", buf );
                }
            }
        } // while
        printf("\n");
        SQLFreeHandle( SQL_HANDLE_STMT, stmt );
        SQLDisconnect( dbc );
    } else { printf("NO HA SIDO POSIBLE LA CONEXIÓN\n"); }
    return 1;
} // fin main
```

Lo compilamos con el compilador **gcc** cargando la librería *-lodbc* y con la opción *-std=gnu99*, puede que aparezca algún error *Warning*; y luego lo ejecutamos:

```
# gcc conexion.c -lodbc -std=gnu99
conexion.c: En la función 'main':
conexion.c:33:8: aviso: formato '%u' espera un argumento de tipo
'unsigned int', pero el argumento 2 es de tipo 'SQLINTEGER' [-Wformat]
# ./a.out
Conectado
* Número de Columnas: 3
* Número de Filas: 4

* Columna 10* Columna CONTABILIDAD* Columna SEVILLA
* Columna 20* Columna INVESTIGACIÓN* Columna MADRID
* Columna 30* Columna VENTAS* Columna BARCELONA
* Columna 40* Columna PRODUCCIÓN* Columna BILBAO
#
#
```

2.6. ACCESO A DATOS MEDIANTE JDBC

JDBC proporciona una librería estándar para acceder a fuentes de datos principalmente orientados a bases de datos relacionales que usan SQL. No solo provee una interfaz sino que también define una arquitectura estándar, para que los fabricantes puedan crear los drivers que permitan a las aplicaciones Java el acceso a los datos. JDBC dispone de una interfaz distinta para cada base de datos (véase Figura 2.8), es lo que llamamos **driver** (controlador o conector). Esto permite que las llamadas a los métodos Java de las clases JDBC se correspondan con el API de la base de datos.

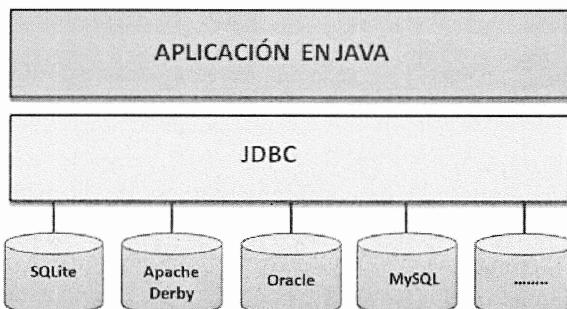


Figura 2.8. Acceso mediante JDBC.

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

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

2.6.1. Dos modelos de acceso a bases de datos

La API JDBC es compatible con los modelos tanto de dos como de tres capas para el acceso a la base de datos. En el **modelo de dos capas**, un applet o una aplicación Java “hablan” directamente con la base de datos, esto requiere un driver JDBC residiendo en el mismo lugar que la aplicación (Figura 2.9). Desde el programa Java se envían sentencias SQL al sistema gestor de base de datos para que las procese y los resultados se envíen de vuelta al programa. La base de datos puede encontrarse en otra máquina diferente a la de la aplicación y las solicitudes se hacen a través de la red (arquitectura cliente-servidor). El driver será el encargado de manejar la comunicación a través de la red de forma transparente al programa.

En el **modelo de tres capas**, los comandos se envían a una capa intermedia que se encargará de enviar los comandos SQL a la base de datos y de recoger los resultados de la ejecución de las sentencias. Es decir, tenemos una aplicación o applet corriendo en una máquina y accediendo a un driver de base de datos situado en otra máquina, véase Figura 2.10. En este caso los drivers no tienen que residir en la máquina cliente.

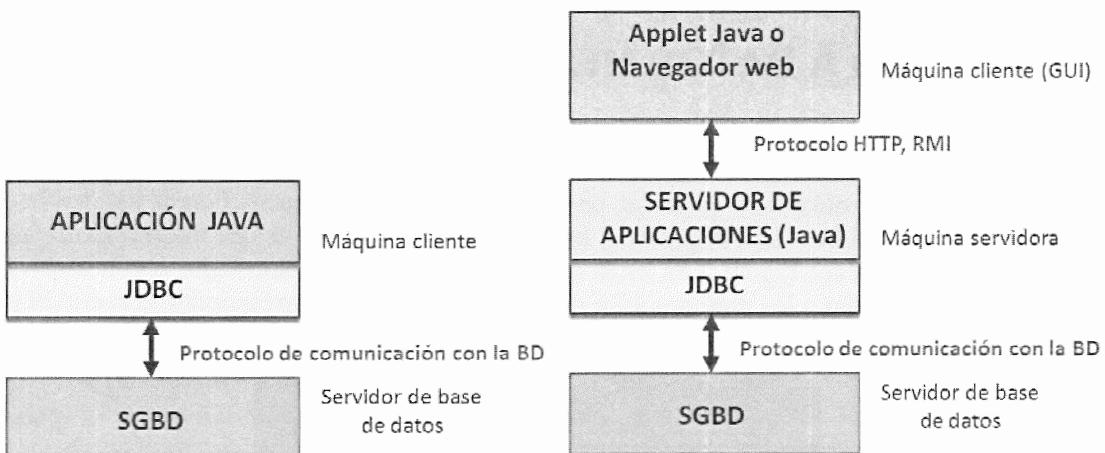


Figura 2.9. Modelo de dos capas.

Figura 2.10. Modelo de tres capas.

Un **servidor de aplicaciones** es una implementación de la especificación J2EE (*Java 2 Platform Enterprise Edition*). J2EE 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. Algunas de ellas son las siguientes: *BEA WebLogic*, *IBM WebSphere*, *Oracle IAS*, *Borland AppServer*, etc.

2.6.2. Tipos de drivers

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

- **Tipo 1. JDBC-ODBC Bridge** (*JDBC-ODBC bridge plus ODBC driver*): permite el acceso a bases de datos JDBC mediante un driver ODBC. Convierte las llamadas al API de JDBC en llamadas ODBC. Exige la instalación y configuración de ODBC en la máquina cliente.
- **Tipo 2. Native** (*Native-API partly-Java driver*): controlador escrito parcialmente en Java y en código nativo de la base de datos. Traduce las llamadas al API de JDBC

Java en llamadas propias del motor de base de datos. Exige instalar en la máquina cliente código binario propio del cliente de base de datos y del sistema operativo.

- **Tipo 3. Network (*JDBC-Net pure Java driver*):** controlador de Java puro que utiliza un protocolo de red (por ejemplo HTTP) para comunicarse con el servidor de base de datos. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red independiente de la base de datos y a continuación son traducidas por un software intermedio (*Middleware*) al protocolo usado por el motor de base de datos. El driver JDBC no comunica directamente con la base de datos, comunica con el software intermedio, que a su vez comunica con la base de datos. Son útiles para aplicaciones que necesitan interactuar con diferentes formatos de bases de datos, ya que usan el mismo driver JDBC sin importar la base de datos específica. No exige instalación en cliente.
- **Tipo 4. Thin (*Native-protocol pure Java driver*):** controlador de Java puro con protocolo nativo. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red usado por el motor de base de datos. No exige instalación en cliente.

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

2.6.3. Cómo funciona JDBC

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

CLASE E 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, etc.
Statement	Permite ejecutar sentencias SQL sin parámetros
PreparedStatement	Permite ejecutar sentencias SQL con parámetros de entrada
CallableStatement	Permite ejecutar sentencias SQL con parámetros de entrada y salida, como llamadas a procedimientos almacenados
ResultSet	Contiene las filas resultantes de ejecutar una orden SELECT.
ResultSetMetadata	Permite obtener información sobre un ResultSet , como el número de columnas, sus nombres, etc.

<http://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html>

La Figura 2.11 muestra las 4 clases principales que usa cualquier programa Java con JDBC. El trabajo con JDBC comienza con la clase **DriverManager** que es la encargada de establecer las conexiones con los orígenes de datos a través de los drivers JDBC. El funcionamiento de un programa con JDBC requiere los siguientes pasos:

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**.

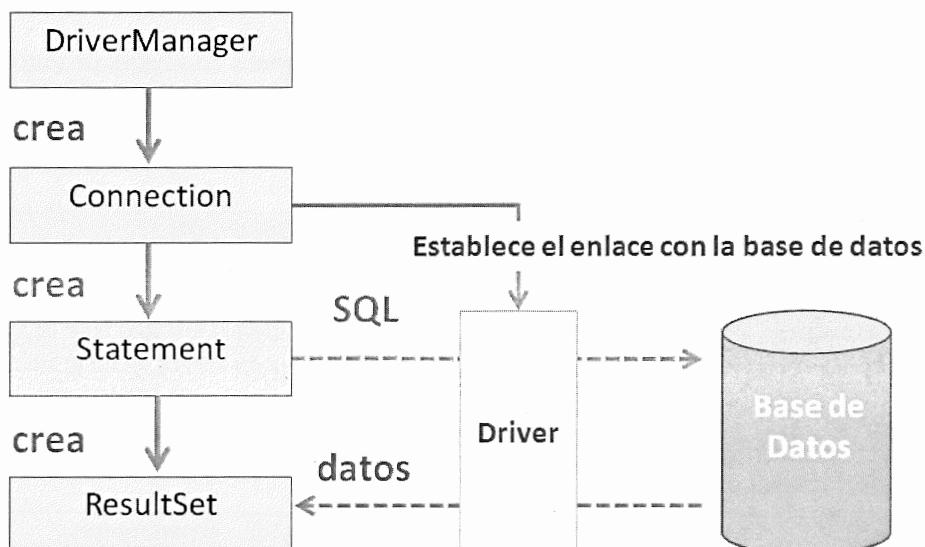


Figura 2.11. Funcionamiento de JDBC.

Para el siguiente ejemplo Java creamos desde MySQL una base de datos y un usuario con nombre *ejemplo*, la clave del usuario es la misma. Este usuario tendrá todos los privilegios sobre esta base de datos. A continuación creamos las siguientes tablas e insertamos datos en ellas, las relaciones se muestran en la Figura 2.12:

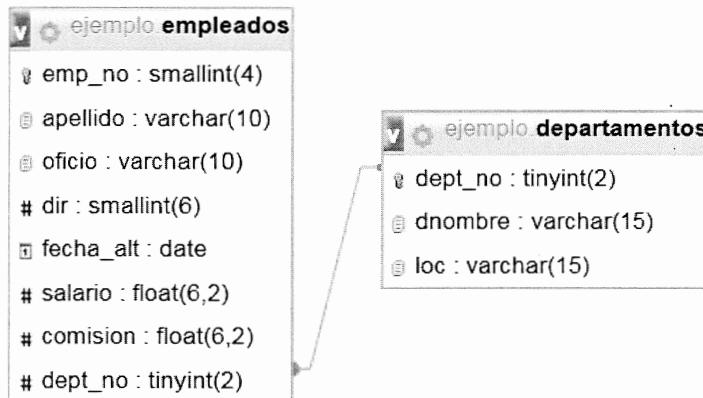


Figura 2.12. Base de datos *ejemplo*.

```

CREATE TABLE departamentos (
    dept_no  TINYINT(2) NOT NULL PRIMARY KEY,
    dnombre  VARCHAR(15),
    loc      VARCHAR(15)
) ENGINE=InnoDB;

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,
    CONSTRAINT FK_DEP FOREIGN KEY (dept_no) REFERENCES
                                         departamentos(dept_no)
) ENGINE=InnoDB;

```

El siguiente programa ilustra los pasos de funcionamiento de JDBC accediendo a la base de datos anterior y mostrando el contenido de la tabla *departamentos*:

```

import java.sql.*;
public class Main {
    public static void main(String[] args) {
        try {
            //Cargar el driver
            Class.forName("com.mysql.jdbc.Driver");

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

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

```

```

//Recorremos el resultado para visualizar cada fila
//Se hace un bucle mientras haya registros y se van mostrando
while (resul.next()) {
    System.out.printf("%d, %s, %s %n",
                      resul.getInt(1),
                      resul.getString(2),
                      resul.getString(3));
}

resul.close();      // Cerrar ResultSet
sentencia.close(); // Cerrar Statement
conexion.close();  // Cerrar conexión

} catch (ClassNotFoundException cn) {
    cn.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
}

}// fin de main
}// fin de la clase

```

La ejecución muestra la siguiente salida:

```

10, CONTABILIDAD, SEVILLA
20, INVESTIGACIÓN, MADRID
30, VENTAS, BARCELONA
40, PRODUCCIÓN, BILBAO

```

Para poder probar el programa hemos de obtener el JAR que contiene el driver MySQL (en el ejemplo se ha utilizado **mysql-connector-java-5.1.18-bin.jar**) e incluirlo en el CLASSPATH o añadirlo a nuestro IDE, por ejemplo, en Eclipse pulsamos en el proyecto con el botón derecho del ratón y seleccionamos **Build Paths-> Add External Archives** para localizar el fichero JAR. Desde la URL <http://www.mysql.com/products/connector/> se puede descargar el conector. Si ejecutamos el programa desde la línea de comandos hemos de asegurarnos que el lugar donde se encuentra el fichero JAR se encuentre definido en la variable CLASSPATH.

Se puede observar que en nuestro programa Java, todos los *import* que necesitamos para manejar la base de datos están en el paquete **java.sql.***. También se ha incluido todo el programa en un **try-catch** ya que casi todos los métodos relativos a la base de datos pueden lanzar la excepción **SQLException**. La llamada al método **forName()** para cargar el driver puede lanzar la excepción **ClassNotFoundException** si este no se encuentra.

Cargar el driver:

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

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

Establecer la conexión:

A continuación se establece la conexión con la base de datos, el servidor MySQL debe estar arrancado, usamos la clase **DriverManager** con el método *getConnection()* de la siguiente manera:

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

La sintaxis del método *getConnection()* es la siguiente:

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

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

`jdbc:mysql://nombre_host:puerto/nombre_basedatos`

Donde

- **jdbc:mysql** indica que estamos utilizando un driver JDBC para MySQL.
- **nombre_host** indica el nombre del servidor donde está la base de datos. Aquí puede ponerse una IP o un nombre de máquina que esté en la red. Si especificamos *localhost* como nombre de servidor, estamos indicando que el servidor de base de datos se encuentra en la misma máquina en la que se ejecuta el programa Java.
- **puerto** es el puerto predeterminado para las bases de datos MySQL, por defecto es **3306**. Si no se pone se asume este valor.
- **nombre_basedatos** es el nombre de la base de datos a la que nos vamos a conectar y que debe existir en MySQL. En este caso el nombre es *ejemplo*.

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

El tercer parámetro es la clave del usuario, que en este caso también es *ejemplo*.

Ejecutar sentencias SQL:

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

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

El resultado nos lo devuelve como un **ResultSet**, que es un objeto similar a una lista en la que está el resultado de la consulta. Cada elemento de la lista es uno de los registros de la tabla *departamentos*. **ResultSet** no contiene todos los datos, sino que los va consiguiendo de la base

de datos según se van pidiendo. Por ello, el método `executeQuery()` puede tardar poco, aunque recorrer los elementos del **ResultSet** puede no ser tan rápido.

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

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

Los métodos `getInt()` y `getString()` nos van devolviendo los valores de los campos de dicho registro. Entre paréntesis se pone la posición de la columna en la tabla, es decir, la columna que deseamos. También se puede poner una cadena que indica el nombre de la columna (se hará referencia a estos métodos más adelante):

```
System.out.printf("%d, %s, %s %n",
                  resul.getInt("dept_no"),
                  resul.getString("dnombre"),
                  resul.getString("loc"));
```

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 <code>true</code> si el puntero se posiciona correctamente y <code>false</code> si no hay registros en el ResultSet
<code>boolean first()</code>	Mueve el puntero del objeto ResultSet al primer registro de la lista. Devuelve <code>true</code> si el puntero se posiciona correctamente y <code>false</code> si no hay registros
<code>boolean last()</code>	Mueve el puntero del objeto ResultSet al último registro de la lista. Devuelve <code>true</code> si el puntero se posiciona correctamente y <code>false</code> si no hay registros
<code>boolean previous()</code>	Mueve el puntero del objeto ResultSet al registro anterior de la lista. Devuelve <code>true</code> si el puntero se posiciona correctamente y <code>false</code> 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

El siguiente código muestra el número de filas recuperadas en la consulta y seguidamente muestra los datos de cada fila acompañada del número de fila:

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

resul.last(); //Nos situamos en el último registro
```

```

System.out.println ("NÚMERO DE FILAS: " + resul.getRow());
resul.beforeFirst(); //Nos situamos antes del primer registro
//Recorremos el resultado para visualizar cada fila
while (resul.next())
    System.out.printf("Fila %d: %d, %s, %s %n",
                      resul.getRow(),
                      resul.getInt(1),
                      resul.getString(2),
                      resul.getString(3));

```

La salida muestra la siguiente información:

```

NÚMERO DE FILAS: 4
Fila 1: 10, CONTABILIDAD, SEVILLA
Fila 2: 20, INVESTIGACIÓN, MADRID
Fila 3: 30, VENTAS, BARCELONA
Fila 4: 40, PRODUCCIÓN, BILBAO

```

Liberar recursos:

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

```

resul.close(); //Cerrar ResultSet
sentencia.close(); //Cerrar Statement
conexion.close(); //Cerrar conexión

```

ACTIVIDAD 2.6

Tomando como base el programa que ilustra los pasos de funcionamiento de JDBC obtén el APELLIDO, OFICIO y SALARIO de los empleados del departamento 10.

Realiza otro programa Java que visualice el APELLIDO del empleado con máximo salario, visualiza también su SALARIO y el nombre de su departamento.

2.6.4. Acceso a datos mediante el puente JDBC-ODBC

Hay productos (aunque muy pocos) para los que no hay controlador (o driver) JDBC, pero si hay un controlador ODBC. En estos casos se utiliza un puente denominado normalmente **JDBC-ODBC Bridge**. El puente JDBC-ODBC es un controlador JDBC que implementa operaciones JDBC traduciéndolas en operaciones ODBC, para ODBC aparece como una aplicación normal. El puente está implementado en Java y usa métodos nativos de Java para llamar a ODBC, se instala automáticamente con el JDK como el paquete **sun.jdbc.odbc** por lo que no es necesario añadir ningún JAR a nuestros proyectos para trabajar con él.

Por ejemplo, para acceder a una base de datos MySQL usando el puente JDBC-ODBC necesitaremos crear un origen de datos o DSN (*Data Source Name*). En Windows nos vamos al **Panel de Control-> Herramientas administrativas-> Orígenes de datos (ODBC)**. Pulsamos en el botón *Agregar*, a continuación seleccionamos el driver *MySQL ODBC 5.3 ANSI Driver* y pulsamos el botón *Finalizar*, véase Figura 2.13.

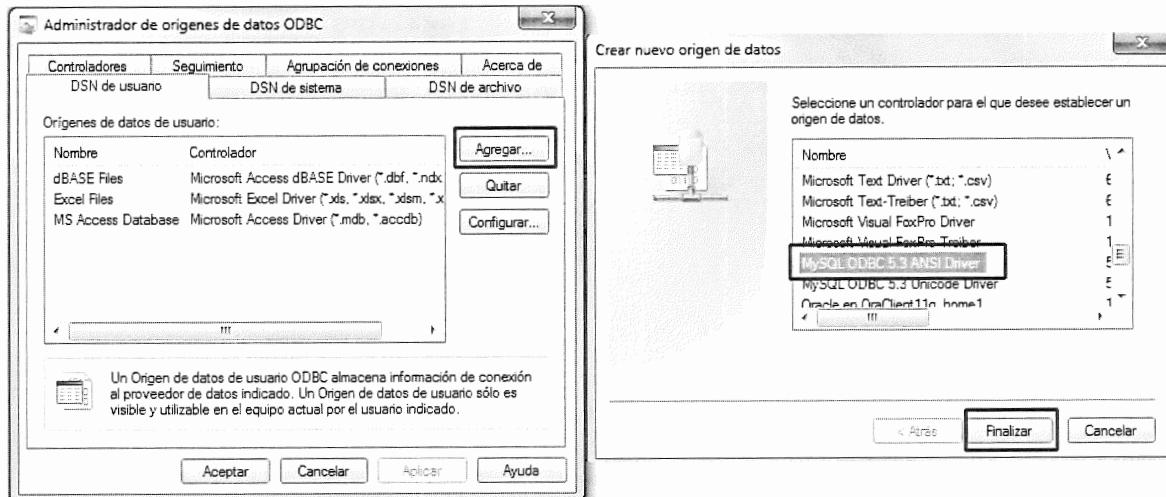


Figura 2.13. Crear un origen de datos ODBC.

En la siguiente pantalla hemos de dar un nombre al origen de datos en el campo **Data Source Name**, por ejemplo, escribimos el nombre: **Mysql-odbc**. El siguiente campo es opcional. En el campo **TCP/IP Server** escribimos el nombre de máquina (o la dirección IP) donde reside la base de datos, si reside en la misma máquina desde la que creamos el origen de datos escribimos **localhost**. En **Port** se escribe el puerto por el que escucha el servidor MySQL, en este caso es **3307**. Como nombre de usuario escribimos **root** y a continuación su password (se escribe un usuario que exista en la base de datos). De la lista **Database** elegimos un esquema de base de datos, en este caso se ha elegido **test**. El botón **Test** nos permite probar la conexión. Pulsamos el botón **OK**, véase Figura 2.14. A continuación se visualiza la pantalla inicial del *Administrador de Orígenes de datos ODBC*, con el nuevo origen creado, clic en el botón **Aceptar** para finalizar. Para cada esquema de base de datos es necesario crear un origen de datos.

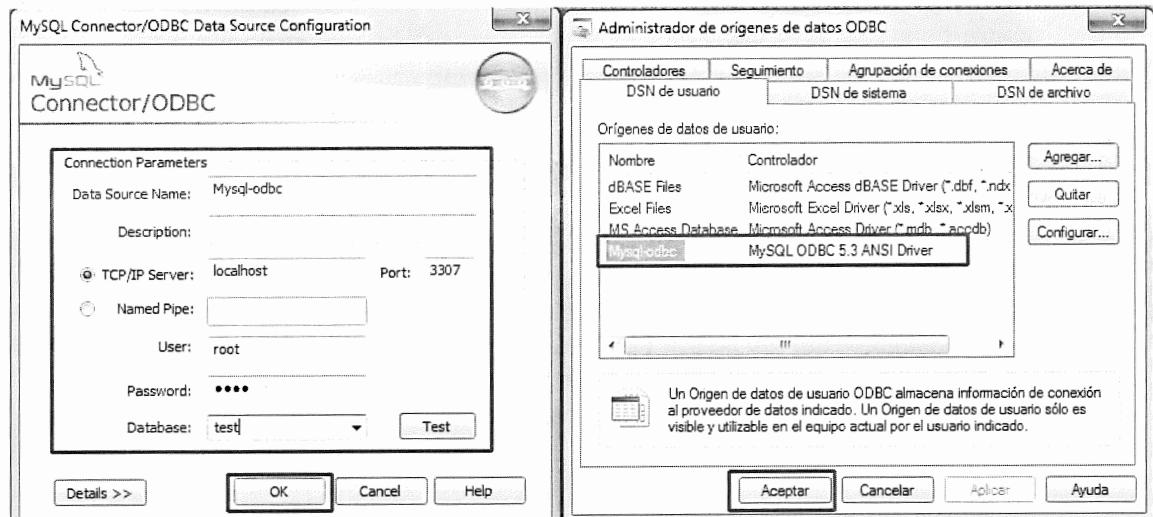


Figura 2.14. Origen de datos ODBC para acceder a MySQL desde Windows.

Puede ocurrir que no esté instalado el driver ODBC para conectar con bases de datos MySQL. En ese caso debemos descargarnos el driver e instalarlo. Accedemos a la Web

<http://www.mysql.com/products/connector/> y descargamos el driver ODBC desde la sección *MySQL Connectors (Connector/ODBC 5.3.6)*. Se elige la plataforma en la que se usará el driver, en este caso se ha descargado la versión para Windows de 32 bits. Puede ser un fichero con el nombre **mysql-connector-odbc-5.3.6-win32.msi**, lo ejecutamos y seguimos los pasos.

Ahora en nuestro programa Java anterior cambiamos 2 líneas, la carga del driver (**sun.jdbc.odbc.JdbcOdbcDriver**) y el establecimiento de la conexión en el que hay que escribir el nombre del origen de datos creado (**jdbc:odbc:MySql-odbc**):

```
//Cargar el driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// Establecemos la conexión con la BD
Connection conexion =
    DriverManager.getConnection("jdbc:odbc:MySql-odbc");
```

Para ejecutar este programa desde Linux (Ubuntu) hemos de instalar los paquetes *unixodbc*, *unixodbc-dev* y *libmyodbc* y crear el origen de datos **Mysql-odbc** como se hizo en el epígrafe de acceso a datos mediante ODBC. El código del programa Java sería el mismo.

IMPORTANTE!!

Estos ejemplos funcionarán en las versiones de Java inferiores a la 8. A partir de la versión **Java SE 8** no se incluye el puente JDBC-ODBC con el JDK. El puente JDBC-ODBC no es compatible con las versiones más recientes de la especificación JDBC.

ACTIVIDAD 2.7

Realiza la Actividad 2.6 utilizando el puente JDBC-ODBC.

2.7. ESTABLECIMIENTO DE CONEXIONES

Hemos visto en ejemplos anteriores cómo se realiza la conexión con una base de datos MySQL utilizando JDBC y el puente JDBC-ODBC, a continuación vamos a ver cómo conectarnos a través de JDBC a las bases de datos embebidas estudiadas anteriormente. Hemos de crear la base de datos *ejemplo* con las tablas *empleados* y *departamentos* y vamos a utilizar el mismo programa Java, solo cambiaremos la carga del driver y la conexión a la base de datos.

En Windows supongamos que la base de datos *ejemplo* la tenemos en las carpetas *D:\DB\SQLite*, *D:\DB\HSQLDB\ejemplo*, *D:\DB\H2* y *D:\DB\DERBY*. En Linux en las carpetas */home/usuario/DB/SQlite/*, */home/usuario/DB/HSQLDB/*, */home/usuario/DB/H2* y */home/usuario/DB/DERBY/*. Para realizar las pruebas necesitaremos tener el conector Java (fichero JAR) correspondiente para cada una de las bases de datos.

Conexión a SQLite

Para conectarnos a SQLite necesitamos la librería **sqlite-jdbc-3.8.11.2.jar** que se puede descargar desde la URL <https://bitbucket.org/xerial/sqlite-jdbc/downloads>. Partimos del programa Java inicial que recorre la tabla *departamentos* (su nombre es *Main.java*) de la base de datos *ejemplo* de MySQL. En el entorno gráfico que usemos para ejecutar el programa incluimos el fichero JAR o lo incluimos en el CLASSPATH si lo ejecutamos desde la línea de comandos.

En el programa Java cambiamos dos cosas: la carga del driver, en este caso se llama **org.sqlite.JDBC** y la conexión a la base de datos:

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

El ejemplo en Linux es similar, solo habría que cambiar en la conexión la carpeta donde se encuentra la base de datos: "jdbc:sqlite:/home/usuario/DB/SQLITE/ejemplo.db".

Conexión a Apache Derby

Para conectarnos a Apache Derby necesitamos la librería **derby.jar** (que se encuentra en la carpeta donde se instaló Derby: /db-derby-10.12.1.1-bin/lib). El driver para la conexión a la base de datos se llama: **org.apache.derby.jdbc.EmbeddedDriver**:

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

Conexión a HSQLDB

Para conectarnos a HSQLDB necesitamos la librería **hsqldb.jar** que se puede obtener de la carpeta *lib* obtenida al descomprimir el fichero **hsqldb-2.3.3.zip**. En este caso el driver se llama **org.hsqldb.jdbcDriver** y la conexión a la base de datos es la siguiente:

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

Conexión a H2

Para conectarnos a H2 necesitamos la librería **h2-1.4.191.jar** que se puede obtener de la carpeta *bin* en la que se encuentra al descomprimir el fichero **h2-2016-01-21.zip**. El driver se llama **org.h2.Driver** y la conexión a la base de datos es la siguiente:

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

En este caso es necesario incluir el nombre del usuario y la clave en la conexión. El nombre es "sa" y la clave se dejó en blanco cuando se creó la base de datos.

Conexión a Access

Para conectarnos a una base de datos Access necesitamos las siguientes librerías: **commons-lang-2.6.jar**, **commons-logging-1.1.1.jar**, **hsqldb.jar**, **jackcess-2.1.2.jar**, **ucanaccess-3.0.2.jar**, y **ucanload.jar**. Se pueden descargar de la URL: <http://ucanaccess.sourceforge.net/site.html>. Al acceder al sitio podremos descargar un fichero similar a **UCanAccess-3.0.4-src.zip** con ejemplos, o el fichero **UCanAccess-3.0.4-bin.zip** que contiene los JAR.

Para conectarnos a una base de datos Access escribiremos:

```
Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
Connection conn = DriverManager.getConnection
    ("jdbc:ucanaccess://mibasedatosaccess");
```

Por ejemplo, quiero conectarme a la base de datos **ejemplo.accdb**, que la tengo guardada en la carpeta raíz del proyecto, en la conexión escribiré lo siguiente:

```
Connection conn = DriverManager.getConnection
    ("jdbc:ucanaccess://./ejemplo.accdb");
```

Conexión a MySQL

Para conectarnos a MySQL necesitamos la librería **mysql-connector-java-5.1.38-bin.jar**, que podemos descargar desde la URL <http://dev.mysql.com/downloads/connector/j/>. Se descarga un fichero ZIP, y dentro de él se encuentra el JAR. El driver se llama **com.mysql.jdbc** y la conexión es la siguiente:

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

Conexión a Oracle

Para conectarlos mediante JDBC usamos el driver **JDBC Thin**. Se puede descargar desde la página Web de Oracle (es necesario comprobar antes la versión de la base de datos instalada), desde la dirección <http://www.oracle.com/technetwork/apps-tech/jdbc-112010-090769.html>. Para el ejemplo se ha descargado el driver **ojdbc6.jar**. Necesitamos saber el nombre de 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**. El driver se llama **oracle.jdbc.driver.OracleDriver**, y la conexión a la base de datos es la siguiente:

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

2.8. EJECUCIÓN DE SENTENCIAS DE DESCRIPCIÓN DE DATOS

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

La interfaz **DatabaseMetaData** proporciona información sobre la base de datos a través de múltiples métodos de los cuales es posible obtener gran cantidad de información. Muchos de estos métodos devuelven un **ResultSet**, algunos de los que veremos en los siguientes ejemplos son:

Método	Descripción
getTables()	Proporciona información sobre las tablas y vistas de la base de datos
getColumns()	Devuelve información sobre las columnas de una tabla
getPrimaryKeys()	Proporciona información sobre las columnas que forman la clave primaria de una tabla
getExportedKeys()	Devuelve información sobre las claves ajena que utilizan la clave primaria de una tabla
getImportedKeys()	Devuelve información sobre las claves ajenas existentes en una tabla
getProcedures()	Devuelve información sobre los procedimientos almacenados
Más información sobre métodos de DatabaseMetaData:	
https://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html	

El siguiente ejemplo conecta con la base de datos MySQL de nombre *ejemplo* y muestra información sobre el producto de base de datos, el driver, la URL para acceder a la base de datos, el nombre de usuario y las tablas y vistas del esquema actual (o de todos los esquemas dependiendo del sistema gestor de base de datos), un esquema se corresponde generalmente con un usuario de la base de datos; el método *getMetaData()* de la interfaz **Connection** devuelve un objeto **DataBaseMetaData** que contiene 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 conexión con la BD
            Connection conexion = DriverManager.getConnection
                ("jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");

            DatabaseMetaData dbmd = conexion.getMetaData();
            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.printf("Nombre : %s %n", nombre );
            System.out.printf("Driver : %s %n", driver );
            System.out.printf("URL : %s %n", url );
            System.out.printf("Usuario: %s %n", usuario );

            //Obtener información de las tablas y vistas que hay
            resul = dbmd.getTables(null, "ejemplo", null, 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.printf("%s - Catalogo: %s, Esquema: %s,
                           Nombre: %s %n", tipo, catalogo, esquema, tabla);
    }
    conexion.close(); //Cerrar conexión
}
catch (ClassNotFoundException cn) {cn.printStackTrace();}
catch (SQLException e) {e.printStackTrace();}
}//fin de main
}//fin de la clase

```

La ejecución del programa visualiza la siguiente información:

INFORMACIÓN SOBRE LA BASE DE DATOS:

```

=====
Nombre : MySQL
Driver : MySQL-AB JDBC Driver
URL   : jdbc:mysql://localhost/ ejemplo
Usuario: ejemplo@localhost
TABLE - Catalogo: ejemplo, Esquema: null, Nombre: departamentos
TABLE - Catalogo: ejemplo, Esquema: null, Nombre: empleados
VIEW  - Catalogo: ejemplo, Esquema: null, Nombre: vista

```

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

```

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

```

- Primer parámetro: catálogo de la base de datos. El método obtiene las tablas del catálogo indicado, al poner *null*, indicamos el catálogo actual.
- Segundo parámetro: esquema de la base de datos (nombre de usuario). Obtiene las tablas del esquema indicado, el valor *null* indica el esquema actual (o todos los esquemas, dependiendo del SGBD, como en Oracle).
- Tercer parámetro: es un patrón en el que se indica el nombre de las tablas que queremos que obtenga el método. Se puede utilizar el carácter guion bajo o porcentaje, por ejemplo, "de%" obtendría todas las tablas cuyo nombre empieza por "de".
- El cuarto parámetro es un array de *String*, en el que indicamos qué tipos de objetos queremos obtener, por ejemplo: *TABLE* (para tablas), *VIEW* (para vistas); al poner *null*, nos devolverá todos los tipos de objetos ya sean tablas o vistas. Los tipos válidos son: *TABLE*, *VIEW*, *SYSTEM TABLE*, *GLOBAL TEMPORARY*, *LOCAL TEMPORARY*, *ALIAS* y *SYNONYM*. El siguiente ejemplo nos devolvería las tablas y los sinónimos:

```

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

```

Cada fila de **ResultSet** que devuelve **getTables()** tiene información sobre una tabla. La descripción de cada columna tiene las siguientes columnas: **TABLE_CAT** (columna 1, el nombre del catálogo al que pertenece la tabla), **TABLE_SCHEM**, (columna 2, el nombre del esquema al que pertenece la tabla), **TABLE_NAME** (columna 3, el nombre de la tabla o vista), **TABLE_TYPE** (columna 4, el tipo TABLE o VIEW), **REMARKS** (columna 5, comentarios), **TYPE_CAT**, **TYPE_SCHEM**, **TYPE_NAME**, **SELF_REFERENCING_COL_NAME**, y **REF_GENERATION**. Para obtener estos resultados también podríamos haber puesto en el código anterior el nombre de la columna en lugar del número:

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

ACTIVIDAD 2.8

Prueba el programa anterior para visualizar información de las bases de datos Oracle y SQLite con las que estás trabajando en este tema.

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

- **getColumns()**: Devuelve un objeto **ResultSet** con información sobre las columnas de una tabla o tablas. La descripción de cada columna tiene las siguientes columnas: **TABLE_CAT**, **TABLE_SCHEM**, **TABLE_NAME**, **COLUMN_NAME**, **DATA_TYPE**, **TYPE_NAME**, **COLUMN_SIZE**, **BUFFER_LENGTH**, **DECIMAL_DIGITS**, **NUM_PREC_RADIX**, **NULLABLE**, **REMARKS**, **COLUMN_DEF**, **SQL_DATA_TYPE**, **SQL_DATETIME_SUB**, **CHAR_OCTET_LENGTH**, **ORDINAL_POSITION**, **IS_NULLABLE**, **SCOPE_CATALOG**, **SCOPE_SCHEMA**, **SCOPE_TABLE**, **SOURCE_DATA_TYPE**, **IS_AUTOINCREMENT** e **IS_GENERATEDCOLUMN**. Su sintaxis es:

```
public abstract ResultSet getColumns(
    String catalogo, String Esquema,
    String patronNombreDeTabla, String patronNombreDeColumna)
throws SQLException
```

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

```
System.out.println("COLUMNAS TABLA DEPARTAMENTOS:");
System.out.println("=====");
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);
}
```

Visualiza la siguiente información:

```
COLUMNAS TABLA DEPARTAMENTOS:
```

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

- **getPrimaryKeys()**: devuelve la lista de columnas que forman la clave primaria de la tabla especificada. La descripción de cada columna de la clave primaria tiene las siguientes columnas: *TABLE_CAT*, *TABLE_SCHEM*, *TABLE_NAME*, *COLUMN_NAME* y *KEY_SEQ*. La sintaxis es la siguiente:

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

El siguiente ejemplo muestra la clave primaria de la tabla *departamentos* (ejemplo en MySQL):

```
ResultSet pk = dbmd.getPrimaryKeys(null, "ejemplo", "departamentos");
String pkDep="", separador="";
while (pk.next()) {
    pkDep = pkDep + separador +
        pk.getString("COLUMN_NAME");//getString(4)
    separador="+";
}
System.out.println("Clave Primaria: " + pkDep);
```

- **getExportedKeys()**:devuelve la lista de todas las claves ajenas que utilizan la clave primaria de la tabla especificada. La descripción de cada columna de clave ajena tiene las siguientes columnas: *PKTABLE_CAT*, *PKTABLE_SCHEM*, *PKTABLE_NAME*, *PKCOLUMN_NAME*, *FKTABLE_CAT*, *FKTABLE_SCHEM*, *FKTABLE_NAME*, *FKCOLUMN_NAME*, *KEY_SEQ*, *UPDATE_RULE*, *DELETE_RULE*, *FK_NAME*, *PK_NAME* y *DEFERRABILITY*. La sintaxis es:

```
public abstract 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 la tabla *empleados*:

```
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);
}
```

Visualiza la siguiente información:

Tabla PK: departamentos, Clave Primaria: dept_no
 Tabla FK: empleados, Clave Ajena: dept_no

El método no devuelve nada si queremos ver las claves ajenas que referencian a la tabla *empleados*, `dbmd.getExportedKeys(null, "ejemplo", "empleados")`, ya que la tabla *empleados* no es referenciada por ninguna clave ajena.

A la hora de crear una tabla es recomendable definir las restricciones de clave ajena asignándolas un nombre, usando la cláusula *CONSTRAINT nombre FOREIGN KEY (col1, col2,...) REFERENCES tabla(col1,col2,...)*. De esta manera el método `getExportedKeys()` nos devolverá la información deseada.

- **`getImportedKeys()`**: devuelve la lista de claves ajenas existentes en la tabla indicada. Se utiliza igual que el método anterior, en este caso `dbmd.getImportedKeys(null, "ejemplo", "empleados")` devuelve la salida anterior, en cambio `dbmd.getImportedKeys(null, "ejemplo", "departamentos")` no devuelve nada ya que no tiene claves ajenas. La sintaxis es:

```
public abstract ResultSet getImportedKeys
    (String catalogo, String esquema, String tabla)
throws SQLException
```

- **`getProcedures()`**: devuelve la lista de procedimientos almacenados. Cada descripción de procedimiento tiene las siguientes columnas: *PROCEDURE_CAT* (columna 1), *PROCEDURE_SCHEM* (columna 2), *PROCEDURE_NAME* (columna 3), *REMARKS* (columna 7), *PROCEDURE_TYPE* (columna 8) y *SPECIFIC_NAME* (columna 9). La sintaxis es:

```
public abstract ResultSet getProcedures
    (String catalogo, esquema, String procedure) throws SQLException
```

Para probar el método `getProcedures()` creamos algunos procedimientos y funciones. Por ejemplo, creamos la función de nombre *SUMAR* que recibe dos números y devuelve la suma:

Ejemplo de función en Oracle:

```
CREATE OR REPLACE FUNCTION SUMAR (N1 NUMBER, N2 NUMBER)
RETURN NUMBER AS
BEGIN
    RETURN N1 + N2;
END SUMAR;
/
```

Para probarla escribimos: `SELECT SUMAR(2,22) FROM DUAL;`

Ejemplo de función en MySQL:

```
DELIMITER //
CREATE FUNCTION SUMAR (N1 INT, N2 INT) RETURNS INT
BEGIN
    RETURN N1 + N2;
END ;
//
```

Para probarla escribimos: `SELECT SUMAR(2,22)`

Ejemplo de creación de un procedimiento de nombre `SUBIDA` que sube 100 euros el salario de los empleados del departamento 30:

Ejemplo en Oracle:

```
CREATE OR REPLACE PROCEDURE SUBIDA AS
BEGIN
    UPDATE EMPLEADOS SET SALARIO = SALARIO +100 WHERE DEPT_NO=30;
    COMMIT;
END SUBIDA;
```

Ejemplo en MySQL:

```
DELIMITER //
CREATE PROCEDURE SUBIDA()
BEGIN
    UPDATE EMPLEADOS SET SALARIO = SALARIO + 100 WHERE DEPT_NO=30;
    COMMIT;
END;
//
```

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

```
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);
}
```

2.8.1. ResultSetMetaData

Se pueden obtener metadatos (datos sobre los datos) a partir de un objeto **ResultSet** mediante la interfaz **ResultSetMetaData**; es decir podemos obtener más información sobre los tipos y propiedades de las columnas de los objetos **ResultSet**, como por ejemplo, el número de columnas devueltas, el tipo, el nombre, etc. El siguiente ejemplo muestra el uso de la interfaz para conocer más información acerca de las columnas devueltas por esta consulta `SELECT * FROM departamentos`; en este caso al usar `*` en la `SELECT` desconocemos el nombre de las columnas devueltas.

Usaremos el método `getMetadata()` del objeto **ResultSet** que devuelve una referencia a un objeto **ResultSetMetaData** con el que se obtendrá la información acerca de las columnas devueltas:

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

```

Statement sentencia = conexion.createStatement();
ResultSet rs = sentencia
    .executeQuery("SELECT * FROM departamentos");

ResultSetMetaData rsmd = rs.getMetaData();

int nColumnas = rsmd.getColumnCount();
String nula;
System.out.printf("Número de columnas recuperadas: %d%n",
    nColumnas);
for (int i = 1; i <= nColumnas; i++) {
    System.out.printf("Columna %d: %n ", i);
    System.out.printf("  Nombre: %s %n  Tipo: %s %n ",
        rsmd.getColumnName(i), rsmd.getColumnTypeName(i));

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

    System.out.printf("  Puede ser nula?: %s %n ", nula);

    System.out.printf("  Máximo ancho de la columna: %d %n",
        rsmd getColumnDisplaySize(i));
} // for
sentencia.close();
rs.close();
conexion.close();

} catch (ClassNotFoundException cn) {
    cn.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
}
} // fin de main
}

```

Obtiene la siguiente información:

```

Número de columnas recuperadas: 3
Columna 1:
Nombre: dept_no
Tipo: TINYINT
Puede ser nula?: NO
Máximo ancho de la columna: 2
Columna 2:
Nombre: dnombre
Tipo: VARCHAR
Puede ser nula?: SI
Máximo ancho de la columna: 15
Columna 3:
Nombre: loc
Tipo: VARCHAR

```

Puede ser nula?: SI
Máximo ancho de la columna: 15

Los métodos usados son los siguientes:

Método	Descripción
<code>int getColumnCount ()</code>	Devuelve el número de columnas devueltas por la consulta
<code>String getColumnName (int índiceColumna)</code>	Devuelve el nombre de la columna cuya posición se indica en <code>índiceColumna</code>
<code>String getColumnTypeName (int índiceColumna)</code>	Devuelve el nombre del tipo de dato específico del sistema de bases de datos que contiene la columna indicada en <code>índiceColumna</code>
<code>int isNullable (int índiceColumna)</code>	Devuelve 0 si la columna no puede contener valores nulos
<code>int getColumnDisplaySize (int índiceColumna)</code>	Devuelve el máximo ancho en caracteres de la columna indicada en <code>índiceColumna</code>
<i>Más información sobre métodos de ResultSetMetaData:</i>	
https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSetMetaData.html	

ACTIVIDAD 2.9

Visualiza información sobre las columnas de la tabla *empleados*.

2.9. EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS

En ejemplos anteriores vimos como se podían ejecutar sentencias SQL mediante la interfaz **Statement** (sentencia), esta proporciona métodos para ejecutar sentencias SQL y obtener los resultados. Como **Statement** es una interfaz no se pueden crear objetos directamente, en su lugar los objetos se obtienen con una llamada al método `createStatement()` de un objeto **Connection** válido:

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

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

- **ResultSet executeQuery(String):** se utiliza para sentencias SQL que recuperan datos de un único objeto **ResultSet**, se utiliza para las sentencias SELECT.
- **int executeUpdate(String):** se utiliza para sentencias que no devuelven un **ResultSet** como son las sentencias de manipulación de datos (DML): INSERT, UPDATE y DELETE; y las sentencias de definición de datos(DDL): CREATE, DROP y ALTER. El método devuelve un entero indicando el número de filas que se vieron afectadas y en el caso de las sentencias DDL devuelve el valor 0.

- **boolean execute(String):** se puede utilizar 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. En este ejemplo **execute()** 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 orden SQL por esta otra: *String sql= "UPDATE departamentos SET dnombre = LOWER(dnombre)"*; entonces la variable *valor* será *false* y la salida del programa será diferente.

A través de un objeto **ResultSet** se puede acceder al valor de cualquier columna de la fila actual por nombre o por posición, también se puede obtener información sobre las columnas como el número de columnas o su tipo; en ejemplos anteriores vimos como se podía averiguar el número de columnas devueltas por una orden SELECT usando el método **getMetadata()** de un objeto **ResultSet**. Algunos de los métodos **getXXX()** para la obtención de valores son los siguientes:

Método	Tipo Java devuelto
getString(int númerodecolumna) getString(String nombredecolumna)	String
getBoolean(int númerodecolumna) getBoolean(String nombredecolumna)	boolean
getByte(int númerodecolumna) getByte(String nombredecolumna)	byte
getShort(int númerodecolumna) getShort(String columna)	short
getInt(int númerodecolumna) getInt(String nombredecolumna)	int
getLong(int númerodecolumna) getLong(String nombredecolumna)	long
getFloat(int númerodecolumna) getFloat(String nombredecolumna)	float
getDouble(int númerodecolumna) getDouble(String nombredecolumna)	double
getBytes(int númerodecolumna) getBytes(String nombredecolumna)	byte[]
getDate(int númerodecolumna) getDate(String nombredecolumna)	Date
getTime(int númerodecolumna) getTime(String nombredecolumna)	Time
getTimestamp(int númerodecolumna) getTimestamp(String nombredecolumna)	Timestamp
Más información sobre métodos de ResultSet:	
https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html	

El siguiente ejemplo inserta un departamento en la tabla *departamentos*, los datos del nuevo departamento se introducen a través de los argumentos de *main()*. El primer parámetro es el departamento, el siguiente el nombre y el tercero la localidad. Antes de ejecutar la orden INSERT construimos la sentencia SQL en un *String*, las cadenas de caracteres (en este caso el nombre del departamento y la localidad) deben ir encerradas entre comillas simples:

```
import java.sql.*;
public class InsertarDep {
    public static void main(String[] args) {
        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");

            //recuperar argumentos de main
            String dep = args[0];           // num. departamento
            String dnombre = args[1];       // nombre
            String loc = args[2];          // localidad

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

```

        System.out.println(sql);

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

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

    } catch (ClassNotFoundException cn) {
        cn.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }

} //fin de main
}//fin de la clase

```

La ejecución desde la línea de comandos y suponiendo que el conector MySQL está en el CLASSPATH visualiza la siguiente información en la que se inserta el departamento 15 de nombre INFORMÁTICA y localidad MADRID:

```

java InsertarDep 15 INFORMÁTICA MADRID
INSERT INTO departamentos VALUES (15, 'INFORMÁTICA', 'MADRID')
Filas afectadas: 1

```

El siguiente código sube el salario a los empleados de un departamento (supongamos que el programa se llama *ModificarSalario.java*). El número de departamento y la subida se reciben a través de los argumentos de *main()*:

```

//recuperar parametros de main
String dep = args[0];      // num. departamento
String subida = args[1];   // subida

//construir orden UPDATE
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);

```

La ejecución desde la línea de comandos y suponiendo que el conector MySQL está en el CLASSPATH visualiza la siguiente información en la que se sube 100 euros a los empleados del departamento 10:

```

java ModificarSalario 10 100
UPDATE empleados SET salario = salario + 100 WHERE dept_no = 10
Empleados modificados: 3

```

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:

```
//construir orden CREATE VIEW
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);
```

La ejecución muestra la siguiente información:

```
CREATE OR REPLACE VIEW totales (dep, dnombre, nemp, media) AS SELECT
d.dept_no, dnombre, COUNT(emp_no), AVG(salario) FROM departamentos d
LEFT JOIN empleados e ON e.dept_no = d.dept_no GROUP BY d.dept_no,
dnombre
Resultado de la ejecución: 0
```

ACTIVIDAD 2.10

Crea un programa Java que inserte un empleado en la tabla *empleados*, el programa recibe desde la línea de argumentos de *main()* los valores a insertar. Los argumentos que recibe son los siguientes: *EMP_NO*, *APELLIDO*, *OFICIO*, *DIR*, *SALARIO*, *COMISIÓN*, *DEPT_NO*. Antes de insertar se deben realizar las siguientes comprobaciones:

- que el departamento exista en la tabla *departamentos*, si no existe no se inserta.
- que el número del empleado no exista, si existe no se inserta.
- que el salario sea > que 0, si es <= 0 no se inserta.
- que el director (DIR, es el número de empleado de su director) exista en la tabla *empleados*, si no existe no se inserta.
- El APELLIDO y el OFICIO no pueden ser nulos.
- La fecha de alta del empleado es la fecha actual.

Cuando se inserte la fila visualizar mensaje y si no se inserta visualizar el motivo (departamento inexistente, número de empleado duplicado, director inexistente, etc.)

2.9.1. Ejecución de Scripts

Algunas bases de datos admiten la ejecución de varias sentencias DDL y/o DML en una misma cadena. Por ejemplo, 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 = 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 las admiten está MySQL. En el siguiente ejemplo se muestra la ejecución del siguiente script almacenado en el fichero *./script/scriptmysql.sql*, dentro del proyecto:

```
SET FOREIGN_KEY_CHECKS = 0;
drop table if EXISTS notas;
drop table if EXISTS alumnos;
drop table if EXISTS asignaturas;
CREATE TABLE IF NOT EXISTS ALUMNOS
( DNI VARCHAR(10) NOT NULL primary key,
  APENOM VARCHAR(30),
  DIREC VARCHAR(30),
  POBLA  VARCHAR(15),
  TELEF  VARCHAR(10) ) ;

CREATE TABLE IF NOT EXISTS ASIGNATURAS
( COD int NOT NULL primary key,
  NOMBRE VARCHAR(25)) ;

CREATE TABLE IF NOT EXISTS NOTAS
( DNI VARCHAR(10) NOT NULL ,
  COD int NOT NULL ,
  NOTA int,
  primary key(DNI,COD)) ;

/* Create Foreign Keys */
ALTER TABLE NOTAS
  ADD CONSTRAINT FKNOTASALUM FOREIGN KEY (DNI)
    REFERENCES ALUMNOS (DNI)          ON UPDATE CASCADE
    ON DELETE RESTRICT;

ALTER TABLE NOTAS
  ADD CONSTRAINT FKNOTASASIG FOREIGN KEY (COD)
    REFERENCES ASIGNATURAS (COD)
    ON UPDATE CASCADE          ON DELETE RESTRICT;

/* Rellenar Datos */
INSERT IGNORE INTO ASIGNATURAS VALUES (1,'Prog. Leng. Estr.');
INSERT IGNORE INTO ASIGNATURAS VALUES (2,'Sist. Informáticos');
INSERT IGNORE INTO ASIGNATURAS VALUES (3,'Análisis');
INSERT IGNORE INTO ASIGNATURAS VALUES (4,'FOL');
INSERT IGNORE INTO ASIGNATURAS VALUES (5,'RET');

INSERT IGNORE INTO ALUMNOS VALUES ('12344345','Alcalde García, Elena',
'C/Las Matas, 24','Madrid','917766545');

INSERT IGNORE INTO ALUMNOS VALUES('4448242','Cerrato Vela, Luis',
'C/Mina 28 - 3A', 'Madrid','916566545');

INSERT IGNORE INTO ALUMNOS VALUES('56882942','Díaz Fernández, María',
'C/Luis Vives 25', 'Móstoles','915577545');
```

```

INSERT IGNORE INTO NOTAS VALUES('12344345', 1,6);
INSERT IGNORE INTO NOTAS VALUES('12344345', 2,5);

INSERT IGNORE INTO NOTAS VALUES('4448242', 4,6);
INSERT IGNORE INTO NOTAS VALUES('4448242', 5,8);

INSERT IGNORE INTO NOTAS VALUES('56882942', 1,8);
INSERT IGNORE INTO NOTAS VALUES('56882942', 3,7);

commit;

```

El método lee el fichero de texto que contiene el script línea a línea, y lo almacena en un *StringBuilder*. Que después lo convierte a *String* para ejecutarlo con *executeUpdate()*:

```

public static void ejecutarScriptMySQL() {
    File scriptFile = new File("./script/scriptmysql.sql");
    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 HAY FILE: " + e.getMessage());
    }
    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 " +
                           e.getMessage());
    }
    String consulta = stringBuilder.toString();
    System.out.println(consulta);
    try {
        Class.forName("com.mysql.jdbc.Driver");
    } catch (ClassNotFoundException e) {
        System.out.println("ERROR Driver:" + e.getMessage());
    }
    try {
        Connection connmysql = DriverManager.getConnection(
            "jdbc:mysql://localhost/ejemplo?allowMultiQueries=true",
            "ejemplo", "ejemplo");
        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());
    }
}

```

2.9.2. Sentencias preparadas

En los ejemplos anteriores hemos creado sentencias SQL a partir de cadenas de caracteres en las que íbamos concatenando los datos necesarios para construir la sentencia completa. La interfaz **PreparedStatement** nos va a permitir construir una cadena de caracteres SQL con *placeholder* o marcadores de posición, que representarán los datos que serán asignados más tarde, el *placeholder* se representa mediante el símbolo interrogación (?). Por ejemplo, la orden INSERT para insertar un departamento se representa así:

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

Cada *placeholder* tiene un índice, el 1 correspondería al primero que se encuentre en la cadena, el 2 al segundo y así sucesivamente. Solo se pueden utilizar para ocupar el sitio de los datos en la cadena SQL, no se pueden usar para representar una columna o un nombre de una tabla, por ejemplo *FROM ?* sería incorrecto. Antes de ejecutar un **PreparedStatement** es necesario asignar los datos para que cuando se ejecute la base de datos asigne variables de unión con estos datos y ejecute la orden SQL. Los objetos **PreparedStatement** se pueden preparar o precompilar una sola vez y ejecutar las veces que queramos asignando diferentes valores a los marcadores de posición, en cambio en los objetos **Statement**, la sentencia SQL se suministra en el momento de ejecutar la sentencia.

Los métodos de **PreparedStatement** tienen los mismos nombres que en **Statement**: *executeQuery()*, *executeUpdate()* y *execute()* pero no se necesita enviar la cadena de caracteres con la orden SQL en la llamada ya que lo hace el método *prepareStatement(String)*. El ejemplo anterior en el que se inserta una fila en la tabla *departamentos* quedaría así:

```
//construir orden INSERT
String sql= "INSERT INTO departamentos VALUES(?, ?, ?)";
PreparedStatement sentencia = conexion.prepareStatement(sql);

sentencia.setInt(1, Integer.parseInt(dep)); // num departamento
sentencia.setString(2, dnombre);           // nombre
sentencia.setString(3, loc);               // localidad

int filas = sentencia.executeUpdate(); // filas afectadas
```

Para asignar valor a cada uno de los marcadores de posición se utilizan los métodos *setXXX()*. La sintaxis es la siguiente:

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

Donde, se asigna el valor indicado en *tipoJava* al parámetro cuyo índice coincide con *indiceDelParametro*, que es transformado por el controlador JDBC en un tipo SQL correspondiente para pasarlo a la base de datos. Los métodos *setXXX()* son los siguientes:

Método	Tipo SQL
void setString(int índice, String valor)	VARCHAR
void setBoolean(int índice, boolean valor)	BIT
void setByte(int índice, byte valor)	TINYINT
void setShort(int índice, short valor)	SMALLINT
void setInt(int índice, int valor)	INTEGER
void setLong(int índice, long valor)	BIGINT
void setFloat(int índice, float valor)	FLOAT
void setDouble(int índice, double valor)	DOUBLE
void setBytes(int índice, byte[] valor)	VARBINARY
void setDate(int índice, Date valor)	DATE
void setTime(int índice, Time valor)	TIME
Más información sobre métodos de PreparedStatement:	
https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html	

Para asignar valores NULL a un parámetro se usa el método *setNull()*, el formato es:

```
void setNull(int índice, int tipoSQL)
```

Donde *tipoSQL* es una constante que se define en la librería **java.sql.Types**. Son las siguientes: *ARRAY*, *BIGINT*, *BINARY*, *BIT*, *BLOB*, *BOOLEAN*, *CHAR*, *CLOB*, *DATALINK*, *DATE*, *DECIMAL*, *DISTINCT*, *DOUBLE*, *FLOAT*, *INTEGER*, *JAVA_OBJECT*, *LONGNVARCHAR*, *LONGVARBINARY*, *LONGVARCHAR*, *NCHAR*, *NCLOB*, *NULL*, *NUMERIC*, *NVARCHAR*, *OTHER*, *REAL*, *REF*, *REF_CURSOR*, *ROWID*, *SMALLINT*, *SQLXML*, *STRUCT*, *TIME*, *TIME_WITH_TIMEZONE*, *TIMESTAMP*, *TIMESTAMP_WITH_TIMEZONE*, *TINYINT*, *VARBINARY*, *VARCHAR*.

El ejemplo en el que se modifica el salario de los empleados quedaría así:

```
//construir orden UPDATE
String sql="UPDATE empleados SET salario=salario + ? WHERE dept_no=?";
PreparedStatement sentencia = conexion.prepareStatement(sql);

sentencia.setInt(2, Integer.parseInt(dep));           // num departamento
sentencia.setFloat(1, Float.parseFloat(subida)); // subida

int filas = sentencia.executeUpdate(); // filas afectadas
```

En los ejemplos anteriores se ha usado *Integer.parseInt(dep)* y *Float.parseFloat(subida)* para convertir la cadena *dep* a un tipo entero y la cadena *subida* a un tipo float.

También se puede utilizar esta interfaz con la orden *SELECT*. El siguiente ejemplo muestra el APELLIDO y SALARIO de los empleados de un departamento y un oficio concreto, el departamento y oficio se introducen desde los argumentos de *main()* al ejecutar el programa desde la línea de comandos:

```
//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 desde la línea de comandos y suponiendo que el conector MySQL está en el CLASSPATH visualiza la siguiente información en la que se visualizan los vendedores del departamento 30:

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

ACTIVIDAD 2.11

Utiliza la interfaz **PreparedStatement** para visualizar el APELLIDO, SALARIO y OFICIO de los empleados de un departamento cuyo valor se recibe desde los argumentos de *main()*. Visualiza también el nombre del departamento.

Visualiza al final el salario medio y el número de empleados del departamento. Si el departamento no existe en la tabla *departamentos* visualiza un mensaje indicándolo. Utiliza la clase **DecimalFormat** para dar formato al salario. Ejemplo:

```
DecimalFormat formato = new DecimalFormat("##,##0.00");  
String valorFormateado = formato.format(resul.getFloat(1));
```

2.10. EJECUCIÓN DE PROCEDIMIENTOS

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

El siguiente ejemplo muestra un procedimiento de nombre *subida_sal* que sube el salario a los empleados de un departamento, el procedimiento recibe dos parámetros de entrada que son el número de departamento (*d*) y la subida (*subida*):

Procedimiento en ORACLE:

```
CREATE OR REPLACE PROCEDURE subida_sal(d NUMBER, subida NUMBER) AS
BEGIN
    UPDATE empleados SET salario = salario + subida WHERE dept_no = d;
    COMMIT;
END;
/
```

Procedimiento en MySQL:

```
delimiter //
CREATE PROCEDURE subida_sal(d INT, subida INT)
BEGIN
    UPDATE empleados SET salario = salario + subida WHERE dept_no = d;
    COMMIT;
END;
//
```

El siguiente ejemplo crea una función (en ORACLE) de nombre *nombre_dep* con dos parámetros, el primero es de entrada y recibe un número de departamento, el segundo es de salida, se utilizará para guardar la localidad del departamento; la función devuelve el nombre del departamento; si el departamento no existe devuelve como nombre “INEXISTENTE”:

```
CREATE OR REPLACE FUNCTION nombre_dep
    (d NUMBER, locali OUT VARCHAR2) RETURN VARCHAR2 AS
    nom VARCHAR2(15);
BEGIN
    SELECT dnombre, loc INTO nom, locali FROM departamentos
    WHERE dept_no = d;
    RETURN nom;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        nom := 'INEXISTENTE';
        RETURN nom;
END;
/
```

El siguiente ejemplo crea una función (en MySQL) de nombre *nombre_dep*, recibe un número de departamento (parámetro de entrada) y devuelve el nombre si existe; si no existe devuelve como nombre “INEXISTENTE”:

```
DELIMITER //
CREATE FUNCTION nombre_dep(d int) RETURNS VARCHAR(15)
BEGIN
    DECLARE nom VARCHAR(15);
    SET nom = 'INEXISTENTE';
    SELECT dnombre INTO nom FROM departamentos
    WHERE dept_no=d;
```

```

    RETURN nom;
END;
//
```

Para ejecutarlo desde MySQL escribimos: `SELECT nombre_dep(10);`

A continuación se muestra un procedimiento (en MySQL) que recibe un número de departamento y devuelve en forma de parámetros de salida el nombre y la localidad (las funciones no pueden usar parámetros OUT), se asigna un valor inicial a los parámetros de salida por si el departamento no existe:

```

DELIMITER //
CREATE PROCEDURE datos_dep
    (d int, OUT nom VARCHAR(15), OUT locali VARCHAR(15))
BEGIN
    SET locali = 'INEXISTENTE';
    SET nom = 'INEXISTENTE';
    SELECT dnombre, loc INTO nom, locali FROM departamentos
    WHERE dept_no=d;
END;
//
```

Para ejecutarlo desde MySQL escribimos las siguientes sentencias:

```

CALL datos_dep(10, @nom, @locali);
SELECT @nom;
SELECT @locali;
```

La interfaz **CallableStatement** permite que se pueda llamar desde Java a los procedimientos almacenados. Para crear un objeto se llama al método **prepareCall(String)** del objeto **Connection**. En el *String* se declara la llamada al procedimiento o función, tiene dos formatos, uno incluye el parámetro de resultado (usado para las funciones) y el otro no:

```

{? = call <nombre_procedure>[(<arg1>,<arg2>, ...)]}
{call <nombre_procedure>[(<arg1>,<arg2>, ...)]}
```

Si los procedimientos y funciones incluyen parámetros de entrada o de salida es necesario indicarlos en forma de marcadores de posición. La referencia a los parámetros es secuencial, por número, el primer parámetro es el 1, el siguiente el 2, etc. El parámetro de resultado y los parámetros de salida deben ser registrados antes de realizar la llamada. El siguiente ejemplo declara la llamada al procedimiento *subida_sal* que tiene dos parámetros de entrada, se usan los marcadores de posición (?) para indicarlo:

```

String sql= "{ call subida_sal (?, ?) } ";
CallableStatement llamada = conexion.prepareCall(sql);
```

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

- **{ call nombre_procedimiento }**: para un procedimiento almacenado sin parámetros.

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

En el siguiente ejemplo se realiza una llamada al procedimiento *subida_sal* (de MySQL); los valores de los parámetros se asignan a partir de los argumentos de *main()*:

```
import java.sql.*;
public class ProcSubida {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            Connection conexion = DriverManager.getConnection
                ("jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");

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

            //construir orden de llamada
            String sql= "{ call subida_sal (?, ?) } ";

            //Preparar la llamada
            CallableStatement llamada = conexion.prepareCall(sql);

            //Dar valor a los argumentos
            llamada.setInt(1,Integer.parseInt(dep));           //primero
            llamada.setFloat(2,Float.parseFloat(subida)); // segundo

            //Ejecutar el procedimiento
            llamada.executeUpdate();
            System.out.println ("Subida realizada....");

            llamada.close();
            conexion.close();
        }
        catch (ClassNotFoundException cn) { cn.printStackTrace(); }
        catch (SQLException e) { e.printStackTrace(); }
    }
} //fin de main
}//fin de la clase
```

La ejecución desde la línea de comandos y suponiendo que el conector MySQL está en el CLASSPATH visualiza la siguiente información:

```
java ProcSubida 30 200
Subida realizada....
```

En MySQL al ejecutarlo puede que se muestre el siguiente error: *java.sql.SQLException: User does not have access to metadata required to determine stored procedure parameter types* ... si el usuario no tiene permisos para ejecutar procedimientos. En este caso debemos darle el privilegio SELECT sobre la tabla de sistema **mysql.proc** que contiene la información sobre todos los procedimientos almacenados en la base de datos; se ejecutaría la siguiente orden desde la línea de comandos de MySQL o desde el entorno gráfico que usemos: *GRANT SELECT ON mysql.proc TO 'ejemplo'@'localhost'*;

Cuando un procedimiento o función tiene parámetros de salida (OUT) deben ser registrados antes de que la llamada tenga lugar, si no se registra se producirá un error. El método que se utilizará es: *registerOutParameter(int indice, int tipoSQL)*, el primer parámetro es la posición y el siguiente es una constante definida en la clase **java.sql.Types**. Estas constantes se nombraron en el apartado anterior. Por ejemplo, si el segundo parámetro de un procedimiento es OUT y de tipo VARCHAR en la base de datos en la llamada al método escribimos lo siguiente:

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

Una vez ejecutada la llamada al procedimiento, los valores de los parámetros OUT e INOUT se obtienen con los métodos *getXXX()* similares a los utilizados para obtener los valores de las columnas en un **ResultSet**. El siguiente ejemplo ejecuta el procedimiento *nombre_dep* (de Oracle); desde los argumentos de *main()* se recibe el número de departamento cuyos datos se visualizarán:

```
import java.sql.*;
public class FuncNombre {
    public static void main(String[] args) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection conexion = DriverManager.getConnection
                ("jdbc:oracle:thin:@localhost:1521:XE", "ejemplo", "ejemplo");

            //recuperar parametro de main
            String dep = args[0]; //departamento

            //Construir orden de llamada
            String sql = "{ ? = call nombre_dep (?, ?) } "; // ORACLE

            //Preparar la llamada
            CallableStatement llamada = conexion.prepareCall(sql);

            //registrar parámetro de resultado
            llamada.registerOutParameter(1, Types.VARCHAR); //valor devuelto

            llamada.setInt(2, Integer.parseInt(dep));           //param de entrada

            //Registrar parámetro de salida
            llamada.registerOutParameter(3, Types.VARCHAR); //parámetro OUT

            //Ejecutar el procedimiento
            llamada.executeUpdate();
            System.out.printf("Nombre Dep: %s, Localidad: %s %n",
                llamada.getString(1), llamada.getString(3));
            llamada.close();
            conexion.close();
        }
    }
}
```

```

        }
        catch (ClassNotFoundException cn) { cn.printStackTrace(); }
        catch (SQLException e) { e.printStackTrace(); }
    } // fin de main
} // fin de la clase

```

La ejecución desde la línea de comandos y suponiendo que el conector ORACLE está en el CLASSPATH visualiza la siguiente información:

```

java FuncNombre 10
Nombre Dep: CONTABILIDAD, Localidad: SEVILLA

java FuncNombre 120
Nombre Dep: INEXISTENTE, Localidad: null

```

ACTIVIDAD 2.12

Crea una función en Oracle, que reciba un número de departamento y devuelva el salario medio de los empleados de ese departamento y como parámetro de salida el número de empleados. Si el departamento no existe debe devolver como salario medio el valor -1 y el número de empleados será 0. Si existe y no tiene empleados debe devolver 0. Realiza después un programa Java que use dicha función. El programa recorrerá la tabla *departamentos* y mostrará los datos del departamento, incluyendo el número de empleados y el salario medio.

Realiza un procedimiento en MySQL que funcione de forma similar a la función en Oracle, es decir, debe recibir un número de departamento y como parámetros de salida debe devolver el número de empleados y el salario medio. Realiza después un programa Java para usar dicho procedimiento, igual que antes el programa recorrerá la tabla *departamentos* y mostrará los datos del departamento, incluyendo el número de empleados y el salario medio.

La función y el procedimiento se crearán desde un programa Java.

2.11. INFORMES CON JASPERREPORTS

JasperReports es una herramienta para generar informes. De código abierto y licencia GPL (*Licencia Pública General*). Genera informes en distintos formatos: PDF, HTML, XLS, RTF, ODT, CSV, TXT y XML. Está escrita en Java y su principal objetivo es ayudar a crear documentos preparados para la impresión de una forma simple y flexible.

La página para descargarse la herramienta es: <http://community.jaspersoft.com/download>. Hay distintas versiones *Server*, *Library* y *Studio*. Y para saber más de JasperReports podemos acceder a <http://community.jaspersoft.com/wiki/jasperreports-library-tutorial>.

JasperReports organiza los datos recuperados de una fuente de datos de acuerdo con un informe de trazado definido en un fichero **JRXML**. Con el fin de llenar el informe con los datos, este informe de diseño (el fichero JRXML) debe ser compilado previamente.

En este capítulo solo nos interesa **generar informes utilizando la plantilla definida** en el fichero **JRXML**. Para nuestros proyectos necesitaremos añadir los JAR de JasperReports y el JAR **tools.jar** del JDK. En estas pruebas se utiliza la versión 6.2.0 de JasperReports. Así pues, añadiremos a los proyectos los siguientes JAR: *commons-beanutils-1.9.0.jar*, *commons-codec-1.5.jar*, *commons-collections-3.2.1.jar*, *commons-digester-2.1.jar*, *commons-javaflow-*

20060411.jar, commons-logging-1.1.1.jar, iText-2.1.7.js4.jar, jackson-annotations-2.1.4.jar, jackson-core-2.1.4.jar, jackson-databind-2.1.4.jar, jasperreports-6.2.0.jar, jasperreports-fonts-6.2.0.jar, jasperreports-javafow-6.2.0.jar y tools.jar.

Ejemplo1: creamos un proyecto para generar un informe de datos de departamentos, de la base de datos MySQL. Utilizaremos las siguientes clases:

- `net.sf.jasperreports.engine.JasperCompileManager`,
- `net.sf.jasperreports.engine.JasperFillManager`,
- `net.sf.jasperreports.engine.JasperPrintManager`
- `net.sf.jasperreports.engine.JasperExportManager`

Para crear un informe con JasperReports seguiremos los siguientes pasos:

1. Generar el fichero `.jrxml`, será la plantilla en la que configuraremos como se desea el informe. En este fichero indicaremos los parámetros del informe, la consulta (SELECT) que se va a realizar, los datos que se van a visualizar, y además, se describirán cómo van a ser las líneas de cabecera, de detalle y de pies.
2. Ya dentro del proyecto Java, se compilará la plantilla, y obtendremos un objeto `JasperReport` de la siguiente manera:

```
JasperReport NombreJasperReport =
    JasperCompileManager.compileReport(MIPLANTILLA.JRXML);
```

3. Para llenar de datos el informe se utiliza el método `fillReport()` de la clase `JasperFillManager` (`JasperFillManager.fillReport()`). Esto generará un fichero `.jrprint`. También se necesita el nombre del objeto `JasperReport`, creado anteriormente, los parámetros del informe y la conexión a la BD.

```
JasperPrint MiInforme = JasperFillManager.fillReport
    (NombreJasperReport, ParámetrosDelInforme, conexiónalaBD);
```

Los parámetros tienen que crearse y almacenarse en un `HashMap` (de `java.util`), ese `Map` se utiliza para crear el `JasperPrint` añadiendo parámetros. Por ejemplo, declaro 3 parámetros *titulo*, *autor* y *fecha* y los guardo en *params*:

```
Map<String, Object> params = new HashMap<String, Object>();
params.put("titulo", "LISTADO DE DEPARTAMENTOS.");
params.put("autor", "ARM");
params.put("fecha", (new java.util.Date()).toString());
```

4. Y finalmente podremos exportar el fichero `JasperPrint`, en el ejemplo *MiInforme*, generado anteriormente al formato que se deseé. Por ejemplo:

- Para visualizar en un visor, por consola, el informe generado escribiremos:

```
JasperViewer.viewReport(MiInforme);
```

Si se desea cerrar el visor sin cerrar la aplicación (por ejemplo, en una aplicación con ventanas) añadiremos *false*, es decir:

```
JasperViewer.viewReport(MiInforme, false);
```

- Para generar el informe en HTML:

```
JasperExportManager.exportReportToHtmlFile
    (MiInforme, nombreFicheroHTML);
```

- Para generar el informe en PDF:

```
JasperExportManager.exportReportToPdfFile
    (MiInforme, nombreFicheroPDF);
```

- Para generar la salida en un documento XML:

```
//Convertir a XML,
//False es para indicar que no hay imágenes
//(isEmbeddingImages)
JasperExportManager.exportReportToXmlFile
    (MiInforme, nombreFicheroXML, false);
```

El código Java será el siguiente, la plantilla *jrxml* se guarda en la carpeta *plantillas*, y los informes de salida en la carpeta *informes*.

```
import java.util.Map;

import com.mysql.jdbc.exceptions.jdbc4.CommunicationsException;
import com.mysql.jdbc.Connection;

import net.sf.jasperreports.engine.JRException;
import net.sf.jasperreports.engine.JasperCompileManager;
import net.sf.jasperreports.engine.JasperExportManager;
import net.sf.jasperreports.engine.JasperFillManager;
import net.sf.jasperreports.engine.JasperPrint;
import net.sf.jasperreports.engine.JasperReport;
import net.sf.jasperreports.view.JasperViewer;

import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.HashMap;

public class Principal {

    public static void main(String[] args) {
        String reportSource = "./plantilla/plantilla.jrxml";
        String reportHTML = "./informes/Informe.html";
        String reportPDF = "./informes/Informe.pdf";
        String reportXML = "./informes/Informe.xml";

        Map<String, Object> params = new HashMap<String, Object>();
        params.put("titulo", "LISTADO DE DEPARTAMENTOS.");
        params.put("autor", "ARM");
        params.put("fecha", (new java.util.Date()).toString());
        try {
            JasperReport jasperReport =
                JasperCompileManager.compileReport(reportSource);
            Class.forName("com.mysql.jdbc.Driver");
            Connection conn = (Connection) DriverManager.getConnection
                ("jdbc:mysql://localhost/ejemplo", "root", "");
            JasperPrint MiInforme =
                JasperFillManager.fillReport(jasperReport, params, conn);
            // Visualizar en pantalla
            JasperViewer.viewReport(MiInforme);
        }
    }
}
```

```

        // Convertir a HTML
        JasperExportManager.exportReportToHtmlFile(MiInforme,
            reportHTML);
        // Convertir a PDF
        JasperExportManager.exportReportToPdfFile(MiInforme,
            reportPDF);
        // Convertir a XML.
        JasperExportManager.exportReportToXmlFile
            (MiInforme, reportXML, false);
        System.out.println("ARCHIVOS CREADOS");
    } catch (CommunicationsException c) {
        System.out.println(" Error de comunicación con la BD. "+
            " No está arrancada.");
    } catch (ClassNotFoundException e) {
        System.out.println(" Error driver. ");
    } catch (SQLException e) {
        System.out.println(" Error al ejecutar sentencia SQL ");
    } catch (JRException ex) {
        System.out.println(" Error Jasper.");
        ex.printStackTrace();
    }
}

```

2.11.1. El fichero .JRXML, la plantilla

Un informe de salida se va a estructurar en las siguientes secciones y el siguiente orden, estas secciones serán representadas en la plantilla:

Sección	Descripción
1. title	Su contenido se imprime solo una vez al comienzo del informe y como su nombre indica es el título que el informe tendrá
2. pageHeader	Esta es la cabecera de cada página, se imprimirá en cada página
3. columHeader	En esta zona se escribe la cabecera que vamos a poner para el detalle. Es decir, los nombres de las columnas que se visualizarán en el detalle (<i>detail</i>)
4. detail	Esta sección es el cuerpo del documento, es decir, donde se colocan la información a desplegar de nuestro informe (en nuestros ejercicios son columnas que devuelve la SELECT). En formato tabular
5. columFooter	En esta sección podremos poner los totales acumulados, y otras informaciones para cada una de las columnas del detalle
6. pageFooter	Este es el pie de página y se imprime al final de cada página. Útil para poner el contador de páginas, o alguna otra información
7. summary	Esto se utiliza para concluir el documento y se imprime una sola vez al final del informe

Para explicar el contenido de un fichero *jrxm1*, lo vemos con el estudio de la plantilla del ejercicio. La siguiente plantilla crea un informe para visualizar los datos de la tabla *departamentos* de la base de datos *ejemplo* de MySQL. El informe a visualizar es el siguiente, véase la Figura 2.15:

LISTADO DE DEPARTAMENTOS.		
Realizado por: ARM on Wed Apr 13 03:22:55 CEST 2016		
Código depart	Nombre departamento	Localidad departamento
10	CONTABILIDAD	SEVILLA
20	INVESTIGACIÓN	MADRID
30	VENTAS	BARCELONA
40	PRODUCCIÓN	BILBAO
61	MARKETING	GUADALAJARA

Total Registros: 5

Página 1 of 1

Figura 2.15. Informe del ejercicio

La plantilla debe empezar con la etiqueta `<jasperReport>`, este es el elemento raíz, debe ir acompañado del *namespace* de JasperReports y del nombre del informe. También en esta etiqueta se especificarán las características del documento, por ejemplo, el ancho de la página (*pageWidth*), el alto (*pageHeight*), ancho de columna (*columnWidth*), los márgenes izquierdo, derecho, superior o inferior (*leftMargin*, *rightMargin*, *topMargin*, *bottomMargin*), o también la orientación, por ejemplo, para escribir en apaisado pondremos *orientation*=*"Landscape"*.

Por ejemplo:

```
<jasperReport
    xmlns="http://jasperreports.sourceforge.net/jasperreports"
    name="Listadodepartamentos" pageWidth="595" pageHeight="600"
    columnWidth="555" leftMargin="20" rightMargin="20" topMargin="30"
    bottomMargin="30" >
```

Si se desea que el informe sea apaisado escribiremos en la etiqueta `<jasperReport>`:

```
<jasperReport name="ejemplo" orientation="Landscape" pageWidth="842">
```

En el ejemplo el fichero se va a llamar *plantilla.jrxml*, las partes de esta plantilla son las siguientes:

- **Declaración del documento**, se indica el espacio de nombres que se van a utilizar y el nombre del documento. Es la raíz del documento (al crearlo en Eclipse se añadirán de forma automática varios *namespaces*). En *xmlns* se indica el *namespace* de JasperReports, para calificar las etiquetas y atributos que pertenecen a este lenguaje:

```
<jasperReport
    xmlns="http://jasperreports.sourceforge.net/jasperreports"
    name="Listadodepartamentos">
```

- **Declaración de los parámetros**, se indica el tipo de dato de los parámetros. Los nombres son los que se ponen en el programa Java, dentro del *HashMap*.

```
<parameter name="titulo" class="java.lang.String" />
<parameter name="autor" class="java.lang.String" />
<parameter name="fecha" class="java.lang.String" />
```

- **Definición de la consulta <queryString>**, y de las columnas de la consulta **<field name>**. Acompañando a la columna se indica el tipo de dato en *class*. Si la consulta tiene campos calculados, contadores, medias, importes, etc., es necesario poner un alias al cálculo para luego referenciar la columna en el **field name**. En el ejercicio se visualizan el código de departamento, el nombre y la localidad:

```
<queryString>
  <![CDATA[SELECT * FROM departamentos]]>
</queryString>
<field name="dept_no" class="java.lang.Integer"/>
<field name="dnombre" class="java.lang.String"/>
<field name="loc" class="java.lang.String"/>
```

Para las columnas tipo *Date* pondremos el class **java.util.Date** y para las tipo *float* **java.lang.Float**.

- **Líneas de título del informe**, etiqueta **<title>**. El título solo se visualiza al inicio del informe. En el ejercicio el título definido en esta sección se muestra en la Figura 2.16:

LISTADO DE DEPARTAMENTOS.

Realizado por: ARM on Wed Apr 13 01:54:02 CEST 2016

Figura 2.16. Líneas de título del informe del ejercicio.

Dentro del título se añade una etiqueta **<band>** para indicar la altura del título. Y dentro de ella se añaden dos etiquetas **textField**, una para visualizar el título y otra para los valores de los parámetros.

Dentro de estas se indica la posición con **reportElement**. Dentro de la etiqueta **textElement** podremos indicar la alineación o la fuente. Y el contenido a visualizar se escribe en la etiqueta **textFieldExpression**, para ello se utiliza la etiqueta **<![CDATA[información]]>**. Donde indicaremos lo que se desea visualizar. Para hacer referencia a los parámetros escribimos el prefijo **\$P** seguido del nombre del parámetro entre llaves, por ejemplo: **\$P{autor}**. Las posiciones de los elementos, el ancho y el alto, se miden en pixel.

El código para el título es el siguiente:

```
<title>
  <band height="60"> <!-- Se indica el alto del título -->
    <textField>
      <reportElement x="0" y="10" width="500" height="40" />
      <textElement textAlign="Center"><font size="24" />
      </textElement>
      <textFieldExpression><![CDATA[$P{titulo}]]>
      </textFieldExpression>
    </textField>
    <textField>
```

```

<reportElement x="0" y="40" width="500" height="20" />
<textElement textAlignment="Center"/>
<textFieldExpression><! [CDATA["Realizado por: " + $P{autor}
+ " on " + $P{fecha}]]></textFieldExpression>
</textField>
</band>
</title>

```

- **Cabecera del informe**, etiqueta `<columnHeader>`. Como en el caso anterior el contenido se encierra entre la etiqueta `band`, donde indicamos la altura. Se utiliza la etiqueta `<rectangle>` para encerrar la cabecera en un rectángulo, dentro se indica la posición, el ancho y el alto del rectángulo. Para cada literal a visualizar se añade la etiqueta `<staticText>`, y dentro de ella se indica la posición x e y, el ancho y el alto del texto a visualizar con `<reportElement>`. Y el contenido del texto a visualizar se escribe dentro de la etiqueta `<text>` y utilizando la etiqueta `<! [CDATA[información]]>`. En la Figura 2.17 se muestra como queda la cabecera.

Código depart	Nombre departamento	Localidad departamento
---------------	---------------------	------------------------

Figura 2.17. Cabecera del informe del ejercicio.

El código para esta sección es el siguiente.

```

<columnHeader>
<band height="30">
<rectangle>
<reportElement x="0" y="0" width="500" height="25" />
</rectangle>
<staticText>
<reportElement x="5" y="5" width="100" height="15" />
<text><! [CDATA[Código depart]]></text>
</staticText>
<staticText>
<reportElement x="105" y="5" width="150" height="15" />
<text><! [CDATA[Nombre departamento]]></text>
</staticText>
<staticText>
<reportElement x="255" y="5" width="150" height="15" />
<text><! [CDATA[Localidad departamento]]></text>
</staticText>
</band>
</columnHeader>

```

- **Línea de detalle del informe**, etiqueta `<detail>`. Como en el caso anterior el contenido se encierra entre la etiqueta `band`, para indicar la altura de cada línea de detalle. Para cada columna a visualizar se añade una etiqueta `<textField>`, y dentro de ella se indica la posición x e y, el ancho y el alto del texto a visualizar con `<reportElement>`. Y el contenido del texto a visualizar se escribe dentro de la etiqueta `<textFieldExpression>` indicando el tipo de dato y utilizando la etiqueta `<! [CDATA[información]]>`. En la Figura 2.18 se muestra cómo queda el detalle,

10	CONTABILIDAD	SEVILLA
20	INVESTIGACIÓN	MADRID
30	VENTAS	BARCELONA
40	PRODUCCIÓN	BILBAO
61	MARKETING	GUADALAJARA

Figura 2.18. Detalle del informe del ejercicio.

Observa que las columnas de la SELECT llevan el prefijo `$F{Nombre_columna}`. El código para esta sección es el siguiente.

```
<detail>
  <band height="30">
    <textField>
      <reportElement x="35" y="7" width="100" height="15" />
      <textFieldExpression><! [CDATA[$F{dept_no}]]>
        </textFieldExpression>
    </textField>
    <textField>
      <reportElement x="105" y="7" width="150" height="15" />
      <textFieldExpression><! [CDATA[$F{dnombre}]]>
        </textFieldExpression>
    </textField>
    <textField>
      <reportElement x="255" y="7" width="150" height="15" />
      <textFieldExpression><! [CDATA[$F{loc}]]>
        </textFieldExpression>
    </textField>
  </band>
</detail>
```

- **Línea de pie del informe**, etiqueta `<pageFooter>`. En esta sección se añade una línea con color utilizando la etiqueta `<line>`. Para añadir el número de página actual se utiliza una etiqueta `<textField>`, y dentro de ella en la etiqueta `<textFieldExpression>` se indica la variable `$V{PAGE_NUMBER}` para obtener el número de página actual. Para obtener el total de páginas se utiliza también la misma variable, pero en el `textField`, se indica cuando se calcula el contador, a nivel de reporte se indica así: `<textField evaluationTime="Report">`. En la Figura 2.19 se muestra como queda el pie:

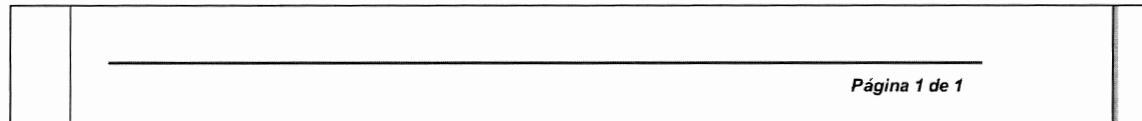


Figura 2.19. Línea de pie.

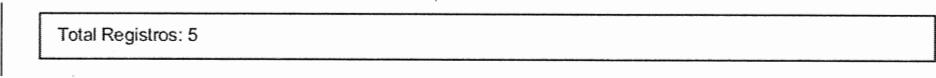
El código para esta sección es el siguiente.

```

<pageFooter>
    <band height="32">
        <line>
            <reportElement positionType="FixRelativeToBottom" x="0"
                y="3" width="500" height="1" />
            <graphicElement>
                <pen lineWidth="2.0" lineColor="#FF0000"/>
            </graphicElement>
        </line>
        <textField>
            <reportElement x="390" y="10" width="90" height="20" />
            <textElement textAlignment="Right">
                <font isBold="true" isItalic="true"/>
            </textElement>
            <textFieldExpression>
                <![CDATA["Página " + $V{PAGE_NUMBER} + " of"]]>
            </textFieldExpression>
        </textField>
        <textField evaluationTime="Report">
            <reportElement x="480" y="10" width="40" height="20" />
            <textElement><font isBold="true" isItalic="true"/>
            </textElement>
            <textFieldExpression>
                <![CDATA[" " + $V{PAGE_NUMBER}]]>
            </textFieldExpression>
        </textField>
    </band>
</pageFooter>

```

- **Línea de sumario**, etiqueta **<summary>**. Se visualiza una vez al final del informe. En el ejercicio se obtiene el número de registros visualizados. Para obtener el número de registros de utiliza la variable **\$V{REPORT_COUNT}**. Las variables llevan el prefijo **\$V**. La salida se muestra en la Figura 2.20.



Total Registros: 5

Figura 2.20. Línea de sumario.

El código para esta sección es el siguiente.

```

<summary>
    <band height="60">
        <rectangle>
            <reportElement x="0" y="0" width="500" height="25" />
        </rectangle>
        <textField>
            <reportElement x="10" y="5" width="300" height="15" />
            <textElement textAlignment="Left"/>
            <textFieldExpression> <![CDATA["Total Registros: "
                +String.valueOf($V{REPORT_COUNT})]]>
            </textFieldExpression>
        </textField>
    </band>
</summary>

```

El código completo para la plantilla es el siguiente:

```
<jasperReport
    xmlns="http://jasperreports.sourceforge.net/jasperreports"
    name="Listadodedepartamentos" >
    <parameter name="titulo" class="java.lang.String"/>
    <parameter name="autor" class="java.lang.String"/>
    <parameter name="fecha" class="java.lang.String"/>
    <queryString>
        <! [CDATA[SELECT * FROM departamentos]]>
    </queryString>
    <field name="dept_no" class="java.lang.Integer"/>
    <field name="dnombre" class="java.lang.String"/>
    <field name="loc" class="java.lang.String"/>

    <title>
        <band height="60">
            <textField>
                <reportElement x="0" y="10" width="500" height="40" />
                <textElement textAlignment="Center"><font size="24"/>
                </textElement>
                <textFieldExpression><! [CDATA[$P{titulo}]]>
                </textFieldExpression>
            </textField>
            <textField>
                <reportElement x="0" y="40" width="500" height="20" />
                <textElement textAlignment="Center"/>
                <textFieldExpression><! [CDATA["Realizado por: " +
                    $P{autor} +" on "+$P{fecha}]]></textFieldExpression>
            </textField>
        </band>
    </title>

    <columnHeader>
        <band height="30">
            <rectangle>
                <reportElement x="0" y="0" width="500" height="25" />
            </rectangle>
            <staticText>
                <reportElement x="5" y="5" width="100" height="15" />
                <text><! [CDATA[Código depart]]></text>
            </staticText>
            <staticText>
                <reportElement x="105" y="5" width="150" height="15" />
                <text><! [CDATA[Nombre departamento]]></text>
            </staticText>
            <staticText>
                <reportElement x="255" y="5" width="150" height="15" />
                <text><! [CDATA[Localidad departamento]]></text>
            </staticText>
        </band>
    </columnHeader>

    <detail>
        <band height="30">
            <textField>
```

```
<reportElement x="35" y="7" width="100" height="15" />
<textFieldExpression>
<! [CDATA[$F{dept_no}]]></textFieldExpression>
</textField>
<textField>
<reportElement x="105" y="7" width="150" height="15" />
<textFieldExpression><! [CDATA[$F{dnombre}]]>
</textFieldExpression>
</textField>
<textField>
<reportElement x="255" y="7" width="150" height="15" />
<textFieldExpression><! [CDATA[$F{loc}]]>
</textFieldExpression>
</textField>
</band>
</detail>

<pageFooter>
<band height="32">
<line>
<reportElement positionType="FixRelativeToBottom" x="0"
y="3" width="500" height="1" />
<graphicElement>
<pen lineWidth="2.0" lineColor="#FF0000"/>
</graphicElement>
</line>
<textField>
<reportElement x="390" y="10" width="90" height="20" />
<textElement textAlignment="Right">
<font isBold="true" isItalic="true"/>
</textElement>
<textFieldExpression><! [CDATA["Página " + $V{PAGE_NUMBER} +
" de"]]></textFieldExpression>
</textField>
<textField evaluationTime="Report">
<reportElement x="480" y="10" width="40" height="20" />
<textElement>
<font isBold="true" isItalic="true"/>
</textElement>
<textFieldExpression><! [CDATA[" " + $V{PAGE_NUMBER}]]>
</textFieldExpression>
</textField>
</band>
</pageFooter>

<summary>
<band height="60">
<rectangle>
<reportElement x="0" y="0" width="500" height="25" />
</rectangle>
<textField>
<reportElement x="10" y="5" width="300" height="15" />
<textElement textAlignment="Left"/>
<textFieldExpression><! [CDATA["Total Registros: "
+String.valueOf($V{REPORT_COUNT})]]>
```

```

        </textFieldExpression>
    </textField>
</band>
</summary>
</jasperReport>

```

ACTIVIDAD 2.13

Sobre el mismo proyecto, crea una nueva plantilla para obtener por cada departamento, además de sus datos, el número de empleados que hay, la media de salario y la suma de salario. Si el departamento no tiene empleados debe de salir 0 en la media, el contador y la suma. Cambia las propiedades para que el documento se visualice en apaisado.

RESUMEN DATOS DE DEPARTAMENTOS.					
Realizado por ARI en WinAge 19.1.2018-CEST 2018					
Código depart.	NOMBRE DEPARTAMENTO	Localidad Departamento	NÚMERO DE EMPLEADOS	MÉDIA DE SALARIO	SUMA DE SALARIO
10	CONTABILIDAD	SEVILLA	3	28978,1	86934
20	INVESTIGACIÓN	MADRID	5	2278,6	11390
30	VENTAS	BARCELONA	4	1779,83	7115,2
40	PRODUCCIÓN	IBÁÑEZ	0	0,0	0,0
50	MARKETING	GRANADA	0	0,0	0,0

Total Registros: 5

Página 1 de 1

Figura 2.21. Informe propuesto.

También se pueden **añadir variables acumuladas**. Para añadir totales acumulados es necesario crear las variables. Se crean debajo de los campos de la consulta (*field name*). Y luego esas variables se pueden añadir o en el sumario en las líneas de pie. Por ejemplo, aquí defino dos totales, uno para sumar la cantidad y otro para contar artículos (*cantidad*, e *idart* son *field name*). En ***calculation*** se escribe la función de grupo:

```

<variable name="sumacant" class="java.lang.Integer" calculation="Sum">
    <variableExpression><! [CDATA[$F{cantidad}]]></variableExpression>
</variable>
<variable name="numart" class="java.lang.Integer" calculation="Count">
    <variableExpression><! [CDATA[$F{idart}]]></variableExpression>
</variable>

```

Las funciones a añadir en ***calculation*** pueden ser: *sum*, *count*, *max*, *min*, *average*, *DistinctCount*, entre otras funciones. Dentro del informe estas variables se añadirán dentro de las secciones **<summary>**, o en **<columnFooter>**, las escribimos dentro de un **textField**. Por ejemplo:

```

<textField>
    <reportElement x="620" y="0" width="50" height="20"/>
    <textFieldExpression><! [CDATA[$V{sumacant}]]></textFieldExpression>
</textField>
<textField>

```

```

<reportElement x="670" y="0" width="50" height="20"/>
<textFieldExpression><![CDATA[$V{numart}]]></textFieldExpression>
</textField>

```

2.12. GESTIÓN DE ERRORES

Hasta ahora en 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 una fila en una tabla inexistente en la base de datos:

```
java.sql.SQLSyntaxErrorException: 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(OracleStatement.java:1814)
at
oracle.jdbc.driver.OracleStatement.executeUpdate(OracleStatement.java:1779)
at
oracle.jdbc.driver.OracleStatementWrapper.executeUpdate(OracleStatementWrapper.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 <i>java.lang.Throwable</i> .

A continuación se utilizan esos métodos para visualizar los mensajes de error:

```
try
{
```

```

    //Código
}
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());
}

```

El siguiente ejemplo muestra la salida que se produce cuando se intenta hacer 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 información diferente:

```

HA OCURRIDO UNA EXCEPCIÓN:
Mensaje : ORA-00942: la tabla o vista no existe

SQL estado: 42000
Cód error : 942

```

Cuando se intenta 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

```

Y en 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 la otra tabla, en MYSQL se muestra la siguiente información:

```

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

```

En ORACLE:

```

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

```

COMPRUEBA TU APRENDIZAJE

1. Rellena la siguiente tabla en donde a la izquierda aparece el nombre de la base de datos y a la derecha debe aparecer la librería necesaria para la conexión mediante un programa Java:

Base de datos	Librería Java necesaria para la conexión
SQLite	sqlite-jdbc-3.8.11.2.jar
Apache Derby	
HSQLDB	
H2	
MySQL	
ORACLE	
Db4o	

2. Rellena la siguiente tabla resumen para que aparezca por cada base de datos estudiada el driver y la URL necesaria para el establecimiento de la conexión:

Base de datos	Driver/URL
SQLite	org.sqlite.JDBC jdbc:sqlite:D:/DB/SQLITE/ejemplo.db
Apache Derby	
HSQLDB	
H2	
MySQL	
ORACLE	

3. Cuál de las siguientes afirmaciones sobre JDBC NO es correcta:

- a) JDBC define una API que pueden usar los programas Java para conectarse a bases de datos relacionales y orientadas a objetos.
- b) JDBC no solo provee una interfaz, sino que también define una arquitectura estándar, para que los fabricantes puedan crear los drivers que permitan a las aplicaciones Java el acceso a los datos.
- c) JDBC dispone de la misma interfaz para todas las bases de datos.
- d) Los tipos de conectores 3 y 4 se usan normalmente cuando el único sistema de acceso final al gestor de bases de datos es ODBC (es decir, no existen drivers disponibles para el SGBD).
- e) Los tipos de conectores 1 y 2 exigen instalación de software en el puesto cliente. El tipo 4 no exige instalación en el cliente.

4. ¿El siguiente código Java es correcto? Razona la respuesta:

```
import java.sql.*;
public class Ejercicio4 {
    public static void main(String[] args) {
        Class.forName("com.mysql.jdbc.Driver");
        Connection conexion = DriverManager.getConnection
```

```

        ("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");
        Statement sentencia = conexion.createStatement();
        ResultSet resul = sentencia.executeQuery
            ("SELECT * FROM empleados");
        while (resul.next())
        {
            System.out.printf("%d, %s %n",
                resul.getInt("EMP_NO"), resul.getString("APELLIDO"));
        }
        resul.close();
        sentencia.close();
        conexion.close();
    }
}

```

5. Crea las tablas PRODUCTOS, CLIENTES y VENTAS en las bases de datos MySQL, Oracle y SQLite. En MySQL crea una base de datos de nombre UNIDAD2, y usuario y clave con el mismo nombre. En Oracle crea un usuario de base de datos de nombre y clave UNIDAD2 y en SQLite guarda las tablas en un fichero que se llame UNIDAD2.DB. Las tablas son las siguientes:

PRODUCTOS: ID numérico, clave primaria. DESCRIPCION varchar(50), no nulo. STOCKACTUAL numérico. STOCKMINIMO numérico. PVP numérico.	CLIENTES: ID numérico, clave primaria. NOMBRE varchar(50), no nulo. DIRECCION varchar(50). POBLACION varchar(50). TELEF varchar(20). NIF varchar(10).
VENTAS: IDVENTA numérico, clave primaria. FECHAVENTA no nulo. IDCLIENTE numérico, clave ajena a CLIENTES. IDPRODUCTO numérico, clave ajena a PRODUCTOS. CANTIDAD numérico. Un cliente puede tener muchas ventas.	

Una vez creadas haz un programa Java que llene las tablas PRODUCTOS y CLIENTES (los datos a insertar se definen en el propio programa). El programa Java recibe un argumento al ejecutarlo desde la línea de comandos cuyo valor válido es 1, 2 o 3. Si el valor es 1 debes llenar las tablas de la base de datos de MySQL, si es 2 debes llenarlas de la base de datos ORACLE y si es 3 debes llenarlas en SQLite.

Una vez rellenas visualiza los datos insertados y el número de filas que se han insertado en cada tabla.

Puedes crear las clases y métodos que creas convenientes.

6. Partimos de las tablas anteriores, realiza un programa Java para insertar ventas en la tabla VENTAS. El programa recibe varios parámetros desde la línea de comandos:

- El primero indica la base de datos donde se insertará la venta (1,2, o 3, su significado es como en el ejemplo anterior).
- El segundo parámetro indica el identificador de venta.
- El tercer parámetro indica el identificador del cliente.
- El cuarto parámetro indica el identificador del producto.
- Y el quinto parámetro indica la cantidad.
- Realiza las siguientes comprobaciones antes de insertar la venta en la tabla:
- El identificador de venta no debe existir en la tabla VENTAS.
- El identificador de cliente debe existir en la tabla CLIENTES.
- El identificador de producto debe existir en la tabla PRODUCTOS.
- La cantidad debe ser > que 0.
- La fecha de venta es la fecha actual.

Una vez insertada la fila en la tabla visualizar un mensaje indicándolo. Si no se ha podido realizar la inserción visualizar el motivo (no existe el cliente, no existe el producto, cantidad menor o igual a 0, etc.)

Ejecuta el programa e inserta varias ventas en las distintas bases de datos.

7. Se pretende realizar un listado de las ventas de un cliente. El programa recibe dos parámetros desde los argumentos de *main()*, el primero indica la base de datos de la que se consultarán las ventas y el segundo el identificador del cliente cuyas ventas se van a consultar. El programa debe visualizar la siguiente información:

Ventas del cliente: *Nombre de cliente*
 Venta: *idventa* , Fecha venta: *fecha*
 Producto: *descripción del producto*
 Cantidad: *cantidad* PVP: *pvp*
 Importe: *cantidad * pvp*
 Venta: *idventa*, Fecha venta: *fecha*
 Producto: *descripción del producto*
 Cantidad: *cantidad* PVP: *pvp*
 Importe: *cantidad * pvp*

 Número total de ventas: _____
 Importe Total: _____

8. Realiza un programa Java con pantalla gráfica que nos permita consultar los datos de la tabla de *departamentos* (en MySQL). La pantalla mostrará los campos de la tabla departamentos y 4 botones. El botón *Primero* muestra el primer departamento, el botón *Siguiente* muestra el siguiente departamento al mostrado actualmente. El botón *Anterior* muestra el departamento anterior al que se está mostrando. El botón *Último* muestra el último departamento de la tabla. El primer departamento es aquel cuyo número de

departamento es el menor. El último departamento es aquel con mayor número de departamento. Mostrar los posibles errores que puedan surgir, por ejemplo, cuando pulsamos el botón *Anterior* y estamos en el primer registro, o cuando pulsamos el botón *Siguiente* y estamos en el último registro. Inicialmente se debe mostrar el primer departamento. La pantalla es la siguiente:

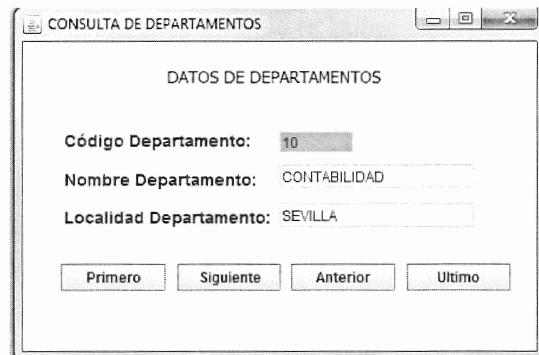


Figura 2.22. Ejercicio 8.

- Realiza un programa con pantalla gráfica para llevar a cabo la gestión de la tabla *empleados* (en MySQL). La aplicación nos debe permitir consultar, insertar, modificar y eliminar datos en la tabla. A la hora de insertar, el departamento y el director se eligen desde una lista. Al visualizar los datos de un empleado se debe mostrar en las listas su director y su departamento correspondiente. A la hora de insertar es obligatorio que todos los campos tengan valor, excepto la comisión que si no se da valor se le asigna valor 0. Por defecto, para la fecha de alta se asume la fecha del sistema. Se muestran varios botones:

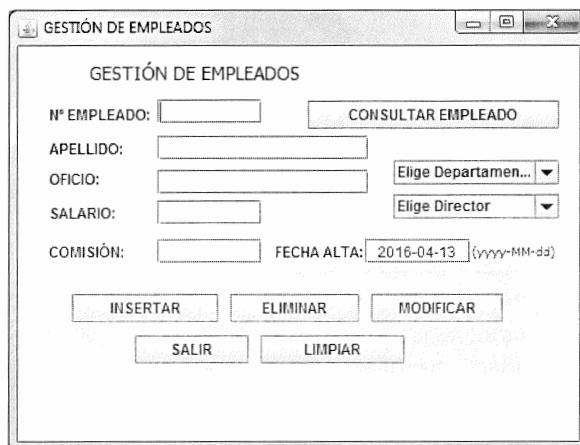


Figura 2.23. Pantalla inicial del Ejercicio 9.

- El botón *CONSULTAR EMPLEADO* muestra los datos del empleado cuyo número se ha introducido. Si no se introduce ningún número no se mostrará nada. Si se introduce un número de empleado y no existe se debe mostrar mensaje indicándolo.
- El botón *INSERTAR* inserta los datos en la tabla siempre y cuando se hayan introducido todos los valores en los campos, se debe controlar que el número del empleado no exista, si existe se debe visualizar un mensaje indicándolo. Cuando

se inserta un empleado se debe añadir a la lista de directores. La Figura 2.24 muestra la inserción de un empleado.

- El botón *ELIMINAR* elimina el empleado que se muestra en pantalla. NO se podrá eliminar un empleado que es director de otros. Se debe mostrar mensaje indicándolo. Cuando se elimina un empleado se debe eliminar de la lista de directores.
- El botón *MODIFICAR* modifica los datos del empleado. Mostrar mensaje si la modificación se ha realizado correctamente. Véase Figura 2.25.
- El botón *LIMPIAR* limpia los campos de la pantalla, se deben mostrar como en la Figura 2.23. El botón *SALIR* finaliza la ejecución.
- Despues de las operaciones de insertar, modificar o eliminar se debe mostrar mensaje de cómo se ha realizado la operación y se debe limpiar la pantalla. Se debe controlar el tamaño de los campos de entrada, número de empleado 4 dígitos, oficio y apellido tendrán el número de caracteres definidos en las columnas de la tabla, etc. Controlar todos los posibles errores que puedan surgir visualizando mensajes indicando lo que ocurre.

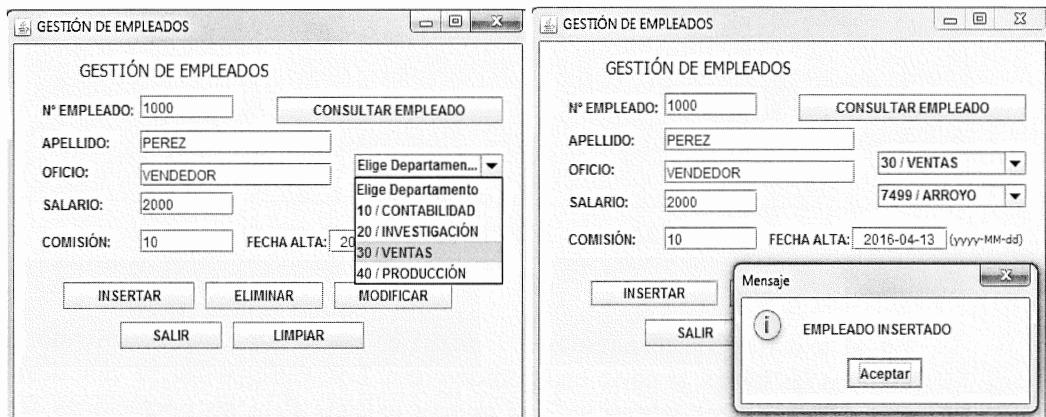


Figura 2.24. Ejercicio 9, inserción de un empleado.

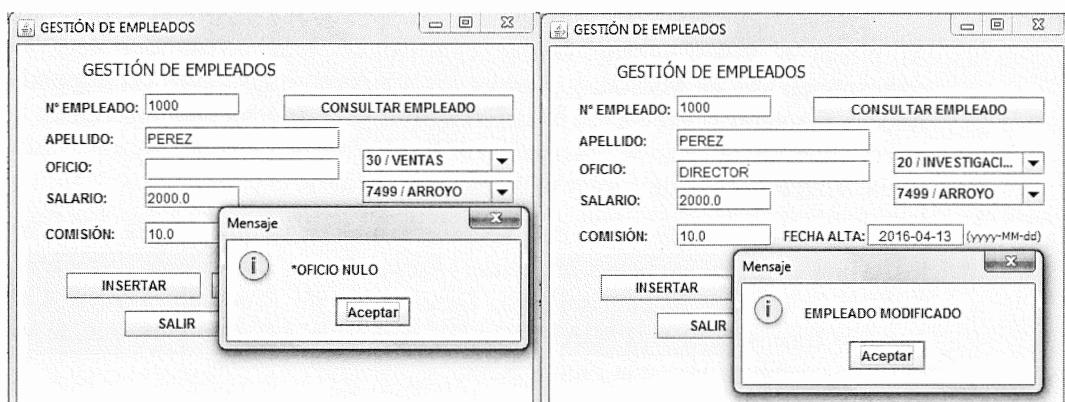


Figura 2.25. Ejercicio 9, modificación de un empleado.

10. Realiza un programa con pantalla gráfica para crear las tablas PRODUCTOS, CLIENTES y VENTAS dentro de la base de datos ACCESS *ejemplo.accdb*. Las tablas tendrán el mismo formato que las del apartado 5. Para los CREATE TABLE podemos utilizar los siguientes tipos de datos: *Integer* para enteros, *Float* para decimales, *Datetime* para la fecha, *Text(tamaño)* para la cadenas, *Long* para enteros largos, entre otros.

Para las claves primarias añadimos ***PRIMARY KEY*** a la columna que es la clave primaria, y para las claves ajenas añadimos:

REFERENCES nombreTabla(nombreColumna) a la columna que es clave ajena.

La pantalla debe ser similar a la que se muestra en la Figura 2.26.

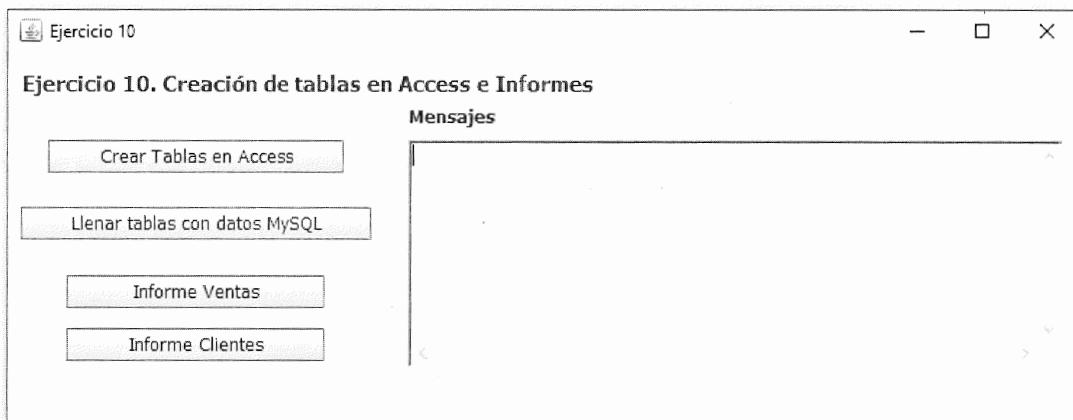


Figura 2.26. Ejercicio 10. Ventana de la actividad.

- En el *textArea* visualizaremos los mensajes de lo que vaya ocurriendo. Si se han creados las tablas, si no se han podido crear. Los registros que se van añadiendo a la BD Access, si ya existen indicad que ya existen. Para las operaciones de crear las tablas en la BD Access capturaremos la excepción ***UcanaccessSQLException***, y visualizaremos en el *textArea* el código de error y el mensaje de error de la excepción. Nos interesa detectar si la tabla ya se ha creado o no.
- Al pulsar el botón *Llenar Tablas con Datos MySQL*, hay que leer las tablas correspondientes en MySQL y guardar los registros en las tablas correspondientes de Access. Capturar la excepción ***SQLException*** para detectar si al insertar en las tablas Access el registro ya se ha insertado y salta el error de clave duplicada. Visualizar en el *textArea* el código de error y el mensaje de error de la excepción.
- Al pulsar el botón *Informe Ventas* se debe visualizar un informe de todas las ventas, los datos a visualizar son: *Id Venta*, *Fecha Venta*, *Id Cliente*, *Nombre Cliente*, *Población Cliente*, *Id de Producto*, *Descripción de Producto*, *Precio*, *Cantidad* e *Importe* (que será el *PVP*CANTIDAD*). Ajustar el informe para que se visualicen todos los campos. Véase la Figura 2.27.

RESUMEN DATOS DE VENTAS.									
Realizado por: ARM on Thu Apr 21 01:02:00 CEST 2016									
Id Venta	Fecha	Id	Nombre Cliente	Población Cliente	Id	Descripción Producto	Precio	Cantidad	Importe
1	2012-07-16	1	MARIA SERRANO	Guadalajara	4	Diccionario María Moliner	43	3	129.0
2	2012-07-17	4	ALICIA PÉREZ	Talavera	5	Impresora HP Deskjet	30	2	61.3
3	2012-07-19	2	PEDRO BRAVO	Guadalajara	5	Impresora HP Deskjet	30	1	30.65
4	2012-08-20	1	MARIA SERRANO	Guadalajara	6	Pen Drive 8 Gigas	7	5	35.0
5	2012-08-22	3	MANUEL SERRA	Guadalajara	4	Diccionario María Moliner	43	1	43.0

Total Registros: 5

Página 1 de 1

Figura 2.27. Ejercicio 10. Informe de ventas.

- Al pulsar el botón *Informe De Clientes* se debe visualizar un informe de clientes con estos datos: Id Cliente, Nombre Cliente, Población Cliente, Número de Ventas (que será el contador de ventas del cliente), Total Importe (la suma de los importes, pvp*cantidad, de las ventas del cliente).

Podemos **crear la BD Access** (si no la tenemos creada) con el siguiente método, el método recibe el nombre de la base de datos, con la extensión. La versión asignada es la 2007. Se crea la BD añadiendo ***newDatabaseVersion=V2007***:

```
public static boolean crearbd(String nombre){
    String url = UcanaccessDriver.URL_PREFIX +
        nombre+";newDatabaseVersion=V2007";
    try {
        Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
        return false;
    }
    try {
        Connection conn = DriverManager.getConnection(url);
        conn.close();
    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
    System.out.println("Base de datos "+ nombre + " creada");
    return true;
}
```

11. Crea un proyecto nuevo y utilizando JAXB estudiado en el Capítulo 1, crea las clases necesarias para obtener un documento XML, con los datos de departamentos y empleados de la BD MySQL. El documento debe presentar los datos de los departamentos y los empleados de cada departamento. Haz un método también, para visualizar el contenido del fichero XML creado. El fichero XML debe tener la siguiente estructura:

```

<Datoseempledepart>
  <Departamento>
    <depno> </depno>
    <dnombre></dnombre>
    <loc></loc>
    <Empleados>
      <Emple>
        <empno> </empno>
        <apellido> </apellido>
        <salario> </salario>
        <oficio> </oficio>
      </Emple>
    </Empleados>
  <Departamento>
  . . .
<Datoseempledepart>

```

12. Crea un proyecto nuevo y realiza un método que proporcione la siguiente información sobre la BD *ejercicio12.db*, creada con SQLITE, que se encuentra en la carpeta de recursos del capítulo. La información a mostrar es la siguiente:

INFORMACIÓN SOBRE LA BASE DE DATOS:

```
=====
Nombre :
Driver :
URL   :
```

INFORMACIÓN SOBRE LAS TABLAS (Solo las tablas), Para cada tabla visualizar:

```
=====
Nombre :
Clave(s) Primaria(s) :
Clave(s) ajena(s) (importadas) :
Clave(s) exportadas(s) :
Columnas: nombre, tipo y si puede ser nula.
Número de registros que tiene:
```

Realiza cada apartado dentro de un bloque **try-catch**, capturando la excepción **SQLException**, y visualiza el mensaje de error que devuelve la excepción.

13. Ahora se desea crear un documento XML similar al creado en el Ejercicio 11. En este caso se pide crear el XSD que describa el documento XML. Una vez creado el XSD se crea una clase Java para generar el fichero *employdepar.xml* y cargar los datos de empleados y departamentos de MySQL. El fichero XSD debe estar formado por tres tipos: **DatoseempledepartType**, que será el elemento raíz y este estará compuesto por el elemento **Departamentos**. Este elemento recogerá los datos de todos los departamentos, y cada departamento estará compuesto por *depno*, *dnombre*, *loc*, y el tipo **Empleados**. Datos del tipo **Empleados** serán *empno*, *apellido*, *oficio* y *salario*.

ACTIVIDADES DE AMPLIACIÓN

- Realiza la modificación de los departamentos de la tabla de *departamentos* (en MySQL) a partir de los datos contenidos en un fichero de texto. El fichero de texto contiene una línea por cada departamento a modificar, los campos del departamento estarán separados por comas. El formato es el siguiente: *departamento a modificar, nuevo nombre de departamento, nueva localidad*. El primer campo tiene que ser numérico. Ejemplo de fichero que contiene los datos a modificar:

```
10, INFORMÁTICA,  
20, NÓMINAS, SEVILLA  
40, , SANTANDER  
40,, SANTANDER  
10, INFORMÁTICA  
30,,  
40  
50,
```

La primera línea indica que se quiere modificar el departamento 10, el nuevo nombre de departamento es INFORMÁTICA. No se modifica la localidad. La quinta línea indica lo mismo.

La segunda línea indica que se quiere modificar el departamento 20, el nuevo nombre de departamento es NÓMINAS y la localidad es SEVILLA.

La tercera y la cuarta línea indican que se quiere modificar el departamento 40, no se modifica el nombre del departamento, se modifica la localidad que es SANTANDER.

Las tres últimas líneas indican que no se modifica el departamento correspondiente. Y así sucesivamente.

Controlar los posibles errores: departamento incorrecto, no hay datos para modificar, el departamento a modificar no existe, etc.

