UD_2: MANEJO DE CONECTORES

Contenido

2.1 Introducción

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 **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 b**ases de datos relacionales**, ya que solucionan las necesidades de aplicaciones más sofisticadas que requieren el tratamiento de elementos más complejos.
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 son los Objetos.

En este sentido, las bases de datos relacionales no están diseñadas para almacenar estos objetos. 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 de las estructuras provistas en el modelo de datos a las provistas por el entorno de programación**. Este tema se tratará más ampliamente en la Unidad 3 (Herramientas de Mapeo Objeto-Relacional).

2.3. Bases de datos embebidas

Cuando desarrollamos pequeñas aplicaciones donde no vamos a tener que manejar una gran cantidad de datos o información, no es necesario que utilicemos un sistema gestor de bases de datos, (Oracle, mySQL…). En su lugar podemos usar una BD embebida- En ella el motor está incrustado en la misma aplicación y su uso es exclusivo para ella.

La base de datos se inicia cuando se arranca la aplicación y finaliza cuando se cierra la misma.

Generalmente son de código abierto (Open Source), pero existen algunas de origen propietario.

2.3.1 SQLite

Sistema gestor de bases de datos multiplataforma, está escrito en C y tiene un motor ligero. Las bases de datos se guardan como ficheros, con esto se facilita el traslado de las bases de datos con la aplicación con la que se usa. Cuenta con una utilidad que nos permitirá ejecutar

comandos SQL en modo consola. Es un proyecto de dominio público. Se puede usar desde programas en C, C++, PHP, Visual Basic, Java, Perl,…

Su instalación es sencilla. Desde la página https://www.sqlite.org/download.html se puede descargar. Dependiendo del sistema operativo descargaremos un ZIP u otro:

- sqlite-tools-win32-x86-3330000.zip para **Windows**. *A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff.exe program, and the sqlite3_analyzer.exe program.*
- sqlite-tools-linux-x86-3330000.zip para **Linux**. *A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff program, and the sqlite3_analyzer program.*
- sqlite-tools-osx-x86-3330000.zip para **Mac OS X**. *A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff program, and the sqlite3_analyzer program*.

Para completar la instalación en Windows ejecutaremos el instalador **sqlite3.exe**. Al ejecutarlo desde la línea de comandos escribimos el nombre del fichero que contendrá la base de datos, si el fichero no existe se creará, si existe cargará la base de datos.

El siguiente ejemplo crea la base de datos ejemplo.db (en la carpeta D:\DB\SQLITE), todas las tablas que creemos en esta sesión se almacenarán en este fichero,

```
D:\>sqlite3 D:\DB\SQLITE\ejemplo.db
SQLite version 3.11.1 2016-03-03 16:17:53
Enter ".help" for usage hints.
sqlite> CREATE TABLE departamentos (
   ...>  dept_no  TINYINT(2) NOT NULL PRIMARY KEY,
   ...>  dnombre  VARCHAR(15),
   ...>  loc      VARCHAR(15)
   ...> );
sqlite> INSERT INTO departamentos VALUES (10,'CONTABILIDAD','SEVILLA');
sqlite> INSERT INTO departamentos VALUES (20,'INVESTIGACIÓN','MADRID');
sqlite> INSERT INTO departamentos VALUES (30,'VENTAS','BARCELONA');
sqlite> INSERT INTO departamentos VALUES (40,'PRODUCCIÓN','BILBAO');
sqlite> .tables
departamentos
sqlite> SELECT * FROM departamentos;
10|CONTABILIDAD|SEVILLA
20|INVESTIGACIÓN|MADRID
30|VENTAS|BARCELONA
40|PRODUCCIÓN|BILBAO
sqlite> .quit

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

ACTIVIDAD 2.1

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

DEPARTAMENTOS:

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

EMPLEADOS:

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

Los datos a insertar pueden ser los siguientes:

Para los departamentos:

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

Para los empleados:

```
7369,'SÁNCHEZ','EMPLEADO',7902,'1990/12/17',1040,NULL,20
7499,'ARROYO','VENDEDOR',7698,'1990/02/20',1500,390,30
7521,'SALA','VENDEDOR',7698,'1991/02/22',1625,650,30
7566,'JIMÉNEZ','DIRECTOR',7839,'1991/04/02',2900,NULL,20
7654,'MARTÍN','VENDEDOR',7698,'1991/09/29',1600,1020,30
7698,'NEGRO','DIRECTOR',7839,'1991/05/01',3005,NULL,30
7782,'CEREZO','DIRECTOR',7839,'1991/06/09',2885,NULL,10
7788,'GIL','ANALISTA',7566,'1991/11/09',3000,NULL,20
7839,'REY','PRESIDENTE',NULL,'1991/11/17',4100,NULL,10
7844,'TOVAR','VENDEDOR',7698,'1991/09/08',1350,0,30
7876,'ALONSO','EMPLEADO',7788,'1991/09/23',1430,NULL,20
7900,'JIMENO','EMPLEADO',7698,'1991/12/03',1335,NULL,30
7902,'FERNÁNDEZ','ANALISTA',7566,'1991/12/03',3000,NULL,20
7934,'MUÑOZ','EMPLEADO',7782,'1992/01/23',1690,NULL,10
```

2.3.2 Apache Derby

Es una base de datos relacional de código abierto, implementado totalmente en Java. Está disponible bajo la licencia Apache 2.0. Algunas ventajas de esta base de datos son su tamaño reducido, que esté basada en Java y soporte los estándares SQL, soporta el paradigma cliente-servidor, es fácil de usar, instalar e implementar.

Para realizar la instalación descargamos la última versión desde la página web https://db.apache.org/derby/derby_downloads.html .

Descargamos el fichero db-derby-10.12.1.1-bin.zip y lo descomprimimos por ejemplo en D:\db-derby-10.12.1.1-bin. A partir de ahora para poder utilizar Derby en nuestros programas Java, solo será necesario tener accesible la librería **derby.jar** en el CLASSPATH de nuestro programa o en nuestro proyecto de Eclipse o Netbeans.

Apache Derby trae una serie de ficheros .BAT que nos permitirán ejecutar por consola órdenes para crear nuestras bases de datos y ejecutar sentencias DDL (Definición de datos) y DML (Manipulación de datos). El fichero es ij.bat y se encuentra en la carpeta *bin* (*D:\db-*

derby-10.12.1.1-bin\bin). Desde la línea de comandos del DOS nos dirigimos a dicha carpeta y ejecutamos el fichero **ij.bat**:

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

En la creación de la base de datos es necesario determinar:

- 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 a crear
- create=true, atributo usado para crear la base de datos.

```
D:\db-derby-10.12.1.1-bin\bin>ij
Versi¾n de ij 10.12
ij> connect 'jdbc:derby:D:\DB\DERBY\ejemplo;create=true';
ij> CREATE TABLE departamentos (
> dept_no  INT NOT NULL PRIMARY KEY,
> dnombre  VARCHAR(15),
> loc      VARCHAR(15)
> );
0 filas insertadas/actualizadas/suprimidas
ij> INSERT INTO departamentos VALUES (10,'CONTABILIDAD','SEVILLA');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamentos VALUES (20,'INVESTIGACIÓN','MADRID');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamentos VALUES (30,'VENTAS','BARCELONA');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamentos VALUES (40,'PRODUCCIÓN','BILBAO');
1 fila insertada/actualizada/suprimida
ij> run 'Empleados.sql';
ij> CREATE TABLE empleados (
 emp_no    INT  NOT NULL PRIMARY KEY,
 apellido  VARCHAR(10),
 oficio    VARCHAR(10),
 dir       INT,
 fecha_alt DATE      ,
 salario   FLOAT,
 comision  FLOAT,
 dept_no   INT NOT NULL REFERENCES departamentos(dept_no)
);
0 filas insertadas/actualizadas/suprimidas
ij> INSERT INTO empleados VALUES (7369,'SANCHEZ','EMPLEADO',7902,'1990-
12-17', 1040,NULL,20);
1 fila insertada/actualizada/suprimida
ij> INSERT INTO empleados VALUES (7499,'ARROYO','VENDEDOR',7698,'1990-
02-20', 1500,390,30);
1 fila insertada/actualizada/suprimida
ij> INSERT INTO empleados VALUES (7521,'SALA','VENDEDOR',7698,'1991-02-
22', 1625,650,30);
1 fila insertada/actualizada/suprimida
ij> exit;
```

El comando **show tables;** muestra las tablas existentes en la base de datos. Para obtener ayuda podemos escribir el comando **help;**
Para volver a usar la base de datos escribiremos la siguiente orden desde la línea de comandos de ij: *connect 'jdbc:derby:D\DB\DERBY\ejemplo';*

ACTIVIDAD 2.2

Crea las tablas EMPLEADOS y DEPARTAMENTOS en Apache Derby e inserta filas en ellas. Emplea los mismos datos que en la Actividad 2.1.

2.3.3 HSQLDB

HSQLDB (Hyperthreaded Structured Query Language database) es un sistema gestor de bases de datos relacional escrito en Java. La suite ofimática OpenOffice lo incluye desde su versión 2.0 para dar soporte a la aplicación Base. Soporta la mayor parte de las características y funciones incluidas en el estándar SQL:2011. Puede mantener la base de datos en memoria o en ficheros de disco.

Desde la URL https://sourceforge.net/projects/hsqldb/files/ podemos descargarnos la última versión estable. Descargamos el fichero y lo descomprimimos, por ejemplo crearemos en la unidad D una carpeta con el nombre del fichero ZIP: (D:\hsqldb-version\hsqldb), dentro de esta 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*. Esto es importante ya que si no creamos la carpeta para la base de datos se mezclarán en *data* todas las bases de datos.

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

D:\hsqldb\bin>runUtil DatabaseManager

Se abre una ventana donde tenemos que configurar los parámetros de la conexión:

- **Setting Name:** nombre para la conexión. P.e. Mi BD Ejemplo
- **Type:** HSQL Database Engine Standalone
  Para que la base de datos la tome de un fichero si existe y si no existe la cree.
- **URL:** Nombre de la carpeta donde se almacenará la base de datos y el de la base de datos. P.e.: jdbc:hsqldb:file:ejemplo/ejemplo

Y pulsamos Ok.

Se abrirá una nueva ventana donde podemos ejecutar comandos DDL y DML para crear y manipular objetos de nuestra base de datos. Para ejecutar una sentencia pulsamos el botón *Execute*. Desde la opción de menú **View->Refresh Tree** podemos actualizar el árbol de objetos.

ACTIVIDAD 2.3

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

Emplea los mismos datos que en la Actividad 2.1.

2.3.4 H2

H2 es un sistema gestor de base de datos relacional programado íntegramente en Java. Desde la web http://www.h2database.com/html/main.html podemos descargarnos la última versión. Una vez descomprimida, desde la línea de comandos del DOS nos dirigimos a la carpeta *D:\h2\bin* y ejecutamos el fichero **h2.bat** para arrancar la consola.

D:\h2\bin>h2

Se abre el navegador Web con la consola de administración de H2. Realizaremos la siguiente configuración:

- Nombre para la configuración de la base de datos. P.e. Mi BD ejemplo
- URL JDBC: Escribimos la URL para la conexión a nuestra base de datos: *jdbc:h2:D:/DB/H2/ejemplo/ejemplo*
  si las carpetas no existen se crean automáticamente.

Pulsamos el botón guardar y cada vez que queramos conectarnos a nuestra base de datos usemos ese nombre y pulsamos el botón *Conectar*. Podemos pulsar el botón *Probar conexión* antes de conectar para ver si todo ha ido bien (mensaje Prueba correcta). Se creará la carpeta ejemplo en H2 y dentro los ficheros de nuestra base de datos.
Una vez conectados se visualiza una nueva pantalla desde la que podremos realizar las operaciones sobre la base de datos.
Desde la zona de instrucciones SQL podremos escribir las sentencias para crear tablas, insertar filas, etc. Los botones Ejecutar y Play nos permitirán ejecutar estas sentencias. El botón desconectar nos lleva a la pantalla inicial donde elegimos la conexión.

Desde el enlace Preferencias de la venta inicial se pueden configurar diversos aspectos como: los clientes permitidos (locales/remotos), conexión segura (uso de SSL), puerto del servidor web o notificar 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.

ACTIVIDAD 2.4

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

Emplea los mismos datos que en la Actividad 2.1.

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. Proporciona algunas características interesantes:

- Se evita el problema del desfase objeto-relacional
- No existe un lenguaje SQL para la manipulación de datos, en su lugar existen métodos delegados.
- Se instala añadiendo un único fichero de librería (JAR para Java o DLL para .NET)
- Se crea un único fichero de base de datos con la extensión .YAP(tamaño de 2GB a 264GB)

Desde la URL http://supportservices-old.actian.com/versant/default.html podemos descargarnos la última versión. Para el ejemplo siguiente se ha descargado la versión **db4o-8.0.276.16149-java.zip**. Al descomprimirla hay un fichero JAR en la carpeta /lib db4o-8.0.276.16149-all-java5.zip que necesitamos para utilizar el motor de la base de datos.

En **Eclipse** para usar el JAR seleccionamos nuestro proyecto, pulsamos el botón derecho del ratón y seleccionamos Build Paths -> Add External Archives. Se recomienda crear una carpeta e ir incluyendo los JAR que después usaremos en los ejercicios. Se mostrará en nuestro proyecto un elemento más, Referenced Libraries, con el fichero JAR añadido.
En **Netbeans** botón derecho sobre el proyecto, propiedades, seleccionamos Libraries y añadimos el .jar con el botón + en Classpath.
Si ejecutamos el programa desde la línea de comandos hemos de asegurarnos que el lugar donde se encuentra el fichero JAR se encuentra definido en la variable CLASSPATH.

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

```
1  import com.db4o.Db4oEmbedded;
2  import com.db4o.ObjectContainer;
3
4  public class Main {
5
6      public static void main(String[] args) {
7          String BDPer = "DBPersonas.yap";
8          ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), BDPer);
9
10         // Creamos Personas
11         Persona p1 = new Persona("Juan", "Guadalajara");
12         Persona p2 = new Persona("Ana", "Madrid");
13         Persona p3 = new Persona("Luis", "Granada");
14         Persona p4 = new Persona("Pedro", "Asturias");
15
16         // Almacenar objetos Persona en la base de datos
17         db.store(p1);
18         db.store(p2);
19         db.store(p3);
20         db.store(p4);
21
22         db.close(); // cerrar base de datos
23
24     }// fin de main
25 }// fin de la clase Main
```

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

Para realizar cualquier acción, ya sea insertar, modificar o realizar consultas debemos manipular una instancia de **ObjectContainer** donde se define el fichero de base de datos, en el ejemplo el fichero se llama DBPersonas.yap y el nombre se almacena en la variable BDPer

donde será necesario incluir el trayecto donde se encuentra el fichero, en nuestro caso en la carpeta del proyecto.

Algunos de los métodos más importantes son:

- **openFile()** para abrir la base de datos
  ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), BDPer);
- **close()** para cerrar la base de datos
  db.close();
- **store()** para almacenar un objeto en la base de datos.
  db.store(p1);

Se ha definido la clase Persona formada por los atributos *nombre* y *ciudad* y los métodos get y set para obtener y almacenar los valores de un objeto Persona:

```
1  public class Persona {
2      private String nombre;
3      private String ciudad;
4
5      public Persona(String nombre, String ciudad) {
6          this.nombre = nombre;
7          this.ciudad = ciudad;
8      }
9
10     public Persona() {
11         this.nombre = null;
12         this.ciudad = null;
13     }
14
15
16
17     public String getNombre() {
18         return nombre;
19     }
20
21     public void setNombre(String nombre) {
22         this.nombre = nombre;
23     }
24
25     public String getCiudad() {
26         return ciudad;
27     }
28
29     public void setCiudad(String ciudad) {
30         this.ciudad = ciudad;
31     }
32
33 }// fin Persona
34
```

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**. A continuación hemos de seleccionar los campos pulsar el botón OK. Para generar los constructores **Generate Constructor using Fields** (con parámetros) **Generate Constructors from Superclass** (sin parámetros).

Para recuperar objetos podemos utilizar el sistema de consultas QBE (Query-By-Example) mediante el método **queryByExample().**

El siguiente ejemplo muestra todos los objetos Persona existentes en la base de datos, los resultados se proporcionan en forma de ObjectSet (set de objetos que son resultado de una petición (query)). Si no existe ningún objeto el método **size()** sobre el objeto **ObjectSet** devolverá 0.

```
1  import com.db4o.Db4oEmbedded;
2  import com.db4o.ObjectContainer;
3  import com.db4o.ObjectSet;
4
5  public class Consulta1 {
6
7      public static void main(String[] args) {
8          String BDPer = "DBPersonas.yap";
9          ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), BDPer);
10
11         Persona per = new Persona(null, null);
12         ObjectSet<Persona> result = db.queryByExample(per);
13         if (result.size() == 0)
14             System.out.println("No existen Registros de Personas..");
15         else {
16             System.out.printf("N@mero de registros: %d %n", result.size());
17
18             while (result.hasNext()) {
19                 Persona p = result.next();
20                 System.out.printf("Nombre: %s, Ciudad: %s %n", p.getNombre(), p.getCiudad());
21             }
22         }
23         db.close(); // cerrar base de datos
24
25     }
26
27 }
```

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

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

La siguiente consulta obtiene los objetos Persona cuyo ciudad es Guadalajara:

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

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

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

public class Modificar {

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

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

        db.close();
    }

}
```

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

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

public class Eliminar {

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

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

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

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

        db.close();
    }

}
```

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

ACTIVIDAD 2.5

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

Datos a insertar:

DEPARTAMENTOS:

10, "CONTABILIDAD", "SEVILLA"

20, "INVESTIGACI"N", "MADRID"

30, "VENTAS", "BARCELONA"

40, "PRODUCCIÓN", "BILBAO"

EMPLEADOS:

7369, "SÁNCHEZ", "EMPLEADO", 7902, fec,1040.0, 0.0, 20

7499, "ARROYO", "VENDEDOR", 7698, fec,1500.0, 390.0, 30

7521, "SALA", "VENDEDOR", 7698, fec,1625.0, 650.0, 30

7566, "JIM…NEZ", "DIRECTOR", 7839, fec,2900.0, 0.0, 20

7782, "CEREZO", "DIRECTOR", 7839, fec,2885.0, 0.0, 10

7839, "REY", "PRESIDENTE", 0, fec, 4100.0, 0.0, 10

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 que pueden usar las aplicaciones para abrir una conexión con una base de datos, enviar consultas, actualizaciones y obtener resultados. Las aplicaciones pueden usar esta API para conectarse a cualquier servidor de base de datos compatible con OBDC. Está escrito en C.

- **JDBC** (*Java Database Connectivity*)

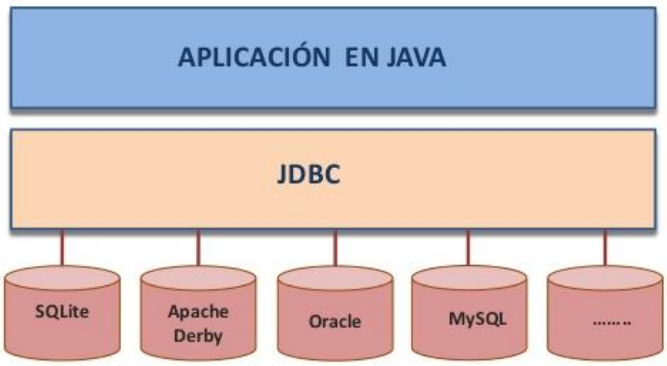
Define una API que pueden usar los programas Java para conectarse a los servidores de bases de datos relacionales.

En ocasiones podemos encontrar orígenes de datos que no son bases de datos relacionales, algunos puede que ni siquiera sean bases de datos, tal es el caso de los ficheros planos y los almacenes de correo electrónico.

OLE-DB de Microsoft es una API de C++ con objetivos parecidos a los de ODBD, pero para orígenes de datos que no son bases de datos. La API ADO (Active Data Objects) ofrece una interfaz sencilla de utilizar con la funcionalidad OLE-DB, que puede llamarse desde los lenguajes de guiones como VBScript y JScript.

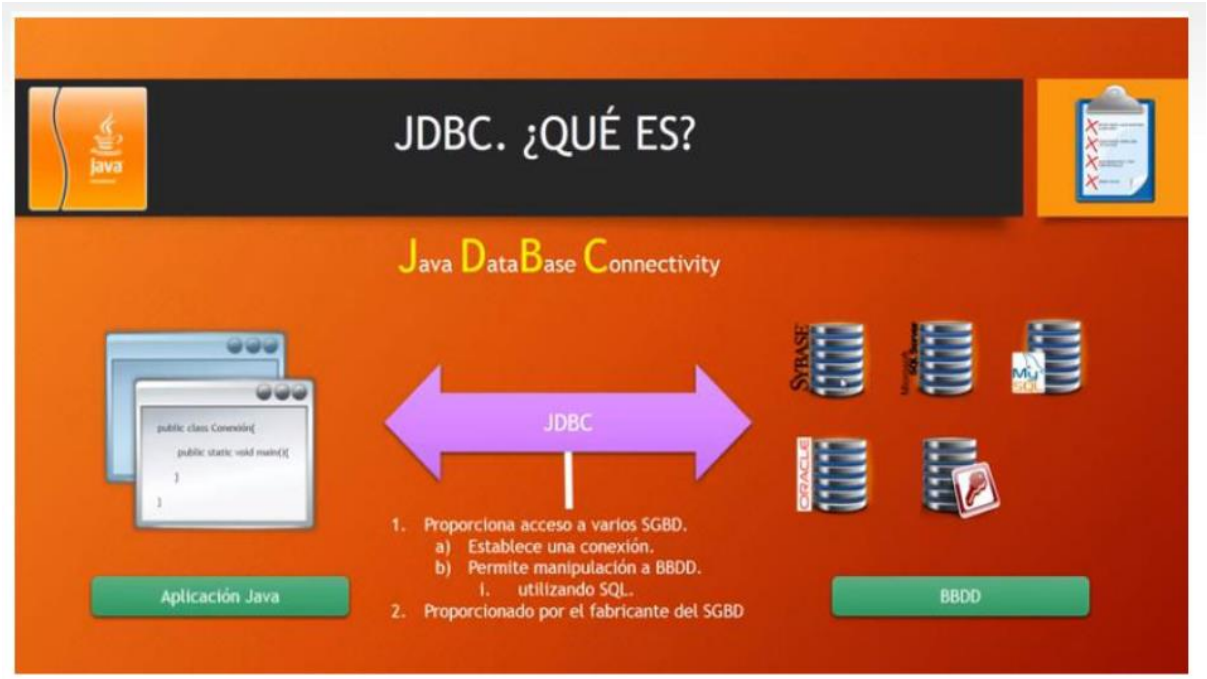
2.5 Acceso a datos mediante JDBC

JDBC proporciona una librería estándar para acceder a las 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 de Java el acceso a los datos. JDBC dispone de una interfaz distinta para cada base de datos (véase la siguiente figura), es lo que llamamos **driver**. Esto permite que las llamadas a métodos Java de las clases JDBC se correspondan con el API de la base de datos.



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



2.5.1 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**.

Como veremos más adelante, necesitaremos manejar:

- 1 paquete: **java.sql**
- 1 clase: **DriverManager.** Carga un driver para un SGBD específico.
- 3 interfaces:
    - **Connection:** Establece conexiones con las bases de datos.
    - **Statement**: Ejecutar sentencias SQL y las envía a las BD's
    - **ResultSet:** Almacena el resultado de la consulta

La siguiente tabla muestra las clases e interfaces más importantes:

Clase / 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: tablas,…
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: nº de columnas, sus nombre, …

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**

2.5.2 Instalación Software necesario

Instalación de XAMPP https://www.apachefriends.org/index.html

XAMPP es una distribución de Apache completamente gratuita y fácil de instalar que contiene MariaDB, PHP y Perl.

2.5.3 Ejemplo JDBC con MySQL

Vamos a crear 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 la base de datos. Crearemos la base de datos ejemplo abriendo PHPMyAdmin de XAMPP. Crearemos en Cuentas de Usuario el usuario ejemplo con la contraseña ejemplo y con todos los privilegios. A continuación, creamos las siguientes tablas e insertamos datos en ellas.

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

INSERT INTO departamentos VALUES (10,'CONTABILIDAD','SEVILLA');
INSERT INTO departamentos VALUES (20,'INVESTIGACI”N','MADRID');
INSERT INTO departamentos VALUES (30,'VENTAS','BARCELONA');
INSERT INTO departamentos VALUES (40,'PRODUCCI”N','BILBAO');
COMMIT;

CREATE TABLE empleados (
 emp_no        SMALLINT(4)  NOT NULL PRIMARY KEY,
 apellido  VARCHAR(10),
 oficio   VARCHAR(10),
 dir       SMALLINT,
 fecha_alt DATE         ,
 salario   FLOAT(6,2),
 comision  FLOAT(6,2),
 dept_no   TINYINT(2) NOT NULL,
 CONSTRAINT FK_DEP FOREIGN KEY (dept_no ) REFERENCES departamentos(dept_no)

) ENGINE=InnoDB;

INSERT INTO empleados VALUES (7369,'SÁNCHEZ','EMPLEADO',7902,'1990/12/17',
            1040,NULL,20);
INSERT INTO empleados VALUES (7499,'ARROYO','VENDEDOR',7698,'1990/02/20',
            1500,390,30);
INSERT INTO empleados VALUES (7521,'SALA','VENDEDOR',7698,'1991/02/22',
            1625,650,30);
INSERT INTO empleados VALUES (7566,'JIMÉNEZ','DIRECTOR',7839,'1991/04/02',
```

```
        2900,NULL,20);
INSERT INTO empleados VALUES (7654,'MARTÓN','VENDEDOR',7698,'1991/09/29',
        1600,1020,30);
INSERT INTO empleados VALUES (7698,'NEGRO','DIRECTOR',7839,'1991/05/01',
        3005,NULL,30);
INSERT INTO empleados VALUES (7782,'CEREZO','DIRECTOR',7839,'1991/06/09',
        2885,NULL,10);
INSERT INTO empleados VALUES (7788,'GIL','ANALISTA',7566,'1991/11/09',
        3000,NULL,20);
INSERT INTO empleados VALUES (7839,'REY','PRESIDENTE',NULL,'1991/11/17',
        4100,NULL,10);
INSERT INTO empleados VALUES (7844,'TOVAR','VENDEDOR',7698,'1991/09/08',
        1350,0,30);
INSERT INTO empleados VALUES (7876,'ALONSO','EMPLEADO',7788,'1991/09/23',
        1430,NULL,20);
INSERT INTO empleados VALUES (7900,'JIMENO','EMPLEADO',7698,'1991/12/03',
        1335,NULL,30);
INSERT INTO empleados VALUES (7902,'FERNÁNDEZ','ANALISTA',7566,'1991/12/03',
        3000,NULL,20);
INSERT INTO empleados VALUES (7934,'MUÑOZ','EMPLEADO',7782,'1992/01/23',
        1690,NULL,10);
COMMIT;
```

EJEMPLO JDBC/Main.java

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

```java
import java.sql.*;

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

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

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

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

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

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

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

Para poder probar el programa hemos de obtener el JAR que contiene el driver MySQL (en el ejemplo se ha utilizado **mysql-connector-java_5.1.38-bin.jar**) e incluirlo en el CLASSPATH o añadirlo a nuestro IDE.

En Eclipse pulsamos en el proyecto con el botón derecho del ratón y seleccionamos Build Paths->Add External Archives para localizar el fichero JAR.

En Netbeans botón derecho sobre el proyecto, propiedades, seleccionamos Libraries y añadimos el .jar con el botón + en Classpath.

Si ejecutamos el programa desde la línea de comandos hemos de asegurarnos que el lugar donde se encuentra el fichero JAR se encuentra definido en la variable CLASSPATH.

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

Los pasos principales son:

1. **Cargar el driver**

Class.forName("com.mysql.jdbc.Driver");
Método **.forName** de la clase **Class**. Se le pasa un objeto String con el nombre de la clase del driver como argumento, en este caso base de datos MySQL.

2. **Establecer la conexión**

Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");
El servidor MySQL debe estar arrancado. Usamos la clase **DriverManager** con el método **getConnection()**.

La sintaxis de **getConnection()** es la siguiente:

public static Connection getConnection (String url, String user, String password) throws SQLException
El primer parámetro representa la URL de conexión a la base de datos. Tiene el siguiente formato para conectarse a MySQL:
jdbc:mysql://nombre_host:puerto/nombre_basedatos
Donde:

- **jdbc:mysql** indica que estamos empleando un driver JDBC para MySQL
- **nombre_host** nombre del servidor donde está la base de datos (IP o nombre máquina en red). Localhost: el servidor de la base de datos está en la misma máquina en la que se ejecuta el programa Java.
- **puerto** por defecto 3306. Si no se pone se asume ese valor.
- **nombre_basedatos** nombre base de datos a la que nos vamos a conectar y que debe existir en MySQL

3. Ejecutar sentencias SQL

Para realizar la consulta 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:

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

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

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

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

                }
```

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

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

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

4. **Liberar recursos:**

Por últimos se liberan todos los recursos y se cierra la conexión:

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

ACTIVIDAD 2.6
Tomando como base el programa que ilustra los pasos de funcionamiento de JDBC obtén el APELLIDO, OFICIO y SALARIO de los empleados del departamento 10.

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

2.6 Establecimiento de conexiones

En este apartado vamos a ver cómo conectarnos a través de JDBC a las bases de datos embebidas estudiadas anteriormente. Hemos de crear la base de datos *ejemplo* con las tablas *empleados* y *departamentos* y vamos a utilizar el mismo programa en Java, solo cambiaremos la carga del driver y la conexión a la base de datos.

En los ejemplos mostrados se supone que las bases de datos ejemplo están situadas en las carpetas: D:\\DB\SQLite, D:\\DB\HSQLDB\ejemplo, D:\\DB\H2 y D:\\DB\DERBY.

Para realizar las pruebas necesitaremos tener el conector Java (fichero JAR) correspondiente para cada una de las bases de datos.

2.6.1 Conexión a SQLite

Driver: **sqlite-jdbc-3.8.11.2.jar**
Nombre del driver para la conexión a la base de datos: **org.sqlite.JDBC**

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

2.6.2 Conexión a Apache Derby

Driver: **derby.jar**
Nombre del driver para la conexión: **org.apache.derby.jdbc.EmbeddedDriver**

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

2.6.3 Conexión a HSQLDB

Driver: **hsqldb.jar**
Nombre del driver para la conexión a la base de datos: **org.hsqldb.jdbcDriver**

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

2.6.4 Conexión a H2

Driver: **h2-1.4.191.jar**
Nombre del driver para la conexión a la base de datos: **org.h2.Driver**

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

En este caso el nombre de usuario que se puso fue “sa” y la contraseña vacía.

2.6.5 Conexión a MySQL

Driver: **mysql-connector-java-5.1.38-bin.jar**
Nombre del driver para la conexión a la base de datos: **com.mysql.jdbc.Driver**

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

2.6.6 Conexión a Oracle

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).
Necesitamos saber el nombre del servicio que usa la base de datos para incluirlo en la URL de la conexión. Normalmente para la versión Express Edition el nombre es XE.
Driver: **ojdbc6.jar**
Nombre del driver para la conexión a la base de datos: **oracle.jdbc.driver.OracleDriver**

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

Actividad 2.7

Instala Oracle 11g Express Edition y SQL Developer y crea las tablas de departamentos y empleados creadas en los ejemplos anteriores. Inserta los mismos datos en ellas y realiza un programa en Java que se conecte a la base de datos de Oracle mediante JDBC y muestre todos los departamentos y todos los empleados.

2.7 Ejecución de sentencias de descripción de datos

Normalmente cuando desarrollamos una aplicación JDBC conocemos la estructura de las tablas y datos que estamos manejando. En caso de no conocer esta información, se puede obtener a través de los *metaobjetos,* que no son más que objetos que proporcionan información sobre la base de datos.

2.7.1 DatabaseMetaData

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

Algunos de los métodos que veremos en los siguientes ejemplos son:

- getTables()
- getColumns()
- getPrimaryKeys() Proporciona información sobre las columnas que forman la clave primaria de una tabla.
- getExportedKeys() Devuelve información sobre las claves ajenas que utilizan la clave primaria de una tabla.
- getImportedKeys() Devuelve información sobre las claves ajenas existentes en una tabla.
- getProcedures() Devuelve información sobre los procedimientos almacenados

A continuación vamos a ver algunos ejemplos de código que podéis importar de la carpeta RECURSOS_ALUMNOS_UD2/Sentencias_descripcion_de_datos.

Ejemplo getMetaData() (EjemploDatabaseMetadata.java)

El siguiente ejemplo conecta con la base de datos MySQL de nombre ejemplo y muestra información sobre la base de datos: nombre, driver, url, usuario, tablas y vistas del esquema actual.

El método **getMetaData()** de la interfaz **Connection** devuelve un objeto **DatabaseMetaData** que contiene información sobre la base de datos representada por el objeto **Connection**.

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

```
ResultSet getTables(String catalog,
                    String schemaPattern,
                    String tableNamePattern,
                    String[] types)
             throws SQLException
```

Este método devuelve un objeto de la clase **ResultSet**. Los parámetros pasados al método sirven para: los dos primeros para especificar el catálogo y el esquema de los que se obtendrá la información; el tercer parámetro es el nombre de la tabla a obtener y el último los tipos (TABLE para tablas, VIEW para vistas, etc(SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS y SYNONYM.). El tercer parámetro, patrón de la tabla, permite utilizar comodines. Por ejemplo, si quisiéramos obtener todas las tablas cuyo nombre empiece por Nom, pasaríamos el parámetro "Nom%".

Cada fila de ResultSet que devuelve getTables() tiene información sobre una tabla. La descripción de cada tabla tiene las siguientes columnas (en orden de numeración):

1. **TABLE_CAT** String => table catalog name(may be null).
2. **TABLE_SCHEM** String => table schema name (may be null)
3. **TABLE_NAME** String => table name (nombre de tabla o vista)
4. **TABLE_TYPE** String => table type. Typical types are "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM".
5. **REMARKS** String => explanatory comment on the table
6. **TYPE_CAT** String => the types catalog (may be null)
7. **TYPE_SCHEM** String => the types schema (may be null)
8. **TYPE_NAME** String => type name (may be null)
9. **SELF_REFERENCING_COL_NAME** String => name of the designated "identifier" column of a typed table (may be null)
10. **REF_GENERATION** String => specifies how values in SELF_REFERENCING_COL_NAME are created. Values are "SYSTEM", "USER", "DERIVED". (may be null)

Ejemplo getColumns (EjemplogetColumns.java)

El siguiente ejemplo muestra información sobre todas las columnas de la tabla departamentos.

getColumns() Devuelve un objeto ResultSet con información sobre las columnas de una tabla o tablas.
Su sintaxis es:

ResultSet getColumns(String catalog,
String schemaPattern,
String tableNamePattern,
String columnNamePattern)
throws SQLException

El valor null en los 4 parámetros indica que obtiene información de todas las columnas y las tablas del esquema actual.

Para el patrón de nombre y columna se puede utilizar el carácter %,por ejemplo “d%” obtendría todos los nombres que comiencen por la letra d.

En ResultSet se obtiene información sobre las columnas de una tabla o tablas. La descripción de cada columna tiene las siguientes columnas:

1. **TABLE_CAT** String => table catalog (may be null)
2. **TABLE_SCHEM** String => table schema (may be null)
3. **TABLE_NAME** String => table name
4. **COLUMN_NAME** String => column name
5. **DATA_TYPE** int => SQL type from java.sql.Types
6. **TYPE_NAME** String => Data source dependent type name, for a UDT the type name is fully qualified
7. **COLUMN_SIZE** int => column size.
8. **BUFFER_LENGTH** is not used.
9. **DECIMAL_DIGITS** int => the number of fractional digits. Null is returned for data types where DECIMAL_DIGITS is not applicable.
10. **NUM_PREC_RADIX** int => Radix (typically either 10 or 2)
11. **NULLABLE** int => is NULL allowed.
    - columnNoNulls - might not allow NULL values
    - columnNullable - definitely allows NULL values
    - columnNullableUnknown - nullability unknown
12. **REMARKS** String => comment describing column (may be null)
13. **COLUMN_DEF** String => default value for the column, which should be interpreted as a string when the value is enclosed in single quotes (may be null)
14. **SQL_DATA_TYPE** int => unused
15. **SQL_DATETIME_SUB** int => unused
16. **CHAR_OCTET_LENGTH** int => for char types the maximum number of bytes in the column
17. **ORDINAL_POSITION** int => index of column in table (starting at 1)
18. **IS_NULLABLE** String => ISO rules are used to determine the nullability for a column.
    - YES --- if the column can include NULLs
    - NO --- if the column cannot include NULLs
    - empty string --- if the nullability for the column is unknown

19. **SCOPE_CATALOG** String => catalog of table that is the scope of a reference attribute (null if DATA_TYPE isn't REF)
20. **SCOPE_SCHEMA** String => schema of table that is the scope of a reference attribute (null if the DATA_TYPE isn't REF)
21. **SCOPE_TABLE** String => table name that this the scope of a reference attribute (null if the DATA_TYPE isn't REF)
22. **SOURCE_DATA_TYPE** short => source type of a distinct type or user-generated Ref type, SQL type from java.sql.Types (null if DATA_TYPE isn't DISTINCT or user-generated REF)
23. **IS_AUTOINCREMENT** String => Indicates whether this column is auto incremented
    - YES --- if the column is auto incremented
    - NO --- if the column is not auto incremented
    - empty string --- if it cannot be determined whether the column is auto incremented
24. **IS_GENERATEDCOLUMN** String => Indicates whether this is a generated column
    - YES --- if this a generated column
    - NO --- if this not a generated column
    - empty string --- if it cannot be determined whether this is a generated column

Ejemplo getPrimaryKeys()

Devuelve la lista de columnas que forman la clave primaria de la tabla especificada. La descripción de cada columna de la clave primaria tiene las siguientes columnas:

TABLE_CAT String => table catalog (may be null)

1. **TABLE_SCHEM** String => table schema (may be null)
2. **TABLE_NAME** String => table name
3. **COLUMN_NAME** String => column name
4. **KEY_SEQ** short => sequence number within primary key(a value of 1 represents the first column of the primary key, a value of 2 would represent the second column within the primary key).
5. **PK_NAME** String => primary key name (may be null)

La sintaxis es la siguiente:

ResultSet getPrimaryKeys(String catalog,
String schema,
String table)
throws SQLException

Ejemplo getExportedKeys()

Devuelve la lista de todas las claves ajenas que utilizan la clave primaria de la tabla especificada. La descripción de cada columna de clave ajena tiene las siguientes columnas:

1. **PKTABLE_CAT** String => primary key table catalog (may be null)
2. **PKTABLE_SCHEM** String => primary key table schema (may be null)
3. **PKTABLE_NAME** String => primary key table name
4. **PKCOLUMN_NAME** String => primary key column name
5. **FKTABLE_CAT** String => foreign key table catalog (may be null) being exported (may be null)
6. **FKTABLE_SCHEM** String => foreign key table schema (may be null) being exported (may be null)
7. **FKTABLE_NAME** String => foreign key table name being exported
8. **FKCOLUMN_NAME** String => foreign key column name being exported
9. **KEY_SEQ** short => sequence number within foreign key(a value of 1 represents the first column of the foreign key, a value of 2 would represent the second column within the foreign key).
10. **UPDATE_RULE** short => What happens to foreign key when primary is updated:
    - importedNoAction - do not allow update of primary key if it has been imported
    - importedKeyCascade - change imported key to agree with primary key update
    - importedKeySetNull - change imported key to NULL if its primary key has been updated
    - importedKeySetDefault - change imported key to default values if its primary key has been updated
    - importedKeyRestrict - same as importedKeyNoAction (for ODBC 2.x compatibility)
11. **DELETE_RULE** short => What happens to the foreign key when primary is deleted.
    - importedKeyNoAction - do not allow delete of primary key if it has been imported
    - importedKeyCascade - delete rows that import a deleted key
    - importedKeySetNull - change imported key to NULL if its primary key has been deleted
    - importedKeyRestrict - same as importedKeyNoAction (for ODBC 2.x compatibility)
    - importedKeySetDefault - change imported key to default if its primary key has been deleted
12. **FK_NAME** String => foreign key name (may be null)
13. **PK_NAME** String => primary key name (may be null)
14. **DEFERRABILITY** short => can the evaluation of foreign key constraints be deferred until commit
    - importedKeyInitiallyDeferred - see SQL92 for definition
    - importedKeyInitiallyImmediate - see SQL92 for definition
    - importedKeyNotDeferrable - see SQL92 for definition

Su sintaxis es la siguiente:

ResultSet getExportedKeys(String catalog,
String schema,
String table)
throws SQLException

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

getImportedKeys()

Devuelve la lista de claves ajenas existentes en la tabla indicada, se emplea igual que el método anterior.

Su sintaxis es la siguiente:

ResultSet getImportedKeys(String catalog,
String schema,
String table)
throws SQLException

Ejemplo getProcedures() (Verprocedimientosyfunciones.java)

Devuelve la lista de procedimientos almacenados. Cada fila de ResultSet es un procedimiento. Cada descripción de procedimiento tiene las siguientes columnas:

1. **PROCEDURE_CAT** String => procedure catalog (may be null)
2. **PROCEDURE_SCHEM** String => procedure schema (may be null)
3. **PROCEDURE_NAME** String => procedure name
4. reserved for future use
5. reserved for future use
6. reserved for future use
7. **REMARKS** String => explanatory comment on the procedure
8. **PROCEDURE_TYPE** short => kind of procedure:
    - procedureResultUnknown - Cannot determine if a return value will be returned
    - procedureNoResult - Does not return a return value
    - procedureReturnsResult - Returns a return value
9. **SPECIFIC_NAME** String => The name which uniquely identifies this procedure within its schema.

Su sintaxis es la siguiente:

ResultSet getProcedures(String catalog,
String schemaPattern,
String procedureNamePattern)
throws SQLException

ACTIVIDAD 2.8
Para probar el método getProcedures() vamos a crear algunos procedimientos y funciones en MySQL:

Ejemplo de función en MySQL
Función que recibe dos números y devuelve su suma.

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

Ejemplo de procedimiento en MySQL
Procedimiento que sube 100 euros el salario de los empleados del departamento 30.

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

Una vez creados realizaremos un programa que muestre la función y el procedimiento, para ello emplearemos:

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

Puede verse el código completo en el ejemplo *Verprocedimientosyfunciones.java* de los recursos de esta unidad.

2.7.2 ResultSetMetaData

La interfaz **ResultSetMetaData()**, **a partir de un objeto ResultSet**, nos permite obtener más información sobre los tipos y propiedades de las columnas de los objetos ResultSet.

Ver *EjemploResultsetmetada.java* de los recursos de esta unidad. En este ejemplo 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.

Los métodos empleados del objeto ResultSetMetaData son los siguientes:

Método	Descripción
int getColumnCount()	Devuelve número de columnas
String getColumnName (int indiceCoumna)	Devuelve el nombre de la columna correspondiente
String getCoumnTypeName(int indiceColumna)	Devuelve el nombre del tipo de dato específico del sistema de bases de datos que contiene la columna indicada en indiceColumna
int isNullable(int indiceColumna)	Devuelve 0 si la columna no puede contener valores nulos
int getColumnDisplaySize(int indiceColumna)	Devuelve el máximo ancho en caracteres de la columna indicada en índiceColumna

ACTIVIDAD 2.9
Visualiza información sobre las columnas de la tabla *empleados*.

2.8 Ejecución de sentencias de manipulación de datos

En ejemplos anteriores vimos cómo se podían ejecutar sentencias SQL mediante la interfaz **Statement** (sentencia). Los objetos se obtienen con una llamada al método **createStatement()** de un objeto **Connection**:

Statement sentencia = conexion.createStatement();

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

- **ResultSet executeQuery(String)**: se utiliza para sentencias SQL que recuperan datos de un único objeto ResultSet, se utiliza para las sentencias SELECT.
- **int executeUpdate(String):** se utiliza para sentencias que no devuelven un ResultSet como son las sentencias de manipulación de datos (DML): INSERT, UPDATE y DELETE; y las sentencias de definición de datos (DDL): CREATE, DROP y ALTER. El método devuelve un entero indicando el número de filas que se vieron afectadas y en el caso de sentencias DDL devuelve el valor 0.
- **boolean execute(String):** se puede utilizar para ejecutar cualquier sentencia SQL. El método devuelve TRUE si devuelve un ResultSet (recuperar filas con getResultSet()) y FALSE si se trata de un recuento de actualizaciones o no hay resultados (usar getUpdateCount() para recuperar el valor devuelto).

Ver *EjemploExecute.java* de los recursos de esta unidad.

A través de un objeto ResultSet se puede acceder al valor de cualquier columna de la fila actual por nombre o por posición, también información como el número de columnas o su tipo.

A continuación se listan algunos métodos disponibles para ResultSet:

Método	Tipo Java devuelto
`getString(int númerodecolumna)` `getString(String nombredecolumna)`	String
`getBoolean(int númerodecolumna)` `getBoolean(String nombredecolumna)`	boolean
`getByte(int númerodecolumna)` `getByte(String nombredecolumna)`	byte
`getShort(int númerodecolumna)` `getShort(String columna)`	short
`getInt(int númerodecolumna)` `getInt(String nombredecolumna)`	int
`getLong(int númerodecolumna)` `getLong(String nombredecolumna)`	long
`getFloat(int númerodecolumna)` `getFloat(String nombredecolumna)`	float
`getDouble(int númerodecolumna)` `getDouble(String nombredecolumna)`	double
`getBytes(int númerodecolumna)` `getBytes(String nombredecolumna)`	byte[]
`getDate(int númerodecolumna)` `getDate(String nombredecolumna)`	Date
`getTime(int númerodecolumna)` `getTime(String nombredecolumna)`	Time
`getTimestamp(int númerodecolumna)` `getTimestamp(String nombredecolumna)`	Timestamp

Ejemplo *InsertarDep.java* de los recursos de esta unidad.
Este ejemplo inserta un departamento en la tabla *departamentos*, los datos del nuevo departamento se introducen a través de los argumentos del *main().* 1º parámetro: departamento, 2º parámetro: nombre, 3º parámetro: localidad.

Antes de ejecutar la orden INSERT construimos la sentencia SQL en un *String,* observad que las cadenas de caracteres deben ir encerradas entre comillas simples.

Ejemplo *ModificarSalario.java* de los recursos de esta unidad.
Este ejemplo sube el salario a los empleados de un departamento, se pasan como parámetros el número de departamento y la subida salarial.

Ejemplo *CrearVista.java* de los recursos de esta unidad.
Este ejemplo crea una vista (de nombre totales) que contiene por cada departamento, el nombre, el número de empleados que tiene y el salario medio.

2.8.1 Ejecución de Scripts

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

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

Ejemplo *ejecutarScriptMySQL()* de los recursos de esta unidad.
En este ejemplo se muestra la ejecución del siguiente script almacenado en el fichero *./scripts/scriptmysql.sql* dentro del proyecto.

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

2.8.2 Sentencias Preparadas

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

Antes de ejecutar un **PreparedStatement** es necesario asignar los datos. Los objetos **PreparedStatement** se pueden preparar o precompilar una sola vez y ejecutar las veces que queramos asignando diferentes valores, en cambio en los objetos **Statement**, la sentencia SQL se suministra en el momento de ejecutar la sentencia.

Los métodos de **PreparedStatement** tienen los mismos nombres que en **Statement:**

- executeQuery()
- executeUpdate()
- execute()

Pero no se necesita enviar la cadena de caracteres con la orden SQL ya que lo hace el método **prepareStatement(String)**

Ejemplo InsertaDepPreparedStatement.java
El ejemplo en el que se inserta una fila en la tabla departamentos quedaría así empleando PreparedStatement.

```
// construir orden INSERT
        String sql = "INSERT INTO departamentos VALUES " + "( ?, ?, ? )";
        System.out.println(sql);
        PreparedStatement sentencia = conexion.prepareStatement(sql);
        sentencia.setInt(1, Integer.parseInt(dep));
        sentencia.setString(2, dnombre);
        sentencia.setString(3, loc);

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

Para asignar valor a cada uno de los marcadores de posición se utilizan los métodos setXXX(), algunos de ellos se describen en la siguiente tabla:

Método	Tipo SQL
void setString(int índice, String valor)	VARCHAR
void setBoolean(int índice, boolean valor)	BIT
void setByte(int índice,byte valor)	TINYINT
void setShort(int índice, short valor)	SMALLINT
void setInt(int índice, int valor)	INTEGER
void setLong(int índice, long valor)	BIGINT
void setFloat(int índice, float valor)	FLOAT
void setDouble(int índice, double valor)	DOUBLE
void setBytes(int índice, byte[] valor)	VARBINARY
void setDate(int índice, Date valor)	DATE
void setTime(int índice, Time valor)	TIME

Para asignar valores NULL a un parámetro se usa el método **setNull()**
void setNull (int índice, int tipoSQL)

donde tipoSQL es una constante que se define en la librería **java.sql.Types.**

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.preparedStatement(sql);

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

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

ACTIVIDAD 2.11

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

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

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

2.9 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 depende del sistema gestor de base de datos. En MySQL, por ejemplo, las funciones no admiten parámetros OUT e INOUT, solo parámetros IN.

La interfaz **CallableStatement** permite que se pueda llamar desde Java a los procedimientos almacenados. Para crear un objeto se llama al método **prepareCall(String)** del objeto **Connection**. En el String se declara la llamada al procedimiento o función, tiene dos formatos, uno incluye el parámetro del resultado (usado para las funciones y el otro no. Además si los procedimientos y funciones incluyen parámetros de entrada o de salida es necesario indicarlos en forma de marcadores de posición (1 primer parámetro, 2 el segundo, etc).

Teniendo en cuenta esto, hay 4 formas de declarar las llamadas:

- {call nombre_procedimiento} procedimiento sin parámetros
- {? = call_nombre_función} función sin parámetros (devuelve valor)
- {call nombre_procedimiento(?,?,..)} procedimiento con parámetros
- {? = call nombre_funcion(?,?,..)} función con parámetros (devuelve valor)

Añadiremos el siguiente procedimiento a nuestra base de datos ejemplo:

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

En el ejemplo **ProcSubida()** (en los recursos de la unidad) se realiza una llamada al procedimiento subida_sal(de MySQL); los valores de los parámetros se asignan a partir de los argumentos del main().

Puede que sea necesario dar permisos al usuario para ejecutar procedimientos. Si es así, añadiremos el permiso SELECT sobre la tabla del sistema mysql.proc (GRANT SELECT ON mysql.proc TO 'ejemplo'@'localhost'

Cuando un procedimiento o función tiene parámetros de salida (OUT) deben ser registrados antes de que la llamada tenga lugar. El método empleado es **registerOutParameter(int indice, int tipoSQL)** el primer parámetro es la posición, es segundo es una constante definida en la clase java.sql.Types.
Una vez ejecutado el procedimiento, los valores de los parámetros OUT o INOUT se obtienen con los métodos getXXX().

2.10 Informes con JasperReports

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

Se puede descargar desde https://community.jaspersoft.com/download, hay distintas versiones: Server, Library y Studio.

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

En este capítulo solo nos interesa generar informes utilizando la plantilla definida en el fichero **JRXML**. Para ello deberemos añadir los JAR de JasperReports y el JAR **tools.jar** del JDK.

En estas pruebas se utiliza la versión 6.2.0 de JasperReports y se pueden encontrar los ficheros en la carpeta de recursos de la unidad.

Ejemplo1_Jasper

Creamos un proyecto para generar un informe de datos de departamentos de la base de datos MySQL.
Para crear un informe con JasperReports seguiremos los siguientes pasos:

1. Generar el fichero **.jrxml**, será la plantilla en la que configuraremos cómo se desea el informe. Indicaremos parámetros del informe, la consulta que se va a realizar, los datos a visualizar y cómo van a ser las líneas de cabecera, de detalle y de pies.
2. Dentro del proyecto Java se compilará la plantilla y obtendremos un objeto **JasperReport**.
3. Para rellenar de datos el informe se utilizará el método **fillReport()** de la clase **JasperFillManager (JasperFillManager.fillReport())** Esto generará un fichero .jrprint. *JasperPrint MiInforme = JasperFillManager.fillReport(NombreJasperReport, ParametrosDelInforme, conexionBD)*
   Los parámetros tienen que crearse y almacenarse en un *HashMap* (de **java.util**) , ese Map se utiliza para crear el *JasperPrint* añadiendo parámetros.
4. Y finalmente podremos exportar el fichero *JasperPrint* al formato que se desee (HTML, PDF, XML…)