

UNIDAD 2

MANEJO DE CONECTORES

OBJETIVOS

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

- 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.
- Definir modelos para comunicar con la base de datos con el patrón Modelo-Vista-Controlador.

CONTENIDOS

2.1	INTRODUCCIÓN	2.7	ESTABLECIMIENTO DE CONEXIONES
2.2	EL DESFASE OBJETO-RELACIONAL	2.8	EJECUCIÓN DE SENTENCIAS DE DESCRIPCIÓN DE DATOS
2.3	BASES DE DATOS EMBEBIDAS	2.8.1	ResultSetMetaData
2.3.1	SQLite	2.9	EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS
2.3.2	Apache Derby	2.9.1	Sentencias preparadas
2.3.3	HSQLDB	2.10	EJECUCIÓN DE PROCEDIMIENTOS
2.3.4	H2	2.11	GESTIÓN DE ERRORES
2.3.5	Db4o	2.12	PATRÓN MODELO-VISTA-CONTROLADOR. ACCESO A DATOS
2.3.6	Otras	2.13	EJERCICIOS
2.4	PROTOCOLOS DE ACCESO A BASES DE DATOS		
2.5	ACCESO A DATOS MEDIANTE ODBC		
2.6	ACCESO A DATOS MEDIANTE JDBC		
2.6.1	Arquitecturas JDBC		
2.6.2	Tipos de drivers		
2.6.3	Cómo funciona JDBC		
2.6.4	Acceso a datos mediante el Puente JDBC-ODBC		

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 Unidad 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

SQL. 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 muere cuando se cierra la aplicación.

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

SQLITE

SQLite es un sistema gestor de base de datos multiplataforma escrito en C que proporciona un rendimiento 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 contra una base de datos SQLite 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 como 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 la biblioteca. Para sistemas Windows podemos descargar el fichero ZIP *sqlite-shell-win32-x86-3070900.zip*, al descomprimirlo obtenemos un único fichero ejecutable (*sqlite3.exe*). Al ejecutarlo desde la línea de mandos 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 creemos en esta sesión se almacenarán en este fichero, para finalizar se escribe el comando *.quit*:

```
D:\>sqlite3 D:\db\SQLite\ejemplo.db
SQLite version 3.7.9 2011-11-01 00:52:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";""
sqlite> BEGIN TRANSACTION;
sqlite> CREATE TABLE departamentos (
...>     dept_no  TINYINT(2) NOT NULL PRIMARY KEY,
...>     dnombre   VARCHAR(15),
...>     loc       VARCHAR(15)
...> );
sqlite>
sqlite>
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> COMMIT;
sqlite> .quit
```

Para instalarlo desde Linux escribimos desde la línea de comandos: `$ sudo apt-get install sqlite3`. Y para ejecutarlo escribimos lo siguiente para crear la base de datos en la carpeta `/home/usuario/db/SQLite`:

```
$ sqlite3 /home/usuario/db/SQLite/ejemplo.db
```

Actividad 1: Crea las tablas EMPLEADOS y DEPARTAMENTOS en SQLite e inserta filas en ellas. La descripción de las tablas es la siguiente:

DEPARTAMENTOS:	EMPLEADOS:
DEPT_NO numérico, clave primaria DNOMBRE VARCHAR(15), LOC VARCHAR(15)	EMP_NO numérico, clave primaria APELLIDO VARCHAR(10), OFICIO VARCHAR(10), DIR numérico, FECHA_ALT DATE, SALARIO numérico, COMISION numérico, DEPT_NO numérico, clave ajena, referencia a DEPARTAMENTOS

2.3.2 APACHE DERBY

Apache Derby, es una base de datos relacional de código abierto, implementado en su totalidad en Java que forma parte del *Apache DB subproject* 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 *Derby Network Server*, 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-10.8.2.2-bin.zip`, y lo descomprimimos por ejemplo en `D:\db-derby-10.8.2.2-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.

Ejemplos para configurar la variable CLASSPATH:

Windows:

```
C:\> SET DERBY_INSTALL=D:\db-derby-10.8.2.2-bin
C:\> SET CLASSPATH=DERBY_INSTALL\lib\derby.jar;DERBY_INSTALL\lib\derbytools.jar;.
```

Linux:

```
$ export DERBY_INSTALL=/opt/db-derby-10.8.2.2-bin
$ export CLASSPATH=$DERBY_INSTALL/lib/derby.jar:$DERBY_INSTALL/lib/derbytools.jar:.
```

Para visualizar la variable CLASSPATH:

Windows:

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

Linux:

```
$ $CLASSPATH
```

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-10.8.2.2-bin\bin*). Desde la línea de comandos del DOS nos dirigimos a dicha carpeta y ejecutamos el fichero IJ.BAT:

```
D:\db-derby-10.8.2.2-bin>ij
```

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

```
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 crea la tabla EMPLEADOS e inserta filas en ella. Al final se ejecuta la orden *exit* para salir:

```
D:\db-derby-10.8.2.2-bin>ij
Versión ij 10.8
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> commit;
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> exit;

```

Para volver a usar la base de datos escribiríamos la siguiente orden desde la línea de comandos de IJ: `connect 'jdbc:derby:D:\db\Derby\ejemplo';`

Para realizar la instalación en Linux (Ubuntu) seguimos los siguientes pasos. En primer lugar es necesario instalar los paquetes JDK y JAVADB de Java (si no los tenemos instalados) ejecutando la siguiente orden desde la línea de comandos:

```
sudo apt-get install sun-java6-jdk sun-java6-plugin sun-java6-javadb
```

En algunas versiones de Linux (Ubuntu) puede no encontrar el paquete debido a un cambio de licencias de Oracle, en ese caso antes hay que añadir un nuevo repositorio y después actualizar el sistema:

```

sudo add-apt-repository ppa:ferramroberto/java
sudo apt-get update

```

A continuación descargamos la versión para Linux y la descomprimimos en la carpeta /opt, en este caso se ha descargado la versión: `db-derby-10.8.2.2-bin.tar.gz`. Se creará la carpeta `/opt/db-derby-10.8.2.2-bin`. Configuramos la variable `DERBY_INSTALL` y `DERBY_HOME` con el nombre de carpeta donde se ha descargado, ejecutando desde la línea de comandos:

```

$ export DERBY_INSTALL=/opt/db-derby-10.8.2.2-bin
$ export DERBY_HOME=/opt/db-derby-10.8.2.2-bin

```

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

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

A continuación nos dirigimos a la carpeta donde está instalado Derby y ejecutamos `setEmbeddedCP`:

```

$ cd $DERBY_INSTALL/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/`, una vez creada finalizamos la conexión ejecutando la orden *exit*:

```
$ java org.apache.derby.tools.ij
Versión ij 10.8
ij> connect 'jdbc:derby:/home/usuario/db/Derby/ejemplo;create=true';
Jan 02 21:37:24 CET 2012 Thread[main,5,main] java.io.FileNotFoundException: derby.
log (Permission denied)

-----
Jan 02 21:37:25 CET 2012:
Arrancando Derby versión The Apache Software Foundation - Apache Derby - 10.8.2.2 -
(01258): instancia a816c00e-0134-a024-0bfb-000000be33d0
en el directorio de base de datos /home/usuario/db/Derby/ejemplo con el cargador de
clases sun.misc.Launcher$AppClassLoader@7d772e

java.vendor=Sun Microsystems Inc.
java.runtime.version=1.6.0_21-b06
user.dir=/opt/db-derby-10.8.2.2-bin/bin
derby.system.home=null

Cargador de clases de base de datos iniciado - derby.database.classpath=''
ij> exit;

-----
Jan 02 21:40:55 CET 2012: cerrando motor de Derby
-----
Jan 02 21:40:57 CET 2012:
Cerrando la instancia a816c00e-0134-a024-0bfb-000000be33d0 en el directorio de
base de datos /home/usuario/db/Derby/ejemplo con cargador de clases sun.misc.
Launcher$AppClassLoader@7d772e
```

Para volver a usar la base de datos escribiríamos 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: 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*. Es compatible con SQL ANSI-92 y SQL: 2008. Puede mantener la base de datos en memoria o en ficheros en disco. Permite integridad referencial (claves foráneas),

procedimientos almacenados en Java, disparadores y tablas en disco de hasta 8GB. Se distribuye con licencia BSD (*Berkeley Software Distribution*) que es una licencia muy cercana al dominio público.

Desde la web <http://sourceforge.net/projects/hsqlDb/files/> podemos descargarnos la última versión. En Windows se ha descargado el fichero *hsqlDb-2.2.6.zip*. Lo descomprimimos por ejemplo en la unidad D, se creará una carpeta con el nombre del fichero ZIP (*D:\hsqlDb-2.2.6\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*.

Abrimos la línea de comandos del DOS 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.

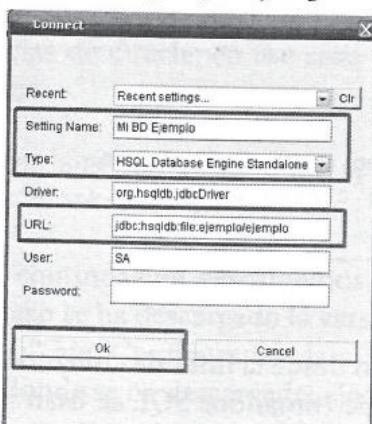


Figura 2.1. Configurar conexión en HSQLDB.

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.

DEPT_NO	DNOMBRE	LOC
10	CONTABILIDAD	SEVILLA
20	INVESTIGACIÓN	MADRID
30	VENTAS	BARCELONA
40	PRODUCCIÓN	BILBAO

Figura 2.2. Ejecución de sentencias SQL en HSQLDB.

Para instalarlo en Ubuntu primero guardamos el fichero *hsqldb-2.2.6.zip* en la carpeta */opt*. Des-
de la línea de comandos nos vamos a dicha carpeta y lo descomprimimos ejecutando la
siguiente orden:

```
$ sudo unzip hsqldb-2.2.6.zip
```

Se creará en *opt* una carpeta con nombre *hsqldb-2.2.6*, dentro hay otra carpeta que se llama *hsqldb*,
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 *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
que se hizo anteriormente. En el campo URL escribimos la carpeta donde se almacenará la base de
datos y su nombre, nos debe quedar: *jdbc:hsqldb:file:/home/usuario/db/Hsqldb/ejemplo*. El resto de
operaciones son similares.

Actividad 3: Crea las tablas EMPLEADOS y DEPARTAMENTOS en HSQLDB e 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://sourceforge.net/projects/hsqldb/files/> podemos descargarnos la última versión. Para probarla descargamos la versión ZIP para todas las plataformas *h2-2011-11-26.zip*. En Windows lo descomprimimos por ejemplo en la unidad D se creará una carpeta con el nombre *D:/h2*. Desde la
línea de comandos de Windows nos dirigimos a la carpeta *D:/h2/bin* y ejecutamos el fichero *h2.bat*
para arrancar la consola:

```
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
las crea), pulsamos el botón *Guardar* para que guarde la configuración y a continuación el botón

Conectar. Cada vez que queramos conectarnos a nuestra base de datos usaremos el nombre dado a la configuración. Podemos pulsar el botón *Probar la 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.

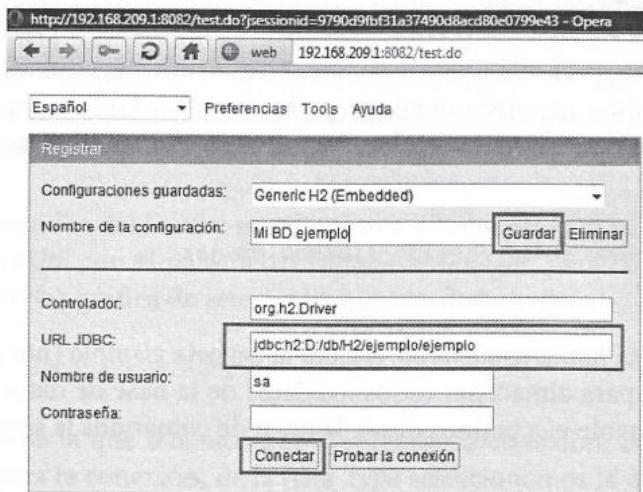


Figura 2.3. Establecer conexión en H2.

Una vez conectados se visualiza una nueva pantalla, 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. Desde la zona de instrucciones SQL podremos escribir las sentencias para crear tablas, insertar filas, etc., véase Figura 2.5.

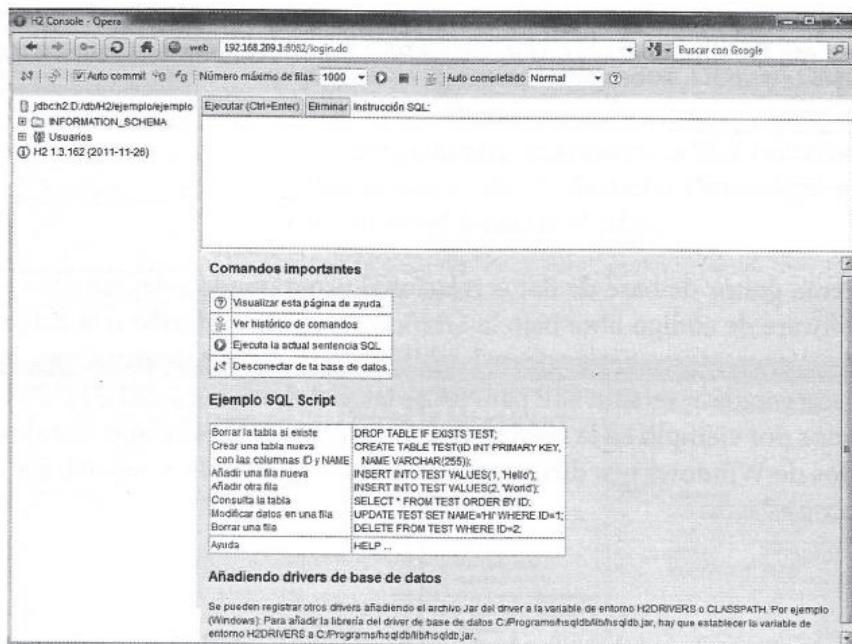


Figura 2.4. Pantalla de manejo de la base de datos en H2.

Los botones **Ejecutar (Ctrl+Enter)** y **Execute** 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.

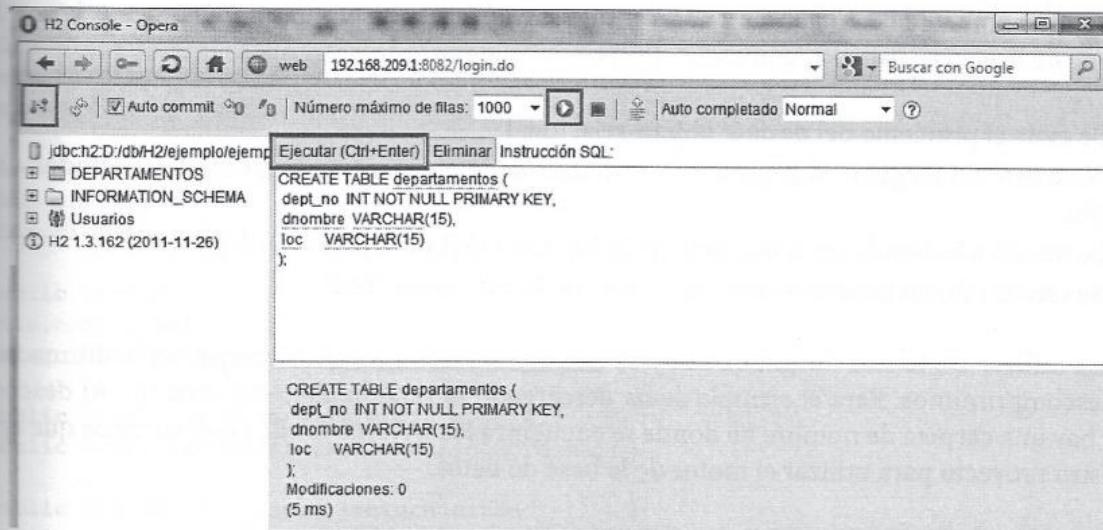


Figura 2.5. Ejecución de sentencias en H2.

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.

En Linux Ubuntu extraemos el fichero *h2-2011-11-26.zip* en la carpeta *opt* de esta manera tenemos */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), 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 URL JDBC podemos escribir 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*, se crearán las carpetas, si no existen, y dentro los ficheros para la base de datos de nombre *ejemplo*.

Actividad 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.

Para utilizar Db4o nos dirigimos a la web <http://www.db4o.com/>, descargamos la última versión y la descomprimimos. Para el ejemplo se ha descargado la versión *db4o-8.0-java.zip*. Al descomprimirla hay una carpeta de nombre *lib* donde se encuentra los JAR (*db4o-.jar*) que tenemos que agregar a nuestro proyecto para utilizar el motor de la base de datos.

Desde Eclipse podemos crear una librería con todos los JAR para ello seleccionamos nuestro proyecto, pulsamos el botón derecho del ratón y seleccionamos **Build Paths-> Add Libraries**. Se visualiza una ventana desde la que hemos de elegir la opción **User Library** y pulsar el botón *Next*. A continuación se visualiza una nueva ventana desde la que pulsamos el botón *User Libraries*, a continuación pulsamos el botón *New* para dar un nombre a la librería, por ejemplo *Db4oLib*. Seguidamente pulsamos el botón *Add JARs*, localizamos todos los *db4o-.jar* de la carpeta *lib* y pulsamos el botón *Abrir*, véase Figura 2.6. A continuación *OK* y para finalizar pulsamos *Finish*. Aunque no es necesario añadir todos los JAR, bastaría con añadir el JAR *db4o-8.0.224.15975-core-java5.jar*.

La librería quedará integrada en nuestro proyecto Eclipse. 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.

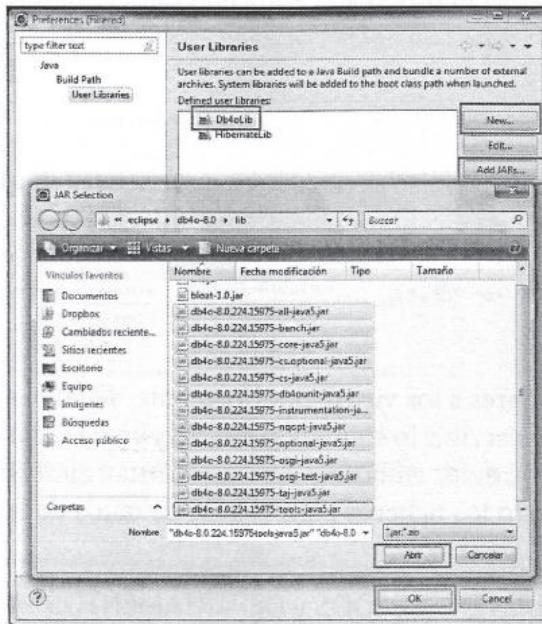


Figura 2.6. Añadir librerías Db4o en Eclipse.

El siguiente ejemplo muestra un programa Java que crea una base de datos y almacena objetos *Persona* en ella. Se ha definido la clase **Persona.java** formada por los atributos *nombre* y *ciudad* y los métodos *get* y *set* para obtener y almacenar los valores de un registro Persona:

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

    public Persona(String nombre, String ciudad) {
        this.nombre=nombre;
        this.ciudad=ciudad;
    }

    public Persona() {
        this.nombre=null;
        this.ciudad=null;
    }

    public String getNombre(){return nombre;}
    public void setNombre(String nom){nombre=nom; }

    public String getCiudad(){return ciudad; }
    public void setCiudad(String dir){ciudad=dir; }

} //fin Persona
```

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 atributos y pulsar el botón OK.

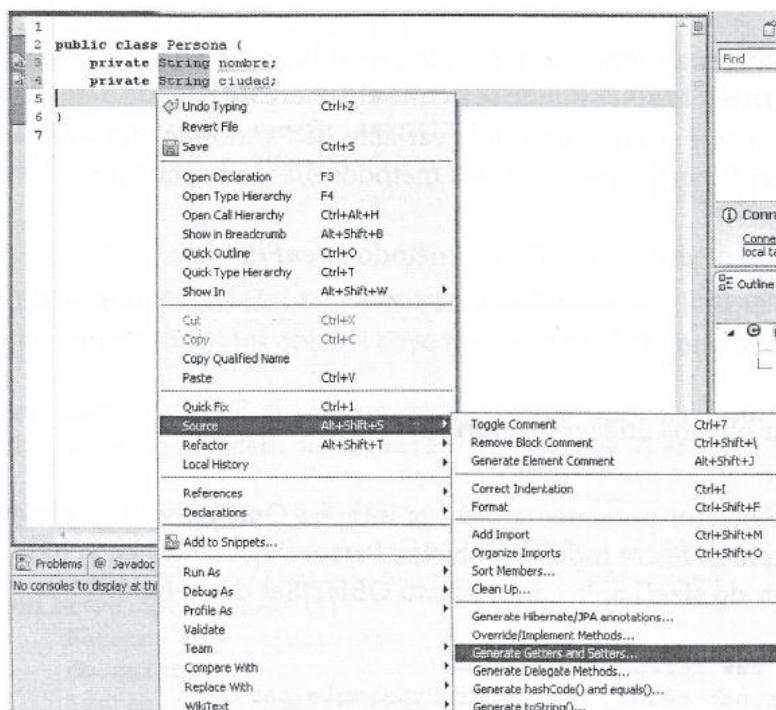


Figura 2.7. Generar *Getters* y *Setters*.

El siguiente código muestra la clase **Main.java** que crea la base de datos (si no existe) e inserta objetos Persona en ella:

```

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

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

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

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

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

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

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

```

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 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 a *close()*:

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

- Para almacenar objetos utilizamos *store()*:

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

- Para recuperar objetos podemos utilizar la interfaz Query-By-Example *queryByExample()*. El siguiente ejemplo muestra todos los objetos Persona existentes en la base de datos, si no existe ninguno el método *size()* aplicado al objeto **ObjectSet** devuelve 0:

```

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.println("Número de registros: "+result.size());

while(result.hasNext()){
    Persona p = result.next();
    System.out.println("Nombre:" + p.getNombre()+
                       ", Ciudad:" + p.getCiudad());
}
db.close(); //cerrar base de datos

```

El siguiente ejemplo obtendría los objetos Persona cuyo nombre es Juan:

```

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

```

El siguiente ejemplo obtendría 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 *store()*. El siguiente ejemplo modifica la ciudad de Juan a Toledo y luego visualiza sus datos:

```

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.println("Nombre:" + existe.getNombre()+
                       " Nueva Ciudad: " + existe.getCiudad());
}

```

- Para eliminar objetos utilizamos *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 {
    Persona existe = (Persona) result.next();
    System.out.println("Registros a borrar: "+ result.size());
    if (result.size()>1) { //varios objetos con nombre Juan
        while(result.hasNext()){
            Persona p = result.next();
            db.delete(p);
            System.out.println("Borrado....");
        }
    } else db.delete(existe); //solo hay un objeto con nombre Juan
}

```

En Linux cambiaríamos la localización del fichero, por ejemplo:

```
final static String BDPer = "/home/usuario/Java/Db4o/DBPersonas.yap";
```

Si queremos ejecutar desde la línea de comandos los programas Java tendríamos que modificar la variable CLASSPATH. En el siguiente ejemplo los ficheros Java están en la carpeta */home/usuario/Java/Db4o*, dentro está la carpeta *lib* que contiene los JAR de Db4o, añadiremos al CLASSPATH el fichero *db4o-8.0.224.15975-core-java5.jar*, después se compilarían los ficheros y al final se ejecutaría Main:

```
/Java/Db4o$ export CLASSPATH=$CLASSPATH:/home/usuario/Java/Db4o/lib/db4o-8.0.224.15975-  
core-java5.jar  
/Java/Db4o$ javac Persona.java  
/Java/Db4o$ javac Main.java  
/Java/Db4o$ java Main
```

En la Unidad 4 se profundizará más acerca de las bases de datos orientadas a objetos.

Actividad 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 de 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.

- **SQL Server Mobile (Microsoft SQL Server 2005 Mobile Edition):** Es una base de datos compacta y con una gran variedad de funciones diseñada para admitir una amplia lista de dispositivos inteligentes y Tablet PC. 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:
 - Un motor de base de datos compacto y un sólido optimizador de consultas.
 - Compatibilidad con la réplica de mezcla y el acceso a datos remotos.
 - Integración con Microsoft SQL Server 2005.
 - Las herramientas de administración son Microsoft SQL Server Management Studio y SQL Server Management Studio Express.
 - Integración con Microsoft Visual Studio 2005.
 - Acceso a datos remotos y réplica de mezcla para sincronizar datos.
 - Microsoft Proveedor de datos .NET Framework y .NET Compact Framework para SQL Server Compact Edition (System.Data.SqlClientCe).
 - Compatibilidad con Microsoft ADO.NET y el proveedor de OLE DB para SQL Server Compact Edition.
 - Un subconjunto de sintaxis SQL.
 - Se implementa como una base de datos incrustada en equipos de escritorio, dispositivos móviles y Tablet PC.
 - Compatibilidad con la tecnología de implementación ClickOnce.
- **Oracle Embedded:** bajo este nombre Oracle ofrece un conjunto de soluciones al desarrollo software que aporta ciertas ventajas como la instalación “silenciosa” (se instala solo sin tener que hacer clic), la configuración automática o el funcionamiento desatendido y menores costes de implementación, licencia, hardware y administración. La tecnología Embedded de Oracle ofrece elementos especializados y una gran libertad de elección según el tipo de necesidades. La oferta de bases de datos permite elegir entre tres elementos:
 - **Oracle Database 11g:** para aplicaciones empresariales.
 - **Oracle TimesTen In-Memory Database:** para aplicaciones en tiempo real que necesitan un tiempo de respuesta de microsegundos. *TimesTen* es una base de datos relacional que se ejecuta en memoria y que gracias a ello ofrece unos altísimos tiempos de respuesta y una caché de datos en tiempo real. Está especialmente diseñada para entornos de misión crítica.
 - **Oracle Berkeley DB:** proporciona a los desarrolladores una base de datos embebida permitiendo varias opciones de almacenamiento: clave/valor, SQL, XML / XQuery o Java Object para su modelo de datos. Entre sus características cabe destacar que se puede ejecutar en disco o en memoria y proporciona un alto rendimiento. Algunas características de Oracle Berkeley DB son:
 - Rendimiento extraordinario, elimina los gastos de la comunicación interprocesos y SQL.
 - Administración cero, toda la administración se realiza a través de una API, se integra por completo en la aplicación y es invisible para los usuarios finales.
 - En dispositivos móviles, el tamaño de las librerías es de menos de 1MB y el tiempo de ejecución de los requisitos de memoria dinámica es de solo unos pocos KB.
 - Bajo coste total de propiedad, al ser embebida se reducen los gastos de implementación, licencias y hardware, y los costes de administración.

El siguiente ejemplo en Java muestra el uso de una base de datos Berkeley para almacenar pares clave/valor. Para poder ejecutarlo hemos de agregar el fichero *je-5.0.34.jar* a nuestro proyecto Eclipse o a nuestro CLASSPATH, este se puede conseguir descargando la versión ZIP de *Berkeley DB Java Edition 5.0.34* (*je-5.0.34.zip*) desde la URL de Oracle: <http://www.oracle.com/technetwork/database/berkeleydb/downloads/index.html>. El fichero se encuentra en la carpeta *lib*. Primero será necesario crear un entorno de base de datos y a continuación la base de datos. En el ejemplo la base de datos se llama *NUMEROS* y se almacena en la carpeta *D:/je-5.0.34/ejemplo*:

```
import java.io.File;
import java.io.UnsupportedEncodingException;

import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
import com.sleepycat.je.Transaction;

public class Main {
    public static void main(String[] args) throws UnsupportedEncodingException {

        String envDir = "D:/je-5.0.34/ejemplo";//carpeta donde se almacenará la BD

        // Creamos un entorno de base de datos transaccional
        EnvironmentConfig envConfig = new EnvironmentConfig();
        envConfig.setAllowCreate(true);
        envConfig.setTransactional(true);
        Environment env = new Environment(new File(envDir), envConfig);

        //Crear una base de datos transaccional para ese entorno
        Transaction tx = env.beginTransaction(null, null);
        DatabaseConfig dbConfig = new DatabaseConfig();
        dbConfig.setTransactional(true);
        dbConfig.setAllowCreate(true);
        dbConfig.setSortedDuplicates(true);
        Database myDb = env.openDatabase(null, "NUMEROS", dbConfig);

        // DatabaseEntry representa los pares clave valor de cada registro
        DatabaseEntry clave =new DatabaseEntry();
        DatabaseEntry valor =new DatabaseEntry();

        //introducimos pares clave/valor
        clave = new DatabaseEntry("1".getBytes("UTF-8"));
        valor = new DatabaseEntry("uno".getBytes("UTF-8"));
        myDb.put(tx, clave, valor);

        clave = new DatabaseEntry("2".getBytes("UTF-8"));
        valor = new DatabaseEntry("dos".getBytes("UTF-8"));
        myDb.put(tx, clave, valor);
    }
}
```

```
clave = new DatabaseEntry("3".getBytes("UTF-8"));
valor = new DatabaseEntry("tres".getBytes("UTF-8"));
myDb.put(tx, clave, valor);
tx.commit();

String laclave;
String elvalor;
//Cursor para recorrer todos los pares Clave/Valor
Cursor cursor = myDb.openCursor(null, null);

while (cursor.getNext(clave, valor, LockMode.DEFAULT) ==
       OperationStatus.SUCCESS) {
    laclave = new String(clave.getData(), "UTF-8");
    elvalor = new String(valor.getData(), "UTF-8");
    System.out.println("Clave = " + laclave + " Valor = " + elvalor);
}

cursor.close(); //cerramos cursor
myDb.close(); //cerramos BD
env.close(); //cerramos entorno

}//fin de main

}//fin de la clase
```

La ejecución del programa visualiza la siguiente información:

```
Clave = 1 Valor = uno
Clave = 2 Valor = dos
Clave = 3 Valor = tres
```

Para probarlo en Linux cambiaríamos la carpeta donde se almacenará la base de datos (que tiene que existir), por ejemplo:

```
String envDir = "/home/usuario/db/Berkeley/ejemplo";
```

Después desde la línea de comandos definimos en el CLASSPATH donde se encuentra el fichero JAR, por ejemplo, si está en la carpeta /home/usuario/Java/Berkeley escribimos:

```
$ export CLASSPATH=$CLASSPATH:/home/usuario/Java/Berkeley/je-5.0.34.jar
```

Compilamos y ejecutamos (suponemos que nuestro programa Java está en la misma carpeta):

```
Java/Berkeley$ javac Main.java
Java/Berkeley$ java Main
Clave = 1 Valor = uno
Clave = 2 Valor = dos
Clave = 3 Valor = tres
Java/Berkeley$
```

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.
- **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 que 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.

Una vez reservados los manejadores el programa abre la conexión a la base de datos usando `SQLDriverConnect()` o `SQLConnect()`.

Una vez realizada la conexión el programa puede enviar órdenes SQL a la base de datos usando `SQLExecDirect()`.

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 interface 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 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 del mismo tienen el mismo nombre de la base de datos. Para poder acceder necesitamos crear un origen de datos ODBC. Desde Linux (Ubuntu) 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 `/usr/lib/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/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 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, &nRows );
        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 conexionODBC.c -lodbc -std=gnu99
uno.c: In function 'main':
uno.c:113: warning: format '%u' expects type 'unsigned int', but argument 2 has type
'SQLINTEGER'
usuario@ubuntu2012:~/ $ ./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 (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.

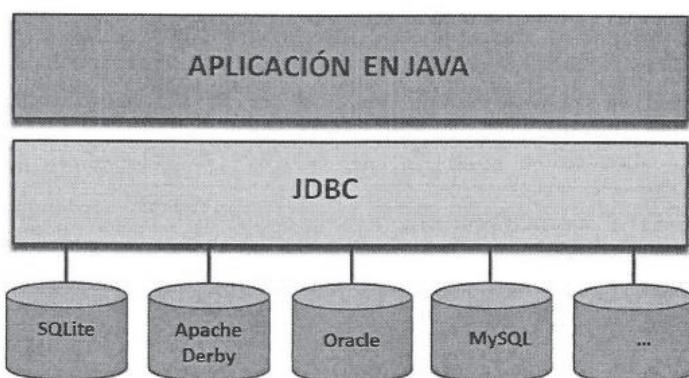


Figura 2.8. Acceso mediante JDBC.

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 ARQUITECTURAS JDBC

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, Figura 2.10. En este caso los drivers no tienen que residir en la máquina cliente.

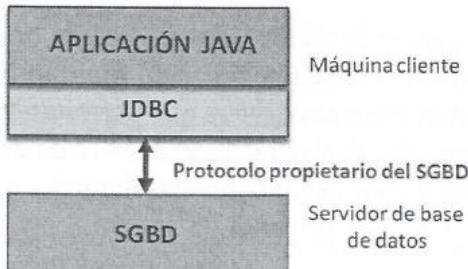


Figura 2.9. Arquitectura JDBC de dos capas.

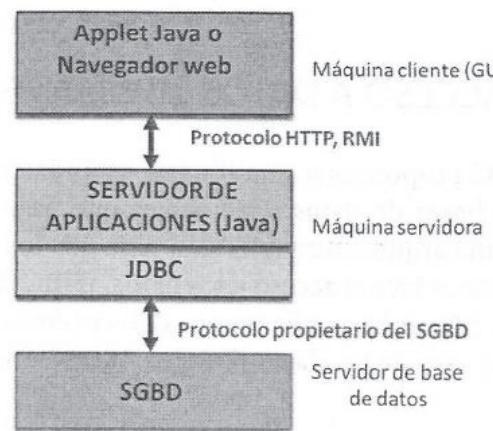


Figura 2.10. Arquitectura JDBC de tres capas.

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. 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. 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 un servidor de base de datos. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red. No exige instalación en cliente.
- **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 INTERFACE	DESCRIPCIÓN
Driver	Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto.
DriverManager	Permite gestionar todos los drivers instalados en el sistema.
DriverPropertyInfo	Proporciona diversa información acerca de un driver.
Connection	Representa una conexión con una base de datos. Una aplicación puede tener más de una conexión.
DatabaseMetadata	Proporciona información acerca de una Base de Datos, como las tablas que contiene, etc.
Statement	Permite ejecutar sentencias SQL sin parámetros.
PreparedStatement	Permite ejecutar sentencias SQL con parámetros de entrada.
CallableStatement	Permite ejecutar sentencias SQL con parámetros de entrada y salida, como llamadas a procedimientos almacenados.
ResultSet	Contiene las filas resultantes de ejecutar una orden SELECT.
ResultSetMetadata	Permite obtener información sobre un ResultSet, como el número de columnas, sus nombres, etc.

<http://docs.oracle.com/javase/6/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*.

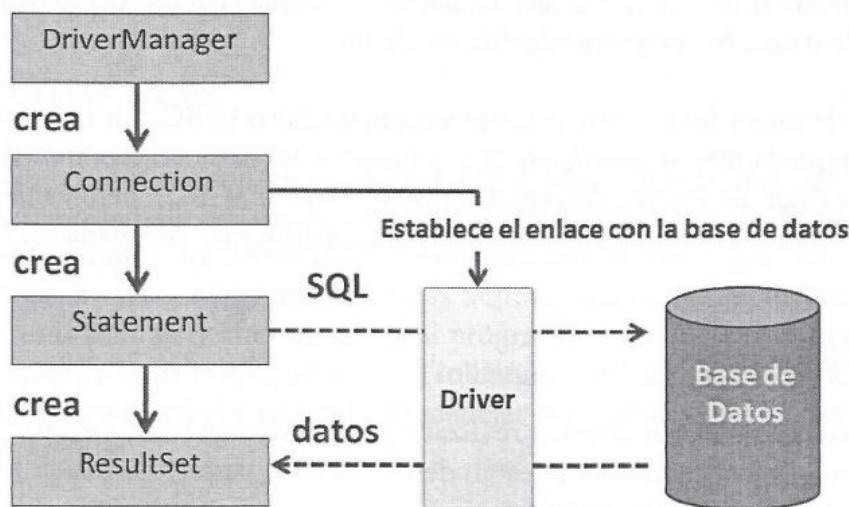


Figura 2.11. Funcionamiento de JDBC.

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 (si ya tenemos creada la BD y las tablas no es necesario realizar estos pasos):

```

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

CREATE TABLE empleados (
  emp_no   SMALLINT(4) UNSIGNED 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 REFERENCES departamentos(dept_no)
;

```

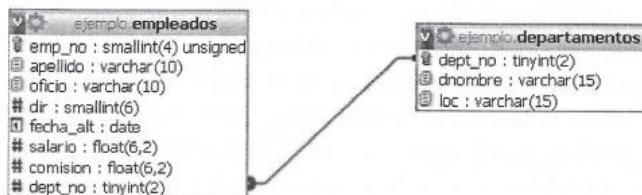


Figura 2.12- Base de datos EJEMPLO.

El siguiente programa ilustra los pasos de funcionamiento de JDBC accediendo a la base de datos EJEMPLO:

```

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();
            ResultSet resul = sentencia.executeQuery ("SELECT * FROM departamentos");

            // Recorremos el resultado para visualizar cada fila
            // Se hace un bucle mientras haya registros, se van visualizando
            while (resul.next())
            {
                System.out.println (resul.getInt(1) + " " + resul.getString(2)+ " " +
                    resul.getString(3));
            }

            resul.close(); // Cerrar ResultSet
            sentencia.close(); // Cerrar Statement
            conexion.close(); // Cerrar conexion
        }
        catch (ClassNotFoundException cn) {cn.printStackTrace();}
        catch (SQLException e) {e.printStackTrace();}
    }//fin de main
}//fin de la clase

```

Visualiza el contenido de la tabla DEPARTAMENTOS de la base de datos EJEMPLO de MySQL:

```
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 *java.sql.**. También se ha incluido todo el programa en un *try-catch* ya que casi todos los métodos relativos a 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()*:

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

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

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

El segundo parámetro es el nombre de usuario que accede a la base de datos, en este caso se llama *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();
ResultSet resul = sentencia.executeQuery ("SELECT * FROM departamentos");
```

El resultado nos lo devuelve como un *ResultSet*, que es un objeto similar a una lista en la que está el resultado de la consulta. Cada elemento de la lista es uno de los registros de tabla DEPARTAMENTOS. *ResultSet* no contiene todos los datos, sino que los va consiguiendo de la base de datos según se pidiendo. Por ello, el método *executeQuery()* puede tardar poco, pero recorrer todos los elementos del *ResultSet* puede no ser tan rápido.

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

```
while (resul.next())
{
    System.out.println (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.println (resul.getInt("dept_no") + " " + resul.getString("dnombre")+
    " " + resul.getString("loc"));
```

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 6: Siguiendo el ejemplo anterior 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.

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 sí 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 ODBC para MySQL y pulsamos el botón *Finalizar*, véase Figura 2.13.

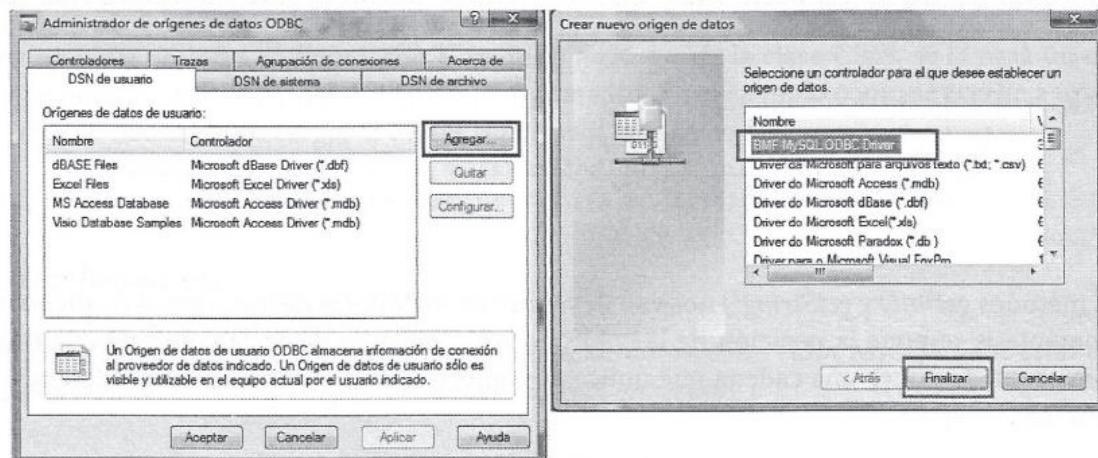


Figura 2.13. Crear un origen de datos ODBC.

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 *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*. Como nombre de usuario escribimos *ejemplo* 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 *EJEMPLO*. Pulsamos el botón *OK*, véase Figura 2.14.

Para cada esquema de base de datos es necesario crear un origen de datos. Desde la pestaña *Connect Options* podemos escribir el número de puerto por el que escucha MySQL; por defecto asume

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.

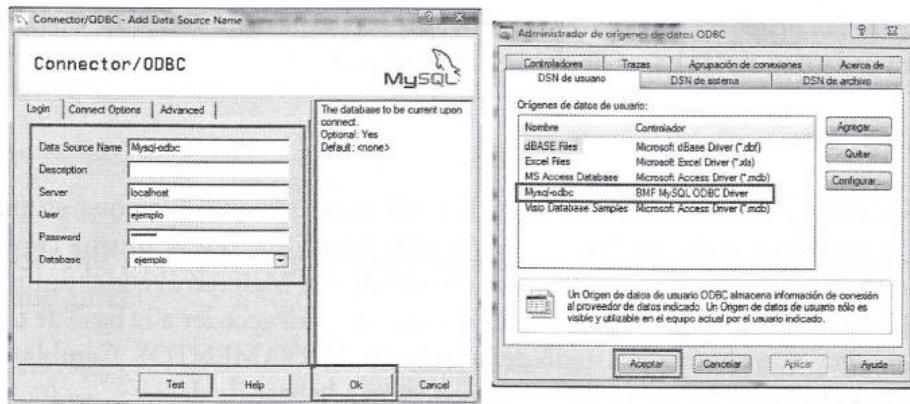


Figura 2.14. Origen de datos ODBC para acceder a MySQL desde Windows.

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 Driver for MySQL (Connector/ODBC)*. Puede ser un fichero con el nombre *mysql-connector-odbc-5.1.9-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.

Actividad 7: Realiza la Actividad 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 como conectarnos a través de JDBC a las bases de datos embebidas estudiadas anteriormente. Hemos de tener creada 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/. También para realizar las pruebas, necesitaremos tener en las carpetas el conector (fichero JAR) correspondiente para cada una de las bases de datos y el programa Java.

Conección a SQLite

Para conectarnos a SQLite necesitamos la librería *sqlite-jdbc-3.7.2.jar* que se puede descargar desde la URL <http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC>. En la misma carpeta donde está la base de datos creamos nuestro programa Java (su nombre es Main.java) e incluimos el fichero JAR. El programa Java es el mismo que se utilizó anteriormente para acceder a la base de datos MySQL de nombre *ejemplo* y que obtenía el listado de la tabla DEPARTAMENTOS. Cambiamos 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");
```

Para compilar y ejecutar nuestro programa desde Windows ejecutamos desde la línea de comandos:

```
D:\db\SQLite>javac Main.java
D:\db\SQLite>Java -classpath ".;sqlite-jdbc-3.7.2.jar" Main
```

En Linux cambiamos la conexión a la base de datos:

```
Connection conexion = DriverManager.getConnection
    ("jdbc:sqlite:/home/usuario/db/SQLite/ejemplo.db");
```

Para ejecutarlo nos vamos a la carpeta */home/usuario/db/SQLite/* definimos la variable CLASSPATH con la carpeta donde estará la clase (Main.class) y el fichero JAR:

```
$ export CLASSPATH=/home/usuario/db/SQLite/:/home/usuario/db/SQLite/sqlite-jdbc-
3.7.2.jar
$ javac Main.java
$ java Main
```

Conección a Apache Derby

Para conectarnos a Apache Derby necesitamos la librería *derby.jar* que ya vimos como obtenerla al instalar la base de datos. En la misma carpeta donde está la base de datos copiamos el programa Java y cambiamos la carga del driver, en este caso se llama *org.apache.derby.jdbc.EmbeddedDriver* y la conexión a la base de datos:

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

Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\Derby>javac Main.java
D:\db\Derby>java -classpath ".;derby.jar" Main
```

En Linux cambiamos la conexión a la base de datos:

```
Connection conexion = DriverManager.getConnection
("jdbc:derby:/home/usuario/db/Derby/ejemplo");
```

Para ejecutarlo nos vamos a la carpeta */home/usuario/db/Derby/* y definimos la variable CLASSPATH como antes:

```
$ export CLASSPATH=/home/usuario/db/Derby/:/home/usuario/db/Derby/derby.jar
$ javac Main.java
$ java Main
```

Conexión a HSQLDB

Para conectarnos a HSQLDB necesitamos la librería *hsqldb.jar* que se puede obtener de la carpeta *hsqldb* que se encuentra al descomprimir el fichero *hsqldb-2.2.6.zip*. En la carpeta *ejemplo* (donde está la base de datos) copiamos el programa Java, el fichero JAR y cambiamos la carga del driver, en este caso se llama *org.hsqldb.jdbcDriver* y la conexión a la base de datos:

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

Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\Hsqldb\ejemplo>javac Main.java
D:\db\Hsqldb\ejemplo>java -classpath ".;hsqldb.jar" Main
```

En Linux cambiamos la conexión a la base de datos:

```
Connection conexion = DriverManager.getConnection
("jdbc:hsqldb:file:/home/usuario/db/Hsqldb/ejemplo/ejemplo");
```

Para ejecutarlo nos vamos a la carpeta */home/usuario/db/Hsqldb/ejemplo* y definimos la variable CLASSPATH:

```
$export CLASSPATH=/home/usuario/db/Hsqldb/ejemplo/:/home/usuario/db/Hsqldb/ejemplo/
hsqldb.jar
$ javac Main.java
$ java Main
```

Conexión a H2

Para conectarnos a H2 necesitamos la librería *h2-1.3.162.jar* que se puede obtener de la carpeta *bin* en la que se encuentra al descomprimir el fichero *h2-2011-11-26.zip*. En la carpeta donde está la base de datos (*D:\db\H2\ejemplo*) copiamos el programa Java, el fichero JAR y cambiamos la carga del driver, en este caso se llama *org.h2.Driver* y la conexión a la base de datos:

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

```

En este caso se ha incluido el nombre de usuario “sa” ya que al crear la base de datos se dejó este nombre de usuario y la clave en blanco. Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\H2\ejemplo>javac Main.java
D:\db\H2\ejemplo>java -classpath ".;h2-1.3.162.jar" Main
```

En Linux cambiamos la conexión a la base de datos:

```
Connection conexion = DriverManager.getConnection
    ("jdbc:h2:/home/usuario/db/H2/ejemplo","sa","");

```

Para ejecutarlo nos vamos a la carpeta */home/usuario/db/H2/ejemplo* y definimos la variable CLASSPATH:

```
$ export CLASSPATH=/home/usuario/db/H2/ejemplo:/home/usuario/db/H2/ejemplo/
h2-1.3.162.jar
$ javac Main.java
$ java Main
```

Conexión a MySQL

Para conectarnos a MySQL necesitamos la librería *mysql-connector-java-5.1.18-bin.jar*. Nos creamos la carpeta MySQL dentro de *D:\db*, copiamos el programa Java, el fichero JAR y cambiamos la carga del driver, en este caso se llama *com.mysql.jdbc.Driver* y la conexión a la base de datos:

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

Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\MySQL>javac Main.java
D:\db\MySQL>java -classpath ".;mysql-connector-java-5.1.18-bin.jar" Main
```

Desde Linux nos vamos a la carpeta */home/usuario/db/MySQL* copiamos el programa Java y el JAR y definimos la variable CLASSPATH:

```
$export CLASSPATH=/home/usuario/db/MySQL/:/home/usuario/db/MySQL/mysql-connector-
$java-5.1.18-bin.jar
$javac Main.java
$java Main
```

Conexión a ORACLE

Para conectarnos a Oracle utilizando el puente ODBC-JDBC hemos de crear un origen de datos. Asumiendo que tenemos instalado en nuestra máquina Windows *Oracle Database 10g Express Edition* creamos un usuario de nombre y clave *EJEMPLO* y creamos las tablas *EMPLEADOS* y *DEPARTAMENTOS*. Al instalarse la base de datos se crea el servicio *XE* para acceder a ella. Para crear el origen de datos seleccionamos *Oracle in XE*, pulsamos el botón *Finalizar* (también se podía haber elegido el driver *Microsoft ODBC for Oracle*) y a continuación rellenamos el resto de campos para la configuración del driver, véase Figura 2.15. Damos el nombre de *ORACLE-XE* al origen de datos.

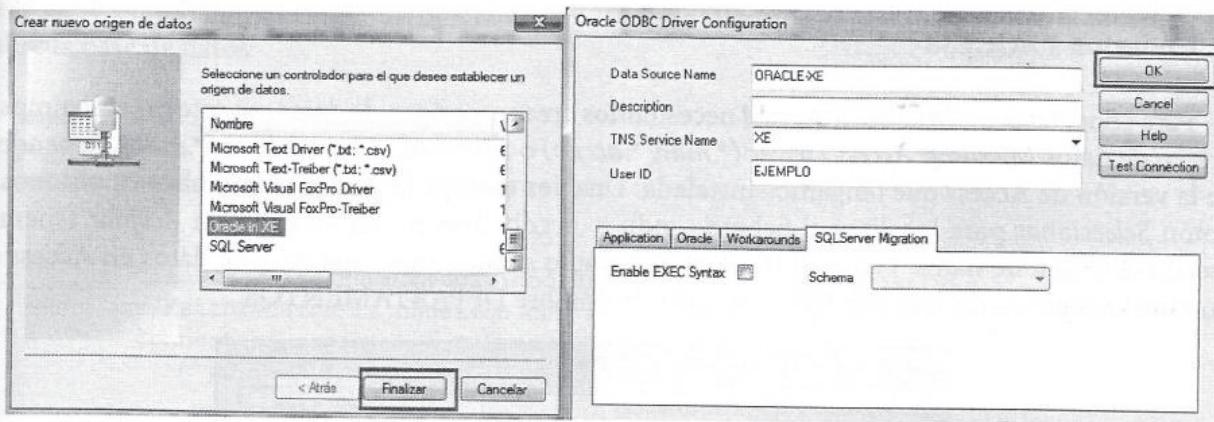


Figura 2.15. Origen de datos ODBC para acceder a Oracle-XE desde Windows.

Nos creamos la carpeta *oracle* dentro de *D:\db*, copiamos el programa Java y cambiamos la carga del driver, en este caso se llama *sun.jdbc.odbc.JdbcOdbcDriver* y la conexión a la base de datos:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conexion = DriverManager.getConnection
        ("jdbc:odbc:ORACLE-XE","ejemplo", "ejemplo");
```

En este caso para compilar y ejecutar el programa desde Windows escribimos: *javac Main.java* y *java Main*.

Para conectarnos 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/database/features/jdbc/index-091264.html>. Para el ejemplo se ha descargado el driver *classes12.jar* para la versión *Oracle Database 10g Release 2 (10.2.0.1.0)*.

Copiamos el JAR en la carpeta *D:\db\oracle* y cambiamos la carga del driver, en este caso se llama *oracle.jdbc.driver.OracleDriver*, y la conexión a la base de datos:

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

Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\oracle>javac Main.java
D:\db\oracle>java -classpath ".;classes12.jar" Main
```

Desde Linux nos vamos a la carpeta `/home/usuario/db/oracle/` copiamos el programa Java y el JAR y definimos la variable CLASSPATH:

```
$ export CLASSPATH=/home/usuario/db/oracle/:/home/usuario/db/oracle/classes12.jar
$ javac Main.java
$ java Main
```

Conexión a ACCESS

Para conectarnos a Microsoft Access necesitamos crear un origen de datos, en este caso elegimos el origen de datos *Microsoft Access Driver (*.mdb, *.accdb)* o *Microsoft Access Driver (*.mdb)*, dependerá de la versión de Access que tengamos instalada. Una vez elegido, le damos un nombre y pulsamos el botón *Seleccionar* para localizar el fichero .mdb o .accdb. Tras pulsar en el botón *Aceptar* tenemos creado el origen de datos, Figura 2.16. Para el ejemplo se ha creado una base de datos en Access de nombre *ejemplo.accdb*, que contiene una tabla de nombre DEPARTAMENTOS.

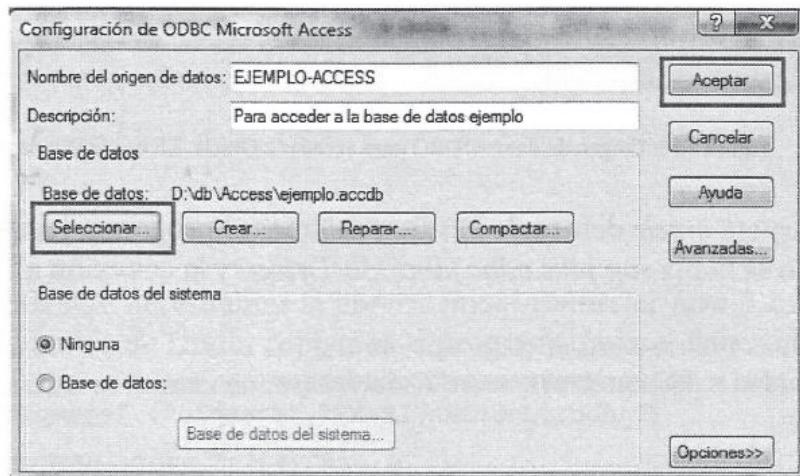


Figura 2.16. Origen de datos ODBC para acceder a una base de datos Access

Nos creamos la carpeta *Access* dentro de *D:\db*, copiamos el programa Java y cambiamos la carga del driver, en este caso se llama *sun.jdbc.odbc.JdbcOdbcDriver* y la conexión a la base de datos:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conexion = DriverManager.getConnection("jdbc:odbc:EJEMPLO-ACCESS");
```

Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\Access>javac Main.java
D:\db\Access>java Main
```

18 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.

El objeto **DatabaseMetaData** proporciona información sobre la base de datos a través de múltiples métodos de los cuales es posible obtener gran cantidad de información. El siguiente ejemplo conecta con la base de datos MySQL de nombre *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 clase *Connection* devuelve un objeto **DataBaseMetaData** con el que se obtendrá la información sobre la base de datos:

```
import java.sql.*;
public class Main {
    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(); //Creamos objeto
            ResultSet resul = null;

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

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

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

            while (resul.next()) {
                String catalogo = resul.getString(1); //columna 1 que devuelve ResultSet
                String esquema = resul.getString(2); //columna 2
                String tabla = resul.getString(3); //columna 3
                String tipo = resul.getString(4); //columna 4
                System.out.println(tipo + " - Catalogo: " + catalogo +
```

```

        ", Esquema : " + esquema + ", Nombre : " + tabla);
    }
    conexion.close(); //Cerrar conexion
}
catch (ClassNotFoundException cn) {cn.printStackTrace();}
catch (SQLException e) {e.printStackTrace();}
}//fin de main
}//fin de la clase

```

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. Necesita 4 parámetros que en el ejemplo anterior tenían el valor `null`:

- Primer parámetro: catálogo de la base de datos. El método obtiene las tablas del catálogo indicado, al poner `null`, indicamos todos los catálogos.
- Segundo parámetro: esquema de la base de datos. Obtiene las tablas del esquema indicado, el valor `null` indica el esquema actual (o todos los esquemas, dependiendo del SGBD, 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 guión bajo o porcentaje, por ejemplo “`de%`” obtendría todas las tablas cuyo nombre empiece por “`de`”.
- El cuarto parámetro es un array de `String`, en el que indicamos qué tipos de tablas queremos: TABLE (para tablas), VIEW (para vistas); al poner `null`, nos devolverá todos los tipos ya sean tablas o vistas. El siguiente ejemplo nos devolvería solo las tablas:

```

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

```

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

```

String catalogo  = resul.getString("TABLE_CAT");
String esquema    = resul.getString("TABLE_SCHEM");
String tabla      = resul.getString("TABLE_NAME");
String tipo       = resul.getString("TABLE_TYPE");

```

Actividad 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(catálogo, esquema, nombre_de_tabla, nombre_de_columna)* devuelve información sobre las columnas de una tabla o tablas. Para el nombre de la tabla y de la columna se puede utilizar el carácter guión 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 nombreCol = 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.println("Columna: " + nombreCol + ", Tipo: " + tipoCol +
        ", Tamaño: " + tamCol + ", ¿Puede ser Nula?:? " + nula);
}
```

Visualiza:

```
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(catálogo, esquema, tabla)* devuelve la lista de columnas que forman la clave primaria. 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(catalogo,esquema,tabla)* devuelve la lista de todas las claves ajenas que utilizan la clave primaria de esta tabla. El siguiente ejemplo muestra las tablas y sus claves ajenas que

referencian a la tabla DEPARTAMENTOS, en este caso solo la tabla EMPLEADOS (ejemplo en ORACLE):

```
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.println("Tabla PK: " + pk_tablename + ", Clave Primaria: " + pk_name);
    System.out.println("Tabla FK: " + fk_tablename + ", Clave Ajena: " + fk_name);
}
```

Visualiza (en Oracle):

```
Tabla PK: DEPARTAMENTOS, Clave Primaria: DEPT_NO
Tabla FK: EMPLEADOS, Clave Ajena: DEPT_NO
```

El método no devuelve nada si escribimos `dbmd.getExportedKeys(null, "EJEMPLO", "EMPLEADOS")` ya que la tabla EMPLEADOS no es referenciada por ninguna clave ajena.

- `getImportedKeys(catálogo, esquema, tabla)` devuelve la lista de claves ajenas existentes en la tabla. Se utiliza igual que el método anterior, en este caso `dbmd.getImportedKeys(null, "EJEMPLO", "EMPLEADOS")` devuelve la salida anterior, en cambio `dbmd.getImportedKeys(null, "EJEMPLO", "DEPARTAMENTOS")` no devuelve nada ya que no tiene claves ajena.
- `getProcedures(catálogo, esquema, procedure)` devuelve la lista de procedimientos almacenados. El siguiente ejemplo muestra los procedimientos y funciones que tiene el esquema de nombre EJEMPLO (ejemplo en ORACLE):

```
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.println("Nombre Procedimiento: " + proc_name +
        " - Tipo: " + proc_type);
}
```

Métodos de DatabaseMetaData:

<http://docs.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html>

2.8.1 RESULTSETMETADATA

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

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

```
Statement sentencia = conexion.createStatement();
ResultSet rs = sentencia.executeQuery("SELECT * FROM departamentos");
ResultSetMetaData rsmd = rs.getMetaData();
int nColumnas = rsmd.getColumnCount();
String nula;
System.out.println("Número de columnas recuperadas: " + nColumnas);
for (int i = 1; i <= nColumnas; i++) {
    System.out.println("Columna " + i + ":");
    System.out.println(" Nombre : " + rsmd.getColumnName(i));
    System.out.println(" Tipo : " + rsmd.getColumnTypeName(i));
    if (rsmd.isNullable(i) == 0) nula = "NO"; else nula = "SI";
    System.out.println(" Puede ser nula? : " + nula);
    System.out.println(" Máximo ancho de la columna: " + rsmd getColumnDisplaySize(i));
```

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 utilizados son:

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

Actividad 9: Visualiza información sobre las columnas de la tabla EMPLEADOS en SQLite.

2.9 EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS

En ejemplos anteriores vimos cómo 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:

- *executeQuery(String)*: se utiliza para sentencias SQL que recuperan datos de un único objeto *ResultSet*, se utiliza para las sentencias SELECT.
- *executeUpdate(String)*: se utiliza para sentencias que no devuelven un *ResultSet* como son las sentencias de manipulación de datos (DML): INSERT, UPDATE 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.
- *execute(String)*: se utiliza para sentencias que devuelven más de un *ResultSet*, se suele utilizar para ejecutar procedimientos almacenados.

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

MÉTODO	DEVUELVEN OBJETOS DE TIPO:
<i>getString(columna)</i>	<i>String</i>
<i>getBoolean(columna)</i>	<i>boolean</i>
<i>getByte(columna)</i>	<i>byte</i>
<i>getShort(columna)</i>	<i>short</i>
<i>getInt(columna)</i>	<i>int</i>
<i>getLong(columna)</i>	<i>long</i>
<i>getFloat(columna)</i>	<i>float</i>
<i>getDouble(columna)</i>	<i>double</i>
<i>getDate(columna)</i>	<i>Date</i>
<i>getTime(columna)</i>	<i>Time</i>

El siguiente ejemplo inserta un departamento en la tabla *departamentos*, los datos del nuevo departamento se introducen al ejecutar el programa desde la línea de comandos, el primer parámetro es el departamento, el siguiente el nombre y el tercero la localidad. Antes de ejecutar la orden INSERT construimos la sentencia en un String, las cadenas de caracteres 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= "INSERT INTO departamentos VALUES (" + dep + ","
            + dnombre + ",'" + loc + "')";
            System.out.println(sql);
            Statement sentencia = conexion.createStatement();
            int filas = sentencia.executeUpdate(sql);
            System.out.println("Filas afectadas: "+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 desde la línea de argumentos:

```

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

// construir orden UPDATE
String sql= "UPDATE empleados SET salario=salario + " + subida
            + " WHERE dept_no = " + dep;
System.out.println(sql);
Statement sentencia = conexion.createStatement();

```

```
int filas = sentencia.executeUpdate(sql);
System.out.println("Filas modificadas: "+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
Filas modificadas: 3
```

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

```
//construir orden CREATE
String sql= "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 " ;
System.out.println(sql);
Statement sentencia = conexion.createStatement();
int filas = sentencia.executeUpdate(sql);
System.out.println("Resultado ejecución: "+filas);
```

La ejecución muestra la siguiente información:

```
java CrearVista
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 ejecucion: 0
```

Actividad 10: Crea un programa Java que inserte un empleado en la tabla EMPLEADOS, el programa recibe desde la línea de argumentos 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.

La fecha de alta del empleado es la fecha actual.

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

2.9.1 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* (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 del ejemplo anterior se representaría 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 (*executeQuery()*, *executeUpdate()* y *execute()*) que en **Statement** pero no se necesita enviar la cadena de caracteres con la orden SQL en la llamada ya que lo hace el método *prepareStatement(String)*:

```
PreparedStatement sentencia = conexion.prepareStatement(sql);
```

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
```

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
```

Se utilizan los métodos `setInt(indice, entero)`, `setString(indice, cadena)` y `setFloat(indice, float)` para asignar los valores a cada uno de los marcadores de posición. Con `Integer.parseInt(dep)` convertimos la cadena `dep` a entero y con `Float.parseFloat(subida)` convertimos la cadena `subida` a 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 la línea de comandos al ejecutar el programa:

```
//recuperar parametros de main
String dep=args[0];           //departamento
String oficio=args[1]; //oficio
//construir 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.println (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 muestran los vendedores del departamento 30:

```
java VerEmpleado 30 VENDEDOR
ARROYO => 1500.0
MARTÍN => 1600.0
SALA => 1625.0
TOVAR => 1350.0
```

Algunos de los métodos de la interfaz `PreparedStatement` para dar valor a los marcadores de posición son:

MÉTODO
<code>void setString(indice, String)</code>
<code>Void setBoolean(indice, boolean)</code>
<code>void setByte(indice, byte)</code>
<code>void setShort(indice, short)</code>
<code>void setInt(indice, int)</code>
<code>void setLong(indice, long)</code>
<code>void setFloat(indice, float)</code>

void setDouble(indice, double)
void setDate(indice, Date)
void setTime(indice, Time)
void setNull(indice, int tipoSQL)

Los valores NULL se insertan en la base de datos usando *setNull()*.

Actividad 11: Utiliza la interfaz **PreparedStatement** para visualizar el APELLIDO, SALARIO y OFICIO de los empleados de un departamento cuyo valor se recibe desde la línea de argumentos. Visualiza también el nombre de 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.

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 sueldo 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; si el departamento no existe devuelve "INEXISTENTE":

Función en ORACLE:

```

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;
/
```

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

```

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

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

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

En el siguiente ejemplo se realiza una llamada al procedimiento *subida_sal* (de MySQL); los valores de los parámetros se introducen al ejecutar el programa desde la línea de comandos:

```

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

```

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

//recuperar parametros de main
String dep=args[0]; //departamento
String subida=args[1]; //subida
//construir orden DE LLAMADA
String sql= "{ call subida_sal (?, ?) } ";

// Preparamos la llamada
CallableStatement llamada = conexion.prepareCall(sql);
// Damos valor a los argumentos
llamada.setInt(1, Integer.parseInt(dep));      // primer argumento-dep
llamada.setFloat(2, Float.parseFloat(subida)); // segundo argumento-subida

llamada.executeUpdate(); //ejecutar el procedimiento
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 puede ocurrir que el usuario (en este caso se llama *ejemplo*) no tenga 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';

```

También es necesario incluir en la conexión el parámetro *noAccessToProcedureBodies* con el valor *true*: "jdbc:mysql://localhost/ejemplo?noAccessToProcedureBodies=true".

Cuando un procedimiento o función tiene parámetros de salida (OUT) deben ser registrados antes de que la llamada tenga lugar, si no se registra se producirá un error. El método que se utilizará es: *registerOutParameter(indice, tipoJDBC)*, el primer parámetro es la posición y el siguiente es una constante definida en la clase *java.sql.Types*. La clase *Types* define una constante para cada tipo genérico SQL, algunas son: TINYINT, SMALLINT, INTEGER, FLOAT, REAL, DOUBLE, NUMERIC,

DECIMAL, CHAR, VARCHAR, DATE, TIME, TIMESTAMP, JAVA_OBJECT, ARRAY, BOOLEAN, ROWID etc. Por ejemplo, si el segundo parámetro de un procedimiento es OUT y de tipo VARCHAR en la base de datos se pondría:

```
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(indice)` 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 la línea de comandos 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 (?, ?) } ";

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

            llamada.registerOutParameter(1,Types.VARCHAR); //valor devuelto

            llamada.setInt(2,Integer.parseInt(dep));           //param de entrada
            llamada.registerOutParameter(3,Types.VARCHAR); //parametro OUT

            llamada.executeUpdate(); //ejecutar el procedimiento
            System.out.println ("Nombre Dep: "+llamada.getString(1) +
                " Localidad: "+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 12: Crea una función en Oracle que reciba un número de departamento y devuelva su nombre si existe y si no existe devuelva INEXISTENTE. Una vez realizada la función modifica la Actividad 11 para que el nombre del departamento se obtenga llamando a dicha función.

2.11 GESTIÓN DE ERRORES

Hasta ahora en todos los ejemplos cuando se producía un error se visualizaba con `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:

```
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Table 'ejemplo.departamento' doesn't exist
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Source)
    at java.lang.reflect.Constructor.newInstance(Unknown Source)
    at com.mysql.jdbc.Util.handleNewInstance(Util.java:411)
    at com.mysql.jdbc.Util.getInstance(Util.java:386)
    at com.mysql.jdbc.SQLException.createSQLException(SQLException.java:1052)
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3609)
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3541)
    at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:2002)
    at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:2163)
    at com.mysql.jdbc.ConnectionImpl.execSQL(ConnectionImpl.java:2618)
    at com.mysql.jdbc.StatementImpl.executeUpdate(StatementImpl.java:1749)
    at com.mysql.jdbc.StatementImpl.executeUpdate(StatementImpl.java:1666)
    at InsertarDep.main(InsertarDep.java:20)
```

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

- `getMessage()`: devuelve una cadena que describe el error.
- `getSQLState()`: es una cadena que contiene un estado definido por el estándar X/OPEN SQL.
- `getErrorCode()`: es un entero que proporciona el código de error del fabricante. Normalmente, este será el código de error real devuelto por la base de datos.

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

```

try
{
    //Código
}
catch (ClassNotFoundException cn) {cn.printStackTrace();}
catch (SQLException e)
{
    System.out.println("HA OCURRIDO UNA EXCEPCIÓN:");
    System.out.println("Mensaje: " + e.getMessage());
    System.out.println("SQL estado: " + e.getSQLState());
    System.out.println("Cód error: " + e.getErrorCode()); }
```

El siguiente ejemplo 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.departamentoss' 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
```

2.12 PATRÓN MODELO-VISTA-CONTROLADOR. ACCESO A DATOS

El patrón MVC (*Model-View-Controller*) es un patrón de diseño que se utiliza como guía para el diseño de arquitecturas software que ofrecen una fuerte interactividad con el usuario y donde se requiere una separación de conceptos para que el desarrollo se realice más eficazmente facilitando la programación en diferentes capas de manera paralela e independiente. Este patrón organiza la aplicación en 3 bloques cada cual especializado en una tarea:

- **El Modelo:** representa los datos de la aplicación y sus reglas de negocio.
- **La Vista:** es la representación del modelo de forma gráfica para interactuar con el usuario, ejemplo son los formularios de entrada y salida de información o páginas HTML con contenido dinámico.
- **El Controlador:** interpreta los datos que recibe del usuario analizando la petición, coordinando la vista y el modelo para que la aplicación produzca los resultados esperados.

Las ventajas de hacer uso de este patrón son:

- Separación de los datos de la representación visual de los mismos.
- Diseño de aplicaciones modulares.

- Reutilización de código.
- Facilidad para probar las unidades por separado.
- Facilita el mantenimiento y la detección de errores.

Entre las desventajas cabe destacar la complejidad que se agrega al sistema al separar los conceptos en capas o la cantidad de ficheros a desarrollar que se incrementa considerablemente.

La Figura 2.17 describe el flujo general de la solicitud de un usuario construida en una arquitectura MVC, como se puede observar, el controlador es el que dirige la aplicación. Los pasos son los siguientes:

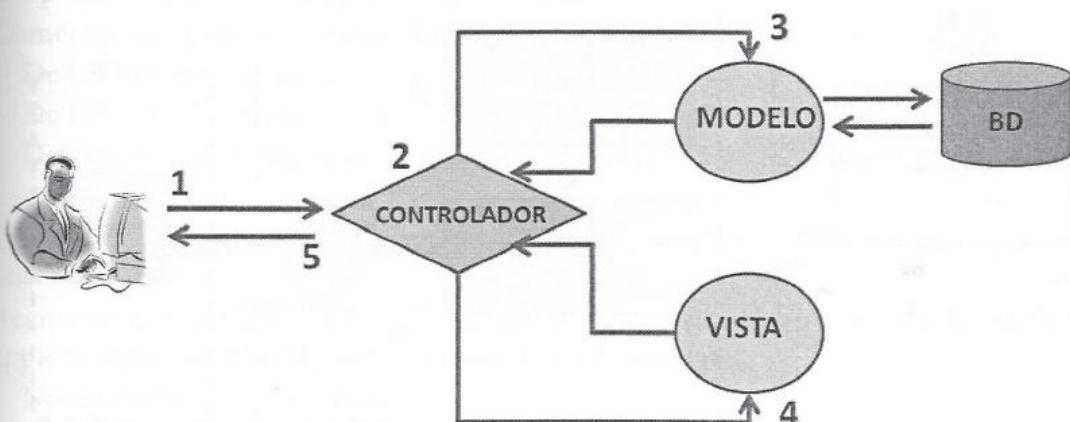


Figura 2.17. Flujo de solicitud en el MVC

1. Un usuario realiza una solicitud a través de la aplicación (pulsa un botón, un enlace,). La solicitud es dirigida al controlador.
2. El controlador examina la solicitud y decide qué regla de negocio aplicar, es decir, determinará el componente de negocio a aplicar para procesar la solicitud, este componente de negocio es el modelo.
3. El modelo contiene las reglas de negocio que procesan la solicitud y que dan lugar al acceso a los datos que necesita el usuario. Estos datos se devuelven al controlador.
4. El controlador toma los datos que devuelve el modelo y selecciona la vista en la que se van a presentar esos datos al usuario.
5. El controlador devuelve los resultados al usuario tras procesar la solicitud.

El patrón MVC es utilizado en múltiples frameworks: Java Enterprise Edition (J2EE), Apache Struts (para aplicaciones web J2EE), Ruby on Rails (para aplicaciones web con Ruby), Google Web Toolkit (GWT, para crear aplicaciones Ajax con Java), ASP.NET MVC Framework (Microsoft), etc. La mayoría de los frameworks para web implementan este patrón. Una aplicación de este patrón en aplicaciones web J2EE es lo que se conoce con el nombre de **Modelo 2**. Esta arquitectura se basa en los siguientes elementos:

- La utilización de Servlets para capturar las peticiones realizadas por el cliente web y redirigirlas a las páginas JSP adecuadas. Estos actúan como controlador.

- Páginas JSP utilizadas para mostrar la interfaz de usuario, implementan la vista. Estas páginas usan los JavaBeans como componente modelo.
- JavaBeans o POJOs (*Plain Old Java Object*) que implementan el modelo.

A continuación se muestra un ejemplo de una aplicación web MVC Java. La arquitectura **Modelo 2** de nuestra aplicación se muestra en la Figura 2.18. Para poder trabajar con ella necesitaremos instalar un contenedor web con soporte para Servlets y JSPs, en este ejemplo se realizará la aplicación sobre Tomcat.

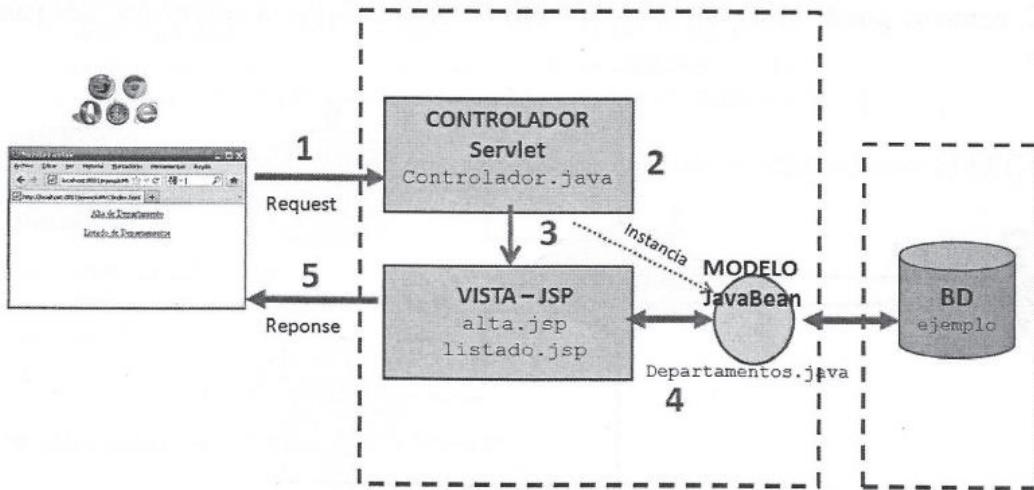


Figura 2.18. Arquitectura Modelo 2.

Se puede descargar la última versión desde la web <http://tomcat.apache.org/>, para el ejemplo se ha descargado la versión *apache-tomcat-7.0.23.zip* (*apache-tomcat-7.0.23.tar.gz* en Linux). La instalación es sencilla, solo hay que descomprimir el fichero que contiene una única carpeta de nombre *apache-tomcat-7.0.23*. Para iniciar el servidor se ejecuta el fichero *startup.bat* en sistemas Windows o *startup.sh* en Linux que se encuentran en la carpeta *\apache-tomcat-7.0.23\bin*.

Antes de empezar con la aplicación introducimos una serie de conceptos sobre los Servlets y las páginas JSP. Los Servlets son clases Java que no tienen el método *main()*, en su lugar se invocan otros métodos cuando se reciben las peticiones. Los métodos son: *init()*, *service()* y *destroy()*. El ciclo de vida de un Servlet se divide en varios pasos:

- El cliente solicita una petición a un servidor a través de una URL. El servidor recibe la petición.
- Si es la primera vez, el servidor carga el Servlet y se llama al método *init()* para iniciararlo.
- Se llama al método *service()* para procesar las peticiones de los clientes web.
- Se llama al método *destroy()* para eliminar al Servlet y liberar los recursos. El servidor los destruye porque cesan las llamadas desde el cliente, un temporizador del servidor así lo indica o el propio administrador lo decide.

Las páginas JSP nos permiten generar contenido dinámico en la web, son páginas web con etiquetas especiales y código Java incrustado. La diferencia con el Servlet es que en este el código es Java puro que recibe peticiones y genera una página web a partir de ellas. Una página JSP consta de 2 partes:

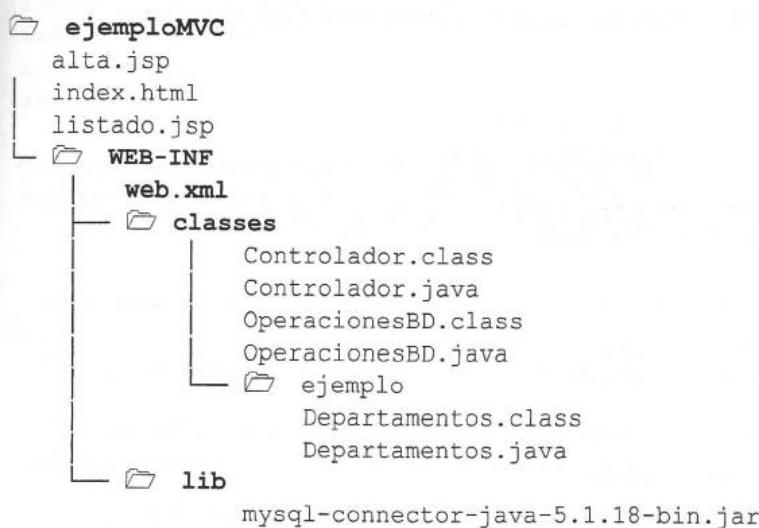
- HTML o XML para el contenido estático.

- Etiquetas JSP y scriplets escritos en lenguaje de programación Java para encapsular la lógica que genera el contenido dinámico.

El código de las partes dinámicas se encierra entre unas etiquetas especiales, la mayoría de las cuales empiezan con “`<%`” y terminan con “`%>`”. Algunos elementos JSP son:

- Declaraciones JSP: dentro de las etiquetas `<%! código Java %>`
- Expresiones: dentro de las etiquetas `<%= código Java %>`
- Scriptlets: dentro de las etiquetas `<% código Java %>`
- Comentarios: se tienen los siguientes tipos:
 - De HTML: `<!-- comentario -->`
 - De JSP: `<%-- comentario --%>`
 - Del lenguaje de script Java: `<%// comentario línea %>` y `<%/* comentario varias líneas */%>`
- Directivas: dentro de las etiquetas `<%@ ... %>`, por ejemplo: `<%@ page import = "ejemplo.*; java.util.*" %>`.
- Acciones: que permiten trabajar con componentes complementarios a la página JSP como applets, otras páginas JSP, javabeans, etc. Son las siguientes:
 - No asociadas a los javabeans:
 - `<jsp:include ... </jsp:include>` o `<jsp:include ... />` si solo tiene atributos
 - `<jsp:plugin ... </jsp:plugin>`
 - `<jsp:forward ... </jsp:forward>` o `<jsp:forward ... />` si solo tiene atributos
 - Asociadas a los javabeans:
 - `<jsp:useBean ... />` si solo tiene atributos o `<jsp:useBean ... > ... </jsp:useBean>`
 - `<jsp:setProperty ... />`
 - `<jsp:getProperty ... />`

Nuestra aplicación web estará formada por una serie de ficheros JSP, HTML, Java, XML y librerías JAR, la estructura de directorios y ficheros es la siguiente:



Esta estructura estará dentro de la carpeta *webapps* (*D:\apache-tomcat-7.0.23\webapps\ejemploMVC*). Hay algunos ficheros y carpetas con un significado especial:

- Carpeta **WEB-INF**: es imprescindible y su nombre debe aparecer siempre en mayúsculas. Contiene las siguientes carpetas y ficheros:
 - Carpeta **clases**: es necesaria si usamos ficheros *.class*. Contiene los paquetes de nuestras clases Java, reproduciendo la estructura de paquetes y subpaquetes. Es dentro de este directorio donde generalmente residen los Servlets (en el ejemplo por simplificar solo hay un paquete de nombre *ejemplo* donde está el JavaBean y 2 clases).
 - Carpeta **lib**: contiene todos los JAR que necesita la aplicación, en el ejemplo se necesita el conector Java para acceder a la base de datos MySQL.
 - Fichero **web.xml**: es el descriptor de despliegue de la aplicación e indica la ubicación de los servlets (mapeo) contenidos en la aplicación (en el ejemplo solo hay un Servlet que es *Controlador.java*). Puede contener otros parámetros para componentes adicionales y manejo de errores.

La aplicación visualizará una página web inicial (**index.html**) con dos enlaces, uno para realizar el alta de un departamento y el otro para realizar el listado de todos. El contenido del fichero es el siguiente:

```
<html>
<body>
  <p align='center'>
    <a href="/ejemploMVC/controlador?accion=alta">Alta de Departamento</a></p>
  <p align='center'>
    <a href="/ejemploMVC/controlador?accion=listado">Listado de Departamentos</a>
  </p>
</body>
</html>
```

Desde los enlaces *href* se llama al controlador con la acción a realizar alta o listado. Para poder llevar a cabo esta acción los contenedores web tienen un fichero llamado **web.xml** que se encarga de mapear la URL. El contenedor web (Tomcat) lee el documento **web.xml** y realiza el mapeo entre el alias encontrado en el path de la URL y el Servlet en cuestión. El contenido del fichero es:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0"
  metadata-complete="true">
<servlet>
  <servlet-name>Controlador</servlet-name>
  <servlet-class>Controlador</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Controlador</servlet-name>
  <url-pattern>/controlador</url-pattern>
</servlet-mapping>
```

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

Nos fijamos en los elementos `<servlet>` y `<servlet-mapping>`:

El elemento `<servlet>` define las características de un Servlet. Está compuesto por los elementos `<servlet-name>` donde se indica el nombre del Servlet (en el ejemplo *Controlador*) y `<servlet-class>` que indica el nombre de la clase Java que contiene el Servlet (*nombredirectorio.ficheroclase*, en el ejemplo el fichero *Controlador.class* no está dentro de ningún directorio (o paquete), por ello se escribe en el elemento solo *Controlador*).

El elemento `<servlet-mapping>` define la ubicación en términos de directorios de un sitio (URL). El Servlet de nombre *Controlador* (`<servlet-name>Controlador</servlet-name>`) será accedido cada vez que se acceda a la URL */controlador* (`<url-pattern>/controlador</url-pattern>`). `<url-pattern>` indica la forma en que se debe invocar al Servlet, en el documento HTML poníamos lo siguiente para invocar al Servlet: ``, además se le envía el parámetro *accion*.

Controlador.java: es el Servlet controlador que maneja todas las solicitudes entrantes. Recibe un parámetro de nombre *accion*. Dependiendo del valor de este parámetro podrá realizar varias acciones: redirigir la petición a *alta.jsp*, redirigir la respuesta a *listado.jsp* enviando los departamentos a *listar* e invocar al método para la inserción de un departamento (cuyos datos se reciben de *alta.jsp*) en la base de datos y después redirigir la respuesta a *index.html*. Antes de realizar el listado carga los datos de la tabla DEPARTAMENTOS en un array, para ello utilizan el método *listarDep()* de la clase OperacionesBD. Los datos para insertar un departamento los recibe de la página *alta.jsp* mediante el atributo *depart*, la inserción la realiza invocando al método *insertaDepartamento()* de la clase OperacionesBD:

```
import ejemplo.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class Controlador extends HttpServlet {
    public void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        //parámetro acción, se obtiene de la URL de index.html, puede ser 'alta' o
        'listado'
        String op=request.getParameter("accion");

        //si se ha pulsado en alta de departamento se visualiza la pantalla de alta
        if(op.equals("alta")) response.sendRedirect("alta.jsp");

        //si se ha pulsado en listado, primero se cargan los datos de departamentos en
```

```

//una lista y luego se envian a listado.jsp
if(op.equals("listado")){
    OperacionesBD operBD=new OperacionesBD();
    ArrayList lista=operBD.listarDep(); //se cargan los datos de los dep
    request.setAttribute("departamentos",lista); //se preparan para enviar al jsp
    RequestDispatcher rd=request.getRequestDispatcher("/listado.jsp");
    rd.forward(request,response);
}

//se inserta departamento en la tabla y luego se visualiza index.html
if(op.equals("insertar")){
    ejemplo.Departamentos dep=
        (ejemplo.Departamentos)request.getAttribute("depart"); //obtener deps
    OperacionesBD operBD=new OperacionesBD();
    operBD.insertaDepartamento(dep); //se insertan en tabla departamentos
    response.sendRedirect("index.html"); //se muestra la página inicial
}
}//fin de la clase Controlador

```

En el ejemplo solo se ha utilizado el método `service()` del Servlet. Este acepta 2 argumentos:

- `HttpServletRequest request`: en este argumento se recibe la petición del cliente.
- `HttpServletResponse response`: es la respuesta que da el servidor al cliente.

Algunos métodos usados en la petición (objeto `HttpServletRequest`) son:

- `request.getParameter("nombre_de_parametro")`: lee el parámetro enviado por el cliente, devuelve el valor del parámetro especificado o null si no existe.
- `request.setAttribute("nombre", objeto)`: se almacena un objeto en la sesión con el nombre indicado.
- `request.getAttribute("nombre")`: devuelve el atributo asociado al nombre especificado en esta sesión o null si no hay ningún objeto asociado bajo el nombre.

Algunos métodos usados en la respuesta al cliente (objeto `HttpServletResponse`) son:

- `response.sendRedirect("url")`: la respuesta al cliente es la url indicada, puede ser un fichero JSP, un Servlet o una página HTML.

La interfaz `RequestDispatcher` encapsula una referencia a otro recurso web. El método `getRequestDispatcher("url")` acepta una ruta de URL que haga referencia al recurso objetivo, la ruta debe ser absoluta, es decir, el nombre debe empezar por /. El método `forward(request,response)` permite reenviar la solicitud a otro servlet, página JSP o fichero HTML.

alta.jsp: Visualiza un formulario para realizar la entrada de datos de un departamento. Los datos del formulario se enviarán a través del JavaBean (el modelo) `Departamentos.java` al controlador; el nombre del parámetro que contendrá los datos del departamento a insertar es `depart (id="depart")` y se usará en el controlador con el método `getAttribute()` (`request.getAttribute("depart")`):

```

<html>
<head>
<title>ALTA DE DEPARTAMENTOS</title>
</head>
<!--Form de entrada de datos e Inserción en el JavaBean-clase Departamentos-->
<jsp:useBean id="depart" scope="request" class="ejemplo.Departamentos" />
<jsp:setProperty name="depart" property="*"/>
<%
    if(request.getParameter("deptno")!= null) {%>
<jsp:forward page="/controlador?accion=insertar"/>
<%}>
</body>
<center><h2>ENTRADA DE DATOS DE DEPARTAMENTOS</h2>
<br>
<form method="post">
    <p>Número de departamento: <input name="deptno" type="text" size="5"></p>
    <p>Nombre: <input name="dnombre" type="text" size="15"> </p>
    <p>Localidad: <input name="loc" type="text" size="15"> </p>
    <input type="submit" name="insertar" value="Insertar departamento.">
    <input type="reset" name="cancelar" value="Cancelar entrada.">
</form>
</center>
</body>
</html>

```

Un JavaBean es una clase Java que nos permite ocultar su implementación mostrando al exterior solo los métodos y propiedades públicos. Para acceder al Bean desde una página JSP se utilizan una serie de etiquetas, algunas son: `<jsp:useBean>`, `<jsp:setProperty>`, `<jsp:getProperty>`, `<jsp:forward>`. En el ejemplo se han usado:

- `<jsp:useBean id="depart" scope="request" class="ejemplo.Departamentos" />`: *id* indica el nombre del Bean, *scope* es el alcance del Bean, un alcance "request" implica que el bean es accesible hasta otra JSP que haya sido invocada por medio de `jsp:forward` o `jsp:include`. El atributo *class* es el nombre de la clase.
- `<jsp:setProperty name="depart" property="*"/>`: el atributo *name* debe ser igual al especificado en *id*, con *property="*"* indicamos que se van a extraer todos los valores de la solicitud (*deptno*, *dnombre* y *loc*).
- `<jsp:forward page="/controlador?accion=insertar"/>`: permite que la solicitud sea enviada a otra página JSP, a un Servlet o a un recurso estático. En este caso se envia al controlador con el parámetro *accion* con valor *insertar* para que inserte los datos del Bean en la tabla.

Departamentos.java: es el JavaBean que facilita el intercambio de datos entre el controlador y el modelo y posteriormente entre el controlador y la vista:

```

package ejemplo;
public class Departamentos {
    private byte deptno;
    private String dnombre;
    private String loc;
}

```

```

public Departamentos() {}
public Departamentos(byte deptno, final String dnombre, final String loc) {
    this.deptno = deptno;
    this.dnombre = dnombre;
    this.loc = loc;
}
public byte getDeptno() {return this.deptno;}
public void setDeptno(byte deptno) {this.deptno = deptno;}
public String getDnombre() {return this.dnombre;}
public void setDnombre(String dnombre) {this.dnombre = dnombre;}
public String getLoc() {return this.loc;}
public void setLoc(String loc) {this.loc = loc; }
}//fin clase Departamentos

```

listado.jsp: Obtiene el listado de los departamentos, es llamado por el controlador y recibe el atributo *departamentos* conteniendo un array con los departamentos a listar:

```

<%@ page import="ejemplo.* ,java.util.*"%>
<html><head><title>LISTADO DE DEPARTAMENTOS</title></head>
<body>
<center>
<h2>LISTADO DE DEPARTAMENTOS</h2>
<table border='1'>
<tr><th>Departamento</th><th>Nombre</th><th>Localidad</th></tr>
<%
ArrayList listadep=(ArrayList)request.getAttribute("departamentos");
if(listadep!=null)
for(int i=0;i<listadep.size();i++){
    Departamentos d=(Departamentos)listadep.get(i);%
    <tr><td><%=d.getDeptno()%></td>
    <td><%=d.getDnombre()%></td>
    <td><%=d.getLoc()%></td>
    </tr>
<%}>
</table><br/><br/>
<a href="index.html">Inicio</a>
</center>
</body>
</html>

```

OperacionesBD.java: es la clase que realiza las operaciones contra la base de datos, la utiliza el controlador:

```

import ejemplo.*;
import java.sql.*;
import java.util.*;

public class OperacionesBD {
    //OBTENER LA CONEXIÓN
    public Connection getConnection(){
        Connection conexion=null;
        try{
            Class.forName("com.mysql.jdbc.Driver");//Cargar el driver

```

```

conexion = DriverManager.getConnection
("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");
}
catch(Exception e){e.printStackTrace();}
return conexion;
}

//LISTAR - Devuelve un array con la lista de departamentos
public ArrayList listarDep(){
    ArrayList departamentos = new ArrayList ();
    try{
        Connection conexion =getConnection();
        Statement sentencia=conexion.createStatement();
        String sql= "SELECT * FROM departamentos";
        ResultSet resul=sentencia.executeQuery(sql);
        while(resul.next()){ //se crea un array con los datos de los
                            departamentos
            ejemplo.Departamentos d=new ejemplo.Departamentos
            resul.getByte("dept_no"),
            resul.getString("dnombre"),resul.getString("loc"));

            departamentos.add(d); //añadir dep al array
        }
        conexion.close();
    }
    catch(Exception e){e.printStackTrace();}
    return(departamentos);
}//fin listarDep

//INSERTAR - Recibe los datos del departamento a insertar en la tabla
public void insertaDepartamento(ejemplo.Departamentos d){
    try{
        Connection conexion=getConnection();
        Statement sentencia=conexion.createStatement();
        String sql = "INSERT INTO departamentos VALUES('"
                    +d.getDeptno()+"','"+d.getDnombre()+"','"+d.getLoc()+"')";

        if (d.getDeptno () != 0) sentencia.execute(sql);
        System.out.println("SQL: "+sql);
        conexion.close();
    }
    catch(Exception e){e.printStackTrace();}
}//fin insertaDepartamento
}//fin clase OperacionesBD

```

Para compilar las clases seguimos estos pasos:

1.Nos dirigimos a la carpeta donde están las clases:

CD D:\apache-tomcat-7.0.23\webapps\ejemploMVC\WEB-INF\classes

2.Definimos la variable CLASSPATH:

SET CLASSPATH=D:/apache-tomcat-7.0.23/webapps/ejemploMVC/WEB-INF/classes/

```
ejemplo;/D:/apache-tomcat-7.0.23/webapps/ejemploMVC/WEB-INF/lib/mysql-connector-java-5.1.18-bin.jar;D:/apache-tomcat-7.0.23/lib/servlet-api.jar;D:/apache-tomcat-7.0.23/webapps/ejemploMVC/WEB-INF/classes/;
```

3. Compilamos la clase Departamentos que está en la carpeta de nombre *ejemplo*:

```
javac ejemplo/Departamentos.java
```

4. Compilamos las otras dos clases:

```
javac -Xlint OperacionesBD.java
```

```
javac Controlador.java
```

Una vez compiladas será necesario recargar la aplicación desde Tomcat. Para ello abrimos el navegador y escribimos la URL: <http://localhost:8080/>. Puede ocurrir que el servidor no se inicie (al ejecutar el fichero *startup.bat* o *startup.sh*) porque el puerto 8080 esté ocupado, en este caso cambiamos el puerto 8080 por otro que no se esté utilizando, por ejemplo 8081. Este cambio se hace en el fichero *D:\apache-tomcat-7.0.23\conf\server.xml* (línea 70). También será necesario crear un usuario para manejar el entorno web de Tomcat. Añadimos al fichero *D:\apache-tomcat-7.0.23\conf\tomcat-users.xml* las siguientes líneas entre las etiquetas *<tomcat-users>* *</tomcat-users>*:

```
<role rolename="manager-gui"/>
<user username="admin" password="admin" roles="manager-gui"/>
```

Donde indicamos que el usuario *admin* con clave *admin* tendrá acceso a la gestión de las aplicaciones web de Tomcat. Hecho esto, iniciamos de nuevo el servidor y escribimos la URL <http://localhost:8081/>, a continuación pulsamos el botón *Manager App*; nos pedirá un nombre de usuario y su clave, escribimos *admin*. Desde la pantalla de gestión de aplicaciones de Tomcat recargamos nuestra aplicación */ejemploMVC*, véase Figura 2.19.

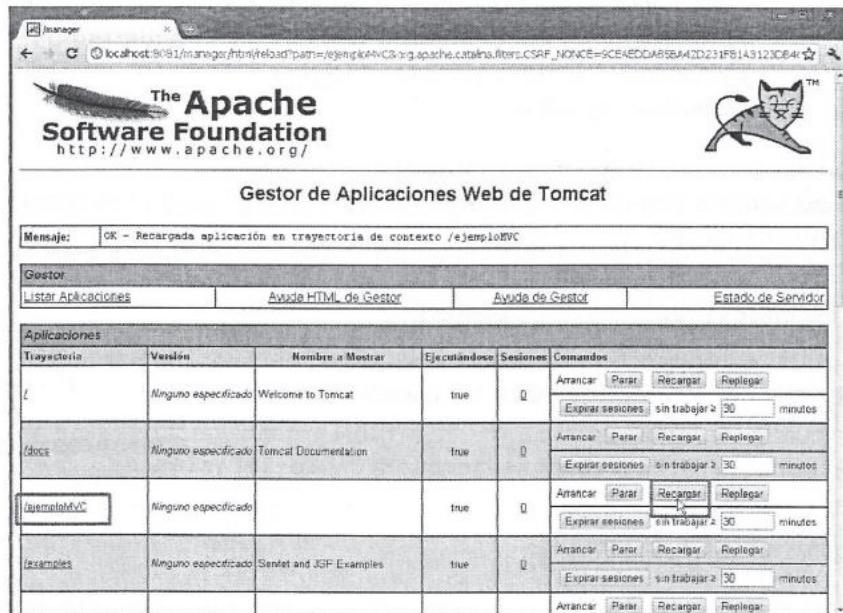


Figura 2.19. Recarga de la aplicación desde Tomcat.