UNIDAD 10

APLICACIONES JAVA CON SGBDR

BIBLIOGRAFÍA:
- Java Data Access JDBC 3, Tood Thomas. O'Rilley (2002).
JDBC Recipes. Mahmoud Apress (2005)
- Manual de MySQL 8
- Documentación oficial de Oracle.

10.1. INTRODUCCIÓN A MYSQL.

10.1.1. INSTALACIÓN DE MYSQL.

Para instalar en GNU/Linux de tipo Debian, primero debes ser root:

```
$ sudo su
# apt-get update
# apt-get upgrade
# apt-get install  mysql-common mysql-server mysql-client
```

Para iniciar el servidor:

```
# /etc/init.d/mysql start
```

Poner un password a la cuenta root del SGBD:

```
$ mysqladmin --user=root password nuevo_password
```

Parar el SGBD:

```
# /etc/init.d/mysql stop
```

Para instalar en Windows se usa la interfaz gráfica. Primero debes decidir si quieres instalar solamente el SGBD o también las herramientas adicionales. Durante el proceso te preguntará si permites instalar software que falte [Execute], si quieres el servidor como un servicio, el modo de Servidor (Standalone), el protocolo de autenticación (mejor Legacy para nuestros propósitcos), desbloquear puertos (TCP 3306 y 33060), cambiar la configuración del sistema, etc. Si eliges la configuración por defecto, instala el servidor en `C:\Program Files\MySQL\MySQL ServerXX\` donde XX es la versión.

Te preguntará también si te interesa crear una cuenta, puedes saltar ese paso. Cuando acabe el proceso, se configura el servidor, debes elegir las configuraciones por defecto. Es cómodo instalar el servidor como un servicio y que añada su ruta a la variable PATH.

Selecciona un nuevo password para el usuario root y espera a que se complete el proceso. Si has instalado como un servicio de Windows el servidor, comenzará a ejecutarse cada vez que se inicie el sistema. Si prefieres activarlo y desactivarlo manualmente usando el nombre del servicio, puedes usar los siguientes comandos siendo administrador:

```
C:\> net start mysql
C:\> net stop mysql
```

Si no lo has instalado como un servicio, debes ejecutarlo manualmente para iniciarlo y paralo de forma manual también:

```
mysqld
```

Si el servidor no arranca, prueba esto:

```
mysqld –no-defaults
```

Para pararlo:

```
mysqladmin –user=root --password=su_password shutdown
```

10.1.2. REPASO DE SQL.

CONECTARSE

Para trabajar en un SGBD, necesitas usar un cliente para conectarte al servidor. El cliente de consola se llama **mysql**. Te aparece un mensaje similar a este:

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 139 to server version: 4.0.1-alpha
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
Mysql>
```

Una vez dentro, puedes saber las bases de datos definidas actualmente en el sistema:

```
mysql> show databases;
```

Para crear una nueva base de datos llamada cuentas:

```
mysql> create database cuentas;
```

Usarla:

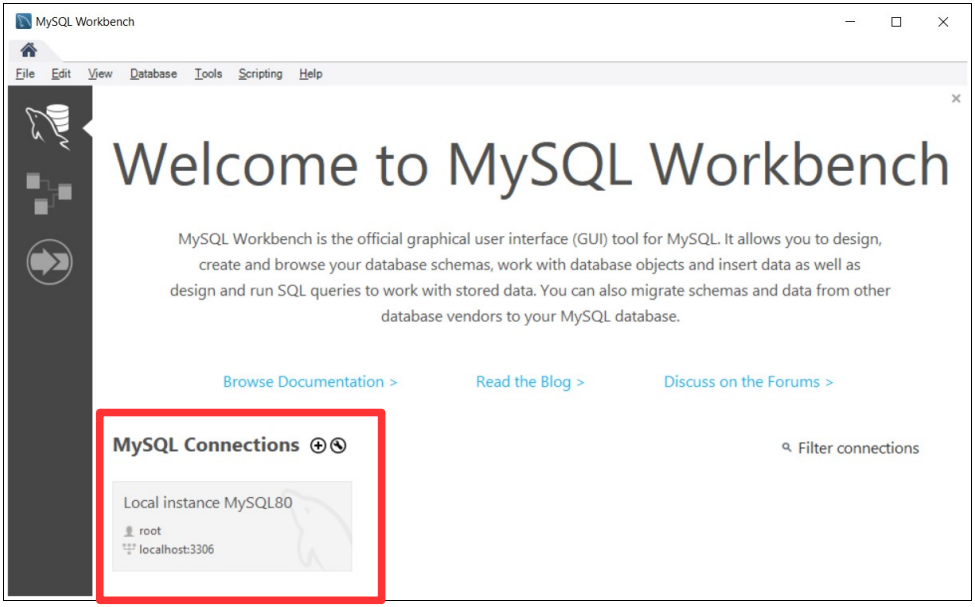
```
mysql> use cuentas;
```

Las tablas son los elementos de una BD donde se almacena la información. Una fila de la tabla describe a un ente (un empleado, una venta, ...) y las columnas son las características que describen a cada ente. Hay varios tipos en MySQL (**motores de almacenamiento**):

- **BDB**—soporta transacciones y recuperación ante fallos.
- **HEAP**—basada en memoria usa índices hash.
- **ISAM**—las tablas originales de MySQL, desfasadas.
- **InnoDB**—soporta transacciones, bloqueo a nivel de fila, claves ajenas y multiversionado.
- **MERGE**—varias tablas MyISAM usadas como una sola. Permite guardar una tabla lógica en varios lugares físicos.
- **MYISAM**—tablas sin soporte de transacciones.

Puedes cambiar el tipo después de crearlas, aunque tengan datos. Hay una gran cantidad de características de cada una que las hacen apropiadas según necesaidades: máximo nº de filas, almacenamiento, passwords para acceder y tipo de columnas que pueden tener.

También hay una aplicación de administración, gestión y modelado llamada **MySQLWorkBench** que puede usar para conectarte y trabajar con un servidor (recuadro marcado en rojo en la figura).



DDL

Tipos de columnas más comunes

- `INT[(ancho)] [UNSIGNED] [ZEROFILL]` números enteros desde `-2,147,483,648` hasta `2,147,483,647`. Si usas `UNSIGNED` el rango va desde 0 hasta `4,294,967,295`. `INT` es una abreviatura de `INTEGER` y puedes usar ambas palabras. Si usas `ZEROFILL` se asume `UNSIGNED`. Necesita 4 bytes de memoria. Ejemplo:

```
mysql> CREATE TABLE numeros(n INT(4) ZEROFILL);
mysql> INSERT INTO numeros VALUES(3), (33), (333), (3333);
mysql> SELECT * FROM numeros;
+-----------+
| n         |
+-----------+
| 0003      |
| 0033      |
```

```
| 0333      |
| 3333      |
+-----------+
6 rows in set (0.00 sec)
```

- **DECIMAL[(ancho[,*decimales*])] [UNSIGNED] [ZEROFILL]** Almacena números con decimales en notación punto fijo. Por ejemplo para definir un precio puedes usar DECIMAL(4,2) que almacena números en el rango -99.99 hasta 99.99.

- **DATE** Almacena y muestra una fecha en formatos *YYYY-MM-DD, YYY:MM:DD, YYYY/MM/DD*. Ejemplos:

```
mysql> CREATE TABLE test (fecha DATE);
mysql> INSERT INTO test VALUES ('2007/02/0');
mysql> SELECT * FROM testdate;
+------------+
| fecha      |
+------------+
| 2007-02-00 |
+------------+
4 rows in set (0.01 sec)
```

- **TIME** almacena una hora en formatos *DD HH:MM:SS, HH:MM:SS, DD HH:MM, HH:MM, DD HH,* o *SS en el rango* -838:59:59 hasta 838:59:59. Ejemplos:

```
mysql> CREATE TABLE tiempos(id SMALLINT, t TIME);
mysql> INSERT INTO tiempos VALUES(1, "2 13:25:59");
```

- **TIMESTAMP** Almacena y muestra una fecha y hora en el formato *YYYY-MM-DD HH:MM:SS en el rango* 1970-01-01 00:00:00 hasta 2037. Otros formatos son *YYYY-MM-DD HH:MM:SS* o *YY-MM-DD HH:MM:SS*. *Ejemplo:*

```
mysql> CREATE TABLE t1(id INT NOT NULL,
-> ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP);
mysql> INSERT INTO t1 VALUES(1,''), (2,'2006-07-16 1:2:3'), (3, NULL);
mysql> SELECT * FROM mytime;
+----+---------------------+
| id | ts                  |
+----+---------------------+
```

```
| 1  | 0000-00-00 00:00:00 |
| 2  | 2006-07-16 01:02:03 |
| 3  | 2006-07-16 01:05:24 |
+----+---------------------+
3 rows in set (0.00 sec)
```

- **CHAR[(ancho)]** Almacena un string de longitud fija (máximo 255 letras). Rellena con espacios hasta alcanzar ese tamaño. Ej:

```
mysql> CREATE TABLE relleno(s CHAR(10) );
mysql> INSERT INTO relleno VALUES ('a'), ('abc'), ('abcde');
mysql> SELECT * FROM show_padding;
+----------+
| mystring |
+----------+
| a        |
| abc      |
| abcde    |
+----------+
4 rows in set (0.01 sec)
```

- **BOOLEAN** almacena un valor booleano, false si es 0, y true si es distinto de cero. Son sinónimos BOOL y BIT. Equivale a TINYINT(1), y consume 1 byte.

- **TINYINT[(*width*)] [UNSIGNED] [ZEROFILL]** almacena enteros en el rango –128 hasta 127 o 0 hasta 255. Consume 1 byte.

- **SMALLINT[(*width*)] [UNSIGNED] [ZEROFILL]** almacena enteros en el rango –32,768 hasta 32,767 o bien de 0 hasta 65,535. Consume 2 bytes.

- **MEDIUMINT[(*width*)] [UNSIGNED] [ZEROFILL]** almacena enteros de -8,388,608 hasta 8,388,607 o desde 0 hasta 16,777,215. Consume 3 bytes.

- **BIGINT[(*width*)] [UNSIGNED] [ZEROFILL]** almacena enteros en el rango –9,223,372,036,854,775,808 hasta 9,223,372,036,854,775,807 y si usas UNSIGNED o ZEROFILL desde 0 hasta

18,446,744,073,709,551,615. Consume 8 bytes.

- **FLOAT[(width, decimals)] [UNSIGNED] [ZEROFILL]** o **FLOAT[(precision)] [UNSIGNED] [ZEROFILL]** almacena números con decimales en punto flotante. Consume 4 bytes.
- **DOUBLE[(*width*, *decimals*)] [UNSIGNED] [ZEROFILL]** almacena números en punto flotante. Consume 8 bytes de memoria.
- **YEAR[(*digits*)]** almacena 2 o 4 dígitos que representan un año. La versión de 2 dígitos representa años desde 1970 hasta 2069.
- **DATETIME** almacena y muestra una fecha y una hora en el formato *YYYY-MM-DD HH:MM:SS*.
- **VARCHAR(ancho)** almacena texto de longitud variable con una cantidad máxima de caracteres. El límite son 65,535 caracteres. Los espacios traseros se eliminan.
- **BINARY(width)** y **VARBINARY(width)** equivale a CHAR y VARCHAR pero almacena strings binarios.
- **BLOB** almacena datos binarios de hasta 65,535 bytes. No puede usar DEFAULT.
- **TEXT** es como BLOB pero para texto.
- **TINYBLOB** y **TINYTEXT** como BLOB y TEXT, pero un máximo de 255 bytes.
- **MEDIUMBLOB** y **MEDIUMTEXT** como BLOB y TEXT, con un máximo de 16,777,215 bytes.
- **LONGBLOB** y **LONGTEXT** como BLOB y TEXT, pero un máximo de 4

gigabytes.

- **ENUM('*value1*'[,'*value2*'[, ...]]** una lista de strings. La columna puede almacenar uno de esos valores. Ejemplo:

```
mysql> CREATE TABLE frutas( f ENUM('manzana', 'kiwi', 'Pera') );
mysql> INSERT INTO frutas VALUES ('kiwi');
```

- **SET('*value1*'[, '*value2*'[, ...]])** la columna puede contener ninguna o varias de las cadenas del conjunto.

```
mysql> CREATE TABLE frutas2( f SET('kiwi', 'pera', 'uva') );
mysql> INSERT INTO frutas2 VALUES ('pera');
mysql> INSERT INTO frutas2 VALUES ('pera,uva');
mysql> SELECT * FROM frutas2;
+--------------+
| f            |
+--------------+
| pera         |
|              |
| pera,uva     |
+--------------+
3 rows in set (0.01 sec)
```

CLAVES E ÍNDICES

Los índices se utilizan como un mecanismo para localizar los datos más rápidamente y cuando son únicos para implementar las claves primarias y alternativas (secundarias). Para ver los índices que tiene una tabla empleas el comando **SHOW INDEX**:

```
mysql> SHOW INDEX FROM artistas;
+--------+-----------+----------+--------------+------------+-------...
| Table  |Non_unique | Key_name | Seq_in_index | Column_name| Collation...
+--------+-----------+----------+--------------+------------+-----...
|artistas| 0         | PRIMARY  | 1            | artist_id  | A   ...
+--------+-----------+----------+--------------+------------+-----...
... +-------------+----------+--------+------+------------+---------+
... | Cardinality | Sub_part | Packed | Null | Index_type | Comment |
... +-------------+----------+--------+------+------------+---------+
... | 6           |     NULL |        |      |      BTREE |         |
... +-------------+----------+--------+------+------------+---------+
1 row in set (0.00 sec)
```

Puedes añadir (**CREATE INDEX**) o borrar índices (**DROP INDEX**).

```
mysql> DROP TABLE artistas;
mysql> CREATE TABLE artistas (
-> id SMALLINT(5) NOT NULL DEFAULT 0,
-> nombre CHAR(128) DEFAULT NULL,
-> PRIMARY KEY (id),
-> KEY nombre (nombre)
-> );
```

La palabra **KEY**, **UNIQUE** e **INDEX** hace que genere un índice extra:

```
mysql> CREATE TABLE clientes (
-> id INT(4) NOT NULL DEFAULT 0,
-> nombre CHAR(50),
-> apellido CHAR(50),
-> mote CHAR(50),
-> PRIMARY KEY (id),
-> KEY names (nombre, apellido, mote) );
```

CREAR TABLAS

```
CREATE TABLE IF NOT EXISTS artistas (
-> id SMALLINT(5) NOT NULL DEFAULT 0,
-> nombre CHAR(128) DEFAULT NULL,
-> PRIMARY KEY (id)
-> );
```

Palabra AUTO_INCREMENT

Permite dar un valor por defecto distinto a una columna:

```
mysql> DROP TABLE artistas;
mysql> CREATE TABLE artistas(
-> id SMALLINT(5) NOT NULL AUTO_INCREMENT,
-> nombre CHAR(128) DEFAULT NULL,
-> PRIMARY KEY (id)
-> );
mysql> INSERT INTO artist VALUES (NULL, "Police");
```

AÑADIR, ELIMINAR, CAMBIAR COLUMNAS

El comando **ALTER TABLE** permite hacerlo. Cambiar el nombre de una columna:

```
mysql> ALTER TABLE jugadores CHANGE anterior nueva tipo;
```

Cambia nombre de de columna nombra a artista:

```
mysal> ALTER TABLE artistas CHANGE nombre artista CHAR(128)
DEFAULT NULL;
```

Cambiar el tipo o las cláusulas debes usar **MODIFY**:

```
mysql> ALTER TABLE artistas MODIFY nombre CHAR(64) DEFAULT
"Unknown";
```

Puedes hacerlo con CHANGE pero indicando el nombre dos veces:

```
mysql> ALTER TABLE artistas CHANGE nombre nombre CHAR(64) DEFAULT
"Unknown";
```

Para añadir una nueva columna y que aparezca la primera usas FIRST:

```
mysql> ALTER TABLE artist ADD formed YEAR FIRST;
```

Si quieres dejarla en una posición determinada usas AFTER:

```
mysql> ALTER TABLE artistas ADD formado YEAR AFTER id;
```

para borrar una columna usas DROP:

```
mysql> ALTER TABLE artistas DROP formado;
```

AÑADIR, ELIMINAR, CAMBIAR ÍNDICES

Primero crearemos un nuevo índice:

```
mysql> ALTER TABLE artistas ADD INDEX idxNombre(nombre);
mysql> SHOW CREATE TABLE artistas;
```

También puedes crear una clave primaria después de crear la tabla:

```
mysql> ALTER TABLE artistas ADD PRIMARY KEY(id);
```

Para borrar un índice:

```
mysql> ALTER TABLE artistas DROP INDEX idxNombre;
mysql> ALTER TABLE artistas DROP PRIMARY KEY;
```

Los índices no se pueden moficar, hay que rehacerlos:

```
mysql> ALTER TABLE artistas DROP INDEX idxNombre;
mysql> ALTER TABLE artistas ADD INDEX idxNombre( nombre(10) );
```

MODIFICAR OTRAS ESTRUCTURAS

Renombrar tablas y bases de datos:

```
mysql> ALTER TABLE escuchado RENAME TO playlist;
mysql> RENAME DATABASE anterior nueva;
```

Borrar una base de datos o una tabla:

```
mysql> DROP DATABASE [IF EXISTS] musica;
mysql> DROP TABLE IF EXISTS temp;
mysql> DROP TABLE IF EXISTS temp, temp1, temp2;
```

DML

CONSULTAS

```
select expresiones
from tablas_origen
[where condición_de_fila]
[group by expresiones_agrupacion [having condición_grupos] ]
[order by expresiones_orden];
```

Operadores de comparación:

- **a = b** Igualdad de a y b.
- **a <> b, a != b** Diferentes a y b.
- **a <= b, a < b, a >= b, a > b**: menor o igual, menor, mayor o igual, mayor.

- **Columna <=> nully IS [NOT] NULL expresion, ISNULL(e1)** Comprobar si una columna/expresión es NULL. Ej:

```
mysql> select login from login where salir = null; MAL!!!!
mysql> SELECT login FROM login WHERE salir <=> null;
```

- **a BETWEEN** *min* **AND** *max* *cierto cuando a está en el intervalo (min, max).*
- **COALESCE(***list, text***)** Si alguna de las columnas de la lista es NULL se sustituye por el texto.
- **e1 [NOT] IN (***value1, …***)**: cierto si e1 [no] está en la lista de valores.
- **INTERVAL(***n, n1, n2, n3***) cierta si** n < *n1* < *n2* < *n3*.

Alias de columnas:
Dar nombre a columnas que se crean usando expresiones. Sintaxis:

mysql> select expresión AS alias from tabla;

Conectores lógicos:

- **NOT expresión_lógica, ! Expresión_lógica**: negar la expresión.
- **exp_log OR exp_log, exp_log || exp_log**: cierta si alguna de las expresiones lo es. Ejemplo:

```
mysql> SELECT login, role FROM login WHERE salario < 100000 OR
ISNULL(fechacierre);
```

- **exp_log AND exp_log,** exp_log **&& exp_log**: cierta si ambas expresiones lógicas lo son. Ejemplo:

```
mysql> SELECT login, role FROM login WHERE salario < 100000 AND
ISNULL(fechacierre);
```

Funciones de control: controlan la salida de una sentencia según se cumpla una condición.

- **CASE *exp* WHEN *c1* THEN *t1* WHEN *c2*… ELSE *f1* END** Selecciona uno de los valores ti cuando el valor de exp es ci. Ej:

```
mysql> SELECT login,
              CASE WHEN salario > 24900 THEN 'forrao'
                   WHEN salario < 1000  THEN 'pelao'
                   ELSE 'medio'
               END AS 'clase'
FROM login;
```

- **IF(*x1,x2,x3*)** **Si x1 es** true, devuelve *x2, sino x3*. Ej:

```
SELECT login,
       fechainicio,
       IF(desc like "Jefe%","Directivo","Empleado") AS "Grupo"
FROM login;
```

- **IFNULL(*x1,x2*)** muestra x2 si x1 es null. Ej:

```
SELECT login, IFNULL(role, 'Debe tener un role') FROM login;
```

- **NULLIF(*x1,x2*)** Si x1 y x2 son iguales devuelve null. Ej:

```
SELECT login, NULLIF(role,"CEO") as "Role" FROM login;
```

Funciones y operadores para Cadenas:

- **ASCII(*s*)** devuelve el valor ASCII de la priemra letra de s.
- **BIN(*n*)** devuelve una cadena con n en su equivalente binario. **CHAR(*N, N1, N2,* …)** Convierte los números en una cadena ASCII.
- **CONCAT(*s1, s2,* …)** une todos los string.
- **CONCAT_WS(*delimiter, s1, s2,* …)** une las cadenas separándolas por un delimitador.
- **CONV(*n, base1, base2*)** devuelve una cadena del número n expresado en base base1 con su expresión en base2.
- **ELT(*n, s1, s2,* …)** devuelve s1 si n es 1, s2 si es 2, etc.
- **FIELD(*s, s1, s2,* …)** devuelve 1 si s es igual que s1, 2 si es s2, etc.

- **FIND_IN_SET(*s, lista*)** si lista es una lista de cadenas separadas por comas e untenta encontrar s. Devuelve su posición comenzando por 1.
- **HEX(*n*)** devuelve un string con n expresado en hexadecimal.
- **INSERT(*s, pos, length*, s2)** inserta s2 en la posición pos de s y escribe length caracteres.
- **LCASE(*s*)** y **LOWER(*s*)** devuelve s en minúsculas.
- **LEFT(*s, length*)** devuelve length caracteres de s desde la izquierda.
- **LENGTH(*s*), OCT_LENGTH(*s*), CHAR_LENGTH(*s*)** y **CHARACTER_LENGTH(*s*)**: longitud de s.
- exp_cadena **[NOT] LIKE patrón [ESCAPE *'char'*]** Operador que comprueba si la expresión casa con un patrón de cadenas. Usa dos comodines: subrayado (_) representa un cualquier letra y porcentaje (%) representa cualquier cadena , incluso nada. Ej: descripciones de al menos dos letras que comiencen por 'C'.

```
SELECT login, descripcion FROM login
WHERE descripcion LIKE "C_%";
```

- **LOCATE(*s1, s2*), POSITION(*s1* IN *s2*)** y **INSTR(*s2, s1*)** devuelve la posición de s1 en s2. Si no está dentro devuelve 0.
- **LOCATE(*s1, s2, p*)** devuelve posición de s1 en s2 a partir de la posición p.
- **LPAD(*s, len, r*)** y **RPAD(s, len, r)** devuelven la cadena s rellena con s hasta alacanzar la longtud len.
- **LTRIM(*s*)** y **RTRIM(s)** elimina espacios de s por la izq. y der.
- **MATCH (*c1,c2*) AGAINST (*exp*)** Permite recuperar expresiones de filas donde una o varias columnas coincidan. Ej:

```
SELECT * FROM documentos MATCH(bibliografia) AGAINST('TCP/IP');
```

- **MID(*s, pos, length*)** Devuelve una subcadena de s desde pos con una longtud de length caracteres.
- **OCT(*n*)** devuelve la versión octal de n.
- **ORD(*s*)** valor ASCII del primer carácter de s usando Unicode.
- **[NOT] REGEXP *patrón, [NOT]* RLIKE *patrón*** Búsquedas usando expresiones regulares. Ej:

```
SELECT login, descripcion FROM login
WHERE login RLIKE "^ja[a-z]*";
```

- **REPLACE(s, dec, ac)** cambia cadena dec en s por su versión en ac.
- **REPEAT(*s, n*)** cadena con s repetida n veces.
- **REVERSE(*s*)** devuelve s en orden contrario.
- **RIGHT(*s, n*)** cadena con los n caracteres de s desde la derecha.
- **STRCMP(*s1, s2*)** compara s1 con s2. Devuelve 0 si son iguales, -1 si s1 es menor y 1 si s1 es mayor. No diferencia mayúsculas.
- **SUBSTRING(*s,p,lon*), SUBSTRING(*s* FROM *p* FOR *lo*), SUBSTRING(*s,p*)** y **SUBSTRING(*s* FROM *p*)** subcadena de s comenzando en posición s de lon de tamaño.
- **SUBSTRING_INDEX(*s, deli, n*)** lee n-1 delimitadores y devuelve la parte izquierda.
- **SOUNDEX(*s*)**: Si quieres usar la codificación de como suena un mensaje, puedes crear el valor usado con SOUNDEX.
- **TRIM([both | leading | trailing] remove FROM *s*)** Es como usar el RTRIM y LTRIM a la vez.
- **UCASE(*s*)** y **UPPER(*s*)** versión en mayúsculas.

Funciones de grupos:

- **AVG(*e*)** media de la expresión e.
- **COUNT(*e*)** cuenta los valores no null. SI e es un * cuenta filas.
- **COUNT(DISTINCT *e*)** cuenta valores distintos y no null de e.

- **MAX(*e*)** devuelve el máximo valor de e.
- **MIN(*e*)** mínimo valor de e.
- **STD(*e*)** y **STDDEV(*e*)** desviación estándar de e.
- **SUM(*e*)** suma de valores no null de e.

Funciones de fecha y hora:

- **CURDATE()** y **CURRENT_DATE** fecha actual, formato YYYY-MM-DD
- **CURTIME()** y **CURRENT_TIME** hora actual.
- **DATE_FORMAT(*d*, *f*)** y **TIME_FORMAT(*t*, *f*)** devuelve fechas y horas cambiando el formato. Los formatos son:

```
%M Month name (January)
%W Weekday name (Sunday)
%D Day with English suffix (1st, 2nd,)
%Y 4-digit year
%y 2-digit year
%X 4-digit year for the week, where Sunday is the first day of the
week combined with '%V'
%x 4-digit year for the week, where Monday is the first day of the
week combined with '%v'
%a Abbreviated weekday name (Sun)
%d Day of the month
%e Day of the month
%m Month (01)
%c Month (1)
%b Abbreviated month name (Jan)
%j Day of the year (001)
%H Hour (00..23)
%k Hour (0..23)
%h Hour (01..12)
%I Hour (01..12)
%l Hour (1..12)
%i Minutes, numeric (00..59)
%r Time, 12-hour (hh:mm:ss [AP]M)
%T Time, 24-hour (hh:mm:ss)
%S Seconds (00..59)
%s Seconds (00..59)
%p AM or PM
%w Day of the week (0=Sunday)
%U Week (0..53), where Sunday is the first day of the week
%u Week (0..53), where Monday is the first day of the week
%V Week (1..53), where Sunday is the first day of the week
combined with '%X'
```

```
%v Week (1..53), where Monday is the first day of the week
combined with '%x'
```

- **DAYNAME(*d*)** nombre del día.
- **DAYOFMONTH(*d*)** día del mes.
- **DAYOFWEEK(*d*)** día de la semana (1 es domingo).
- **DAYOFYEAR(*d*)** Día del año.
- **FROM_DAYS(*dias*)** fecha representada por días.
- **FROM_UNIXTIME(unix_timestamp) y FROM_UNIXTIME(unix_timestamp, formato)** fecha/hora a partir de un timestamp de UNIX.
- **HOUR(*t*)** hora de un time.
- **MINUTE(*t*)** minutos de un time.
- **MONTH(*d*)** mes de una fecha.
- **MONTHNAME(*d*)** nombre del med de la fecha d.
- **NOW(), SYSDATE()** y **CURRENT_TIMESTAMP** fecha y hora actuales en el formato YYYY-MMDD HH:MM:SS.
- **PERIOD_DIFF(*d1, d2*)** número de meses de diferencia entre las dos fechas d1 y d2.
- **QUARTER(*d*)** el trimestre de la fecha d.
- **SECOND(*t*)** segundos de t.
- **SEC_TO_TIME(*s*)** convierte una cantidad s de segundos a hora HH:MM:SS.
- **TIME_TO_SEC(*t*)** convierte una hora a una cantidad de segundos.
- **TO_DAYS(*d*)** número de días hasta la fecha.
- **UNIX_TIMESTAMP()** y **UNIX_TIMESTAMP(*d*)** devuelven un Unix timestamp basado en el número de segundos desde 1970-01-01 00:00:00 GMT.
- **WEEK(*d*)** semana de la fecha (de 0 a 53).
- **WEEK(*d, start*)** si start es 0 la semana comienza en domingo, si es 1 comienza en lunes.
- **WEEKDAY(*d*)** número del dia de la semana, 0 es lunes.

- `YEAR(d)` año de la fecha d.
- `YEARWEEK(d)` devuelven año y med de la fecha en formato YYYYMM.
- `YEARWEEK(d, s)` si s es 0 comienza en domingo y si es 1 en lunes.

Otras funciones:

- `BINARY` un operador que pasa a binario una columna. Permite hacer comparaciones case sensisitive por ejemplo.
- `CONNECTION_ID()` un entero que identifica el hilo de la conexión al servidor.
- `DATABASE()` nombre de la base de datos actual.
- `DECODE(s1, s2)` desencripta *s1* basándose en *s2*.
- `ENCRYPT(s [,semilla])` encrita s utilizando la función de Unix crypt(). Si no encuentra esta llamada al sistema, devuelve null.
- `ENCODE(s, s2)` encripta s basándose en s2.
- `FORMAT(value, d)` formatea el número con comas y lo redondea a d decimales.
- `LAST_INSERT_ID()` cuando una tabla tiene una columna autoincremento, esta función devuelve el último valor usado.
- `MD5(s)` calcula un checksum de s.
- `PASSWORD(s)` convierte s en una cadena encriptada.
- `USER()`, `SYSTEM_USER()` y `SESSION_USER()` usuario logueado.
- `VERSION()` versión del SGBD.

Alias de Tablas:

Permiten poner un mote a una tabla. Son necesarias cuando usamos la misma tabla varias veces en la misma sentencia y útiles al usar más de una tabla (aunque no sean la misma) en sentencias SQL.

Cláusula DISTINCT:

Elimina valores duplicados tras haberlos recuperado. Si se usa dentro

de una agregada, se eliminan los valores repetidos de la columna antes de hacer el cálculo. Si se usa en la select, elimina las filas repetidas del resultado. Ejemplo:

```
mysql> SELECT DISTINCT nombre FROM
-> artistas INNER JOIN album USING (artista_id);
```

Cláusula GROUP BY:

Agrupa filas que coincidan en los valores de una o varias columna o expresiones (criterio de agrupación). Cambia la granularidad del resultado, ya no puedes pedir información de cada fila, sino de grupo (común a todas las filas del grupo), constantes y agregadas. Ejemplos:

```
SELECT artista_nombre, album_nombre, COUNT(*)
-> FROM artistas INNER JOIN album USING (artista_id)
-> INNER JOIN canciones USING (artista_id, album_id)
-> GROUP BY artistas.artista_id, album.album_id;
```

JOINS de tablas:

Las reuniones de tablas más comunes son las internas (INNER JOIN) que unen las filas de dos tablas que casan en una condición que puedes indicar en USING(). También puedes usar la palabra NATURAL para indicar que el criterio es la igualdad entre la clave primaria y ajena de las tablas que intervienen. Ej:

```
mysql> SELECT ar.nombre, al.nombre FROM
-> artistas ar INNER JOIN album al USING (artista_id);
mysql> SELECT artist_name, album_name FROM
-> artistas NATURAL JOIN album;
```

Las filas que no casan en un join interno de cualquiera de las dos tablas que intervienen se pierden. A veces no te interesa. Para recuperarlas se usa el left join (recupera filas perdidas en la tabla de la izquierda) y right join (recupera las de la derecha).

```
mysql> SELECT track_name, played FROM
```

```
-> track LEFT JOIN played USING(artista_id,album_id, track_id)
-> ORDER BY played DESC;
```

Consultas anidadas:

Consiste en escribir sentencias que contengan otras sentencias hijas en su interior. Dan mucha potencia a SQL. Las subconsultas se escriben encerradas entre paréntesis, pueden ser independientes de los datos que recupera la sentencia padre, o que sus datos dependan de los de la sentencia padre. Pueden recuperar un solo valor o un cjunto de ellos, en cualquier caso deben combinarse con operadores adecuados. Ej: artistas que han hecho un album titulado "In A Silent Way". La primera sentencia lo encuentra con joins, la segunda con subconsultas:

```
mysql> SELECT artist_name FROM
-> artist INNER JOIN album USING (artist_id)
-> WHERE album_name = "In A Silent Way";

mysql> SELECT artist_name FROM artist WHERE artist_id =
-> (SELECT artist_id FROM album WHERE album_name = "In A Silent
Way");
```

Las subconsultas se ayudan de operadores para trabajar con conjuntos de datos como son **ANY**, **SOME**, **ALL**, **IN**, y sus versiones **NOT** además de la cláusula **EXISTS()** sobre todo para correlacionadas.

```
mysql> SELECT engineer_name, years
-> FROM engineer WHERE years > ANY (SELECT years FROM producer);

SELECT producer_name FROM producer WHERE producer_name
-> IN (SELECT engineer_name FROM engineer);
```

Las subconsultas además de en la parte WHERE, pueden utilizarse en la parte SELECT de las sentencias. En este caso solo pueden devilver un valor. Ej: estas dos sentencias son equivalentes.

```
mysql> SELECT producer_name, producer.years FROM
-> producer, engineer WHERE producer_name = engineer_name AND
```

```
-> producer.years = engineer.years;

mysql> SELECT producer_name, years FROM producer WHERE
-> (producer_name, years) IN
-> (SELECT engineer_name, years FROM engineer);
```

Las consultas correlacionadas (su resultado depende de la fila con la que trabaja la sentencia padre en cierto momento) utiliza la cláusula `[Not] EXISTS()` que es true cuando se recupera algo y false cuando no existe. Ej:

```
mysql> SELECT * FROM artist WHERE EXISTS(SELECT * FROM played);

mysql> SELECT artist_name FROM artist WHERE EXISTS
-> (SELECT * FROM album WHERE album_name = artist_name);
```

Las subconsultas también pueden estar en el FROM, sirven para preprocesar los datos de alguna manera para la sentencia padre.

```
mysql> SELECT producer_name, months FROM
-> (SELECT producer_name, years*12 AS months FROM producer) AS
prod;

mysql> SELECT AVG(albums) FROM
-> (SELECT COUNT(*) AS albums FROM artist INNER JOIN album
-> USING (artist_id) GROUP BY artist.artist_id) AS alb;
```

INSERCIONES

Sentencia que permite añadir filas a una tabla. Ejemlo:

```
mysql> INSERT INTO played VALUES (7, 1, 2, DEFAULT);
mysql> INSERT INTO played (artist_id, album_id, track_id)
-> VALUES (7, 1, 1);
```

BORRADOS

Sentencia que permite eliminar filas de una tabla. Ejemplos:

```
mysql> DELETE FROM played WHERE played < "2006-08-15";
```

MODIFICACIONES

Sentencia que permite modificar las filas exisentes en una tabla. Ej:

```
mysql> UPDATE artist SET artist_name = UPPER(artist_name);
```

10.1.3. CÓDIGO ALMACENADO EN EL SGBD.

Declaracion de variables:

La sentencia **DECLARE** crea una variable dentro de un bloque de código. Sintaxis y Ejemplos:

```
DECLARE variable [,variable1... ] tipoDato [DEFAULT valor];

DECLARE l_int1 INT DEFAULT -2000000;
DECLARE l_double DOUBLE DEFAULT 2e45;
DECLARE l_date DATE DEFAULT '1999-12-31';
DECLARE l_char CHAR(255) DEFAULT 'Se rellena hasta 255';
DECLARE l_varchar VARCHAR(255) DEFAULT 'No se rellena';
DECLARE l_text TEXT DEFAULT 'Una cadena grande';
```

Asignar valores a variables:

Usas la sentencia **SET**. Sintaxis:

```
SET variable1 = expresion [, variable2 = expresion ... ]
```

Parámetros de funciones y procedimientos almacenados

Pueden definirse de entrada (datos que se pasan al subprograma) o de salida (datos que pasa el subprograma a quien lo llame) o ambas cosas.

```
CREATE PROCEDURE|FUNCTION( [ [IN|OUT|INOUT] parámetro1 tipo...] )
```

Variables de usuario:

Pueden manipularse dentro o fuera de subprogramas, son accesibles desde cualquier parte, como variables globales.

Comentarios

Hay de dos tipos:

- Una línea: lo que sigue a dos guiones: (--)
- De varias líneas: /* comienza y acaba */

OPERADORES

operadores aritméticos: permiten realizar cálculos numéricos.

Table 3-2. MySQL mathematical operators

Operator	Description	Example
+	Addition	SET var1=2+2; → 4
-	Subtraction	SET var2=3-2; → 1
*	Multiplication	SET var3=3*2; → 6
/	Division	SET var4=10/3; → 3.3333
DIV	Integer division	SET var5=10 DIV 3; → 3
%	Modulus	SET var6=10%3 ; → 1

Operadores a nivel de bits: permiten manipular bits.

Operator	Use
|	OR
&	AND
<<	Shift bits to left
>>	Shift bits to right
~	NOT or invert bits

Estructuras de bloques:

Los subprogramas y algunas sentencias de control de flujo agrupan sentencias en bloques. Cada bloque comienza con la palabra **BEGIN** y acaba con **END** en el caso de funciones, procedimientos y triggers. Ejemplo:

```
CREATE {PROCEDURE|FUNCTION|TRIGGER} nombre
BEGIN
      sentencias...
END;
```

EJECUCIÓN CONDICIONAL

La sentencia **IF-ELSE-ELSEIF** es muy similar a la mayoría de lenguajes. Sintaxis:

```
IF expression THEN commands
[ELSEIF expression THEN commands ....]
[ELSE commands]
END IF;
```

Ejemplo:

```
IF venta < 200 THEN
        CALL venta_libre(id_venta);
ELSEIF venta > 200 AND cliente_estado = 'PREMIUM' THEN
        CALL venta_libre(id_venta);
        CALL aplica_descuento(id_venta, 10);
ELSE
        CALL aplica_descuento(id_venta, 5);
END IF;
```

SENTENCIA LOOP

Es el bucle más sencillo, un bucle incondicional. Sintaxis:

```
[label:] LOOP
        Sentencias...
END LOOP [label];
```

Ejemplos:

```
bucle_infinito: LOOP
        SELECT 'Bienvenido al infierno de un bucle infinito!!';
END LOOP bucle_inifinito;
```

Sentencia LEAVE

Sale de cualquier bucle. Es como el break de Java o de C++. Sintaxis:

```
LEAVE label;
```

Ejemplo:

```
SET i= 1;
unBucle: LOOP
```

```
        SET i = i + 1;
        IF i = 10 THEN
                LEAVE unBucle;
        END IF;
END LOOP unBucle;
SELECT 'He contado hasta 10';
```

Sentencia ITERATE:

Termina la iteración actual y continúa con la siguiente. Es como el continue de Java o de C++. Sintaxis:

```
ITERATE label;
```

Bucle REPEAT … UNTIL

Repite las sentencias que delimita hasta que la expresión del UNTIL sea cierta. Sintaxis:

```
[ label:] REPEAT
        statements
<UNTIL expresion END REPEAT [ label]
```

Ejemplo:

```
SET i=0;
bucle1: REPEAT
        SET i = i + 1;
        IF MOD(i,2) <> 0 THEN /* par */
                Select concat(i," es un número par");
        END IF;
UNTIL i >= 10
END REPEAT;
```

Bucle WHILE

Repite las sentencias del bloque que encabeza mientras la condición del WHILE sea cierta. Sintaxis:

```
[label:] WHILE expresion DO
        sentencias...
END WHILE [ label]
```

Ejemplo:

```
SET i=1;
loop1: WHILE i <= 10 DO
       IF MOD(i,2)<>0 THEN /* par */
              SELECT CONCAT(i, " es par");
       END IF;
       SET i = i + 1;
END WHILE loop1;
```

USAR SQL EN PROGRAMAS

Puedes usar cualquier sentencia SQL (DML, DDL y sentencias de utilidades) dentro de programas. Eso si, hay algunas diferencias. Por ejemplo, la sentencia SELECT necesita una cláusula **INTO** para almacenar expresiones en variables del programa. Sintaxis:

```
SELECT expresson1 [, expresion2 ....]
INTO variable1 [, variable2 ...]
FROM ...
[WHERE ...]
[GROUP BY ... [HAVING ...] ]
[ORDER BY ...]
```

Ejemplos:

```
CREATE PROCEDURE simple_sql()
BEGIN
    DECLARE i INT DEFAULT 1;
    /* Ej. De sentencia de utilidad */
    SET autocommit = 0;
    /* Ej. de DDL */
    DROP TABLE IF EXISTS test_tabla;
    CREATE TABLE t1(id INT PRIMARY KEY, d VARCHAR(30)) ENGINE=innodb;
    /* Ej. de DML */
    WHILE (i <= 10) DO
        INSERT INTO test_tabla VALUES(i, CONCAT("fila",i));
        SET i = i + 1;
    END WHILE;
    SET i=5;
    UPDATE test_tabla SET d = CONCAT("He actualizado fila ",i) WHERE id= i;
    DELETE FROM test_tabla WHERE id>i;
END;
```

Ejemplo: usar una select para recuperar valores de una sola fila.

```
CREATE PROCEDURE detalles_cliente(p_cliente INT)
BEGIN
        DECLARE l_cli_nombre VARCHAR(30);
        DECLARE l_cli_ape1 VARCHAR(30);
        DECLARE l_cli_puesto VARCHAR(30);
        SELECT cli_nombre, cli_ape1,cli_puesto
               INTO l_cli_nombre,l_cli_ape1,l_cli_puesto
        FROM clientes
        WHERE cli_id = p_cliente;
        /* Hacer algo... */
END;
```

Creando y usando cursores:

Cuando se recuperan valores de varias filas necesitamos usar cursores que puedan recorrer el conjunto de datos resultado de la consulta.

Definir un Cursor

Defines el cursor con la sentencia **DECLARE** escrita detrás de la definición del resto de variables. Sintaxis:

```
DECLARE nombre_cursor CURSOR FOR sentencia_SELECT;
```

Un cursos puede referenciar variables del programa dentro de la cláusula WHERE o en la lista de columnas.

```
/* Declarar un cursor antes de una variable genera error 1337 */
mysql> CREATE PROCEDURE cursor_malo()
BEGIN
        DECLARE c CURSOR FOR SELECT * from departmentos;
        DECLARE i INT;
END;
ERROR 1337 (42000): Variable or condition declaration after cursor or
handler declaration

/* Declaración correcta de un cursor */
DECLARE cursor1 CURSOR FOR
                SELECT cli_nombre, cli_ape1,cli_puesto
                FROM clientes;

/* Incluyendo variables del programa */
CREATE PROCEDURE cursor_demo(p_cli INT)
BEGIN
```

```
DECLARE v_id INT;
DECLARE v_nombre VARCHAR(30);
DECLARE c1 CURSOR FOR
         SELECT cli_id, cli_nombre
         FROM clientes
         WHERE cli_id >= p_cli;
```

Abrir el cursor (OPEN)

Ejecuta la sentencia y obtiene los datos resultado. Sintaxis:

```
OPEN nombre_cursor;
```

Recuperar una fila (FETCH)

Para recuperar la fila actual y mover el puntero a la siguiente. Sintaxis:

```
FETCH cursor INTO lista_variables;
```

Cerrar el cursor (CLOSE)

Desactiva el cursor y libera la memoria que consume. Sintaxis:

```
CLOSE nombre_cursor;
```

Ejemplo:

```
DECLARE cursor1 CURSOR FOR SELECT cli_nombre FROM clientes
DECLARE CONTINUE HANDLER FOR NOT FOUND SET ultima_fila= 1;
SET ultima_fila= 0; /* false */
OPEN cursor1;
cursor_loop:LOOP
        FETCH cursor1 INTO cli_nombre;
        IF ultima_fila = 1 THEN
                LEAVE cursor_loop;
        END IF;
        /* hacer algo... */
END LOOP cursor_loop;
CLOSE cursor1;
SET ultima_fila = 0;
```

Manejar errores SQL Errors:

Cuando aparece un error en tiempo de ejecución en un programa, el error se devuelve al programa que ha realizado la llamada. Si no

quieres que ocurra esto, puedes definir un manejador de errores. Sintaxis:

```
DECLARE {CONTINUE | EXIT} HANDLER FOR
{SQLSTATE sqlstate_code| MySQL_error_code| nombre_condición}
sentencia_programa_almacenado;
```

SUBPROGRAMAS ALMACENADOS EN EL SGBD

PROCEDIMIENTOS ALMACENADOS:

Permiten ejecutar código que está almacenado en una BD del SGBD. Se le pueden pasar parámetros de entrada, salida y entrada-salida. Sintacis:

```
CREATE PROCEDURE nombre ([ parametro[,...])
[LANGUAGE SQL]
[ [NOT] DETERMINISTIC ]
[ {CONTAINS SQL|MODIFIES SQL DATA|READS SQL DATA|NO SQL} ]
[SQL SECURITY {DEFINER|INVOKER} ]
[COMMENT comment_string]
bloque_sentencias
```

FUNCIONES ALMACENADAS

Como procedures pero además pueden devolver un valor con la sentencia **RETURN valor**, lo que hace posible su uso en SQL. No pueden devolver result sets. Sintaxis:

```
CREATE FUNCTION nombre ([ parámetro[,...])
RETURNS tipo_dato
[LANGUAGE SQL]
[ [NOT] DETERMINISTIC ]
[ { CONTAINS SQL|NO SQL|MODIFIES SQL DATA|READS SQL DATA} ]
[SQL SECURITY {DEFINER|INVOKER} ]
[COMMENT comment_string]
bloque_sentencias
```

TRIGGERS

Código que no se llama para ejecutarlo, se dispara ante la aparición de

un evento al que está asociado. Sintaxis:

```
CREATE [DEFINER = {user|CURRENT_USER }] TRIGGER nombre
{BEFORE|AFTER} {UPDATE|INSERT|DELETE} ON tabla
FOR EACH ROW
bloque_sentencias;
```

10.2 DRIVERS Y CONEXIÓN CON EL SGBD.

JDBC es un conjunto de API's que permiten conectar tus programas Java con una gran variedad de bases de datos relacionales (BDR). La API JDBC se define en 2 packages:

- **java.sql** aporta acceso y procesamiento de datos (objetos como `Connection`, `ResultSet`, `Statement` y `PreparedStatement`).
- **javax.sql** ofrece acceso a orígenes de datos del lado del servidor (objetos como `DataSource` y `RowSet`).

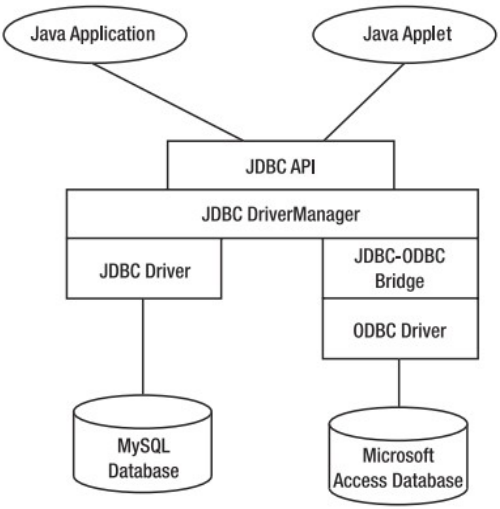


Figura: componentes de aplicación JDBC.

La API JDBC realiza la mayor parte de sus tareas usando la clase **`java.sql.DriverManager`** que es una clase factoría para crear conexiones (cada conexión con una BDR es una instancia de **`java.sql.Connection`**) y el `DriverManager` usa drivers para crearlas. Cada vendedor de BDR (Oracle, MySQL, etc.) ofrece un conjunto de drivers. El funcionamiento es:

- El programa Java llama a JDBC (usando java.sql y javax.sql).
- JDBC carga un driver, por ejemplo Oracle con la sentencia: `Class.forName("oracle.jdbc.driver.OracleDriver")`
- Esto crea una instancia del driver y lo registra en `DriverManager`.
- El driver interactúa con una BDR en concreto.

¿Quién crea estos drivers JDBC? Los fabricantes de BDR los ofrecen junto a sus BDR y son un conjunto de clases que implementan interfaces como `java.sql.Driver` para un determinado SGBDR. Una aplicación Java usa la clase `DriverManager` para obtener un objeto `java.sql.Connection`. A través de él, puedes crear objetos `Statement/PreparedStatement/CallableStatement`, que pueden ejecutar sentencias SQL y procedimientos almacenados. Los datos que devuelven en objetos `ResultSet`. Las clases más importantes que debes conocer para trabajar con BD desde programas Java son:

- **DriverManager**: carga drivers JDBC en memoria y crea conexiones.
- **Connection**: una intrerface que te permite crear objetos Statement, PreparedStatement y CallableStatement.
- **Statement**: interface que representa una sentencia SQL. Puedes usarla para crear objetos ResultSet.
- **PreparedStatement**: interface que extiende a Statement y representa una sentencia SQL precompilada.
- **CallableStatement**: representa un procedimiento almacenado en el SGBDR y te permite ejecutarlo.
- **ResultSet**: interface que representa los datos devueltos por una sentencia SELECT ejecutada en el SGBD.
- **SQLException**: clase que da información sobre errores ocurridos en el SGBD.

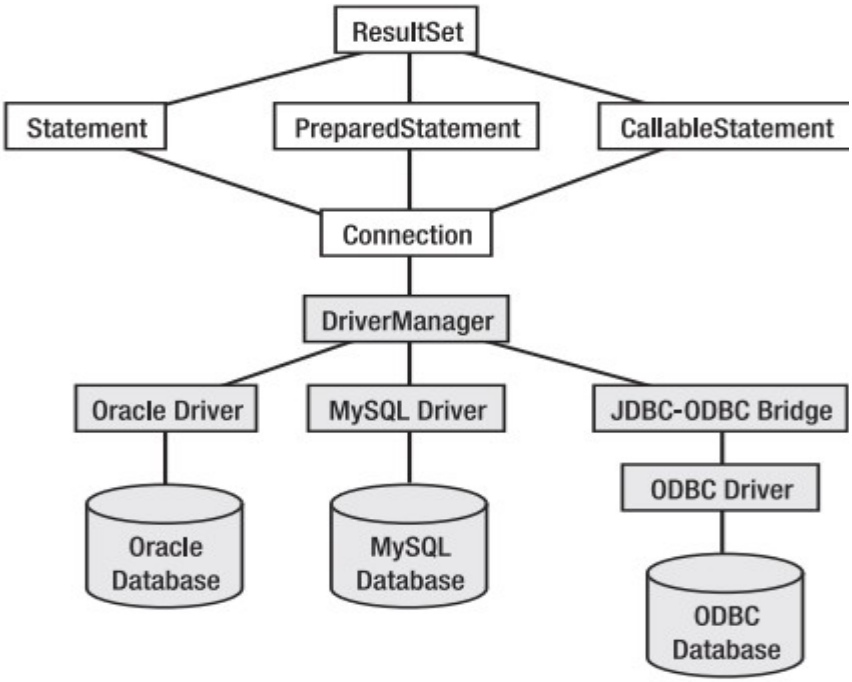


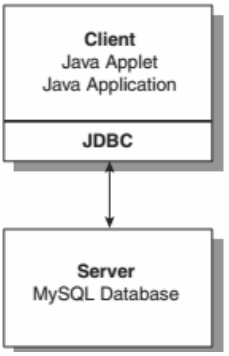
Figura: elementos usados en un programa Java.

Si observas la figura, en la parte derecha y abajo verás una de las BD con las siglas ODBC ¿Qué es ODBC? Son las siglas de Open Database Connectivity y es una API de Microsoft que ofrece una interfaz común para que las aplicaciones Windows puedan acceder a SGBDR sobre una red. Los fabricantes de los SGBD también aportan estos drivers pero Java a partir de Java 8 ya no da soporte. En este enlace tienes acceso a tutoriales de operaciones básicas:

https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html

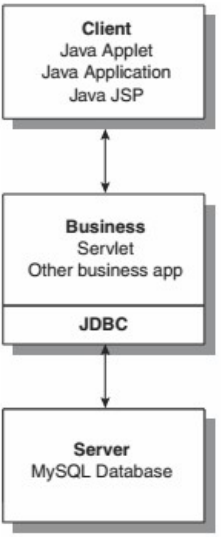
10.2.1. MODELOS DE APLICACIONES (2 y 3 CAPAS).

JDBC permite los modelos de dos y tres capas para el acceso a BD.



En el modelo de dos capas (two-tier)
Una aplicación se comunica directamente con la BD. Requiere un driver JDBC que se pueda comunicar con el SGBD al que se accede.

Las sentencias SQL del usuario son enviadas a la BD y los resultados de la ejecución de estas sentencias son devueltos al usuario. La BD puede estar en otra máquina a la que el usuario accede a través de la red (Internet o Intranet). Es una arquitectura cliente/servidor, con la máquina del usuario como cliente y la máquina que contiene el SGBD como servidor.



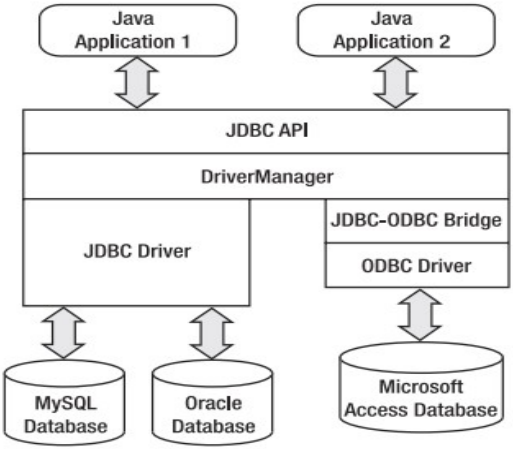
Modelo de 3 capas (three-tier)
Los comandos son enviados a una capa intermedia (middle-tier) de servicios, la cual envía sentencias SQL a la BD. La BD procesa las sentencias y devuelve los resultados a la capa intermedia que se los enviará al usuario.

Este modelo es bastante interesante, ya que la aplicación intermedia no tiene las restricciones de seguridad de las aplicaciones web y da más libertad al programador.

Otra ventaja es que el usuario puede utilizar una API de más alto nivel, y por lo tanto más sencilla de manejar, que será traducida por la capa intermedia a las llamadas apropiadas, en este caso utilizando el JDBC.

9.2.2. EL DRIVER JDBC.

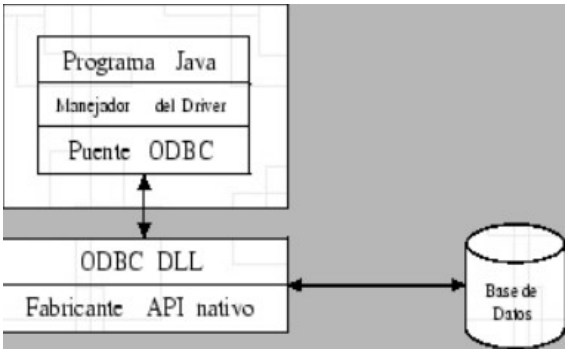
La columna vertebral de JDBC es el **Driver Manager**. JDBC ofrece dos conjuntos de clases e interfaces, unas de más alto nivel para los programadores de aplicaciones y otras de más bajo nivel enfocadas a los programadores de drivers.



Tipos de Drivers JDBC

Existen 4 tipos de drivers JDBC, cada tipo presenta una filosofía de trabajo diferente:

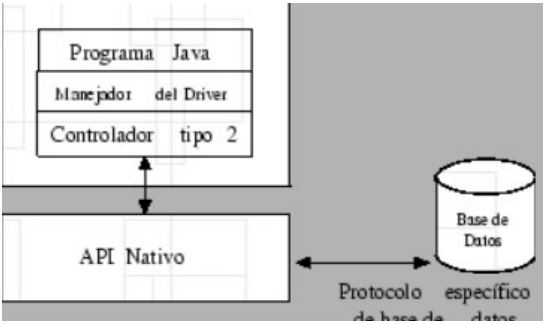
JDBC-ODBC bridge plus ODBC driver (tipo 1): permite acceder a fuentes de datos ODBC existentes mediante JDBC. Implementa operaciones JDBC traduciéndolas a operaciones ODBC, se encuentra dentro del paquete **sun.jdbc.odbc** incluido dentro del JDK a partir de la versión 1.1 y contiene librerías nativas para acceder a ODBC. Ya no está soportado en las versiones actuales de Java.



Native-API partly-Java driver (tipo 2): similares a los drivers de tipo 1 porque también necesitan una configuración en la

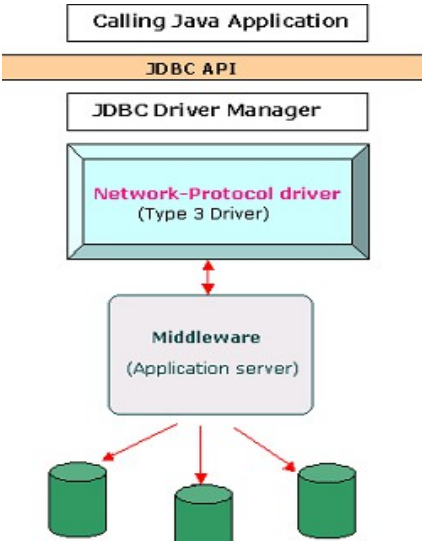
máquina cliente.

Este tipo de driver convierte llamadas JDBC a llamadas de Oracle, MySQL, Sybase, Informix, DB2 u otros SGBD. Tampoco se pueden utilizar dentro de applets al poseer código nativo.



JDBC-Net pure Java driver (tipo 3): traduce las llamadas JDBC a un protocolo independiente del SGBD, que se traduce por un servidor para comunicarse con un SGBD concreto. **Con este tipo de drivers no se necesita ninguna configuración especial en el cliente**, permiten el diálogo con un componente negociador que dialoga a su vez con las BD.

Es ideal para aplicaciones con arquitectura basada en el modelo de 3 capas. Un ejemplo de utilización de este driver puede ser un applet/aplicación web que se comunica con una aplicación intermedia en el servidor y es esta aplicación intermedia la encargada de acceder a la BD. Esta forma de trabajo hace a este tipo de drivers ideales para trabajar con Internet y grandes intranets, ya que el **applet/aplicación web es independiente de la plataforma** y puede ser descargado



completamente desde el servidor Web. El intermediario puede estar escrito en cualquier lenguaje de programación, ya que se ejecutará en el servidor.

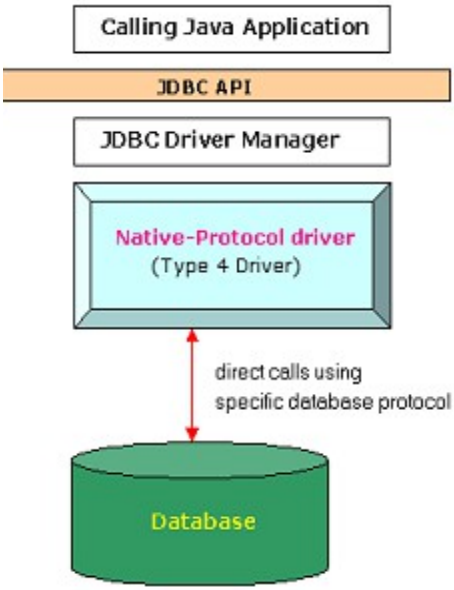
Pero si el intermediario se programa en Java, se tendrá la ventaja de la independencia de la plataforma, además se debe tener en cuenta que el intermediario al ser una aplicación Java no posee las restricciones de seguridad de apliaciones ejecutadas en el cliente, por lo que se podrá acceder a cualquier servidor, no solamente desde el que se cargó la aplicación. El problema que presentan es que su utilización es bastante compleja.

Native-protocol pure Java driver (tipo 4): convierte directamente las llamadas a JDBC al protocolo usado por el SGBD. Esto permite una comunicación directa entre la máquina cliente y el servidor en el que se encuentra el SGBD.

Son como los drivers de tipo 3 pero sin el intermediario y tampoco requieren ninguna configuración en la máquina cliente.

La complejidad del programa intermedio es eliminada. Los drivers de tipo 4 se pueden utilizar para servidores Web de tamaño pequeño y medio así como para intranets.

Estos drivers utilizan protocolos propietarios, por lo tanto son los propios fabricantes de BD los que



ofrecerán estos drivers. **Nosotros usaremos un driver de tipo 4 llamado mysql connector/J-8**. Este driver implementa la versión 3.0 y 4.2 de JDBC. Mediante este driver podemos acceder a BD en el SGBD MySQL 8 a 5.6 (ver https://dev.mysql.com/doc/connector-j/8.0).

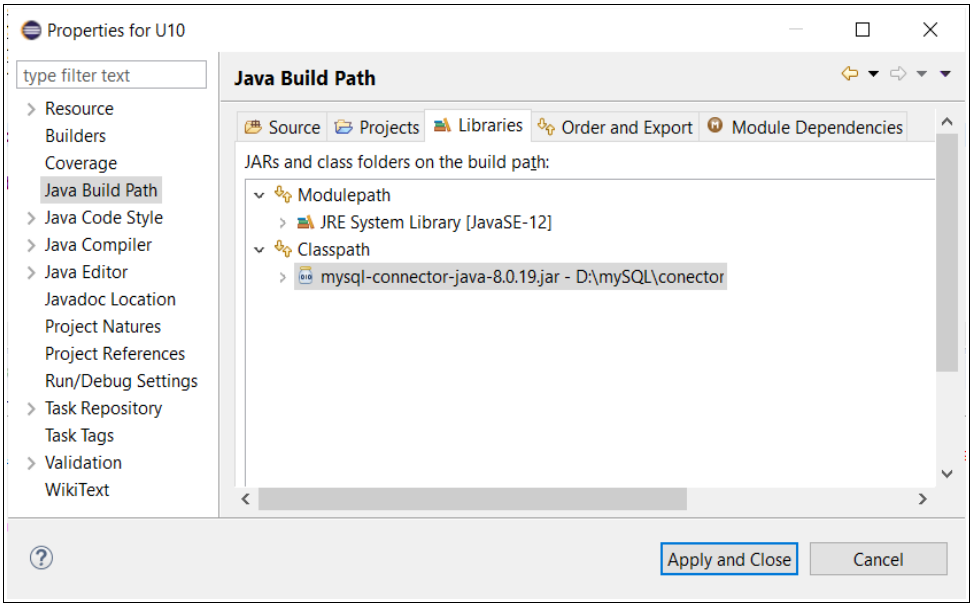
Nota: en Eclipse debemos añadir al fichero module-info la línea:

```
requires java.sql;
```

INSTALAR EL DRIVER DE CONEXIÓN.

Hay varias formas de instalar el driver después de descargarlo. La primera es copiar los ficheros /com y /org en un directorio incluido en tu classpath.

Otra opción es añadir la ruta completa al fichero JAR a la variable CLASSPATH del sistema donde desarrolles.



También puedes copiar el fichero JAR a la carpeta $JAVA_HOME/jre/lib/ext del JRE.

En Windows, copia el fichero JAR del driver a la carpeta donde tengas instalado el JRE, en la carpeta /lib/ext. En nuestro caso no funcionará si el JRE lo tienes aparte.

Por último, si usas Eclipse por ejemplo, prueba a añadir al proyecto un enlace a un JAR externo. Proyecto -> propiedades -> Java Builder Path -> Add Externall Jar... como se ve en la figura anterior.

En GNU/Linux, en /usr/java/versión.../jre/lib/ext

Probar la Instalación del Conector

Según el entorno que uses para desarrollar y como lo tengas configurado, las pruebas se harán de una manera o de otra. Las posibilidades son muchas. Si te toca usarlo en un entorno en concreto tendrás que buscarte la vida. Un buen comienzo es mirar la documentación del fabricante.

10.2.3. CONECTANDO A UNA BD CON JDBC.

Antes de comenzar necesitamos saber exactamente cuál es el nombre de la clase para el driver de MySQL. En nuestro caso vamos a utilizar el driver **JDBC de MySQL**: `com.mysql.jdbc.Driver` pero que en versiones recientes cambia a `com.mysql.cj.jdbc.Driver`

La clase DriverManager

La clase *`java.sql.DriverManager`* trabaja entre el usuario y los drivers. **Tiene en cuenta los drivers disponibles y a partir de ellos establece una conexión entre una BD y el driver adecuado**. Además de esta labor principal, se ocupa de mostrar mensajes de log (registro de actividad) del driver y también el tiempo límite en espera de

conexión (time-out). Normalmente, en un programa Java el único método que un programador deberá utilizar de la clase **DriverManager** es el método ***getConnection()***. Como su nombre indica este método establece una conexión con una BD.

JDBC permite al programador utilizar los métodos ***getDriver()***, ***getDrivers()*** y ***registerDriver()*** de la clase DriverManager, así como el método ***connect()*** de la clase **Driver**, pero en la mayoría de los casos es mejor dejar a la clase DriverManager manejar los detalles del establecimiento de la conexión.

La interfaz Driver

JDBC simplemente ofrece una especificación para acceso a BD, el fabricante de un driver debe seguir las recomendaciones de esta especificación si quiere construir un driver para JDBC.

La interfaz **java.sqlDriver** pertenece al subconjunto del API de JDBC denominado de bajo nivel, ya que lo deberán utilizar los programadores de drivers. Todo driver JDBC debería implementar la interfaz **Driver**.

Los métodos más importantes que ofrece esta interfaz son: ***connect()***, que realiza la conexión con la BD; ***acceptsURL()***, que devolverá **true** si el driver puede abrir la conexión con la URL ofrecida; ***jdbcCompliant()***, devolverá **true** si el driver cumple con la especificación JDBC.

Registrando los Drivers

La clase **DriverManager** mantiene una lista de clases **Driver** que se han registrado ellas mismas llamando a su método ***registerDriver()***. Todas las clases **Driver** deberían estar escritas con una sección estática que crea una instancia de la clase y acto seguido se registra con el método estático ***DriverManager.registerDriver()***, este proceso se ejecuta automáticamente cuando se carga la clase del driver.

Por tanto, gracias a este mecanismo, un programador normalmente no debe llamar directamente al método ***registerDriver()***, lo hace automáticamente el driver cuando es cargado. Una clase Driver es cargada, y por tanto registrada con el DriverManager, de dos maneras posibles. La primera es llamar al método estático *forName()* de la clase ***Class***. Esto carga explícitamente la clase del driver.

```
//Registrando el Driver
String driver = "com.mysql.jc.jdbc.Driver";
Class.forName(driver).newInstance();
```

La otra forma (menos recomendable), es añadir el driver a la propiedad del sistema ***jdbc.drivers***. Esta es una lista que carga el DriverManger, y que está formada por los nombres de las clases de los drivers separados por dos puntos (:).

Cuando la clase DriverManager es inicializada, busca la propiedad del sistema jdbc.drivers, y si el programador ha añadido a esta propiedad uno o más drivers el DriverManager los carga.

Estableciendo la conexión. Interfaz Connection.

Una vez que las clases de los drivers se han cargado y registrado con el DriverManager, están disponibles para establecer una conexión con una BD. Cuando se realiza una petición de conexión con una llamada al método *DriverManager.getConnection()*, la clase DriverManager chequea cada uno de los drivers disponibles para comprobar si puede establecer la conexión.

Un objeto ***Connection*** representa una conexión con una BD. Una sesión de conexión con una BD incluye las sentencias SQL que se ejecuten y los resultados que devuelvan.

Una sola aplicación puede tener una o más conexiones con una misma BD, o puede tener varias conexiones con diferentes BD. ***Connection*** es un interfaz porque JDBC ofrece una plantilla o especificación que deben implementar los fabricantes de drivers de JDBC.

EJEMPLO 1: Conectamos con la BD prueba que está en un SGBD MySQL instalado en la misma máquina (localhost o 127.0.0.1).

```
package p10;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class P1 {

    static public void main(String[] args) {
       // Abrir la conexión con la Base de Datos
       System.out.println( "Conectando con la Base de datos..." );
       String url = "jdbc:mysql://localhost:3306/prueba" +
                    "?serverTimezone=UTC";     // una opción
       String u = "root";  // usuario
       String p = "clave"; // password
       try {
           Connection c = DriverManager.getConnection(url, u, p);
           System.out.println( "Conexión establecida con la BD..." );
           c.close();
           System.out.println( "Conexión cerrada con la BD." );
       } catch (SQLException e) {
           e.printStackTrace();
       }
    }  // fin main
}  // fin clase
```

URL de conexión a mysql con JDBC:
jdbc:mysql://cadena_conexión
Cadena de conexión:
host:puerto/BD?op1=valor1&op2=valor2…

EJERCICIO 1: Modifica el programa anterior y crea dos métodos:

- `public Connection conectaMySQL(host, puerto, BD, user, password)`
- `public boolean desconectaMySQL(Connection c)`

Desde el main(), pregunta al usuario por el servidor (nombre o IP), el puerto en el que da servicio MySQL en ese host, la BD a la que

conectarse, el usuario y su password. Haz una llamada a `conectaMySQL()` y comprueba que devuelve una conexión no nula, lo que significa que ha tenido éxito. Indica si se ha conseguido o no y llama a `desconectaMySQL()` en caso de que se haya establecido la conexión.

EJERCICIO 2: Haz lo mismo, pero desde el entorno gráfico con cajas de texto y un botón conectar y otro desconectar que usen la información escrita en cajas de edición.

10.2.4. TIPOS DE DATOS ENTRE MYSQL Y JAVA.

Los tipos de datos usados en una BD casi nunca coinciden con los de un lenguaje de programación. En el caso de MySQL aquí tienes las equivalencias con Java:

Tipo MySQL	Java Class	Tipo MySQL	Java Class
BIT(1)	Boolean	LONGTEXT	String
BIT(>1)	byte[]	MEDIUMBLOB	byte[]
BIGINT[(M)]	Long/BigInteger	MEDIUMINT[(M)]	[UNSIGNED]Integer
BINARY(M)	byte[]	MEDIUMTEXT	String
BLOB	byte[]	SET('v1','v2',..)	String
BOOL, BOOLEAN	Boolean/Integer	SMALLINT[(M)]	[UNSIGNED]Integer
CHAR(M)	String ó byte[]	TEXT	String
DATE	java.sql.Date	TIME	java.sql.Time
DATETIME	java.sql.Timestamp	TIMESTAMP[(M)]	java.sql.Timestamp
DECIMAL[(M[,D])]	BigDecimal	TINYBLOB	byte[]
DOUBLE[(M,B)]	Double	TINYINT	Boolean/Integer
ENUM('v1','v2',...)	String	TINYTEXT	String
FLOAT[(M,D)]	Float	VARBINARY(M)	byte[]
INT,INTEGER[(M)]	Integer/Long	VARCHAR(M)	String ó byte[]
LONGBLOB	byte[]	YEAR[(2|4)]	java.sql.Date

10.3 USANDO EL DRIVER DESDE JAVA.

9.3.1. EJECUTAR SENTENCIAS SQL.

Los objetos **Statement** te permiten ejecutar sentencias SQL básicas como consultas y recuperar los resultados en clases **ResultSet**.

Para crear una instancia de `Statement` puedes llamar a **`createStatement()`** del objeto **`Connection`** devuelto por **`DriverManager.getConnection()`** o **`DataSource.getConnection()`**.

Puedes ejecutar la consulta llamando al método **`executeQuery(String)`**.

Puedes ejecutar una sentencia que modifique filas con el método **`executeUpdate(String)`** que devuelve el número de filas que coinciden con la sentencia, no el nº de filas modificadas.

Si no sabes si la sentencia va a ser una SELECT o bien un UPDATE/INSERT/DELETE, puedes usar **`execute(String)`** que devuelve true si la sentencia era una consulta y false en otro caso. Si era una consulta puedes acceder a los resultados llamando a **`getResultSet()`**. Si no lo era, puedes averiguar cuantas filas han sido afectadas por la sentencia llamando a **`getUpdateCount()`** sobre la instancia Statement.

EJEMPLO 2: Ejecutar una sentencia select.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;

public class P2 {
    static Connection conex;

    public static Connection conectaBD(String con, String user, String password) {
        try {
          return DriverManager.getConnection( con, user, password );
        }
        catch(SQLException e) {
            e.printStackTrace();
        }
            return null;
        }

        public static void main(String[] args) {
            String jdbcURL= "jdbc:mysql://localhost:3306/prueba" +
```

```
                        "?serverTimezone=UTC";
            Statement sentencia = null;
            ResultSet rs = null;
            try {
                conex= conectaBD(jdbcURL, "root", "1234");
                sentencia = conex.createStatement();
                rs = sentencia.executeQuery("SELECT * FROM alumnos");
                // o si no sabes si la sentencia es una select...
                if (rs != null) {
                    rs = sentencia.getResultSet();
                }
                // Ahora trabajamos con los resultados...
            }
            catch (SQLException e){
                System.out.println( "SQLException: " + e.getMessage() );
                System.out.println( "SQLState: " + e.getSQLState() );
                System.out.println( "VendorError: " + e.getErrorCode() );
            }
            finally {
                if (rs != null) {
                    try { rs.close();  } catch (SQLException e1) { } // ignora
                    rs = null;
                }
                if (sentencia != null) {
                 try { sentencia.close();  } catch (SQLException e1) { } // ignora
                 sentencia = null;
                }
                if (conex != null) {
                    try {  conex.close();  } catch (SQLException e1) { } // ignora
                    conex = null;
                }
            }
        }
}
```

10.3.2. MODELO VISTA CONTROLADOR.

Como recordarás, los componentes y los eventos son los pilares básicos de una interfaz gráfica de usuario. Los componentes de Swing basan su construcción en una arquitectura denominada **model-view-controller** (**MVC**) que proviene de SmallTalk.

Esta arquitectura establece que en la programación de aplicaciones con GUI deben aparecer 3 partes fundamentales:

- **Modelo**: contiene los datos y su construcción será totalmente ajena a la forma en que se representen en la interfaz gráfica.
- **Vista**: encargada de la representación gráfica de los datos.
- **Controlador**: se ocupará de la interpretación de las entradas del usuario, que permiten modificar el modelo (datos) o modificar la vista.

La arquitectura MVC puede aplicarse a muchos niveles. Se puede aplicar, por ejemplo, para la creación de un simple botón de una GUI y se puede aplicar también, por ejemplo, en la estructuración de una aplicación distribuida en Internet, en la cual se observan las tres partes fundamentales. En nuestro caso vamos a ver la arquitectura MVC aplicada en dos niveles:

- **Construcción interna de componentes:** Internamente los componentes de Swing utilizan la arquitectura MVC.
- **Visualizar datos en una aplicación con GUI**: Se guia el desarrollo de las aplicaciones GUI aplicando la arquitectura MVC.

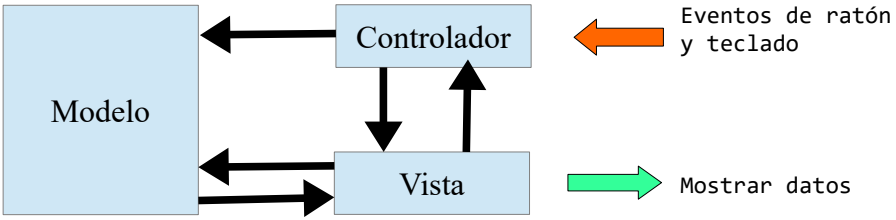
MVC EN LA CONSTRUCCIÓN INTERNA DE COMPONENTES

Los componentes de la GUI no son los únicos objetos capaces de generar eventos. Se puede asociar código a estos eventos que se ejecutará cuando el evento se produzca.

Por ejemplo, un temporizador puede configurarse para generar un evento cada 20 minutos. En realidad, cualquier objeto puede generar eventos cuando su estado cambia si se implementa su clase para que tenga esta capacidad.

En una primera aproximación, la implementación interna de cada componente se realiza delegando la funcionalidad en tres objetos: un

objeto será el encargado de albergar los datos; otro de representar gráficamente esos datos y el tercero de interpretar las acciones del usuario. Estos objetos se relacionan entre sí de la forma que se muestra en la figura.



La responsabilidad de cada uno de los objetos anteriores es la siguiente:

- **Modelo**: Objeto encargado de almacenar los datos. Eleva eventos cuando su estado cambia. Por ejemplo, si el componente es un JCheckbox, el modelo es el objeto que guarda la información sobre si el botón de chequeo está seleccionado o no. Y cada vez que su estado cambie, el objeto eleva un evento.
- **Vista**: Objeto encargado de representar visualmente los datos que están almacenados en el modelo. Este objeto cambia la representación visual cada vez que el modelo eleva un evento de cambio. Si el evento es ligero, la vista debe preguntar al modelo sobre su estado para poder visualizarlo. Si el evento es informativo, con la información que viene en el propio evento es suficiente para poder representar los datos.
- **Controlador**: Es el responsable de interpretar las entradas del usuario. En el ejemplo del botón de chequeo este objeto interpreta las teclas que pulsa el usuario, y si es la tecla espacio cambia el estado del botón. En los componentes más complejos, la vista y el controlador intercambian mucha información.

Los desarrolladores de Swing se basaron en esta idea para construir los componentes, pero realizaron algunos cambios importantes: fusionaron en un solo objeto la responsabilidad de visualización y control, debido al fuerte acoplamiento que tienen y a la complejidad de la interacción entre ambos. A esta nueva clase se la denomina **delegado UI** (UI delegate).

De esta forma, los componentes de Swing tienen lo que se conoce como **arquitectura de modelo delegado** o arquitectura de modelo separable. Los componentes tienen principalmente 2 atributos: el modelo y el delegado, tal y como se muestra en la siguiente figura:

Para cada uno de los componentes existe una interfaz para su modelo y una clase para su delegado UI. Algunos componentes comparten la misma interfaz para definir su modelo. En la siguiente tabla se muestra las interfaces para cada uno de los modelos de los componentes de la interfaz de usuario de Swing:

COMPONENTE MODELO	INTERFACE
javax.swing.AbstractButton	ButtonModel
javax.swing.JButton	ButtonModel
javax.swing.JMenuItem	ButtonModel
javax.swing.JCheckBoxMenuItem	ButtonModel
javax.swing.JMenu	ButtonModel
javax.swing.JRadioButtonMenuItem	ButtonModel
javax.swing.JToggleButton	ButtonModel
javax.swing.JCheckBox	ButtonModel
javax.swing.JRadioButton	ButtonModel
javax.swing.JScrollBar	BoundedRangeModel
javax.swing.JSlider	BoundedRangeModel
javax.swing.JSpinner	SpinnerModel

COMPONENTE MODELO	INTERFACE
javax.swing.JList	ListModel, ListSelectionModel
javax.swing.JComboBox	ComboBoxModel
javax.swing.JProgressBar	BoundedRangeModel
javax.swing.JTabbedPane	SingleSelectionModel
javax.swing.JTable	TableModel, TableColumnModel
javax.swing.JTree	TreeModel, TreeSelectionModel
javax.swing.text.JTextComponent	Document
javax.swing.JEditorPane	Document
javax.swing.JTextPane	Document
javax.swing.JTextArea	Document
javax.swing.JTextField	Document
javax.swing.JFormattedTextField	Document
javax.swing.JPasswordField	Document

La mayoría de los componentes tienen un modelo y un delegado como atributo. Ahora vamos a ver los métodos y las clases relacionadas con los modelos. Todos los componentes que definen modelos tienen las siguientes características:

Métodos de acceso en la clase de cada uno de los componentes para el modelo (siendo XXModel la interfaz del modelo que use cada clase):

- `public XXModel getModel()`
- `public void setModel(XXModel modelo)`

En los componentes que tienen el modelo **Document**, estos métodos cambian por:

- `public Document getDocument()`
- `public void setDocument(Document d)`

En aquellos componentes que tienen más de un modelo, los métodos **getModel()** y **setModel(…)**, se usan para acceder al modelo de datos principal. Para los otros modelos existen los métodos siguientes:

- `public XXModel getXXModel()`
- `public void setXXModel(XXModel modelo)`

La mayoría de los componentes tienen un constructor al que se le puede pasar el modelo o los modelos del componente (siendo JXX la

clase del componente):

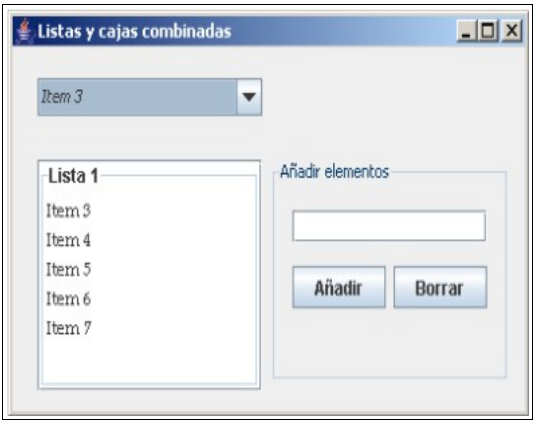
```
public JXX(XXModel modelo, ...)
```

Si se usa un constructor en el que no se especifica ningún modelo, se instancia un modelo por defecto de la clase **DefaultXXModel** (siendo XXModel la interfaz del modelo).

Varios componentes representando los mismos datos.

Para poder implementar esto, necesitamos utilizar la propiedad **model** de cada uno de los componentes que vayan a compartir los datos (el modelo).

En la figura, se ha utilizado el modelo por defecto en el componente **JComboBox**, que es **DefaultComboBoxModel** y se ha puesto en el componente **JList** el modelo que usa el componente **JcomboBox**. Para compartir modelos puedes usar las utilidades del IDE que tengas o hacerlo de forma manual, por ejemplo:




```
cb1.setModel(
            new javax.swing.DefaultComboBoxModel(
                 new String[]{"Item 1", "Item 2", "Item 3", ...})
          );
jl1.setModel( cb1.getModel() );
```

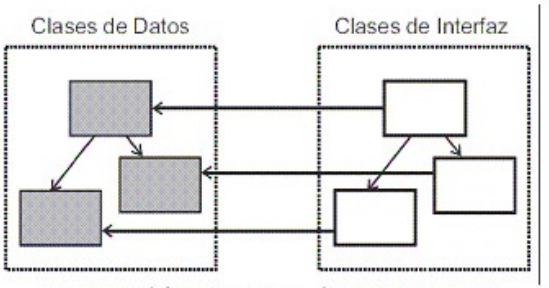
En este caso hemos realizado la compartición de dos modelos que son de componentes distintos debido a que el **ComboBoxModel** hereda de **ListModel** (al revés no podríamos realizarlo).

Eventos en el modelo

Los cambios en los datos de un componente que suceden en respuesta a una acción del usuario no generan eventos en el propio componente. Esto es debido a que **estos eventos se generan en el modelo**. Por tanto cuando se esté trabajando con un componente, se tendrá que ver cuáles son los eventos generados en el propio componente y cuáles los generados en el modelo o modelos que usa el componente.

MVC EN LA CONSTRUCCIÓN DE UNA APLICACIÓN CON GUI

La idea general de MVC es la separación de los datos y de la interfaz gráfica en dos entidades distintas con el mínimo acoplamiento entre ellas. El desarrollo del código de gestión de datos no debe estar influenciado por la forma en que estos datos se van a visualizar y manejar en la GUI. Como consecuencia, el código de gestión de datos no debe hacer ninguna referencia al código de la GUI. La siguiente imagen muestra la estructura general.

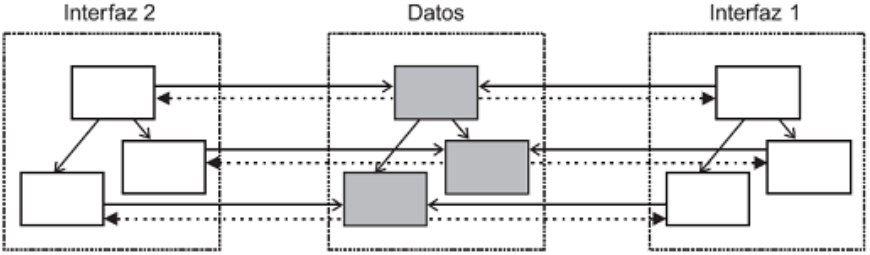


Separación datos - interfaz MVC

Con clases de datos se hace referencia a todas aquellas clases que representan los datos de la aplicación y la gestión que se hace de ellas, sin ninguna consideración sobre la interfaz gráfica. Las clases de interfaz serán aquellas que configuran aspectos como botones, ventanas, colores, etc…

Para permitir que la GUI cambie cuando los datos cambian, bien porque se han modificado en otro punto de la GUI o de cualquier otro modo, se necesita que las clases de datos generen eventos cada vez que su estado cambia.

Separación datos – interfaz MVC Modelo compartido



Modelo compartido en dos visualizaciones diferentes

Las clases que se construyan deberán elevar eventos al cambiar de estado si se pretende tener dos visualizaciones consistentes, o bien, si se quiere que ante un cambio en los datos que no provenga de la interfaz gráfica, la visualización cambie.

10.3.3. UTILIZAR EL COMPONENTE JTABLE.

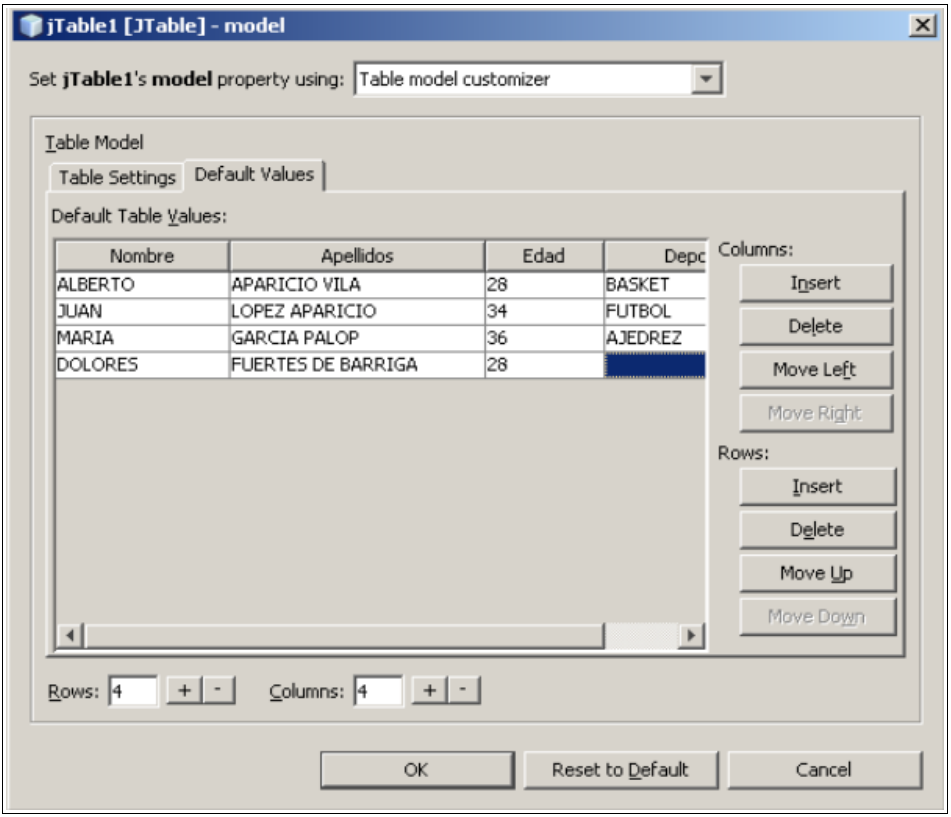
Con la clase ***JTable***, se pueden mostrar tablas de datos y opcionalmente permitir que el usuario los edite. ***JTable*** no contiene ni almacena datos, simplemente es una vista de esos datos (MVC).

Es un componente complejo. Debemos utilizar las clases: `JTable`, `JTableHeader`, `TableModel`, `AbstractTableModel`, `TableCellRenderer`, `DefaultTableCellRenderer`, `TableCellEditor`, `DefaultCellEditor`, `TableColumnModel`, `DefaultTableColumnModel`, `TableColumn` y `DefaultTableModel`.

Un componente ***JTable*** debe de estar ubicado dentro de un

JScrollPane, de esta forma, si nuestra tabla contiene más datos de los que admite el área de la vista, podremos desplazarnos por medio del scroll. En algunos IDE se añade automáticamente y en otros hay que hacerlo de manera manual.

Toda tabla contiene un modelo que es el que representa los datos de forma visual, si queremos modificar el conjunto de datos que contiene la tabla seleccionaremos su propiedad ***model****. Por defecto el IDE que uses podría crear un* ***DefaultTableModel*** *para poder trabajar con los elementos de un Jtable.*



Esta otra figura es un ejemplo de un JTable básico que puede editar los datos.

Código autogenerado por NetBeans cuando se define la propiedad model:

```
 jTableBasico.setModel(
     new javax.swing.table.DefaultTableModel(
         new Object [][] {
{"ALBERTO", "APARICIO VILA", new Integer(28), "BASKET",null},
{"JUAN", "LOPEZ APARICIO", new Integer(34),"FUTBOL", new Boolean(true)},
{"MARIA","GARCIA PALOP", new Integer(36), "AJEDREZ", new Boolean(true)},
{"DOLORES", "FUERTES DE BARRIGA", new Integer(28), "FUTBOL", null}
         },
         new String [] {
            "Nombre", "Apellidos", "Edad", "Deporte", "Vegetariano"
         }
     ) {
          Class[] types = new Class [] {
              String.class, String.class, Integer.class,
              String.class, Boolean.class };
          public Class getColumnClass(int columnIndex) {
              return types [columnIndex];
          }
       }
 );
```

Como se ve, crea de forma automática una clase **DefaultTableModel** con los datos y las columnas que hemos definido mediante la propiedad model.

EJERCICIO 3: Crea la aplicación SimpleTableModel.java en una ventana que contenga un TabbedPane y una tabla en la que defines el TableModel del ejemplo.

PERSONALIZAR EL MODELO DE UN JTABLE

En ocasiones nos interesa crear nuestro propio modelo de tabla para tener más control. El uso de la clase **DefaultTableModel** tiene un problema de rendimiento importante porque utiliza la clase **Vector**, que tiene todos sus métodos sincronizados y en la mayor parte de los casos innecesariamente.

Sería más conveniente usar un **ArrayList** para almacenar los datos, que es equivalente a **Vector**, pero más eficiente.

EJEMPLO 3: En el siguiente fragmento de código, la clase **SimpleTableModel** que hereda de **AbstractTableModel** implementa un modelo de tabla basado en ArrayList:

```
public class SimpleTableModel extends AbstractTableModel {
    ArrayList datos = new ArrayList();
    String[] columnas= {"Nombre", "Apellidos", "Edad", "Deporte", "Vegetariano"};
    Class[] types = new Class[] { java.lang.String.class, java.lang.String.class,
        java.lang.Integer.class, java.lang.String.class, java.lang.Boolean.class};
    /** Crea una nueva instancia de SimpleTableModel*/
    public SimpleTableModel() {
        Object[] fila = new Object[5];  // Creamos los objetos de una fila
        fila[0] = "Alberto";            fila[1] = "Aparicio vila";
        fila[2] = new Integer(29);      fila[3] = "BASKET";
        fila[4] = new Boolean(false);
        //El array list contendra un array de Object[] en cada fila
        datos.add(fila);
    }

    public String getColumnName(int col) { return columnas[col].toString(); }
```

```
    public int getRowCount() { return datos.size(); }
    public int getColumnCount() { return columnas.length; }

    public Object getValueAt(int row, int col) {
        Object[] fila = (Object[]) datos.get(row);
        return fila[col];
    }

    public Class getColumnClass(int columnIndex) { return types[columnIndex];  }

    public boolean isCellEditable(int row, int col) { return true; }

    public void setValueAt(Object value, int row, int col) {
        Object[] fila = (Object []) datos.get(row);
        fila[col] = value;
        fireTableCellUpdated(row, col);
    }

    public void addRow(Object[] fila) {
        datos.add(fila);
        fireTableDataChanged();
    }

    public void removeRow(int fila) {
        datos.remove(fila);
        fireTableDataChanged();
    }
}
```

EJERCICIO 4: Modifica el ejercicio anterior `SimpleTableModel.java`, para que permita una nueva columna `Localidad` de tipo String tal y como muestra la figura.

Utilizando un combo box como editor de celda

Si queremos que alguna de nuestras columnas de la tabla permita seleccionar un valor a partir de un `JComboBox` tal y como muestra la figura, necesitamos especificar para la columna, que dicho valor se ha de seleccionar a partir de un `JComboBox`.

Para añadir un editor de celda de tipo ComboBox:

```
JComboBox cb= new JComboBox();
cb.addItem("TENIS"),
cb.addItem("BASCKET");
cb.addItem("TENIS");
cb.addItem("AJEDREZ");
cb.addItem(NINGUNO");
cb.addItem(" ");
tabla.getColumnModel().getColumn(3).setCellEditor(
                                        new DefaultCellEditor(cb) );
```

Personalizando la Visualización de celdas

Si queremos que nuestras celdas tengan un aspecto personalizado necesitamos crear nuestra propia clase para tener este tipo de

funcionalidad. Necesitamos utilizar la interfaz ***TableCellRenderer*** que permite definir el método ***getTableCellRendererComponent()*** que devuelve un objeto de tipo ***Component***, es decir, cualquier tipo de componente Swing. La siguiente figura muestra como queda la columna Nombre cuando definimos un Renderer especial que hace aparecer en rojo el nombre de los alumnos vegetarianos:

Y el código que nos permite realizar esta personalización lo podemos ver en la clase ***ColorRenderer*** que implementa la interface ***TableCellRenderer***.

```
public class ColorRenderer extends JLabel implements TableCellRenderer {
    public ColorRenderer() { }
    /**
    * Devuelve un componente para personalizar la celda
    * @param table Componente JTable padre
    * @param value Valor de la celda
    * @param isSelected True si la celda está seleccionada
    * @param hasFocus Si tiene el foco
    * @param row Indice de la fila
    * @param column Indice de la columna
    * @return Componente para ser visualizado
    */
    public Component getTableCellRendererComponent( JTable table,
                    Object value, boolean isSelected,
```

```
                    boolean hasFocus, int row, int column) {
            if(!isSelected) {
                this.setBackground(Color.YELLOW);
                this.setForeground(Color.PINK);
            }
            else {
                this.setBackground(Color.LIGHT_GRAY);
                this.setForeground(Color.RED);
            }
            this.setText( (String)value );
            this.setToolTipText("El valor de esta celda es: "+ value);
            this.setOpaque(true);
            return this;
        }
    }
```

Una vez definida la clase, para asociarla a la columna del Nombre seguiremos los siguientes pasos:

```
TableColumn nombre = jTableRenderer.getColumnModel().getColumn(0);
nombre.setCellRenderer( new ColorRenderer() );
```

EJERCICIO 5: Añade una columna Edad y crea un Renderer para la columna edad (`ColorRenderer`) y que realice lo siguiente:

- Si la Edad >=0 y <=15 entonces el color debe ser Rojo
- Si la Edad >15 y <=30 entonces el color debe ser Verde.
- Si la Edad >30 entonces el color debe ser Azul

10.4. TRABAJANDO CON RESULT SETS.

En un objeto ***ResultSet*** se encuentran los resultados de la ejecución de una sentencia SQL, por lo tanto, un objeto **ResultSet** contiene las filas que satisfacen las condiciones de una sentencia SQL y ofrece acceso a los datos de las filas a través de una serie de métodos ***getXXX()*** que permiten acceder a las columnas de la fila actual.

El método ***next()*** de la interface ***ResultSet*** es utilizado para desplazarse a la siguiente fila del **ResultSet**, haciendo que la próxima fila sea la actual, además de este método de desplazamiento básico, según el tipo de *ResultSet* podremos realizar desplazamientos libres utilizando métodos como ***last()***, ***relative()*** o ***previous()***.

El aspecto que suele tener un ***ResultSet*** es una tabla con cabeceras de columnas y los valores devueltos por una consulta. Un ***ResultSet*** **mantiene un cursor que apunta a la fila actual de datos**. El cursor se mueve hacia abajo cada vez que el método ***next()*** se llama. Inicialmente está posicionado antes de la primera fila, de esta forma, la primera llamada a *next()* situará el cursor en la primera fila, pasando a ser la fila actual.

Las filas del *ResultSet* son devueltas de arriba a abajo según se va desplazando el cursor con las sucesivas llamadas al método ***next()***. Un cursor es válido hasta que el objeto **ResultSet** o su objeto padre **Statement** es cerrado.

OBTENER DATOS DEL ResultSet

Los métodos ***getXXX()*** *permiten* recuperar los valores de las columnas (campos) de la fila (registro) actual del ***ResultSet***. Dentro de cada fila, no es necesario que las columnas sean recuperadas utilizando un orden

determinado, pero para una mayor portabilidad entre diferentes BD se recomienda que los valores de las columnas se recuperen de izquierda a derecha y solamente una vez.

Para designar una columna podemos utilizar su nombre o su número de orden en la fila. Por ejemplo, si la segunda columna de un objeto rs de la clase ResultSet se llama "titulo" y almacena datos de tipo String, se podrá recuperar su valor de las formas que muestra el siguiente fragmento de código:

```
// rs es un objeto de tipo ResultSet y queremos el valor de la
// tercera columna que se etiqueta como "titulo"
String valor = rs.getString(1);
String valor = rs.getString("titulo");
```

Las columnas se numeran de izquierda a derecha empezando con la columna 1 y los nombres de las columnas no son case sensitive, es decir, no se distingue entre mayúsculas y minúsculas.

La información referente a las columnas que contiene el *ResultSet* se encuentra disponible llamando al método ***getMetaData()***, este método devolverá un objeto ***ResultSetMetaData*** que contendrá el número, tipo y propiedades de las columnas del *ResultSet*, más adelante veremos con más detalle la interfaz *ResultSetMetaData*.

Si conocemos el nombre de una columna, pero no su índice, el método ***findColumn(String)*** puede utilizarse para obtener el número de columna, pasándole como argumento un objeto String que sea el nombre, este método nos devolverá un entero que es el índice de la columna dentro de la fila.

TIPOS DE DATOS Y CONVERSIONES

Cuando se ejecuta un método *getXXX()* sobre un objeto

ResultSet para obtener el valor de un campo del registro actual, el driver JDBC convierte el dato que se quiere recuperar al tipo Java especificado y entonces devuelve un valor Java adecuado. Por ejemplo si utilizamos el método ***getString()*** y el tipo del dato en la BD es VARCHAR, el driver JDBC convertirá el dato VARCHAR a un objeto String de Java, por lo tanto el valor de retorno de getString() será un objeto de la clase String.

Esta conversión de tipos se puede realizar gracias a la clase **java.sql.Types**. En esta clase se definen lo que se denominan tipos de datos JDBC, que se corresponde con los tipos de datos SQL estándar. Esto nos permite abstraernos del tipo SQL específico de la BD con la que estamos trabajando, ya que los tipos JDBC son tipos de datos SQL genéricos. Normalmente estos tipos genéricos nos servirán para todas nuestras aplicaciones JDBC.

La clase **Types** está definida como un conjunto de constantes (final static), estas constantes se utilizan para identificar los tipos SQL. Si el tipo del dato SQL que tenemos en nuestra BD es específico de esa BD y no se encuentra entre las constantes definidas por la clase Types, se utilizará el tipo `Types.OTHER`.

Al llamar un método getXXX() sobre un objeto ResultSet obtenemos el objeto Java de la clase correspondiente. Esta recuperación de datos suele ser bastante flexible, es decir, podemos recuperar un entero de SQL, que se correspondería con el tipo JDBC Types.INTEGER, o con el método getString() que nos devolvería un objeto String de Java.

Estas acciones, aunque posibles, no son recomendables, ya que además de no ser un buen estilo de programación, pueden ocasionar problemas de portabilidad.

NAVEGAR POR LAS FILAS DE UN ResultSet

La forma de navegar (movernos) por las filas de un objeto ResultSet, depende de su tipo. El método **next()** de la interface **ResultSet** lo utilizamos para desplazarnos al registro (o fila) siguiente dentro de un **ResultSet**. El método **next()** devuelve un valor booleano, true si el registro siguiente existe y false si ya estamos al final y no hay más registros. Este era el único método que ofrecía la interface ResultSet en la versión 1.0 de JDBC, pero en JDBC 2.0, además de tener el método next(), disponemos de los siguientes métodos para el desplazamiento y movimiento dentro de un objeto ResultSet:

- **boolean absolute(int registro)**: desplaza el cursor (fila actual) al número de registro indicado. Si el valor es negativo, se posiciona en el número registro indicado pero empezando por el final. Este método devolverá false si nos hemos desplazado después del último registro o antes del primer registro. Para poder usarlo, el ResultSet debe ser de tipo `TYPE_SCROLL_SENSITIVE` o `TYPE_SCROLL_INSENSITIVE`, a un ResultSet que es de cualquiera de estos tipos se dice que es de tipo scrollable. Si a este método le pasamos un valor cero se lanzará una excepción SQLException.
- **void afterLast()**: se desplaza al final del objeto ResultSet, después del último registro. Si el ResultSet no posee registros este método no tienen ningún efecto. Sólo se puede utilizar en objetos ResultSet de tipo scrollable.
- **void beforeFirst()**: mueve el cursor al comienzo del objeto ResultSet, antes del primer registro. Sólo se puede utilizar en ResultSet de tipo scrollable.
- **boolean first()**: desplaza el cursor al primer registro. Devuelve true si el cursor se ha desplazado a un registro válido y

false en otro caso o si el ResultSet no contiene registros. Sólo se puede utilizar en ResultSet de tipo scrollable.

- **void last():** desplaza el cursor al último registro del objeto ResultSet. Devolverá true si el cursor se encuentra en un registro válido, y false en otro caso o si el objeto ResultSet no tiene registros. Sólo es valido para ResultSet de tipo scrollable, en caso contrario lanza una excepción SQLException.
- **void moveToCurrentRow()**: mueve el cursor a la posición recordada, normalmente el registro actual. Este método sólo tiene sentido cuando estamos situados dentro del ResultSet en un registro que se ha insertado. Este método sólo es válido con ResultSet que permiten la modificación, es decir, están definidos mediante la constante `CONCUR_UPDATABLE`.
- **boolean previous()**: desplaza el cursor al registro anterior. Es el método contrario a next(). Devolverá true si el cursor se encuentra en un registro o fila válidos y false en caso contrario. Sólo es válido con ResultSet de tipo scrollable, en caso contrario lanzará una excepción SQLException.
- **boolean relative(int registros)**: mueve el cursor un número relativo de registros, este número puede ser positivo (avanzar) o negativo (retroceder). Si el número es negativo el cursor se desplazará hacia el principio del ResultSet el número de registros indicados, y si es positivo se desplazará hacía el final del ResultSet. Sólo se puede utilizar si el ResultSet es de tipo scrollable.

También existen otros métodos dentro del interface ResultSet que están relacionados con el desplazamiento:

- **boolean isAfterLast()**: indica si nos encontramos después del último registro del ResultSet. Sólo se puede utilizar con

ResultSet de tipo scrollable.

- **boolean isBeforeFirst()**: indica si nos encontramos antes del primer registro del ResultSet. Sólo se puede utilizar con ResultSet de tipo scrollable.
- **boolean isFirst()**: indica si el cursor se encuentra en el primer registro. Sólo se puede utilizar en ResultSet de tipo scrollable.
- **boolean isLast()**: indica si nos encontramos en el último registro. Sólo se puede utilizar en ResultSet de tipo scrollable.
- **int getRow()**: devuelve el número de registro actual. El primer registro será el número 1, el segundo el 2, etc. Devolverá cero si no hay registro actual.

EJEMPLO 4: deseamos recorrer un ResultSet completo hacia adelante usando un **bucle while** en el que se ejecuta el método **next()** del objeto ResultSet. La ejecución de este bucle finaliza cuando hayamos alcanzado el final del conjunto de registros. Adapta el código, crea las tablas que no tengas e inserta algunos datos para probarlo.

```
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
    // Registrando el Driver
    String driver = "com.mysql.cj.jdbc.Driver";
    Class.forName(driver).newInstance();
    System.out.println("Driver " + driver + " Registrado");
    // Abrir la conexión con la Base de Datos
    System.out.println("Conectando con el SGBD...");
    String jdbcURL= "jdbc:mysql:...";  // Hay que completarla
    conn = DriverManager.getConnection(jdbcURL,"root", "");
    System.out.println("Conexión establecida con la BD.");
    stmt = conn.createStatement();
    // Ejecutamos la SELECT sobre la tabla articulos
    String sql = "select * from articulos";
    rs = stmt.executeQuery(sql);
    int id;
```

```
            String nombre;
            java.math.BigDecimal precio;
            String codigo;
            int grupo;
            System.out.println("Cursor antes de la primera fila = " +
                               rs.isBeforeFirst() );
            while( rs.next() ) {
               // Obtener información por nombre de columna
               id = rs.getInt("id");
               nombre = rs.getString("nombre");
               precio = rs.getBigDecimal("precio");
               // Obtener información por indice de las columnas
               codigo = rs.getString(4);
               grupo = rs.getInt(5);
                // Mostrar la información
                System.out.print("Numero de Fila =" + rs.getRow() );
                System.out.print(", id: " + id);
                System.out.print(", nombre: " + nombre);
                System.out.print(", precio: " + precio.floatValue()+"€");
                System.out.print(", codigo: " + codigo);
                System.out.println(", grupo: " + grupo);
            }
            System.out.println("Cursor despues de la ultima fila = " +
                               rs.isAfterLast() );
        }
        catch(SQLException se) {
            se.printStackTrace();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        finally {
            try {
                if(rs!=null) rs.close();
                if(stmt!=null) stmt.close();
                if(conn!=null) conn.close();
            }
            catch(SQLException e) {
                e.printStackTrace();
            } // finally try
        } // try
```

EJERCICIO 6: Crea una aplicación en Java que conecte a una BD, crea

un ResultSet de tipo scrollable y muestra el conjunto de filas en orden inverso de como están en el ResultSet (recorrerlas hacia atrás). Después mostramos por este orden las siguientes filas: última, primera, tercera y segunda.

10.4.1. MODIFICAR DATOS (Updatable ResultSets).

Para poder modificar los datos que tiene un ResultSet debemos crear un ResultSet de tipo modificable, para ello utilizamos la constante ***ResultSet.CONCUR_UPDATABLE*** en el método ***createStatement()***.

Aunque un ResultSet que permite modificaciones suele permitir distintos desplazamientos, es decir, se suele utilizar la constante ***ResultSet.TYPE_SCROLL_INSENSITIVE*** o ***ResultSet.TYPE_SCROLL_SENSITIVE***, pero no es del todo necesario ya que también puede ser del tipo sólo hacia delante (forward-only).

Para modificar los valores de un registro existente se utilizan una serie de métodos ***updateXXX()*** de la interface *ResultSet*. Las XXX indican el tipo del dato (igual que ocurre con los métodos *getXXX()*). **El proceso para realizar la modificación de una fila de un *ResultSet* es el siguiente:**

- Nos situamos sobre el registro que queremos modificar.
- Ejecutamos los métodos *updateXXX()* adecuados, pasándole como argumento los nuevos valores.
- A continuación ejecutamos el método ***updateRow()*** para que los cambios tengan efecto sobre la base de datos.

El método ***updateXXX()*** recibe dos parámetros, la columna a modificar y el nuevo valor. La columna la podemos indicar por su número de orden o por su nombre, igual que en los métodos *getXXX()*.

EJEMPLO 5: El siguiente fragmento de código muestra como se puede modificar el campo dirección del último registro de un *ResultSet* que contiene el resultado de una SELECT sobre la tabla de clientes:

```
Statement stmt=
conn.createStatement( ResultSet.TYPE_SCROLL_SENSITIVE,
                      ResultSet.CONCUR_UPDATABLE );
// Ejecutar la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
System.out.println("Situamos el cursor al final");
// Situarnos en el último registro del ResultSet y modificarlo
if( rs.last() ) {
    rs.updateString("direccion", "C/.Pepe, 3");
    rs.updateRow();
}
```

Si nos desplazamos dentro del ResultSet después de cambiar valores y antes de llamar al método updateRow(), se pierden las modificaciones realizadas. Y si queremos cancelar las modificaciones sin movernos, ejecutamos el método ***cancelRowUpdates()*** sobre el objeto *ResultSet*, en lugar del método *updateRow()*.

Una vez que hemos invocado el método *updateRow()*, el método *cancelRowUpdates()* no tendrá ningún efecto. El método *cancelRowUpdates()* cancela las modificaciones de todos los campos de un registro, es decir, si hemos modificado dos campos con el método *updateXXX()* se cancelan ambas modificaciones.

INSERTAR NUEVAS FILAS Y BORRAR EXISTENTES

Además de poder realizar modificaciones directamente sobre las filas de un *ResultSet*, también podemos añadir nuevas filas (registros) y eliminar las existentes. Estos métodos son: ***moveToInsertRow()***, ***insertRow()*** y ***deleteRow()***.

El primer paso para insertar un registro en un *ResultSet* es mover el cursor (puntero que indica el registro actual) del *ResultSet*, esto se consigue mediante el método ***moveToInsertRow()***. El siguiente paso es dar un valor a cada uno de los campos que van a formar parte del nuevo registro, para ello se utilizan los métodos ***updateXXX()***. Para finalizar el proceso se lanza el método ***insertRow()***, que creará el nuevo registro tanto en el *ResultSet* como en la tabla de la BD correspondiente. Hasta que no se lanza el método *insertRow()*, la fila no se incluye dentro del *ResultSet*, es una fila especial denominada "fila de inserción" (insert row) y es similar a un buffer completamente independiente del objeto *ResultSet*.

EJEMPLO 6: El siguiente fragmento de código da de alta un nuevo registro en la tabla de clientes:

```
Statement stmt = conn.createStatement(
                        ResultSet.TYPE_SCROLL_SENSITIVE,
                        ResultSet.CONCUR_UPDATABLE );
// Ejecutar la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
//Crear un nuevo registro en la tabla de clientes
rs.moveToInsertRow();
rs.updateString(2,"Kiko Lopez");
rs.updateString(3,"Wall Street 3674");
rs.insertRow();
```

Si no facilitamos valores a todos los campos del nuevo registro con los métodos *updateXXX()*, ese campo tendrá un valor *NULL*, y si en la BD no está definido ese campo para admitir nulos se producirá una excepción *SQLException*.

Cuando hemos insertado nuestro nuevo registro en el objeto *ResultSet*, podremos volver a la antigua posición en la que nos encontrábamos dentro del *ResultSet*, antes de haber ejecutado el

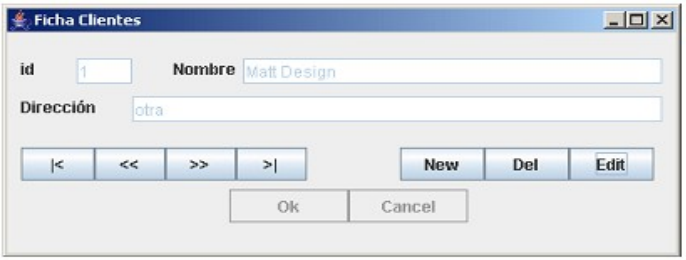
método *moveToInsertRow()*, llamando al método ***moveToCurrentRow()***, este método sólo se puede utilizar en combinación con el método ***moveToInsertRow()***.

Además de insertar filas en nuestro objeto *ResultSet* también podremos eliminar filas o registros del mismo. El método que se debe utilizar es ***deleteRow()***. Para eliminar un registro no tenemos que hacer nada más que movernos a ese registro y ejecutar el método ***deleteRow()*** sobre el objeto ***ResultSet*** correspondiente.

EJEMPLO 7: El siguiente fragmento de código borra el último registro de la tabla de clientes:

```
Statement stmt = conn.createStatement(
                         ResultSet.TYPE_SCROLL_SENSITIVE,
                         ResultSet.CONCUR_UPDATABLE );
// Ejecuta la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
// Nos situamos al final del ResultSet
rs.last();
rs.deleteRow();
```

EJERCICIO 7: Realiza una aplicación que se conecte a una BD de forma automática y muestre un formulario que permita realizar el mantenimiento de la tabla "CLIENTES = = Id + nombre + dirección" como muestra la figura (navegar, insertar, borrar y modificar datos).



10.4.2. SENTENCIAS SQL PRECOMPILADAS.

La interface **PreparedStatement** igual que la interface `Statement`, nos permite ejecutar sentencias SQL sobre una conexión establecida con una BD. Pero en este caso vamos a ejecutar sentencias SQL más especializadas, estas sentencias SQL se van a denominar **sentencias SQL precompiladas** y van a recibir parámetros de entrada.

La interface `PreparedStatement` hereda de la interface `Statement` y se diferencia en dos cosas:

- Las instancias de `PreparedStatement` contienen sentencias SQL que se compilan una vez (en el SGBD) y se usan más de una. Esto es lo que hace a una sentencia "prepared" (preparada) para ejecutarse. Esto permite:
  - Aumentar el rendimiento del sistema de BD (ahorras compilaciones en la BD). Una sentencia SQL que va a ser ejecutada más de una vez, se suele crear como un objeto PreparedStatement para ganar eficiencia.
  - Aumentar la velocidad de respuesta de la aplicación.
  - Mejoras la seguridad (evita ataques de inyección de SQL). También se utiliza este tipo de sentencias para pasarles parámetros de entrada a las sentencias SQL. Si estos valores provienen del usuario, usar parámetros mejora la seguridad al evitar inyección de SQL.
  - Simplificas la escritura de sentencias en el programa.
- La sentencia SQL que contiene un objeto PreparedStatement puede contener uno o más parámetros de entrada. Un parámetro de entrada es aquel cuyo valor no se especifica cuando la sentencia se crea, en su lugar, la sentencia SQL va a tener un signo de interrogación (?) por cada parámetro de entrada. Antes de ejecutarse la sentencia se debe especificar

un valor para cada uno de los parámetros a través de los métodos **setXXX()** apropiados. Estos métodos **setXXX()** los añade la interface PreparedStatement.

Al heredar de la interface Statement, PreparedStatement tiene todas sus funcionalidades. Además, añade una serie de métodos que permiten asignar un valor a cada uno de los parámetros de entrada de este tipo de sentencias.

Los métodos ***execute()***, ***executeQuery()*** y ***executeUpdate()*** se sobrecargan y ahora no toman ningún tipo de argumentos, de esta forma, nunca se les debe pasar por parámetro el objeto *String* que representaba la sentencia SQL a ejecutar. En este caso, un objeto *PreparedStatement* ya es una sentencia SQL por sí misma, a diferencia de lo que ocurría con las sentencias *Statement* que tiene significado real sólo en el momento en el que se ejecuta una en concreto.

Creando objetos PreparedStatements

Para crear un objeto ***PreparedStatement*** se ejecuta el método ***prepareStatement()*** de la interface ***Connection*** sobre el objeto que representa la conexión establecida con la base de datos.

En el siguiente fragmento de código se puede ver como se crea un objeto *PreparedStatement* que representa una sentencia SQL con dos parámetros de entrada. El objeto *ps* contendrá la sentencia SQL indicada en el string después de enviarla al SGBD y prepararla para su ejecución. En este caso se ha creado una sentencia SQL con dos parámetros de entrada. Puede ser cualquier sentencia SQL (selects, updates, etc.).

```
Connection con = DriverManager.getConnection(jdbcUrl, "root", "");
```

```
PreparedStatement ps = con.prepareStatement(
                        "update facturas set serie=? where id=?" );
```

UTILIZANDO PARÁMETROS DE ENTRADA

Antes de poder ejecutar un objeto *PreparedStatement* se debe asignar un valor para cada uno de sus parámetros. Esto se realiza mediante la llamada a un método ***setXXX()***, donde *XXX* es el tipo apropiado para el parámetro. Por ejemplo, si el parámetro es de tipo *long*, el método a utilizar será ***setLong()***.

El primer argumento de los métodos *setXXX()* es la posición ordinal del parámetro al que se le va a asignar valor, y el segundo argumento es el valor a asignar. Por ejemplo en el siguiente fragmento de código crea un objeto *PreparedStatement* y acto seguido le asignas al primero de sus parámetros un valor de tipo String y al segundo un valor de tipo long.

```
String jdbcUrl = "jdbc:mysql:...";
Connection conn = DriverManager.getConnection(jdbcUrl,"root","");
PreparedStatement ps= conn.prepareStatement(
                        "update facturas set serie=? where id=?");
ps.setString(1, "B");
ps.setLong(2, 1);
ps.executeUpdate();
```

Una vez que se ha asignado unos valores a los parámetros de entrada de una sentencia, el objeto *PreparedStatement* se puede ejecutar múltiples veces, hasta que sean borrados los parámetros con el método ***clearParameters()***, aunque no es necesario llamar a este método cuando se quieran modificar los parámetros de entrada, sino que al ejecutar nuevos métodos ***setXXX()*** los valores de los parámetros se reemplazan. Para finalizar se llama al método ***executeUpdate()*** del objeto *PreparedStatement* y de esta forma se verán reflejados los cambios en la base de datos.

EJERCICIO 8: Realiza una aplicación que muestre un formulario que utilizando sentencias PrepapedStatement realice altas, bajas, ediciones y búsquedas en la tabla "VENDEDORES = Id + nombre + ingreso + salario" como muestra la figura. Crea la tabla, e inserta algunos datos de prueba si es necesario. Observa que el botón Busca puede utilitarse para buscar un vendedor por su Id (Select * from vendedores where id = ?).

10.4.3. LLAMAR A CÓDIGO ALMACENADO EN EL SGBD.

El último tipo de sentencias que podemos utilizar en JDBC son las ***CallableStatement***. Esta interface hereda de ***PreparedStatement*** y ofrece la posibilidad de manejar parámetros de salida y de realizar llamadas a procedimientos almacenados de la base de datos.

Creando Procedimientos Almacenados en MySQL con Scripts SQL

Desde MySQL podemos crear procedimientos almacenados de forma interactiva o creando scripts con las sentencias SQL a ejecutar.

EJEMPLO 8: script SQL que crea las tablas productos y proveedores y un procedimiento almacenado llamado muestra_proveedores sin parámetros.

```
select 'Creando tablas...' as ' ';
create table proveedores(
provID int not null auto_increment,
provNombre varchar(30),
provTelefono varchar(15),
constraint prod_pk primary key(provID)
);

create table productos(
prodID int not null auto_increment,
prodNombre varchar(30),
prodProveedor int,
prodPrecio numeric(10,2),
constraint prod_pk primary key(prodID),
constraint prod_fk_prov foreign key(prodProveedor)
                        references proveedores(provID)
);

select 'Borrando procedimiento muestra_proveedores' as ' ';
drop procedure if exists muestra_proveedores;
delimiter = '|';
select 'Creando procedure muestra_proveedores' as ' '|
create procedure muestra_proveedores()
begin
    select provNombre,
           prodNombre
    from proveedores join productos on provID = prodProveedor
    order by provNombre;
end|
```

En MySQL las sentencias de un procedimiento almacenado se separan por puntos y coma, pero se necesita otro delimitador para acabar la

sentencia que crea el propio procedimiento. En este caso se usa el carácter tubería (|) pero puedes usar otro con la sentencia DELIMITER para indicarlo. En un proyecto ant, puedes usar el fichero de configuración siguiente para ejecutar las sentencias del script SQL en el fichero crea_procedimientos.sql:

```
<target name="mysql-crea-procedures">
  <sql driver="${DB.DRIVER}"
       url="${DB.URL}" userid="${DB.USER}"
       password="${DB.PASSWORD}"
       classpathref="CLASSPATH"
       print="true"
       delimiter="|"
       autocommit="false"
       onerror="abort">
       <transaction
         src="./sql/${DB.VENDOR}/crea_procedimientos.sql">
       </transaction>
  </sql>
</target>
```

Creando Procedimientos desde Java

También puedes crearlos desde Java con el API JDBC.

EJEMPLO 9: método que crea un procedimiento almacenado desde código Java. El procedimiento devuelve un resulset aunque no devuelva nada de manera explícita en sus sentencias ni use parámetros.

```
public void creaProcMuestraProveedores() throws SQLException {
    String qd = "drop procedure if exists muestra_proveedores";
    String cp = "create procedure muestra_proveedores() " +
                "begin " +
                    "select provNombre, prodNombre " +
                    "from proveedores join productos " +
                          "on provID = prodProveedor " +
                    "order by provNombre;" +
                "end";
    try(Statement s = con.createStatement()) {
        System.out.println("Borrando si existe");
        s.execute(qd);
```

```
            } catch (SQLException e) {
                e.printStackTrace();
            }
            try(Statement s = con.createStatement()) {
                s.executeUpdate(cp);
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
```

En este método el delimitador no cambia. La llamada al procedure genera un resultset aunque no use ninguna sentencia return ni use parámetros. Para ejecutarlo desde Java usaremos la sentencia `CallableStatement.executeQuery`:

```
CallableStatement cs = null;
cs = con.prepareCall("{call muestra_proveedores}");
ResultSet rs = cs.executeQuery();
```

EJEMPLO 10: procedimiento quien_suministra(producto) que devuelve los proveedores de un producto concreto cuyo código le pasaremos en un parámetro de entrada.

```
drop procedure if exists quien_suministra";
delimiter = '|'
create procedure quien_suministra(
        in producto int,
        out proveedor varchar) |
begin
    select provNombre into proveedor
    from proveedores join productos on provID = prodProveedor
    where prodID = producto;
    select proveedor;
end |
delimiter = ';'
```

Para asignar el valor al parámetro de salida usa una sentencia select al final.

EJEMPLO 11: creación de un procedimiento `calcula_precio(producto,`

beneficio, nuevoPrecio) al que le pasamos tres parámetros, uno de entrada (el id de un producto), otro de entrada (el porcentaje de beneficio que queremos obtener) y el tercero de entrada y salida (el nuevo precio que se fija al producto).

```
drop procedure if exists calcula_precio";
delimiter = '|';
create procedure calcula_precio(
   in producto int,
   in beneficio float,
   inout nuevoPrecio numeric(10,2)
)
begin
  main: begin
    declare max numeric(10,2);
    declare ant numeric(10,2);
    select prodPrecio into ant
    from productos
    where prodID = producto;
    set max = ant * (1 + beneficio);
    if (nuevoPrecio > max) then
        set nuevoPrecio = max;
    end if;
    if (nuevoPrecio <= ant) then
        set nuevoPrecio = ant;
        leave main;
    end if;
    update productos
    set prodPrecio = nuevoPrecio
    where prodID = producto;
    select nuevoPrecio;
  end main;
end|
```

Llamando a los procedimientos.

Un objeto ***CallableStatement*** ofrece la posibilidad de realizar llamadas a procedimientos almacenados de una forma estándar para todos los SGBD. La llamada a un procedimiento es lo que contiene un objeto ***CallableStatement***. Esta llamada está escrita con una sintaxis

de escape y puede tener dos formas diferentes:

- **una con un parámetro de resultado**, es un tipo de parámetro de salida que representa el valor devuelto por el procedimiento (como si fuese una función).
- **Otra sin ningún parámetro de resultado**. Ambas formas pueden tener un número variable de parámetros de entrada, de salida o de entrada/salida. Una interrogación representará al parámetro.

La sintaxis para realizar la llamada a un procedimiento almacenado:

```
{call nombre_del_procedimiento[(?,?,...)]}
```

Si devuelve un parámetro de resultado (como una función):

```
{?=call nombre_del_procedimiento[(?.?...)]}
```

La sintaxis de una llamada a un procedimiento sin parámetros ni valor de retorno sería:

```
{call nombre_del_procedimiento}
```

Normalmente al crear un objeto ***CallableStatement*** el programador debe saber si el SGBD soporta procedimientos almacenados y como se usa el procedimiento.

La interface ***CallableStatement*** hereda los métodos de ***Statement***, que se encargan de sentencias SQL generales y también los métodos de ***PreparedStatement***, que maneja parámetros de entrada.

Los métodos que define la interface ***CallableStatement*** se encargan de manejar parámetros de salida: registrando los tipos JDBC de los parámetros de salida, recuperando los valores o testeando si un valor devuelto es un JDBC NULL.

EJEMPLO 12: llamada al procedimiento almacenado `quien_suministra(1)` y recoger el valor del parámetro de salida.

```
cs = con.prepareCall("{call quien_suministra(?, ?)}");
cs.setString(1, 1);
cs.registerOutParameter(2, Types.VARCHAR);
cs.executeQuery();
String proveedor = cs.getString(2);
```

Una sentencia ***CallableStatement*** puede tener parámetros de entrada, de salida y de entrada/salida. Para pasarle parámetros de entrada a un objeto ***CallableStatement***, se utilizan los métodos ***setXXX()*** que heredaba de la interface ***PreparedStatement***.

Si el procedimiento almacenado devuelve parámetros de salida, el tipo JDBC de cada parámetro de salida debe ser registrado antes de ejecutar el objeto ***CallableStatement*** correspondiente. Para registrar los tipos JDBC de los parámetros de salida se debe ejecutar el método ***CallableStatement.registerOutParameter()***.

Después de ejecutar la sentencia, se pueden recuperar los valores de estos parámetros llamando al método ***getXXX()*** adecuado. El método *getXXX()* debe recuperar el tipo Java que se correspondería con el tipo JDBC con el que se registró el parámetro. A los métodos *getXXX()* se le pasará un entero que indicará el valor ordinal del parámetro a recuperar.

EJEMPLO 13: Vamos a ver un ejemplo de llamada a un procedimiento remoto que devuelve un valor numérico en un parámetro de entrada y salida y un string que pasamos como parámetro de entrada lo devuelve como un ResultSet:

```
-- creamos el procedimiento en mysql
create procedure demo(in iparam varchar(255), inout ioparam int)
begin
```

```
    declare z int;
    set z = ioparam + 1;
    set ioparam = z;
    select iparam;
    select concat('zyxw', iparam);
end
```

Para usar el procedure hay que realizar varios pasos. Preparar la sentencia con `Connection.prepareCall()`. Debes usar la sintaxis de escapar caracteres de JDBC donde las llaves externas no son opcionales:

```
import java.sql.CallableStatement;
...
// Prepara llamada a 'demo' y dar valor a parámetros de entrada
// Observa la sintaxis de JDBC-escape: ({call ...})
CallableStatement cSQL = conn.prepareCall("{call demo(?, ?)}");
```

Nota: *el método* ***Connection.prepareCall()*** *consume muchos recursos debido a que recupera muchos metadatos desde el driver. Por motivos de rendimiento, hay que minimizar las llamadas innecesarias y reutilizar las instancias.*

Ahora se registran los parámetros de salida (OUT e INOUT)

```
import java.sql.Types;
...
cSQL.registerOutParameter(2, Types.INTEGER);
cSQL.registerOutParameter("ioParam", Types.INTEGER);
...
```

Ahora se da valor a los parámetros de entrada (IN) y opcionalmente a los (INOUT).

```
cSQL.setString(1, "abcdefg");
// alternativamente: cSQL.setString("iParam", "abcdefg");
cSSQL.setInt(2, 1);
// alternativamente cSQL.setInt("ioParam", 1);
...
```

Por último se ejecuta y se recuperan los valores de vuelta. Aunque soporta todos los métodos de ejecución (executeUpdate(), executeQuery(), execute()), el más flexible es **execute()**, que te permite comprobar si el procedimiento almacenado devuelve un result set y procesarlo si es lo que quieres.

```
...
boolean tieneResultados = cSQL.execute();
// Procesar resultados
while(tieneResultados){
    ResultSet rs = cSQL.getResultSet();
    // procesa el result set...
    tieneResultados = cSQL.getMoreResults();
}
// Recupera parámetros de salida por índice o por nombre
int salida1= cSQL.getInt(2); // por índice
salida1= cSQL.getInt("ioParam"); // por nombre
...
```

EJEMPLO 14: En el siguiente fragmento de código se registra un parámetro de salida, se ejecuta el procedimiento llamado por el objeto *CallableStatement* y luego recupera el valor de los parámetros de salida.

```
String jdbcUrl = "jdbc:mysql:...";
Connection conn = DriverManager.getConnection(jdbcUrl,"root","");
// Prepara la llamada al procedimiento
CallableStatement cSQL = conn.prepareCall(
                              "{call compras_clientes(?,?)}" );
cSQL.registerOutParameter(2, java.sql.Types.INTEGER);
cSQL.setInt(1, 360);
cSQL.execute();
```

EJERCICIO 9: Realizar una aplicación que muestre un formulario que utilizando una sentencia *CallableStatement* invoque al procedimiento *total_factura(IN factura, OUT total)* para cada una de las líneas de factura de la factura elegida a partir de un *JComboBox*.

```
FACTURA = Id + fecha + cliente + vendedor
LINEAFACTURA = Id + linea + descripcion + cantidad + precio
       clave ajena: Id --> FACTURA(Id)
```

10.4.4. CASOS ESPECIALES.

Recuperar valor de una columna AUTO_INCREMENT

Cuando tienes una columna de tipo AUTO_INCREMENT e insertas una fila, el programa desconoce el valor de la clave en la fila que se almacena en la BD ¿Cómo puede averiguarlo?

EL método más usado es `getGeneratedKeys()` y otra forma sería usando la consulta `SELECT LAST_INSERT_ID()`. Y en un ResultSet modificable puedes usar el propio método **`insertRow()`**.

EJEMPLO 15: Recuperar con Statement.getGeneratedKeys()

```
...
Statement sent = null;
ResultSet rs = null;
try {
    // Asumiendo que la conexión conn está realizada...
    sent = conn.createStatement();
    // Creamos la tabla con DDL
    sent.executeUpdate("DROP TABLE IF EXISTS autoInc");
    sent.executeUpdate(
            "CREATE TABLE autoInc("
            + "pk INT NOT NULL AUTO_INCREMENT, "
            + "dato VARCHAR(64), PRIMARY KEY (pk))");
    // Inserta una fila
    sent.executeUpdate(
            "INSERT INTO autoInc(dato) "
            + "values('Puedo saber el valor auto_increment?')",
            Statement.RETURN_GENERATED_KEYS);
    int autoPk = -1;
    rs = sent.getGeneratedKeys();
    if( rs.next() ) {
        autoPk = rs.getInt(1);
    }
    else {
```

```
            // throw una excepción...
        }
        System.out.println("Clave devuelta por getGeneratedKeys():"
            + autoPk);
    }
    finally {
        if( rs != null ) {
            try { rs.close(); } catch(SQLException e) { /*ignora*/ }
        }
        if( sent != null ) {
            try { sent.close(); } catch(SQLException e) { /*ignora*/ }
        }
    }
```

EJEMPLO 16: Usando la consulta SELECT LAST_INSERT_ID();

```
    Statement sent= null;
    ResultSet rs = null;
    try {
        sent= conn.createStatement();
        sent.executeUpdate("DROP TABLE IF EXISTS autoInc");
        sent.executeUpdate(
                "CREATE TABLE autoInc("
                + "pk INT NOT NULL AUTO_INCREMENT, "
                + "dato VARCHAR(64), PRIMARY KEY(pk))");
        sent.executeUpdate(
                "INSERT INTO autoInc(dato) "
                + "values('Puedo saber el valor de la clave auto?')");
        // Uso de la función MySQL LAST_INSERT_ID()
        int autoPk = -1;
        rs = sent.executeQuery("SELECT LAST_INSERT_ID()");
        if( rs.next() ) {
            autoPk= rs.getInt(1);
        }
        else {
            // throw una excepción
        }
        System.out.println("Clave devuelta por " +
                           "'SELECT LAST_INSERT_ID()': " + autoPk);
    }
    finally {
        if( rs != null ) {
```

```
            try { rs.close(); } catch(SQLException e) { /*ignora*/ }
        }
        if( sent != null ) {
            try { sent.close(); } catch(SQLException e) { /*ignora*/ }
        }
    }
```

EJEMPLO 17: usando un ResultSet Updatable

```
    Statement sent = null;
    ResultSet rs = null;
    try {
      sent= conn.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                                 java.sql.ResultSet.CONCUR_UPDATABLE);
      sent.executeUpdate("DROP TABLE IF EXISTS autoInc");
      sent.executeUpdate( "CREATE TABLE autoInc("
                          + "pk INT NOT NULL AUTO_INCREMENT, "
                + "dato VARCHAR(64), PRIMARY KEY (pk))");
        rs = sent.executeQuery( "SELECT pk, dato FROM autoInc" );
        rs.moveToInsertRow();
        rs.updateString("dato", "Valor de la PK?");
        rs.insertRow();
        rs.last();
        int autoPk = rs.getInt("pk");
        System.out.println("Clave devuelta: " + autoPk);
    }
    finally {
        if( rs != null ) {
            try { rs.close(); } catch(SQLException e) { /*ignora*/ }
        }
        if( sent != null ) {
           try { sent.close(); } catch(SQLException e) { /*ignora*/ }
        }
    }
```

El valor devuelto por la consulta `SELECT LAST_INSERT_ID()` es válido en el marco de una conexión, si alguna otra sentencia se ejecuta en la misma conexión, el valor se sobreescribe. Pero el método `getGeneratedKeys()` es válido porque está aislado en cada sentencia, así que mientras en el mismo objeto Statement no se ejecute otra, será válido aunque en la misma conexión si se hayan ejecutado más sentencias.

CONFIGURAR BALANCEO DE CARGA

El conector/J ofrece mecanismos para distribuir lecturas/escrituras entre varios servidores MySQL en una configuración en cluster o servidores desplegados con replicación maestro <-> maestro. Puedes definir balanceo de conexiones dinámicas. Se configura modificando la URL con la que se conecta la aplicación:

```
jdbc:mysql:loadbalance://[host1][:port1],[h2][:p2]...[/[database]]
[?property1=Value1[&property2=Value2]...]
```

A parte de las propiedades normales aparecen dos nuevas:

- **loadBalanceConnectionGroup** – Ofrece la habilidad de agrupar conexiones de diferentes orígenes. Te permite agrupar estos orígenes JDBC como una única clase dándoles un mismo nombre al grupo. Si no defines un nombre para la propiedad, todas las conexiones balanceadas comparten el mismo nombre y no se diferencian entre sí.
- **ha.enableJMX** – Si una vez definido el nombre del grupo de conexiones balanceadas, quieres administrarlas externamente, debes activar esta propiedad dándole el valor true. Aunque si inicias la aplicación con el flag `Dcom.sun.management.jmxremote` JVM puedes conectarte y realizar operaciones usando un cliente JMX como jconsole.

Una vez que hay conexiones balanceadas creadas, cada una tiene propiedades para monitorizarlas:

- Contador de hosts activos.
- Conexiones físicas activas.
- Conexiones lógicas activas.
- Total de conexiones lógicas creadas.
- Total de transacciones.

También pueden realizarse las siguientes operaciones:

- Añadir un nuevo host.
- Eliminar un host.

Los métodos de la interface JMX definida en `com.mysql.cj.jdbc.jmx.LoadBalanceConnectionGroupManagerMBean` son:

- `int getActiveHostCount(String group);`
- `int getTotalHostCount(String group);`
- `long getTotalLogicalConnectionCount(String group);`
- `long getActiveLogicalConnectionCount(String group);`
- `long getActivePhysicalConnectionCount(String group);`
- `long getTotalPhysicalConnectionCount(String group);`
- `long getTotalTransactionCount(String group);`
- `void removeHost(String group, String host) throws SQLException;`
- `void stopNewConnectionsToHost(String group, String host) throws SQLException;`
- `void addHost(String group, String host, boolean forExisting);`
- `String getActiveHostsList(String group);`
- `String getRegisteredConnectionGroups();` Devuelve los nombres de todos los grupos de conexión activos.

EJEMPLO 18: monitoriza conexiones balanceadas:

```
public class Test {

    private static String URL = "jdbc:mysql:loadbalance://" +
        "localhost:3306,localhost:3310/test?" +
        "loadBalanceConnectionGroup=first&ha.enableJMX=true";

    public static void main(String[] args) throws Exception {
        new Thread(new Repeater()).start();
        new Thread(new Repeater()).start();
        new Thread(new Repeater()).start();
    }

    static Connection getNewConnection() throws
```

```
                                SQLException, ClassNotFoundException {
            Class.forName("com.mysql.cj.jdbc.Driver");
            return DriverManager.getConnection(URL, "root", "");
        }

        static void executeSimpleTransaction(
                    Connection c, int conn, int trans) {
            try {
                c.setAutoCommit(false);
                Statement s = c.createStatement();
                s.executeQuery("SELECT SLEEP(1) /* Connection: " +
                           conn + ", transaction: " + trans + " */");
                c.commit();
            }
            catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    public static class Repeater implements Runnable {
         public void run() {
            for(int i=0; i < 100; i++){
              try {
                 Connection c = getNewConnection();
                 for(int j=0; j < 10; j++){
                    executeSimpleTransaction(c, i, j);
                    Thread.sleep(Math.round(100 * Math.random()));
                 }
                 c.close();
                 Thread.sleep(100);
              }
              catch (Exception e) {
                     e.printStackTrace();
              }
            }
         }
      }
    }
```

Nº FILAS QUE INTERCAMBIAN APP <-> DRIVER<-> SGBD

Un programa cliente que usa JDBC cuando lanza una consulta y tiene

como respuesta muchas filas, no las recibe todas de golpe (puede obligarle a consumir muchos recursos y el tiempo de mover esos datos desde el servidor hasta el equipo por la red puede ser excesivo).

En realidad el driver va copiando poco a poco troozos de los datos resultado. El programa puede limitar la cantidad de filas que recibe en cada trozo, e ir recibiendo los resultados a trozos del tamaño que le interese. A esta cantidad de filas se le llama **fetch size**.

Lo óptimo sería ajustar esta cantidad según se vayan a utilizar los datos de cada consulta. **Se cambia a nivel de sentencia**, cuando creas un objeto de **java.sql.Statement**. Al cambiar la propiedad, todas las sentencias que ejecutes con esa instancia recuperan trozos de esa cantidad de filas.

También puedes hacerlo a nivel de ResultSet, pero en este caso, no tiene efecto hasta que el driver recupere la siguiente cantidad de datos.

EJEMPLO 19: Averiguar el Fetch Size utilizado.

```
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.Connection;
import java.sql.SQLException;
import jcb.util.DatabaseUtil;
...
ResultSet rs = null;
Statement st = null;
Connection co = null;
try {
    co = <...obtener una conexión con la BD...>;
    // A nivel de sentencia
    st= co.createStatement();
    String q = "select isbn,titulo from libros";
    rs = st.executeQuery(q);
    int fs1 = st.getFetchSize();
```

```
        // A nivel de result set
        int fs2= rs.getFetchSize();
        System.out.printf("Fetch size a nivel de:\n" +
                          "Sentencia %d\n Resultset %d", fs1, fs2);
    }
    catch (SQLException e) {
        // manejar excepciones
    }
    finally {
        DatabaseUtil.close(rs);
        DatabaseUtil.close(st);
        DatabaseUtil.close(co);
    }
```

EJEMPLO 20: Modificar el Fetch Size utilizado.

```
    import java.sql.ResultSet;
    import java.sql.Statement;
    import java.sql.Connection;
    import java.sql.SQLException;
    import jcb.util.DatabaseUtil;
    ...
    ResultSet rs = null;
    Statement st = null;
    Connection co = null;
    try {
        co = <...obtener una conexión con la BD...>;
        // A nivel de sentencia
        st= co.createStatement();
        st.setFetchSize(20);
        rs = st.executeQuery("SELECT isbn, titulo FROM libros");
        // A nivel de resultset, tiene efectos la próx. vez que
        // recupere...
        rs.setFetchSize(40);
    }
    catch (SQLException e) {
        // manejar excepciones
    }
    finally {
        DatabaseUtil.close(rs);
        DatabaseUtil.close(st);
        DatabaseUtil.close(co);
    }
```

10.4.5. TIPOS DE DATOS AVANZADOS.

Son tipos de datos especiales que dan a las BD relacionales más flexibilidad al poder definirse columnas para que los almacenen. Por ejemplo una columna de tipo BLOB (binary large object) puede almacenar una cantidad enorme de bytes sin formato (raw bytes) lo que permite que una celda contenga un documento .pdf, un programa, un fichero, una imagen, un vídeo, un audio, etc. Una columna de tipo `CLOB` (character large object) es capaz de almacenar una enorme cantidad de caracteres (páginas web, programas fuente, etc.

Por ejemplo el estándar de SQL de 2003 define los siguientes tipos de datos especiales:

- Los tipos aceptados desde SQL92: `CHAR`, `FLOAT`, `DATE`, etc.
- Los de SQL99: `BOOLEAN`, `BLOB` y `CLOB`.
- Los de SQL2003: objetos XML
- Tipos definidos por el usuario:
    - Estructurados:
      `CREATE TYPE Punto2D AS (X FLOAT, Y FLOAT) NOT FINAL`
    - Tipos `DISTINCT` basados en predefinidos:
      `CREATE TYPE moneda AS NUMERIC(10,2) FINAL`
- Tipos construidos basados en tipos base:
    - `REF(`*`structured-type`*`)`: puntero a filas que residen en el SGBD.
    - *Arrays de datos:* `ARRAY[`*`n`*`]`
- Localizadores: actúan como apuntadores a datos del SGBD. Hay utilidades para recuperar trozos de los datos con el localizador.
    - `LOCATOR(`*`structured-type`*`)`
    - `LOCATOR(`*`array`*`)`
    - `LOCATOR(`*`blob`*`)`
    - `LOCATOR(`*`clob`*`)`

- Datalink: localizadores pero a objetos externos al SGBD.

MAPEAR TIPOS DE DATOS AVANZADOS

JDBC aporta mapeos por defecto de los tipos especificados en el estándar SQL:2003 a tipos de JAva:

- BLOB: Blob interface
- CLOB: Clob interface
- NCLOB: NClob interface
- ARRAY: Array interface
- XML: SQLXML interface
- Structured types: Struct interface
- REF(structured type): Ref interface
- ROWID: RowId interface
- DISTINCT: el tipo base que utilice el SGBD. Por ejemplo, un tipo DISTINCT basado en el tipo SQL NUMERIC se mapea a java.math.BigDecimal porque NUMERIC se mapea aBigDecimal.
- DATALINK: objeto java.net.URL

USANDO TIPOS DE DATOS AVANZADOS

Puedes recuperar, almacenar y modificar estos datos de manera similar a como manejas los tipos de datos habituales. Usarás tanto el método `ResultSet.get`*`DataType`* como `CallableStatement.get`*`DataType`* para recuperarlos, `PreparedStatement.set`*`DataType`* para almacenarlos y `ResultSet.update`*`DataType`* para modificarlos (donde el sufijo *DataType* es el nombre de la interface Java a la que se mapea cada uno). La siguiente tabla resume los métodos usados:

Advanced Data Type	get*DataType* Method	set*DataType* method	update*DataType* Method
BLOB	getBlob	setBlob	updateBlob

Advanced Data Type	get*DataType* Method	set*DataType* method	update*DataType* Method
CLOB	getClob	setClob	updateClob
NCLOB	getNClob	setNClob	updateNClob
ARRAY	getArray	setArray	updateArray
XML	getSQLXML	setSQLXML	updateSQLXML
Structured type	getObject	setObject	updateObject
REF(structured type)	getRef	setRef	updateRef
ROWID	getRowId	setRowId	updateRowId
DISTINCT	getBigDecimal	setBigDecimal	updateBigDecimal
DATALINK	getURL	setURL	updateURL

Nota: *el tipo DISTINCT es un poco diferente.*

EJEMPLO 21: este código recupera un valor SQL ARRAY suponiendo que la columna notas de la tabla estudiantes contiene valores de tipo ARRAY y *s es un objeto de tipo* Statement.

```
ResultSet rs = s.executeQuery("select notas from estudiantes " +
                              "where nia = 002238");
rs.next();
Array notas = rs.getArray("notas");
```

USANDO OBJETOS GRANDES

Los objetos Blob, Clob y NClob de Java ayudan a manipular estos objetos. Algunas implementaciones representan las instancias de estos tipos mediante un locator (un puntero lógico del peso de un entero) al objeto de la BD para mejorar el rendimiento, pues estos datos podrían ser muy pesados, sin embargo otras implementaciones mueven todos esos datos a la computadora donde se ejecuta la aplicación. En Java este tipo de objetos se materializan mediante streams.

Añadir un Clob a la BD

Imaginemos que en la tabla comentarios hay una columna llamada comentarios de tipo CLOB (un texto arbitrariamente grande). El objeto Java `Clob` contiene los datos del fichero especificado en `fileName`.

```
public void insertaComentario(String nia, String fileName)
throws SQLException {
    String sql = "insert into comentarios values(?,?)";
    Clob clob = conexion.createClob();
    try(PreparedStatement ps = conexion.prepareStatement(sql);
        Writer w = clob.setCharacterStream(1);){
        String str = leeFichero(fileName, w);
        System.out.println("Comentario: " + w.toString());
        if(sgbd.equals("mysql")) {
            System.out.println("MySQL añade Clob con setString");
            clob.setString(1, str);
        }
        System.out.println("Longitud: " + clob.length());
        ps.setString(1, nia);
        ps.setClob(2, clob);
        ps.executeUpdate();
    } catch (SQLException sqlex) {
        sqlex.printStack();
    } catch (Exception e) {
        System.out.println("Error no esperado: " + e.toString());
    }
}
```

El argumento 1 del método `setCharacterStream(1)` indica que el objeto Writer comienza a escribir el stream de caracteres al principio de los datos. Y falta el método leeFichero que es el que pasa los datos del fichero al Clob, que escribe en la BD.

```
private String leeFichero(String fichero, Writer w)
throws IOException {
    try (BufferedReader br = new BufferedReader(
                                    new FileReader(fileName))
        ) {
        String linea = "";
```

```
            StringBuffer sb = new StringBuffer();
            while((linea = br.readLine()) != null) {
                System.out.println("Escribiendo: " + linea);
                w.write(linea);
                sb.append(linea);
            }
            // Convertir contenido a string
            String datos = sb.toString();
            return datos;
        }
    }
```

Leer un CLOB de la BD

Este método recupera el comentario de un alumno que es un CLOB:

```
    public String leeComentario(String nia, int numChar)
    throws SQLException {
        String coment = null;
        Clob clob = null;
        String q = "select comentario from comentarios where nia = ?";
        try(PreparedStatement ps = conexion.prepareStatement(q)) {
            ps.setString(1, nia);
            ResultSet rs = ps.executeQuery();
            if(rs.next()) {
                clob = rs.getClob(1);
                System.out.println("Longitud: " + clob.length());
            }
            coment = clob.getSubString(1, numChar);
        } catch (SQLException sqlex) {
            sqlex.printStack();
        } catch (Exception ex) {
            System.out.println("Error inesperado: " + ex.toString());
        }
        return coment;
    }
```

Guardar y Leer Objetos BLOB

Se trabaja de forma similar a los tipos BLOB. Guardar imágenes o ficheros en una BD para su posterior uso es algo fundamental en programación. Veremos los 2 métodos principales que nos permitiran hacer esto, un metodo para guardar una Imagen en la BD en un campo

de **tipo blob** y un metodo que nos permite obtener el contenido de ese campo y convertilo a un array de bytes para su posterior conversion a objeto Image de Java. Se utilizan los métodos **`Blob.setBinaryStream()`** y **`getBlob()`**

EJEMPLO 22: Imagina que tenemos una aplicación GUI que divide su interfaz en dos zonas. En la superior, podemos seleccionar una imagen del disco, visualizarla y pulsando un botón guardarla en una BD. En la parte inferior, tenemos una vista previa de todas las imágenes de la BD y un botón que al pulsarlo copia la imagen actualmente seleccionada y la deja en el disco. La conexión es a una BD MySQL, donde existe una tabla llamada imagenes con las columnas idImagen, imagen y nombre:

```
create table imagenes(
idImagen int not null auto_increment,
imagen blob not null,
nombre varchar(30) not null,
constraint imagnes_pk primary key(idImagen),
constraint imagenes_ak unique(nombre)
)
engine= Innodb;
```

Las conexión suponemos que está realizada y que la sentencia está creada:

```
Connection conexion;
Statement sent;
```

El siguiente metodo recibe una cadena String con la ruta de la imagen en disco y el nombre de la imagen (para cuando se quiera leer y guardar de nuevo a disco), después se utiliza el metodo **setBinaryStream()** para insertarla en la Base de Datos.

```
public boolean guardarImagen(String ruta, String nombre){
    String insert = "insert into Imagenes(imagen,nombre) values(?,?)";
    FileInputStream fis = null;
    PreparedStatement ps = null;
    try {
```

```
            conexion.setAutoCommit(false);
            File file = new File(ruta);
            fis = new FileInputStream(file);
            ps = conexion.prepareStatement(insert);
            ps.setBinaryStream(1, fis, (int)file.length() );
            ps.setString(2, nombre);
            ps.executeUpdate();
            conexion.commit();
            return true;
        }
        catch (Exception e) {
            e.printStack();
        }
        finally{
            try { ps.close(); fis.close();
            } catch (Exception e) {
                e.printStack();
            }
        }
        return false;
    }
```

El siguiente método devuelve un Arraylist de Objetos tipo Imagen, este tipo Imagen es una clase definida por nosotros para guardar tanto la imagen como el nombre de esta. Después seleccionamos todas las imágenes y nombres de la BD, obtenemos el campo tipo blob y convertimos a imagen, obtenemos el nombre de la imagen y luego los agregamos en nuevo tipo Imagen, este proceso se repite hasta que se agregen todas las imágenes de la base de datos en nuestro ArrayList, luego retornamos esta lista con todas las imagenes y nombres de imagen.

```
ArrayList<Imagen> getImagenes() {
   ArrayList<Imagen> lista = new ArrayList();
   try {
      ResultSet rs= st.executeQuery(
                              "select imagen,nombre from Imagenes");
      while( rs.next() ) {
          Imagen imagen= new Imagen();
          Blob blob = rs.getBlob("imagen");
```

```
            String nombre = rs.getObject("nombre").toString();
            byte[] datos = blob.getBytes(1, (int)blob.length() );
            BufferedImage img = null;
            try {
                img = ImageIO.read(new ByteArrayInputStream(datos) );
            }
            catch( IOException ex ) {
               ex.printStack();
            }
            imagen.setImagen(img);
            imagen.setNombre(nombre);
            lista.add(imagen);
          }
          rs.close();
        }
        catch (SQLException ex) {
            ex.prinstStack();
        }
        return lista;
    }
```

Liberar Recursos

Los objetos Java Blob, Clob y NClob permanecen creados hasta que la transacción en que aparecieron no finaliza. Esto puede provocar escasez de recursos si la transacción se alarga mucho en el tiempo. Las aplicaciones deberían liberarlos de manera explícita tan pronto como no los necesiten más usando el método `free()`.

```
Clob c = conexion.createClob();
int numWritten = c.setString(1, val);
c.free();
```

USANDO OBJETOS SQLXML

La interface `Connection` de JDBC da soporte a la creación de objetos SQLXML usando los métodos **`createSQLXML()`**. El objeto creado no contiene datos, estos se añaden usando a `setString()`, `setBinaryStream()`, `setCharacterStream()` o `setResult()` de la interface SQLXML.

Creando Objetos SQLXML

Usamos el objeto Connection y el método `createSQLXML()` y lo llenamos de datos con `setString()`.

```
Connection con = DriverManager.getConnection(url, props);
SQLXML xml = con.createSQLXML();
xml.setString(valor);
```

Recuperar valores SQLXML de un ResultSet

El tipo de dato SQLXML se trata como otro tipo cualquiera (a partir de la especificación 4.0 de JDBC) y puede recuperarse desde un ResultSet llamando a los métodos **`getSQLXML()`** del ResultSet o de la interface `CallableStatement`.

```
SQLXML xml = rs.getSQLXML(1); // existirá toda la transacción
```

Acceder a los datos del Objeto SQLXML

La interface ofrece `getString()`, `getBinaryStream()`, `getCharacterStream()` y `getSource()`.

```
SQLXML xml = rs.getSQLXML(1);
String val = xml.getString();
```

El método `getBinaryStream()` y `getCharacterStream()` pueden utilizarse respectivamente para obtener un `InputStream` o un `Reader` y pasarlo directamente a un parser XML:

```
SQLXML xml = rs.getSQLXML(columna);
InputStream bis = xml.getBinaryStream();
DocumentBuilder parser =
    DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document result = parser.parse(bis);
```

El método `getSource()` devuelve un objeto `javax.xml.transform.Source` que es usado como entrada de parser XML y transformadores XSLT.

```
SQLXML xml = rs.getSQLXML(1);
```

```
SAXSource saxS = xml.getSource(SAXSource.class);
XMLReader xmlR = saxS.getXMLReader();
xmlR.setContentHandler(myHandler);
xmlR.parse(saxS.getInputSource());
```

Almacenando Objetos SQLXML

Un objeto SQLXML puede pasarse como parámetro de entrada de un objeto `PreparedStatement` como otro tipo de dato usando el método `setSQLXML()`.

```
PreparedStatement ps =
    con.prepareStatement("insert into biografia(xmlDatos, d)" +
                         " values (?, ?)");
ps.setSQLXML(1, autorDatos);
ps.setInt(2, autorId);
```

Inicializando Objetos SQLXML

El siguiente fragmento usa el método `setResult()` para devolver un objeto `SAXResult` para poblar un objeto SQLXML:

```
SQLXML xml = con.createSQLXML();
SAXResult saxR = xml.setResult(SAXResult.class);
ContentHandler ch = saxR.getXMLReader().getContentHandler();
contentHandler.startDocument();
// Fija los elementos XML y atributos...
contentHandler.endDocument();
```

Este código usa `setCharacterStream()` para obtener un objeto `java.io.Writer` para inicializarlo.

```
SQLXML xml = con.createSQLXML();
Writer out= xml.setCharacterStream();
BufferedReader in = new BufferedReader(
                        new FileReader("xml/ejemplo.xml"));
String linea = null;
while((linea = in.readLine() != null) {
    out.write(linea);
}
```

Liberar recursos de SQLXML

Como ocurría con los objetos large, es conveniente liberarlos explícitamente tan pronto como no se necesiten.

```
SQLXML xml = con.createSQLXML();
xml.setString(valor);
xml.free();
```

EJEMPLO 23: MySQL y su driver JDBC no soportan completamente los datos SQLXML. Como no tiene el tipo SQLXML usaremos un longtext para guardarlo. Imagina que queremos almacenar en formato XML noticias de un canal especializado en el sector de la empresa. Las noticias pueden por ejemplo estar en el fichero de nombre "rss-coffee-industry-news.xml" y puede tener este contenido:

```
<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Coffee Industry News</title>
    <link>http://www.example.com/thecoffeebreak/blog</link>
    <description>Coffee Industry News: Your source for the latest news in the coffee and beverage industry</description>

    <item>
      <title>The Coffee Break Revamps Its Online Presence</title>
      <link>https://www.example.com/coffeeindustrynews/2010/09/the-coffee-break-revamps-its-online-presence.html</link>
      <description>The Coffee Break has dramatically revamped its Web site! Check out their site at <a
href="http://www.example.com/thecoffeebreak">https://www.example.com/thecoffeebreak</a>.</description>
      <pubDate>Wed, 01 Sept 2010 16:00:00 GMT</pubDate>
      <guid>https://www.example.com/coffeeindustrynews/2010/09/the-coffee-break-revamps-its-online-presence.html</guid>

    </item>
    <item>
      <title>Home Espresso Machine Sales up 25% from Last Year</title>
      <link>https://www.example.com/thecoffeebreak/blog/2010/09/home-espresso-machine-espresso-sales-are-up-25.html</link>
      <description>Industry analysts have observed that home espresso machine sales are up 25% from last year. This should be good news to stores
that sell coffee beans!</description>
      <pubDate>Wed, 8 Sept 2010 16:00:00 GMT</pubDate>
      <guid>https://www.example.com/thecoffeebreak/blog/2010/09/home-espresso-machine-espresso-sales-are-up-25.html</guid>

    </item>
  </channel>
</rss>
```

Vamos a crear un programa que copie las noticias y las vuelque en la tabla de una base de datos mysql que tiene esta estructura:

```
mysql> create table noticias(
    -> rssId int not null auto_increment,
    -> nombreRSS varchar(60) not null,
    -> XML longtext not null,
    -> primary key(rssId)
    -> );
Query OK, 0 rows affected (0.04 sec)
```

```java
package p10;

import java.io.IOException;

import java.io.StringWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.SQLXML;
import java.sql.Statement;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Node;

import org.xml.sax.SAXException;

public class DemoXML {
  static Connection con = null;
```

```
  public static void insertaNoticias(String nombreFile)
  throws ParserConfigurationException, SAXException, IOException,
         XPathExpressionException, TransformerConfigurationException,
         TransformerException, SQLException {
      // Parse the document and retrieve the name of the RSS feed
      String sTitulo = null;
      javax.xml.parsers.DocumentBuilderFactory factory =
                  javax.xml.parsers.DocumentBuilderFactory.newInstance();
      factory.setNamespaceAware(true);
      DocumentBuilder builder = factory.newDocumentBuilder();
      Document doc = builder.parse(nombreFile);
      XPathFactory xPathfactory = XPathFactory.newInstance();
      XPath xPath = xPathfactory.newXPath();
      Node elementoTitulo =
(Node)xPath.evaluate("/rss/channel/title[1]", doc, XPathConstants.NODE);
      if (elementoTitulo == null) {
          System.out.println("No puedo recuperar el título");
          return;
      } else {
          sTitulo = elementoTitulo.getTextContent()
                                  .trim()
                                  .toLowerCase()
                                  .replaceAll("\\s+", "_");
          System.out.println("elemento título: [" + sTitulo + "]");
      }
      System.out.println( docToString(doc) );
      PreparedStatement insertaFila = null;
      SQLXML rssDato = null;
      try {
          System.out.println("Añadir fichero XML " + nombreFile);
          String q = "insert into noticias(nombreRSS, XML) values(?, ?)";
          insertaFila = con.prepareStatement(q);
          insertaFila.setString(1, sTitulo);
          System.out.println("Creando objeto SQLXML para MySQL");
          rssDato = con.createSQLXML();
          System.out.println("Creando objeto DOMResult");
          DOMResult dom = (DOMResult)rssDato.setResult(DOMResult.class);
          dom.setNode(doc);
          insertaFila.setSQLXML(2, rssDato);
          System.out.println("Ejecutando modificación()");
          insertaFila.executeUpdate();
      } catch (SQLException e) {
```

```
            e.printStackTrace();
        } catch (Exception ex) {
            System.out.println("Error inesperado:");
            ex.printStackTrace();
        }
        finally {
            if (insertaFila != null) { insertaFila.close(); }
        }
    }

    public static String docToString(Document doc)
    throws TransformerConfigurationException, TransformerException {
        Transformer t = TransformerFactory.newInstance().newTransformer();
        StringWriter sw = new StringWriter();
        t.transform(new DOMSource(doc), new StreamResult(sw));
        return sw.toString();
    }

    public static void verTabla()
    throws SQLException, ParserConfigurationException, SAXException,
    IOException, TransformerConfigurationException, TransformerException {
        try (Statement stmt = con.createStatement()) {
            String q = "select nombreRSS, XML from rss";
            ResultSet rs = stmt.executeQuery(q);
            while (rs.next()) {
                String rssNombre = rs.getString(1);
                SQLXML rssXML = rs.getSQLXML(2);
                javax.xml.parsers.DocumentBuilderFactory factory =
                   javax.xml.parsers.DocumentBuilderFactory.newInstance();
                factory.setNamespaceAware(true);
                DocumentBuilder builder = factory.newDocumentBuilder();
                Document doc = builder.parse(rssXML.getBinaryStream());
                System.out.println("Identificador RSS: " + rssNombre);
                System.out.println(docToString(doc));
            }
        } catch (SQLException e) {
        e.printStackTrace();
      }
    }

    public static void main(String[] args) {
      Connection con = null;
      try {
```

```
        String jdbcURL= "jdbc:mysql://localhost:3306/demodao?
&serverTimezone=UTC";
        con = DriverManager.getConnection(jdbcURL, "userdao",
"UserDAO_1");
        DemoXML.insertaNoticias("rss-coffee-industry-news.xml");
        DemoXML.verTabla();
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      try{ if(con != null) con.close(); } catch(Exception e) {};
    }
  }
}
```

El resultado de su ejecución:

```
elemento título: [coffee_industry_news]
<?xml version="1.0" encoding="UTF-8" standalone="no"?><rss version="2.0">
   <channel>
      <title>Coffee Industry News</title>
      <link>http://www.example.com/thecoffeebreak/blog</link>
      <description>Coffee Industry News: Your source for the latest news
```

USANDO OBJETOS ARRAY

Nota: MySQL no soporta el tipo de dato Array. Otros SGBD como Oracle y su driver JDBC implementan la interface `java.sql.Array` con la clase `oracle.sql.ARRAY`.

Creando Objetos Array

Usando el método `Connection.createArrayOf` se crean objetos `Array`. Imagina que en la BD tenemos una tabla llamada regiones que contiene una columna donde queremos guardar todos los códigos postales que tiene en un array (es para Oracle):

```
create table regiones(
nombreRegion varchar(32) not null,
zips varchar32 array[10] not null,
primary key(nombreRegion)
);
```

```
insert into regiones values('Noroeste', '{"93101", "97201",
"99210"}');
insert into regiones values('Suroeste', '{"94105", "90049",
"92027"}');
Connection c = DriverManager.getConnection(url, props);
String[] noreste = { "10022", "02110", "07399" };
Array a = c.createArrayOf("VARCHAR", noreste);
```

Leer Valores Array de un ResultSet

A partir de JDBC 4.0 puedes manipular objetos Array sin tener que materializar los datos en el dispositivo donde se ejecuta la aplicación (como el caso de objetos large). Cuando necesites hacerlo:

```
ResultSet rs= s.executeQuery("select regionNombre, zips from regiones" );
while( rs.next() ) {
    Array z = rs.getArray("zips");
    String[] zips = (String[])z.getArray();
    for(int i = 0; i < zips.length; i++) {
        if (!ZipCode.esValido(zips[i])) {
            // ...
        }
    }
}
```

Almacenando y Actualizando Objetos Array

Usando el método `setArray()` y `setObject()` de la clase `PreparedStatement` pasamos valores de un objeto Array como parámetro de entrada:

```
PreparedStatement ps = c.prepareStatement(
    "insert into regiones(nombreRegion, zips) values(?, ?)");
ps.setString(1, "NorthEast");
ps.setArray(2, anArray);
ps.executeUpdate();
```

De forma similar usamos los métodos `updateArray()` and `updateObject()` del objeto `PreparedStatement` para modificar una columnas.

Liberar Recursos

Es conveniente liberar recursos lo antes posible:

```
Array a = con.createArrayOf("VARCHAR", nombre);
// ...
a.free();
```

USANDO TIPOS DE DATOS DISTINCT

Nota: *MySQL no soporta el tipo de dato DISTINCT.*

El tipo de dato DISTINCT tiene un comportamiento distinto al resto. El usuario define uno basándose en los que ya existen, y que tienen un mapeo definido a tipos Java. Por tanto los tipos se mapean a los tipos en los que se basan.

Por ejemplo imagina que usamos un código de tres letras para representar la provincia en la que un usuario tiene fijada su residencia. Podemos crear un tipo de datos en una BD de la siguiente forma:

```
create type provincia as char(3);
```

Otras bases de datos usarían esta sintaxis:

```
create distinct type provincia as char(3);
```

Puedes dar el tipo provincia a las columnas de tus tablas. Pero como los valores de estas columnas se basan en el tipo char, su mapeo en Java es a un String. Por ejemplo si la columna 4 de un ResultSet llamado rs es de tipo provincia, reccuperarlo sería:

```
String state = rs.getString(4);
```

De la misma forma podrías usar el método `setString()` para almacenar un valor y el método `updateString()` para modificarlo.

USANDO OBJETOS ESTRUCTURADOS

Nota: *MySQL no soporta tipos definidos por el usuario.*

Los tipos de datos estructurados en SQL junto con los tipos DISTINCT son dos tipos de atos que los usuarios pueden definir. Ambos se denominan a veces UDTs (user-defined types) y se crean con la sentencia CREATE TYPE de SQL.
Por ejemplo, si una empresa tiene sucursulas decide almacenar en una tabla información de cada una.

- SucursalID para identificarla.
- Ubicación para situarla y localizarla
- servicios: los servicios que ofrece
- reponsable el nombre/código del director.

La columna ubicacion puede hacerse de tipo estructurado, servicios de tipo array y responsable una referencia a un tipo estructurado. Por ejemplo la ubicación:

```
create type direccion (
num integer,
calle varchar(40),
ciudad varchar(40),
provincia char(2),
municipio char(5)
ZIP char(5)
);
```

USANDO REFERENCIAS A TIPOS ESTRUCTURADOS

En la tabla sucursales, para sucursal se almacena el gerente del establecimiento. Si un empleado es gerente de varias sucursales, para evitar repetir la misma información, en vez de almacenar repetidamente los datos del gerente, se almacena una referencia al responsable. El modelo relacional usaría una clave ajena, si el gerente es un objeto el tipo de dato REF que no es más que un puntero lógico (como una clave ajena) al dato que se desea aparezca: REF(gerente).

Como un valor de tipo REF necesita estar asociado a los datos que referencia, se almacena en una tabla especial junto al dato. Cada tipo estructurado que sea referenciado tendrá su propia tabla:

```
create table gerentes of gerente(
oid ref(gerente)
vales are system generated
);
```

Esta sentencia crea esta tabla especial que tiene una sola columna oid que identifica a cada empleado que se cree y otra columna implícita donde se almacena cada empleado creado. El siguiente código al insertar cada empleado crea entradas en esa tabla especial.

```
insert into gerentes(geId, geApes, geNombre, geTelefono)
values(000001, 'montoya', 'alfredo', '8317225600');

insert into gerentes(id, apellidos, nombre, telefono)
values(000002, 'Cruz', 'Helena', '4153785600');
```

La tabla empleados tendrá 3 filas. La columna oid tendrá 3 identificadores únicos de tipo REF(empleado). Los oids se generan y almacenan automáticamente. Para acceder a una instancia REF(MANAGER), un empleado) lo seleccionas de su tabla. Por ejemplo si queremos acceder a los datos de "Alfredo Montoya" cuyo número de ID es 000001:

```
String jefe = "select oid from gerentes where gerenteID = 000001";
ResultSet rs = s.executeQuery(jefe);
rs.next();
Ref gerente = rs.getRef("oid");
```

EJEMPLO 24: conectarnos a un SGBD Oracle para crear tipos REF y usarlos desde Java.

```
package p10;

import java.sql.*;
```

```
public class DemoREF {

  public static void main(String args[]) {
      Connection c = null;
      Statement s = null;
      try {
         String creaGerentes = "create table generentes of gerente " +
                       "(oid ref(gerente) values are system generated)";
         String insertaG1 =
           "insert into gerentes(geId, geApes, geNombre, geTelefono) " +
           "values(000001, 'Montoya', 'Alfredo', '963123456')";
         String insertaG2 =
           "insert into gerentes(geId, geApes, geNombre, geTelefono) " +
           "values(000002, 'Cruz', 'Helena', '963234567')";
         String url = "jdbc:oracle:thin:@localhost:1521:XE";
         c = DriverManager.getConnection(url, "usu1", "pwd1");
         c.setAutoCommit(false);
         s = c.createStatement();
         s.executeUpdate(creaGerentes);
         s.addBatch(insertaG1);
         s.addBatch(insertaG2);
         int [] contadoresUpdate = s.executeBatch();
         c.commit();
         System.out.println("Contadores de modificación para:  ");
         for(int i = 0; i < contadoresUpdate.length; i++) {
            System.out.print("    comando " + (i + 1) + " = " +
                             contadoresUpdate[i] );
         }
      } catch(BatchUpdateException b) {
         System.err.println("-----Excepciones BatchUpdate-----");
         System.err.println("Mensaje:   " + b.getMessage());
         System.err.println("SQLState:  " + b.getSQLState());
         System.err.println("Vendedor:  " + b.getErrorCode());
         System.err.print("Contadores para comandos correctos:  ");
         int [] filasModificadas = b.getUpdateCounts();
         for (int i = 0; i < filasModificadas.length; i++) {
             System.err.print(filasModificadas[i] + "   ");
         }
         System.err.println("");
      } catch(SQLException ex) {
         System.err.println("------SQLException------");
         System.err.println("Mensaje:   " + ex.getMessage());
```

```
            System.err.println("SQLState:   " + ex.getSQLState());
            System.err.println("Vendedor    " + ex.getErrorCode());
        } finally {
            try {
                if(s != null) { s.close(); }
                if(c != null) { c.close(); }
            } catch(Exception e1) { e1.printStackTrace(); }
        }
    }
}
```

USANDO MAPAS OBJETO JAVA <-> TIPO SQL

> **Nota**: *actualmente MySQL no soporta tipos definidos por el usuario ni tipos DINTINCT, por tanto no es posible usar este sgbd para dar un ejemplo.*

Implementar SQLData

El primer paso para definir un mapeo personalizado es crear una clase que implemente la interface `SQLData`. Por ejemplo podemos usar el tipo `DIRECCION` definido en un SGBD que soporte definir tipos nuevos con estructura:

```
create type direccion(
  num integer,
  calle varchar(40),
  municipio varchar(40),
  provincia char(2),
  ZIP char(5)
);
```

Una clase que implemente `SQLData` para mapear el tipo dirección:

```
public class Dirección implements SQLData {
    public int num;
    public String calle;
    public String municipio;
    public String provincia;
    public String zip;
    private String sql_type;
```

```
    public String getSQLTypeName() { return sql_type; }

    public void readSQL(SQLInput stream, String type)
    throws SQLException {
        sql_type = type;
        num = stream.readInt();
        calle = stream.readString();
        municipio = stream.readString();
        provincia = stream.readString();
        zip = stream.readString();
    }

    public void writeSQL(SQLOutput stream) throws SQLException {
        stream.writeInt(num);
        stream.writeString(calle);
        stream.writeString(municipio);
        stream.writeString(provincia);
        stream.writeString(zip);
    }
}
```

Usando el Mapa de una Conexión

Después de escribir la clase que implementa la interface `SQLData`, hay que configurar el mapeo personalizado. Esto significa indicar el nombre SQL completamente cualificado del tipo DIRECCION y la clase que representa los objetos de este tipo.

```
java.util.Map map = con.getTypeMap();
map.put("esquema.direccion", Class.forName("Direccion"));
con.setTypeMap(map);
```

Puedes llamar al método `getObject()` para recuperar una instancia del tipo Direccion y el Driver hará automáticamente todo lo necesario para crear una instancia en el programa.

De manera similar puedes llamar a `setObject()` para almacenarla. Puedes observar la diferencia entre trabajar con el mapeo estandar a un objeto Struct y usar el mapeo personalizado comparando el siguiente

trozo de código que mapea usando el método predefinido con otro tipo distinto.

```
ResultSet rs = s.executeQuery("select lugar where sucursal = 13");
rs.next();
Struct direccion = (Struct)rs.getObject("lugar");
```

La variable direccion contiene los valores : 4344, "Calle Principal", "Torrent", "46", "46015" que se corresponde con el tipo direccion de SQL. Mientras que si está definido el mapeo, las cosas se harían así:

```
ResultSet rs = s.executeQuery("select lugar where sucursal = 13");
rs.next();
Address sucur13 = (Direccion)rs.getObject("lugar");
```

Para que veas las diferencias de usar o no el mapa, imagina que la sucursal13 se ha movido de lugar al municipio vecino de Aldaia y hay que actualizar su lugar en la BD. Usando el mapa:

```
ResultSet rs = s.executeQuery("select lugar where sucursal = 13");
rs.next();
Address sucur13 = (Direccion)rs.getObject("lugar");
sucur13.num = 18;
sucur13.calle = "Nueva";
sucur13.municipio = "Aldaia";
sucur13.zip = "46123";
PreparedStatement ps = con.prepareStatement(
              "update suscursales set lugar = ? where num = 13");
ps.setObject(1, sucur13);
ps.executeUpdate();
```

Y si no usamos el mapa:

```
PreparedStatement ps = con.prepareStatement(
    "update sucursales " +
    "set lugar.num = 18, lugar.calle= 'Nueva', " +
    "lugar.municipio = 'Aldaia', lugar.zip = '46123' " +
    "where num = 13");
ps.executeUpdate();
```

USANDO OBJETOS DATALINK

Un valor de tipo DATALINK es una referencia a un recurso externo al SGBD. En Java se mapea a una URL.

Alamacenar una Referencia a un Dato Externo

Usa el método `setURL()` de `PreparedStatement` para indicar un objeto `java.net.URL`. Por ejemplo vamos a añadir a una tabla que tiene una columna el lugar donde se almacena un documento.

```
create table docs(
docNombre varchar(50),
docURL varchar(200)
);
```

```
public void insertaFila(String descripcion, String url)
throws SQLException {
    String q = "insert into docs(docNombre, docurl) values(?,?)";
    try (PreparedStatement ps = con.prepareStatement(q)) {
        ps.setString(1, descripcion);
        ps.setURL(2, new URL(url));
        ps.execute();
  } catch (SQLException sqlex) {
      sqlex.printStackTrace();
  } catch (Exception ex) {
      System.out.println("Error inesperado");
      ex.printStackTrace();
  }
}
```

Recuperar Referencia a Dato Externo

Usamos el método `ResultSet.getURL` para recuperar la referencia al dato externo como un objeto `java.net.URL`. En los casos en que el driver del SGBD no soporte el tipo URL, usa un varchar y lo intercambias como un string y usas Java para convertirlo a string y url.

```
public static void verTabla(Connection c, Proxy p)
throws SQLException, IOException {
  String q = "select docNombre, docURL from docs";
```

```
    try (Statement s = c.createStatement()) {
      ResultSet rs = s.executeQuery(q);
      if ( rs.next() )  {
        String dn = null;
        java.net.URL url = null;
        dn = rs.getString(1);
        url = rs.getURL(2);
        if (url != null) {
          URLConnection uc = url.openConnection(p);
          BufferedReader br = new BufferedReader(
                              new InputStreamReader(uc.getInputStream()));
          System.out.println("Documento: " + dn);
          String pageContent = null;
          while ((contenido = br.readLine()) != null ) {
              System.out.println(contenido);
          }
        } else {
          System.out.println("URL es null");
        }
      }
    } catch (SQLException e) {
        e.printStackTrace();
    } catch(IOException ioEx) {
        System.out.println("IOException: " + ioEx.toString());
    } catch (Exception ex) {
        System.out.println("Error inesperado");
        ex.printStackTrace();
    }
  }
```

El método `URLConnection.openConnection()` puede no tener argumentos si usas una conexión directa a Intenet, pero si necesitas utilizar un proxy puedes usar el objeto `java.net.Proxy` como argumento. La siguiente sentencia muestra como usar un proxy HTTP con el nombre de servidor www-proxy.ejemplo.com y puerto 80:

```
Proxy p;
InetSocketAddress ps;
ps = new InetSocketAddress("www-proxy.example.com", 80);
p = new Proxy(Proxy.Type.HTTP, ps);
```

10.4.6. TRABAJAR CON TRANSACCIONES.

Las transacciones son un potente mecanismo de protección que ofrecen los SGBD relacionales a sus usuarios. Ya sabes que la ejecución de una sentencia SQL es atómica (o toda o nada). Básicamente una transacción consiste en extender esta atomicidad a un grupo de varias sentencias SQL. Esto permite que si cualquier fallo impide a ese grupo de sentencias ejecutarse de forma completa, todos los cambios que haya realizado cualquiera de ellas pueden deshacerse, dejando a los datos en un estado consistente.

Un programa con el driver JDBC es un usuario más del SGBD al que ataca. Por tanto, si este SGBD usa transacciones, puede aprovecharse de esta característica.

EJEMPLO 25: *Averiguar si el SGBD soporta transacciones:*

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.DatabaseMetaData;
import jcb.util.DatabaseUtil;
import jcb.db.VeryBasicConnectionManager;

public class TestSoporteTransacciones {
    public static boolean soportaTransacciones(Connection co)
    throws SQLException {
        if(co == null) return false;
        DatabaseMetaData dbMD = co.getMetaData();
        if( dbMD == null ) return false; // no soporta metadata
        return dbMD.supportsTransactions();
    }

    public static void main(String[] args) {
        Connection co = null;
        try {
            String dbVendor = args[0];
            co=VeryBasicConnectionManager.getConnection(dbVendor);
            System.out.println("--- Comienza el Test ---");
            System.out.println("dbVendor= " + dbVendor);
```

```
                System.out.println("conexión=" + co);
                System.out.println("Sporta Transacciones:" +
                                   soportaTransacciones(co) );
                System.out.println("--- Final del Test ---");
            }
            catch(Exception e){
                e.printStackTrace();
                System.exit(1);
            }
            finally {
                DatabaseUtil.close(conn);
            }
        }
    }
```

La interface **Connection** tiene varios métodos que controlan la confirmación (commit) o deshacer (rollback) los cambios en los datos de una base de datos. Estos métodos son los siguientes:

- **void commit()**: Hace permanentes todos los cambios realizados desde el anterior commit/rollback y libera cualesquiera bloqueos realizados en la conexión.
- **boolean getAutoCommit()**: Recupera el estado del modo autocommit, true indica que está activo.
- **boolean isReadOnly()**: Recupera el estado de una conexión en modo aislamiento de solo lectura.
- **void setReadOnly(boolean ro)**: establece el modo de la conexión en solo lectura, lo que permite al driver activar ciertas optimizaciones en la BD.
- **void rollback()**: Deshace todos los cambios realizados en la transacción actual y libera los bloqueos aplicados en la BD.
- **void rollback(Savepoint s)**: Deshace todos los cambios realizados tras el Savepoint indicado.
- **void setAutoCommit(boolean ac)**: Fija el modo autocommit.
- **Savepoint setSavepoint()**: Crea un savepoint sin nombre en la

transacción actual y lo devuelve.

- **Savepoint setSavepoint(String n)**: Crea y devuelve un savepoint con el nombre indicado.
- **void setTransactionIsolation(int nivel)**: Intenta cambiar el nivel de aislamiento de la transacción actual para el objeto conexión.

Por defecto, una conexión a base de datos con el driver, comitea todos los cambios a los datos de forma automática después de ejecutar cada sentencia SQL que hace cambios, por ejemplo un UPDATE. Si quieres utilizar transacciones debes desactivar el autocommit.

EJEMPLO 26: autocommit y commits explícitos.

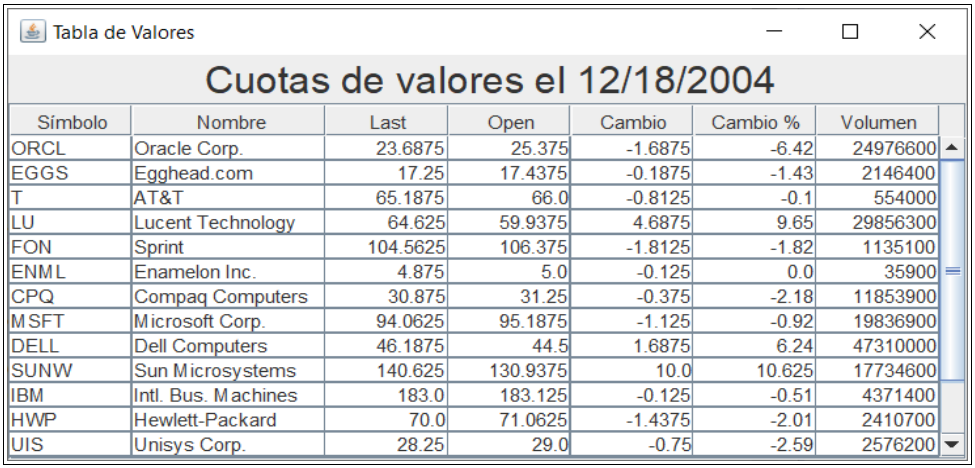
```
import java.sql.Connection;
import java.sql.SQLException;
import jcb.util.DatabaseUtil;
...
Connection co = null;
try {
    co = <...Crear una conexión a una BD...>;
    co.setAutoCommit(false);    // desactivar el autocommit
    // ejecutar cualquier número de sentencias SQL...
    co.commit();  // commit a los cambios
}
catch( SQLException e ) {
    co.rollback(); // deshacer cambios en la transacción
}
finally {
    DatabaseUtil.close(co);
}
```

10.4.7. MOSTRAR DATOS EN UN JTABLE.

Las tablas se usan frecuentemente para mostrar al usuario los datos almacenados en una BD y manipularlos: crear nuevos datos, borrar los que no se necesitan, modificar los existentes, ordenarlos, buscar los

que interesan o navegar por ellos.

EJEMPLO 27. Vamos a comenzar con una versión básica de una tabla que muestre información sobre la bolsa. Lo que debe mostrar es:



Símbolo	Nombre	Last	Open	Cambio	Cambio %	Volumen
ORCL	Oracle Corp.	23.6875	25.375	-1.6875	-6.42	24976600
EGGS	Egghead.com	17.25	17.4375	-0.1875	-1.43	2146400
T	AT&T	65.1875	66.0	-0.8125	-0.1	554000
LU	Lucent Technology	64.625	59.9375	4.6875	9.65	29856300
FON	Sprint	104.5625	106.375	-1.8125	-1.82	1135100
ENML	Enamelon Inc.	4.875	5.0	-0.125	0.0	35900
CPQ	Compaq Computers	30.875	31.25	-0.375	-2.18	11853900
MSFT	Microsoft Corp.	94.0625	95.1875	-1.125	-0.92	19836900
DELL	Dell Computers	46.1875	44.5	1.6875	6.24	47310000
SUNW	Sun Microsystems	140.625	130.9375	10.0	10.625	17734600
IBM	Intl. Bus. Machines	183.0	183.125	-0.125	-0.51	4371400
HWP	Hewlett-Packard	70.0	71.0625	-1.4375	-2.01	2410700
UIS	Unisys Corp.	28.25	29.0	-0.75	-2.59	2576200

- **Símbolo**: abreviatura de una empresa que cotiza en bolsa.
- **Nombre**: nombre de la empresa.
- **Last**: precio al comienzo de la sesión
- **Open**: Precio al final del día
- **Cambio**: Cambio absoluto de precio (Last – open)
- **Cambio %**: Porcentaje del cambio
- **Volumen**: volumen de valores (acciones) intercambiadas.

```
import java.awt.*;
import java.util.*;
import java.text.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.table.*;

public class TablaValores extends JFrame {
    private static final long serialVersionUID = 1L;
    protected JTable m_tabla;
    protected DatoTablaValores m_dato;
    protected JLabel m_titulo;
```

```
    public TablaValores() {
         super("Tabla de Valores");
         setSize(600, 300);
         UIManager.put( "Table.focusCellHighlightBorder",
                        new LineBorder(Color.black, 0) );
         m_dato = new DatoTablaValores();
         m_titulo = new JLabel( m_dato.getTitulo(), new ImageIcon("money.gif"),
                                SwingConstants.CENTER );
         m_titulo.setFont(new Font("Helvetica",Font.PLAIN,24));
         getContentPane().add(m_titulo, BorderLayout.NORTH);
         m_tabla = new JTable();
         m_tabla.setAutoCreateColumnsFromModel(false);
         m_tabla.setModel(m_dato);
         for (int k = 0; k < m_dato.getColumnCount(); k++) {
             DefaultTableCellRenderer renderer = new DefaultTableCellRenderer();
            renderer.setHorizontalAlignment(
                                    DatoTablaValores.m_columns[k].m_alineacion );
             TableColumn column = new TableColumn(k,
                             DatoTablaValores.m_columns[k].m_ancho, renderer, null);
                                 m_tabla.addColumn(column);
         }
         JTableHeader header = m_tabla.getTableHeader();
         header.setUpdateTableInRealTime(false);
         JScrollPane ps = new JScrollPane();  // ***** tabla sobre el ScrollPane!!
         ps.getViewport().setBackground(m_tabla.getBackground());
         ps.getViewport().add(m_tabla);
         getContentPane().add(ps, BorderLayout.CENTER);
        }

         public static void main(String argv[]) {
             TablaValores frame = new TablaValores();
             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
             frame.setVisible(true);
         }
    }

class DatoValor {
         public String m_abrevia;
         public String m_nombre;
         public Double m_last;
         public Double m_open;
         public Double m_cambio;
         public Double m_cambioPct;
         public Long m_volumen;

    public DatoValor(String simbolo, String nombre, double last, double open,
                             double cambio, double cambopPct, long volumen) {
                 m_abrevia = simbolo;
                 m_nombre = nombre;
                 m_last = last;
                 m_open = open;
                 m_cambio = cambio;
                 m_cambioPct = cambopPct;
                 m_volumen = volumen;
    }
```

```
}

class DatoColumna {
         public String m_titulo;
         public int m_ancho;
         public int m_alineacion;
         public DatoColumna(String titulo, int ancho, int alineacion) {
             m_titulo = titulo;
             m_ancho = ancho;
             m_alineacion = alineacion;
        }
}

class DatoTablaValores extends AbstractTableModel {
    static final public DatoColumna m_columns[] = {
                 new DatoColumna( "Símbolo", 100, JLabel.LEFT ),
                 new DatoColumna( "Nombre", 160, JLabel.LEFT ),
                 new DatoColumna( "Last", 100, JLabel.RIGHT ),
                 new DatoColumna( "Open", 100, JLabel.RIGHT ),
                 new DatoColumna( "Cambio", 100, JLabel.RIGHT ),
                 new DatoColumna( "Cambio %", 100, JLabel.RIGHT ),
                 new DatoColumna( "Volumen", 100, JLabel.RIGHT )
             };
         protected SimpleDateFormat m_frm;
         protected Vector m_vector;
         protected Date m_fecha;

         public DatoTablaValores() {
                 m_frm = new SimpleDateFormat("MM/dd/yyyy");
                 m_vector = new Vector();
                 setDatosPorDefecto();
         }

         public void setDatosPorDefecto() {
             try {
                 m_fecha = m_frm.parse("12/18/2004");
         }
         catch (java.text.ParseException ex) {
                 m_fecha = null;
         }
         m_vector.removeAllElements();
         m_vector.addElement(new DatoValor("ORCL", "Oracle Corp.", 23.6875, 25.375, -1.6875,
-6.42, 24976600));
         m_vector.addElement(new DatoValor("EGGS", "Egghead.com", 17.25, 17.4375, -0.1875,
-1.43, 2146400));
         m_vector.addElement(new DatoValor("T", "AT&T", 65.1875, 66, -0.8125, -0.10, 554000));
         m_vector.addElement(new DatoValor("LU", "Lucent Technology", 64.625, 59.9375, 4.6875,
9.65, 29856300));
         m_vector.addElement(new DatoValor("FON", "Sprint", 104.5625, 106.375, -1.8125, -1.82,
1135100));
         m_vector.addElement(new DatoValor("ENML", "Enamelon Inc.", 4.875, 5, -0.125, 0,
35900));
         m_vector.addElement(new DatoValor("CPQ", "Compaq Computers", 30.875, 31.25, -0.375,
-2.18, 11853900));
         m_vector.addElement(new DatoValor("MSFT", "Microsoft Corp.", 94.0625, 95.1875,
-1.125, -0.92, 19836900));
```

```
        m_vector.addElement(new DatoValor("DELL", "Dell Computers", 46.1875, 44.5, 1.6875,
6.24, 47310000));
        m_vector.addElement(new DatoValor("SUNW", "Sun Microsystems", 140.625, 130.9375, 10,
10.625, 17734600));
        m_vector.addElement(new DatoValor("IBM", "Intl. Bus. Machines", 183, 183.125, -0.125,
-0.51, 4371400));
        m_vector.addElement(new DatoValor("HWP", "Hewlett-Packard", 70, 71.0625, -1.4375,
-2.01, 2410700));
        m_vector.addElement(new DatoValor("UIS", "Unisys Corp.", 28.25, 29, -0.75, -2.59,
2576200));
        m_vector.addElement(new DatoValor("SNE", "Sony Corp.", 96.1875, 95.625, 1.125, 1.18,
330600));
        m_vector.addElement(new DatoValor("NOVL", "Novell Inc.", 24.0625, 24.375, -0.3125,
-3.02, 6047900));
        m_vector.addElement(new DatoValor("HIT", "Hitachi, Ltd.", 78.5, 77.625, 0.875, 1.12,
49400));
        }

        public int getRowCount() { return m_vector==null ? 0 : m_vector.size(); }

        public int getColumnCount() { return m_columns.length; }

        public String getColumnName(int column) { return
                                                m_columns[column].m_titulo; }

        public boolean isCellEditable(int nRow, int nCol) { return false; }

        public Object getValueAt(int nRow, int nCol) {
                if (nRow < 0 || nRow>=getRowCount()) return "";
                DatoValor row = (DatoValor)m_vector.elementAt(nRow);
                switch (nCol) {
                        case 0: return row.m_abrevia;
                        case 1: return row.m_nombre;
                        case 2: return row.m_last;
                        case 3: return row.m_open;
                        case 4: return row.m_cambio;
                        case 5: return row.m_cambioPct;
                        case 6: return row.m_volumen;
                }
                return "";
        }

        public String getTitulo() {
                if (m_fecha==null) return "Cuotas de Valores";
                return "Cuotas de valores el " + m_frm.format(m_fecha);
        }
}
```

La clase **TablaValores** extiende a **JFrame** para mostrar de forma gráfica una serie de datos usando la tabla **JTable** m_tabla, que utiliza el modelo **DatoTablaValores** m_dato y un título (JLabel). Al crear la tabla ponemos **autoCreateColumnsFromModel** a false porque vamos a

crear las columnas a mano.

Luego, creamos una instancia de **JTableHeader** y la propiedad **updateTableInRealTime** se deja a false para que al arrastrar una sola columna, se visualice mientras se arrastra. Por último, comentar que la tabla se añade al **JViewport** de un **JScrollPane** para que puedan visualizarse las filas y columnas ocultas.

EJEMPLO 28: Ahora, extendemos **TablaValores** para usar colores y pequeños iconos al dibujar las celdas de la tabla:

- Los valores positivos del cambio absoluto y porcentaje se dibujan en verde si son positivos y en rojo si son negativos.
- Añadimos un icono a cada símbolo: flecha hacia arriba si el cambio es positivo, hacia abajo si es negativa o nada si ni sube ni baja.

Para hacer esto, debemos construir nuestro propio **CellRenderer**.

Símbolo	Nombre	Last	Open	Cambio	Cambio %	Volumen
ORCL	Oracle Corp.	23.6875	25.375	-1.6875	-6.42	24976600
EGGS	Egghead.com	17.25	17.4375	-0.1875	-1.43	2146400
T	AT&T	65.1875	66.0	-0.8125	-0.1	554000
LU	Lucent Technology	64.625	59.9375	4.6875	9.65	29856300
FON	Sprint	104.5625	106.375	-1.8125	-1.82	1135100
ENML	Enamelon Inc.	4.875	5.0	-0.125	0.0	35900
CPQ	Compaq Computers	30.875	31.25	-0.375	-2.18	11853900
MSFT	Microsoft Corp.	94.0625	95.1875	-1.125	-0.92	19836900
DELL	Dell Computers	46.1875	44.5	1.6875	6.24	47310000
SUNW	Sun Microsystems	140.625	130.9375	10.0	10.625	17734600

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
```

```
import java.text.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class TablaValores1 extends JFrame {
    protected JTable m_tabla;
    protected DatoTablaValores m_dato;
    protected JLabel m_titulo;

    public TablaValores1() {
        // Código del ejemplo anterior
        super("Tabla de Valores");
        setSize(600, 300);
        UIManager.put( "Table.focusCellHighlightBorder",
                       new LineBorder(Color.black, 0) );
        m_dato = new DatoTablaValores();
        m_titulo = new JLabel( m_dato.getTitulo(),
                      new ImageIcon("imagen\\money.gif"), SwingConstants.CENTER );
        m_titulo.setFont(new Font("Helvetica",Font.PLAIN,24));
        getContentPane().add(m_titulo, BorderLayout.NORTH);
        m_tabla = new JTable();
        m_tabla.setAutoCreateColumnsFromModel(false);
        m_tabla.setModel(m_dato);
        // Nuevo código
        for (int k = 0; k < m_dato.getColumnCount(); k++) {
            DefaultTableCellRenderer renderer = new  ColoredTableCellRenderer();
            renderer.setHorizontalAlignment(
                                DatoTablaValores.m_columnas[k].m_alineacion );
            TableColumn columna = new TableColumn(k,
                        DatoTablaValores.m_columnas[k].m_ancho, renderer, null);
            columna.setHeaderRenderer( createDefaultRenderer() );
            m_tabla.addColumn(columna);
        }
        // Código del ejemplo anterior
        JScrollPane ps = new JScrollPane();
        ps.getViewport().setBackground(m_tabla.getBackground());
        ps.getViewport().add(m_tabla);
        getContentPane().add(ps, BorderLayout.CENTER);
    }

    public static void main(String argv[]) {
        TablaValores1 frame = new TablaValores1();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    class ColoredTableCellRenderer extends DefaultTableCellRenderer {
```

```
        public void setValue(Object value) {
           if (value instanceof DatoColor) {
               DatoColor cvalue = (DatoColor)value;
               setForeground(cvalue.m_color);
               setText(cvalue.m_data.toString());
           }
           else if (value instanceof DatoIcono) {
               DatoIcono ivalue = (DatoIcono)value;
               setIcon(ivalue.m_icon);
               setText(ivalue.m_dato.toString());
           }
           else super.setValue(value);
        }
    }

    protected TableCellRenderer createDefaultRenderer() {
        DefaultTableCellRenderer label = new DefaultTableCellRenderer() {
           public Component getTableCellRendererComponent(
                JTable table, Object value, boolean isSelected,  boolean hasFocus,
                int row, int column) {
               if (table != null) {
                   JTableHeader header = table.getTableHeader();
                   if (header != null) {
                      setForeground(header.getForeground());
                      setBackground(header.getBackground());
                      setFont(header.getFont());
                   }
               }
               setText((value == null) ? "" : value.toString());
               setBorder(UIManager.getBorder("TableHeader.cellBorder"));
               return this;
            }
        };
        label.setHorizontalAlignment(JLabel.CENTER);
        return label;
    }
}

class DatoColor {
    public Color m_color;
    public Object m_data;
    public static Color GREEN = new Color(0, 128, 0);
    public static Color RED = Color.red;

    public DatoColor(Color color, Object data) {
        m_color = color;
        m_data = data;
    }
```

```
    public DatoColor(Double data) {
        m_color = data.doubleValue() >= 0 ? GREEN : RED;
        m_data = data;
    }

    public String toString() { return m_data.toString(); }
}

class DatoIcono {
    public ImageIcon m_icon;
    public Object m_dato;

    public DatoIcono(ImageIcon icon, Object data) {
        m_icon = icon;
        m_dato = data;
    }

    public String toString() { return m_dato.toString(); }
}

class DatoValor {
    //Código del ejemplo anterior
    public DatoIcono m_abrevia;
    public String m_nombre;
    public Double m_last;
    public Double m_open;
    public DatoColor m_cambio;
    public DatoColor m_cambioPct;
    public Long m_volumen;
    static private ImageIcon ICON_UP = new ImageIcon("imagen\\uparrow.gif");
    static private ImageIcon ICON_DOWN = new ImageIcon("imagen\\downarrow.gif");
    static private ImageIcon ICON_BLANK = new ImageIcon("imagen\\blank.gif");

    // Nuevo código
    public DatoValor(String symbol, String name, double last,
        double open, double change, double changePr, long volume) {
            m_abrevia = new DatoIcono( getIcon(change), symbol);
            m_nombre = name;
            m_last = last;
            m_open = open;
            m_cambio = new DatoColor( change );
            m_cambioPct = new DatoColor( changePr );
            m_volumen = volume;
        }

        public ImageIcon getIcon(double cambio) {
            return (cambio > 0 ? ICON_UP : (cambio < 0 ? ICON_DOWN:ICON_BLANK));
        }
    }
```

```
    class DatoColumna {
        public String m_titulo;
        public int m_ancho;
        public int m_alineacion;

        public DatoColumna(String titulo, int ancho, int alineacion) {
                m_titulo = titulo;
                m_ancho = ancho;
                m_alineacion = alineacion;
        }
    }

    class DatoTablaValores extends AbstractTableModel {
        // Código anterior
        static final public DatoColumna m_columnas[] = {
            new DatoColumna( "Símbolo", 100, JLabel.LEFT ),
            new DatoColumna( "Nombre", 160, JLabel.LEFT ),
            new DatoColumna( "Last", 100, JLabel.RIGHT ),
            new DatoColumna( "Open", 100, JLabel.RIGHT ),
            new DatoColumna( "Cambio", 100, JLabel.RIGHT ),
            new DatoColumna( "Cambio %", 100, JLabel.RIGHT ),
            new DatoColumna( "Volumen", 100, JLabel.RIGHT )
        };
        protected SimpleDateFormat m_frm;
        protected NumberFormat m_volumeFormat;
        protected Vector m_vector;
        protected Date m_fecha;

        public DatoTablaValores() {
            m_frm = new SimpleDateFormat("MM/dd/yyyy");
            m_volumeFormat = NumberFormat.getInstance();
            m_volumeFormat.setGroupingUsed(true);
            m_volumeFormat.setMaximumFractionDigits(0);
            m_vector = new Vector();
            setDatosPorDefecto();
        }

        public void setDatosPorDefecto() {
            try {
                m_fecha = m_frm.parse("12/18/2004");
            }
            catch (java.text.ParseException ex) {
                m_fecha = null;
            }
            m_vector.removeAllElements();
            m_vector.addElement(new DatoValor("ORCL", "Oracle Corp.", 23.6875,
25.375, -1.6875, -6.42, 24976600));
            m_vector.addElement(new DatoValor("EGGS", "Egghead.com", 17.25,
```

```
17.4375, -0.1875, -1.43, 2146400));
            m_vector.addElement(new DatoValor("T", "AT&T", 65.1875, 66, -0.8125,
-0.10, 554000));
            m_vector.addElement(new DatoValor("LU", "Lucent Technology", 64.625,
59.9375, 4.6875, 9.65, 29856300));
            m_vector.addElement(new DatoValor("FON", "Sprint", 104.5625, 106.375,
-1.8125, -1.82, 1135100));
            m_vector.addElement(new DatoValor("ENML", "Enamelon Inc.", 4.875, 5,
-0.125, 0, 35900));
            m_vector.addElement(new DatoValor("CPQ", "Compaq Computers", 30.875,
31.25, -0.375, -2.18, 11853900));
            m_vector.addElement(new DatoValor("MSFT", "Microsoft Corp.", 94.0625,
95.1875, -1.125, -0.92, 19836900));
            m_vector.addElement(new DatoValor("DELL", "Dell Computers", 46.1875,
44.5, 1.6875, 6.24, 47310000));
            m_vector.addElement(new DatoValor("SUNW", "Sun Microsystems", 140.625,
130.9375, 10, 10.625, 17734600));
            m_vector.addElement(new DatoValor("IBM", "Intl. Bus. Machines", 183,
183.125, -0.125, -0.51, 4371400));
            m_vector.addElement(new DatoValor("HWP", "Hewlett-Packard", 70,
71.0625, -1.4375, -2.01, 2410700));
            m_vector.addElement(new DatoValor("UIS", "Unisys Corp.", 28.25, 29,
-0.75, -2.59, 2576200));
            m_vector.addElement(new DatoValor("SNE", "Sony Corp.", 96.1875,
95.625, 1.125, 1.18, 330600));
            m_vector.addElement(new DatoValor("NOVL", "Novell Inc.", 24.0625,
24.375, -0.3125, -3.02, 6047900));
                m_vector.addElement(new DatoValor("HIT", "Hitachi, Ltd.", 78.5,
77.625, 0.875, 1.12, 49400));
        }

        // Similar al anterior
        public int getRowCount() { return m_vector==null ? 0 : m_vector.size(); }

        public int getColumnCount() { return m_columnas.length; }

        public String getColumnName(int column) {
             return m_columnas[column].m_titulo;
        }

        public boolean isCellEditable(int nRow, int nCol) { return false; }

        public Object getValueAt(int nRow, int nCol) {
                if (nRow < 0 || nRow >= getRowCount()) return "";
                DatoValor row = (DatoValor)m_vector.elementAt(nRow);
                switch (nCol) {
                case 0: return row.m_abrevia;
                case 1: return row.m_nombre;
                case 2: return row.m_last;
```

```
                case 3: return row.m_open;
                case 4: return row.m_cambio;
                case 5: return row.m_cambioPct;
                case 6: return row.m_volumen;
                }
                return "";
        }

        // Unchanged code from example 18.1
        public String getTitulo() {
                if (m_fecha==null) return "Cuotas de Valores";
                return "Cuotas de valores el " + m_frm.format(m_fecha);
        }
}
```

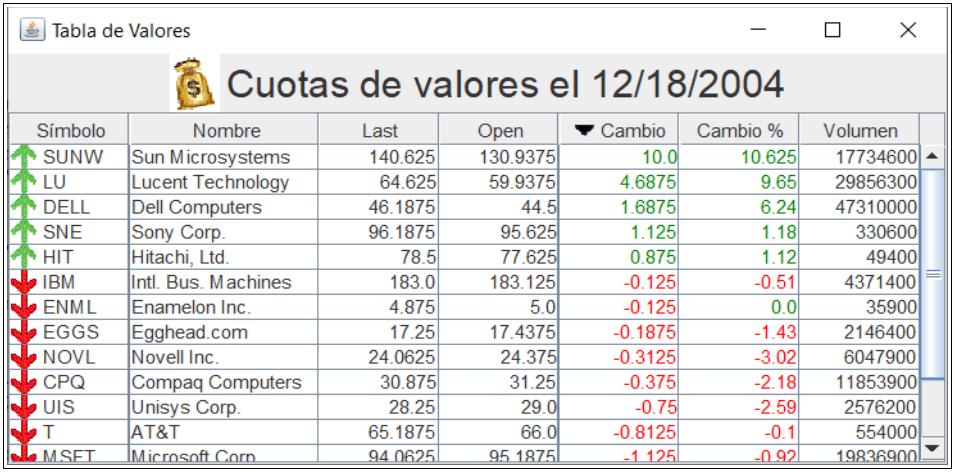
La clase `ColoredTableCellRendered` extiende a `DefaultTableCellRenderer` y sobreescribe el método **setValue()** que se llama antes de dibujar la celda para recuperar los datos como un Object. Esto nos permite diferenciar entre datos de tipo **DatoColor** y **DatoIcono** (se asigna el icono con setIcon()) y un dato normal.

EJEMPLO 29: **Ordenar columnas**. Ahora añadimos la funcionalidad de poder ordenar las filas por cualquier columna tanto de forma ascendente como ascendente. Queremos que sew comporte así:

- Un clic en la cabecera de una columna reordena las filas según esa columna.
- Si vuelves a hacer clic en la misma columna, se cambia la dirección (de menor a mayor y viceversa).
- La cabecera de la columna usada para ordenar tendrá una marca que lo indique (un icono con la dirección del orden).

Para conseguirlo debemos añadir un mouse listener a la cabecera de la tabla. Cuando escuche el evento de pulsar dispara el código que se encarga de ordenar, usando la API de las colecciones, puesto que las filas están almacenadas en un vector (la clase `java.util.Collections` tiene métodos estáticos para este fin) y creamos un objeto que

implementa la interface **Comparator**.



Símbolo	Nombre	Last	Open	Cambio	Cambio %	Volumen
SUNW	Sun Microsystems	140.625	130.9375	10.0	10.625	17734600
LU	Lucent Technology	64.625	59.9375	4.6875	9.65	29856300
DELL	Dell Computers	46.1875	44.5	1.6875	6.24	47310000
SNE	Sony Corp.	96.1875	95.625	1.125	1.18	330600
HIT	Hitachi, Ltd.	78.5	77.625	0.875	1.12	49400
IBM	Intl. Bus. Machines	183.0	183.125	-0.125	-0.51	4371400
ENML	Enamelon Inc.	4.875	5.0	-0.125	0.0	35900
EGGS	Egghead.com	17.25	17.4375	-0.1875	-1.43	2146400
NOVL	Novell Inc.	24.0625	24.375	-0.3125	-3.02	6047900
CPQ	Compaq Computers	30.875	31.25	-0.375	-2.18	11853900
UIS	Unisys Corp.	28.25	29.0	-0.75	-2.59	2576200
T	AT&T	65.1875	66.0	-0.8125	-0.1	554000
MSFT	Microsoft Corp	94.0625	95.1875	-1.125	-0.92	19836900

Que obliga a implementar dos métodos:

- **int compare(Object o1, Object o2)**: devuelve 0 si son iguales, negativo si o1 es menor que o2 y positivo en caso contrario.
- **boolean equals(Object obj)**: true si iguales, false en otro caso.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class TablaValores2 extends JFrame {
    protected JTable m_tabla;
    protected DatoTablaValores m_dato;
    protected JLabel m_titulo;

    public TablaValores2() {
        super("Tabla de Valores");
        setSize(600, 300);
        UIManager.put( "Table.focusCellHighlightBorder",
```

```
                    new LineBorder(Color.black, 0) );
        m_dato = new DatoTablaValores();
        m_titulo = new JLabel( m_dato.getTitulo(),
                    new ImageIcon("imagen\\money.gif"), SwingConstants.CENTER);
        m_titulo.setFont(new Font("Helvetica",Font.PLAIN,24));
        getContentPane().add(m_titulo, BorderLayout.NORTH);
        m_tabla = new JTable();
        m_tabla.setAutoCreateColumnsFromModel(false);
        m_tabla.setModel(m_dato);
        for (int k = 0; k < m_dato.getColumnCount(); k++) {
            DefaultTableCellRenderer renderer = new  ColoredTableCellRenderer();
            renderer.setHorizontalAlignment(
                                DatoTablaValores.m_columnas[k].m_alineacion );
            TableColumn columna = new TableColumn(k,
                        DatoTablaValores.m_columnas[k].m_ancho, renderer, null);
            columna.setHeaderRenderer( createDefaultRenderer() );
            m_tabla.addColumn(columna);
        }
        JTableHeader header = m_tabla.getTableHeader();
        header.setUpdateTableInRealTime(true);
        header.addMouseListener( new ColumnaListener() );
        header.setReorderingAllowed(true);
        JScrollPane ps = new JScrollPane();
        ps.getViewport().setBackground(m_tabla.getBackground());
        ps.getViewport().add(m_tabla);
        getContentPane().add(ps, BorderLayout.CENTER);
    }

    protected TableCellRenderer createDefaultRenderer() {
        DefaultTableCellRenderer label = new DefaultTableCellRenderer() {
            public Component getTableCellRendererComponent(
                   JTable table, Object value, boolean isSelected,
                                boolean hasFocus, int row, int column) {
                    if (table != null) {
                        JTableHeader header = table.getTableHeader();
                        if (header != null) {
                           setForeground(header.getForeground());
                           setBackground(header.getBackground());
                           setFont(header.getFont());
                        }
                    }
                    setText((value == null) ? "" : value.toString());
                    setBorder( UIManager.getBorder("TableHeader.cellBorder") );
                    return this;
                }
            };
            label.setHorizontalAlignment(JLabel.CENTER);
            return label;
    }
```

```
    public static void main(String argv[]) {
        TablaValores2 frame = new TablaValores2();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

class ColumnaListener extends MouseAdapter {
        public void mouseClicked(MouseEvent e) {
        TableColumnModel colModel = m_tabla.getColumnModel();
        int columnModelIndex = colModel.getColumnIndexAtX(e.getX());
        int modelIndex = colModel.getColumn(columnModelIndex).getModelIndex();
        if (modelIndex < 0) return;
        if (m_dato.m_sortCol == modelIndex)
            m_dato.m_sortAsc = !m_dato.m_sortAsc;
        else
            m_dato.m_sortCol = modelIndex;
        for (int i=0; i < m_dato.getColumnCount(); i++) {
            TableColumn column = colModel.getColumn(i);
            int index = column.getModelIndex();
            JLabel renderer = (JLabel)column.getHeaderRenderer();
            renderer.setIcon(m_dato.getColumnIcon(index));
        }
        m_tabla.getTableHeader().repaint();
        m_dato.sortDatos();
        m_tabla.tableChanged(new TableModelEvent(m_dato));
        m_tabla.repaint();
    }
}  // Acaba clase ColumnaListener

}  // Acaba la clase TablaValores2

class ColoredTableCellRenderer extends DefaultTableCellRenderer {
    public void setValue(Object value) {
       if (value instanceof DatoColor) {
           DatoColor cvalue = (DatoColor)value;
           setForeground(cvalue.m_color);
           setText(cvalue.m_data.toString());
       }
       else if (value instanceof DatoIcono) {
           DatoIcono ivalue = (DatoIcono)value;
           setIcon(ivalue.m_icon);
           setText(ivalue.m_dato.toString());
       }
       else super.setValue(value);
    }

    protected TableCellRenderer createDefaultRenderer() {
       DefaultTableCellRenderer label = new DefaultTableCellRenderer() {
```

```
            public Component getTableCellRendererComponent(
            JTable table, Object value, boolean isSelected,  boolean hasFocus,
                                    int row, int column) {
                if (table != null) {
                    JTableHeader header = table.getTableHeader();
                    if (header != null) {
                       setForeground(header.getForeground());
                       setBackground(header.getBackground());
                       setFont(header.getFont());
                    }
                }
                setText((value == null) ? "" : value.toString());
                setBorder(UIManager.getBorder("TableHeader.cellBorder"));
                return this;
            }
        };
        label.setHorizontalAlignment(JLabel.CENTER);
        return label;
    }
}

class DatoColor {
        public Color m_color;
        public Object m_data;
        public static Color GREEN = new Color(0, 128, 0);
        public static Color RED = Color.red;

        public DatoColor(Color color, Object data) {
            m_color = color;
            m_data = data;
        }

        public DatoColor(Double data) {
            m_color = data.doubleValue() >= 0 ? GREEN : RED;
            m_data = data;
        }

        public String toString() { return m_data.toString(); }
}

class DatoIcono {
        public ImageIcon m_icon;
        public Object m_dato;

        public DatoIcono(ImageIcon icon, Object data) {
            m_icon = icon;
            m_dato = data;
        }
```

```
        public String toString() { return m_dato.toString(); }
}

class DatoValor {
    public DatoIcono m_abrevia;
    public String m_nombre;
    public Double m_last;
    public Double m_open;
    public DatoColor m_cambio;
    public DatoColor m_cambioPct;
    public Long m_volumen;
    static private ImageIcon ICON_UP = new ImageIcon("imagen\\uparrow.gif");
    static private ImageIcon ICON_DOWN = new ImageIcon("imagen\\downarrow.gif");
    static private ImageIcon ICON_BLANK = new ImageIcon("imagen\\blank.gif");

    public DatoValor(String symbol, String name, double last,
        double open, double change, double changePr, long volume) {
        m_abrevia = new DatoIcono( getIcon(change), symbol);
        m_nombre = name;
        m_last = last;
        m_open = open;
        m_cambio = new DatoColor( change );
        m_cambioPct = new DatoColor( changePr );
        m_volumen = volume;
    }

    public ImageIcon getIcon(double cambio) {
        return (cambio > 0 ? ICON_UP : (cambio < 0 ? ICON_DOWN:ICON_BLANK));
    }
}

class DatoColumna {
        public String m_titulo;
        public int m_ancho;
        public int m_alineacion;

        public DatoColumna(String titulo, int ancho, int alineacion) {
                m_titulo = titulo;
                m_ancho = ancho;
                m_alineacion = alineacion;
        }
}

class DatoTablaValores extends AbstractTableModel {
    static final public DatoColumna m_columnas[] = {
                new DatoColumna( "Símbolo", 100, JLabel.LEFT ),
                new DatoColumna( "Nombre", 160, JLabel.LEFT ),
                new DatoColumna( "Last", 100, JLabel.RIGHT ),
                new DatoColumna( "Open", 100, JLabel.RIGHT ),
```

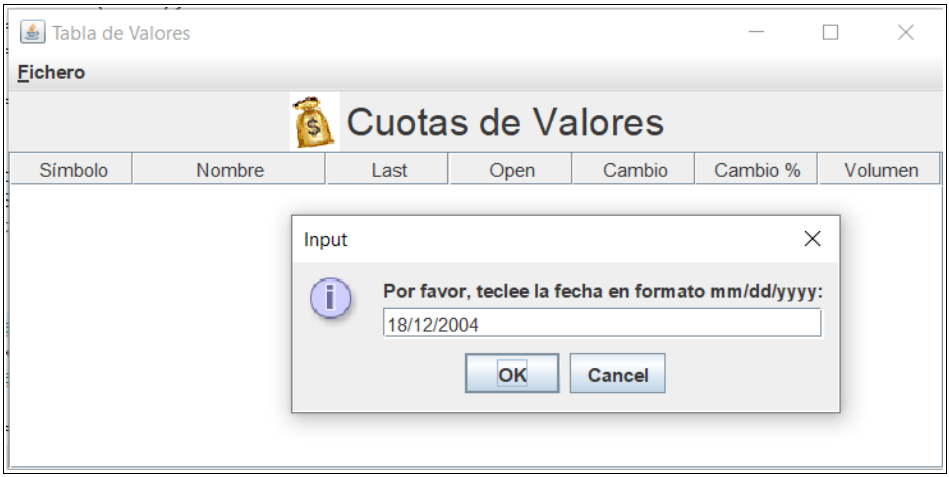
```
                new DatoColumna( "Cambio", 100, JLabel.RIGHT ),
                new DatoColumna( "Cambio %", 100, JLabel.RIGHT ),
                new DatoColumna( "Volumen", 100, JLabel.RIGHT )
    };

    public static ImageIcon COLUMNA_UP = new ImageIcon("imagen\\sortup.gif");
    public static ImageIcon COLUMNA_DOWN = new ImageIcon("imagen\\sortdown.gif");
    protected SimpleDateFormat m_frm;
    protected NumberFormat m_volumeFormat;
    protected Vector m_vector;
    protected Date m_fecha;
    public int m_sortCol = 0;
    public boolean m_sortAsc = true;

    public DatoTablaValores() {
        m_frm = new SimpleDateFormat("MM/dd/yyyy");
        m_volumeFormat = NumberFormat.getInstance();
        m_volumeFormat.setGroupingUsed(true);
        m_volumeFormat.setMaximumFractionDigits(0);
        m_vector = new Vector();
        setDatosPorDefecto();
        sortDatos();
    }

    public void setDatosPorDefecto() {
        try {
                m_fecha = m_frm.parse("12/18/2004");
        }
        catch (java.text.ParseException ex) {
                m_fecha = null;
        }
        m_vector.removeAllElements();
        m_vector.addElement(new DatoValor("ORCL", "Oracle Corp.", 23.6875, 25.375, -1.6875,
-6.42, 24976600));
        m_vector.addElement(new DatoValor("EGGS", "Egghead.com", 17.25, 17.4375, -0.1875,
-1.43, 2146400));
        m_vector.addElement(new DatoValor("T", "AT&T", 65.1875, 66, -0.8125, -0.10, 554000));
        m_vector.addElement(new DatoValor("LU", "Lucent Technology", 64.625, 59.9375, 4.6875,
9.65, 29856300));
        m_vector.addElement(new DatoValor("FON", "Sprint", 104.5625, 106.375, -1.8125, -1.82,
1135100));
        m_vector.addElement(new DatoValor("ENML", "Enamelon Inc.", 4.875, 5, -0.125, 0,
35900));
        m_vector.addElement(new DatoValor("CPQ", "Compaq Computers", 30.875, 31.25, -0.375,
-2.18, 11853900));
        m_vector.addElement(new DatoValor("MSFT", "Microsoft Corp.", 94.0625, 95.1875,
-1.125, -0.92, 19836900));
        m_vector.addElement(new DatoValor("DELL", "Dell Computers", 46.1875, 44.5, 1.6875,
6.24, 47310000));
        m_vector.addElement(new DatoValor("SUNW", "Sun Microsystems", 140.625, 130.9375, 10,
10.625, 17734600));
```

```
        m_vector.addElement(new DatoValor("IBM", "Intl. Bus. Machines", 183, 183.125, -0.125,
-0.51, 4371400));
        m_vector.addElement(new DatoValor("HWP", "Hewlett-Packard", 70, 71.0625, -1.4375,
-2.01, 2410700));
        m_vector.addElement(new DatoValor("UIS", "Unisys Corp.", 28.25, 29, -0.75, -2.59,
2576200));
        m_vector.addElement(new DatoValor("SNE", "Sony Corp.", 96.1875, 95.625, 1.125, 1.18,
330600));
        m_vector.addElement(new DatoValor("NOVL", "Novell Inc.", 24.0625, 24.375, -0.3125,
-3.02, 6047900));
        m_vector.addElement(new DatoValor("HIT", "Hitachi, Ltd.", 78.5, 77.625, 0.875, 1.12,
49400));
    }

    public int getRowCount() { return m_vector==null ? 0 : m_vector.size(); }

    public int getColumnCount() { return m_columnas.length; }

    public String getColumnName(int column) {
        return m_columnas[column].m_titulo;
    }

    public boolean isCellEditable(int nRow, int nCol) { return false; }

    public Object getValueAt(int nRow, int nCol) {
        if (nRow < 0 || nRow >= getRowCount()) return "";
        DatoValor row = (DatoValor)m_vector.elementAt(nRow);
        switch (nCol) {
        case 0: return row.m_abrevia;
        case 1: return row.m_nombre;
        case 2: return row.m_last;
        case 3: return row.m_open;
        case 4: return row.m_cambio;
        case 5: return row.m_cambioPct;
        case 6: return row.m_volumen;
        }
        return "";
    }

    public String getTitulo() {
        if (m_fecha==null) return "Cuotas de Valores";
        return "Cuotas de valores el " + m_frm.format(m_fecha);
    }

    public Icon getColumnIcon(int column) {
        if (column==m_sortCol) return m_sortAsc ? COLUMNA_UP : COLUMNA_DOWN;
        return null;
    }

    public void sortDatos() {
```

```
        Collections.sort(m_vector, new StockComparator(m_sortCol, m_sortAsc) );
    }
}

class StockComparator implements Comparator {
    protected int m_sortCol;
    protected boolean m_sortAsc;

    public StockComparator(int sortCol, boolean sortAsc) {
        m_sortCol = sortCol;
        m_sortAsc = sortAsc;
    }

    public int compare(Object o1, Object o2) {
        if( !(o1 instanceof DatoValor) || !(o2 instanceof DatoValor)) return 0;
        DatoValor s1 = (DatoValor) o1;
        DatoValor s2 = (DatoValor) o2;
        int result = 0;
        double d1, d2;
        switch (m_sortCol) {
        case 0: // abreviatura
                String str1 = (String)s1.m_abrevia.m_dato;
                String str2 = (String)s2.m_abrevia.m_dato;
                result = str1.compareTo(str2);
                break;
        case 1:// nombre
               result = s1.m_nombre.compareTo(s2.m_nombre);
               break;
        case 2: // last
                d1 = s1.m_last.doubleValue();
                d2 = s2.m_last.doubleValue();
                result = d1 < d2 ? -1 : (d1 > d2 ? 1 : 0);
                break;
        case 3:// open
               d1 = s1.m_open.doubleValue();
               d2 = s2.m_open.doubleValue();
               result = d1 < d2 ? -1: (d1 > d2 ? 1 : 0);
               break;
        case 4:// cambio
               d1 = ((Double)s1.m_cambio.m_data).doubleValue();
               d2 = ((Double)s2.m_cambio.m_data).doubleValue();
               result = d1 < d2 ? -1 : (d1 > d2 ? 1 : 0);
               break;
        case 5:// % cambio
               d1 = ((Double)s1.m_cambioPct.m_data).doubleValue();
               d2 = ((Double)s2.m_cambioPct.m_data).doubleValue();
               result = d1 < d2 ? -1 : (d1 > d2 ? 1 : 0);
               break;
        case 6:// volumen
```

```
                    long l1 = s1.m_volumen.longValue();
                    long l2 = s2.m_volumen.longValue();
                    result = l1<l2 ? -1 : (l1>l2 ? 1 : 0);
                    break;
            }
            if (!m_sortAsc) result = -result;
            return result;
        }

        public boolean equals(Object obj) {
          if(obj instanceof StockComparator) {
             StockComparator compObj = (StockComparator)obj;
             return (compObj.m_sortCol==m_sortCol) && (compObj.m_sortAsc== m_sortAsc);
          }
          return false;
        }
}
```

EJEMPLO 30: Añadimos un `JMenuBar` y hacemos un método `creaMenuBar()` para añadir al frame un menú titulado Fichero que tenga las opciones "Recuperar Datos..." y "Salir" con un separador. A cada opción le añadimos ActionListeners anónimos, el primero llama al método recuperarDatos() y el segundo acaba el programa con una llamada a System.exit(0). El método muestra un diálogo `JOptionPane` para leer una fecha y se conecta mediante JDBC para recuperar datos de una BD.



Crear la BD: puedes guardar las sentencias en un script y ejecutarlo con el comando: `mysql -u usuario -p BD < script.sql` o primero te autenticas (`mysql -u usuario -p BD`) y luego ejecutas: `source [ruta\]script.sql;`

```
create database valores;
use valores;

create table simbolos(
simbolo varchar(10) primary key,
nombre varchar(30)
);

insert into simbolos values('ORCL', 'Oracle Corp.');
insert into simbolos values('EGGS', 'Egghead.com');
insert into simbolos values('T', 'AT&T');
insert into simbolos values('LU', 'Lucent Technology');
insert into simbolos values('FON', 'Sprint');
insert into simbolos values('ENML', 'Enamelon Inc.');
insert into simbolos values('CPQ', 'Compaq Computers');
insert into simbolos values('MSFT', 'Microsoft Corp.');
insert into simbolos values('DELL', 'Dell Computers');
insert into simbolos values('SUNW', 'Sun Microsystems');
insert into simbolos values('IBM', 'Intl. Bus. Machines');
insert into simbolos values('HWP', 'Hewlett-Packard');
insert into simbolos values('UIS', 'Unisys Corp.');
insert into simbolos values('SNE', 'Sony Corp.');
insert into simbolos values('NOVL', 'Novell Inc.');
insert into simbolos values('HIT', 'Hitachi, Ltd.');

create table datos(
simbolo varchar(10) not null,
fecha date not null,
last double,
open double,
cambio double,
cambioPct double,
volumen int unsigned,
primary key (simbolo, fecha),
foreign key(simbolo) references simbolos(simbolo)
);

insert into datos values('ORCL', '2004/12/18', 23.6875, 25.375, -1.6875, -6.42, 24976600);
insert into datos values('EGGS', '2004/12/18', 17.25, 17.4375, -0.1875, -1.43, 2146400);
insert into datos values('T', '2004/12/18', 65.1875, 66, -0.8125, -0.10, 554000);
insert into datos values('LU', '2004/12/18', 64.625, 59.9375, 4.6875, 9.65, 29856300);
insert into datos values('FON', '2004/12/18', 104.5625, 106.375, -1.8125, -1.82, 1135100);
insert into datos values('ENML', '2004/12/18', 4.875, 5, -0.125, 0, 35900);
insert into datos values('CPQ', '2004/12/18', 30.875, 31.25, -0.375, -2.18, 11853900);
insert into datos values('MSFT', '2004/12/18', 94.0625, 95.1875, -1.125, -0.92, 19836900);
insert into datos values('DELL', '2004/12/18', 46.1875, 44.5, 1.6875, 6.24, 47310000);
insert into datos values('SUNW', '2004/12/18', 140.625, 130.9375, 10, 10.625, 17734600);
```

```
insert into datos values('IBM', '2004/12/18', 183, 183.125, -0.125, -0.51, 4371400);
insert into datos values('HWP', '2004/12/18', 70, 71.0625, -1.4375, -2.01, 2410700);
insert into datos values('UIS', '2004/12/18', 28.25, 29, -0.75, -2.59, 2576200);
insert into datos values('SNE', '2004/12/18', 96.1875, 95.625, 1.125, 1.18, 330600);
insert into datos values('NOVL', '2004/12/18', 24.0625, 24.375, -0.3125, -3.02, 6047900);
insert into datos values('HIT', '2004/12/18', 78.5, 77.625, 0.875, 1.12, 49400);
```

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.text.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class TablaValores3 extends JFrame {
    private static final long serialVersionUID = 1L;
    protected JTable m_tabla;
    protected DatoTablaValores m_dato;
    protected JLabel m_titulo;

    public TablaValores3() {
        super("Tabla de Valores");
        setSize(600, 300);
        UIManager.put( "Table.focusCellHighlightBorder",
                       new LineBorder(Color.black, 0) );
        m_dato = new DatoTablaValores();
        m_titulo = new JLabel( m_dato.getTitulo(),
                   new ImageIcon("imagen\\money.gif"), SwingConstants.CENTER );
        m_titulo.setFont(new Font("Helvetica",Font.PLAIN,24));
        getContentPane().add(m_titulo, BorderLayout.NORTH);
        m_tabla = new JTable();
        m_tabla.setAutoCreateColumnsFromModel(false);
        m_tabla.setModel(m_dato);
        for (int k = 0; k < m_dato.getColumnCount(); k++) {
            DefaultTableCellRenderer renderer = new  ColoredTableCellRenderer();
            renderer.setHorizontalAlignment(
                                DatoTablaValores.m_columnas[k].m_alineacion );
            TableColumn columna = new TableColumn(k,
                         DatoTablaValores.m_columnas[k].m_ancho, renderer, null);
            columna.setHeaderRenderer( createDefaultRenderer() );
            m_tabla.addColumn(columna);
        }
```

```
        JTableHeader header = m_tabla.getTableHeader();
        header.setUpdateTableInRealTime(true);
        header.addMouseListener( new ColumnaListener() );
        header.setReorderingAllowed(true);
        setJMenuBar( creaMenuBar() );
        JScrollPane ps = new JScrollPane();
        ps.getViewport().setBackground(m_tabla.getBackground());
        ps.getViewport().add(m_tabla);
        getContentPane().add(ps, BorderLayout.CENTER);
        setJMenuBar( creaMenuBar() );
    }

    protected JMenuBar creaMenuBar() {
        JMenuBar menuBar = new JMenuBar();
        JMenu mFile = new JMenu("Fichero");
        mFile.setMnemonic('f');
        JMenuItem mData = new JMenuItem("Recuperar Datos...");
        mData.setMnemonic('r');
        ActionListener lstData = new ActionListener() {
                                    public void actionPerformed(ActionEvent e) {
                                          recuperaDatos();
                                    }
                                };
        mData.addActionListener(lstData);
        mFile.add(mData);
        mFile.addSeparator();
        JMenuItem mExit = new JMenuItem("Salir");
        mExit.setMnemonic('s');
        ActionListener lstExit = new ActionListener() {
                                     public void actionPerformed(ActionEvent e) {
                                         System.exit(0);
                                     }
                                 };
        mExit.addActionListener(lstExit);
        mFile.add(mExit);
        menuBar.add(mFile);
        return menuBar;
    }

    public void recuperaDatos() {
        Runnable updater = new Runnable() {
            public void run() {
               SimpleDateFormat frm = new SimpleDateFormat("dd/MM/yyyy");
               if (m_dato==null) m_dato = new DatoTablaValores();  // Para 1ª vez
               if(m_dato.m_fecha == null) m_dato.m_fecha= new Date();
               String fechaUsada = frm.format(m_dato.m_fecha);
               String resultado = (String)JOptionPane.showInputDialog(null,
                           "Por favor, teclee la fecha en formato mm/dd/yyyy:",
                           "Input", JOptionPane.INFORMATION_MESSAGE,
```

```
                        null, null, fechaUsada);
                if(resultado == null) return;   // Si cancela y no elige fecha
                java.util.Date date = null;
                try {
                    date = frm.parse(resultado);
                }
                catch (java.text.ParseException ex) {
                    date = null;
                }
                if (date == null) {
                    JOptionPane.showMessageDialog(TablaValores3.this,
                                    resultado + " no es una fecha correcta",
                                    "Warning", JOptionPane.WARNING_MESSAGE);
                    return;
                }
                setCursor( Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR) );
                try {
                    JOptionPane.showMessageDialog( null, "Se ejecuta la consulta");
                    m_dato.recuperaDatos(date);
                }
                catch (Exception ex) {
                    JOptionPane.showMessageDialog(TablaValores3.this,
                                                "Error recuperando datos:\n" +
                                                 ex.getMessage(), "Error",
                                                 JOptionPane.ERROR_MESSAGE);
                }
                setCursor( Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR) );
                m_titulo.setText( m_dato.getTitulo() );
                m_tabla.tableChanged( new TableModelEvent(m_dato) );
            }
        };
        SwingUtilities.invokeLater(updater);
    }

    protected TableCellRenderer createDefaultRenderer() {
        DefaultTableCellRenderer label = new DefaultTableCellRenderer() {
            public Component getTableCellRendererComponent(
                    JTable table, Object value, boolean isSelected,
                                boolean hasFocus, int row, int column) {
                if (table != null) {
                    JTableHeader header = table.getTableHeader();
                    if (header != null) {
                        setForeground(header.getForeground());
                        setBackground(header.getBackground());
                        setFont(header.getFont());
                    }
                }
                setText((value == null) ? "" : value.toString());
                setBorder( UIManager.getBorder("TableHeader.cellBorder") );
```

```
                    return this;
                }
            };
            label.setHorizontalAlignment(JLabel.CENTER);
            return label;
    }

    public static void main(String argv[]) {
        TablaValores3 frame = new TablaValores3();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

class ColumnaListener extends MouseAdapter {
        public void mouseClicked(MouseEvent e) {
        TableColumnModel colModel = m_tabla.getColumnModel();
        int columnModelIndex = colModel.getColumnIndexAtX(e.getX());
        int modelIndex = colModel.getColumn(columnModelIndex).getModelIndex();
        if (modelIndex < 0) return;
        if (m_dato.m_sortCol == modelIndex)
            m_dato.m_sortAsc = !m_dato.m_sortAsc;
        else
            m_dato.m_sortCol = modelIndex;
        for (int i=0; i < m_dato.getColumnCount(); i++) {
            TableColumn column = colModel.getColumn(i);
            int index = column.getModelIndex();
            JLabel renderer = (JLabel)column.getHeaderRenderer();
            renderer.setIcon(m_dato.getColumnIcon(index));
        }
        m_tabla.getTableHeader().repaint();
        m_dato.sortDatos();
        m_tabla.tableChanged(new TableModelEvent(m_dato));
        m_tabla.repaint();
    }
}  // Acaba clase ColumnaListener

}  // Acaba la clase TablaValores2

class ColoredTableCellRenderer extends DefaultTableCellRenderer {
    public void setValue(Object value) {
       if (value instanceof DatoColor) {
           DatoColor cvalue = (DatoColor)value;
           setForeground(cvalue.m_color);
           setText(cvalue.m_data.toString());
       }
       else if (value instanceof DatoIcono) {
           DatoIcono ivalue = (DatoIcono)value;
           setIcon(ivalue.m_icon);
           setText(ivalue.m_dato.toString());
```

```
        }
        else super.setValue(value);
    }

    protected TableCellRenderer createDefaultRenderer() {
        DefaultTableCellRenderer label = new DefaultTableCellRenderer() {
            public Component getTableCellRendererComponent(
            JTable table, Object value, boolean isSelected,  boolean hasFocus,
                        int row, int column) {
                if (table != null) {
                    JTableHeader header = table.getTableHeader();
                    if (header != null) {
                        setForeground(header.getForeground());
                        setBackground(header.getBackground());
                        setFont(header.getFont());
                    }
                }
                setText((value == null) ? "" : value.toString());
                setBorder(UIManager.getBorder("TableHeader.cellBorder"));
                return this;
            }
        };
        label.setHorizontalAlignment(JLabel.CENTER);
        return label;
    }
}

class DatoColor {
        public Color m_color;
        public Object m_data;
        public static Color GREEN = new Color(0, 128, 0);
        public static Color RED = Color.red;

        public DatoColor(Color color, Object data) {
            m_color = color;
            m_data = data;
        }

        public DatoColor(Double data) {
            m_color = data.doubleValue() >= 0 ? GREEN : RED;
            m_data = data;
        }

        public String toString() { return m_data.toString(); }
}

class DatoIcono {
        public ImageIcon m_icon;
        public Object m_dato;
```

```
        public DatoIcono(ImageIcon icon, Object data) {
            m_icon = icon;
            m_dato = data;
        }

        public String toString() { return m_dato.toString(); }
}

class DatoValor {
        public DatoIcono m_abrevia;
        public String m_nombre;
        public Double m_last;
        public Double m_open;
        public DatoColor m_cambio;
        public DatoColor m_cambioPct;
        public Long m_volumen;
        static private ImageIcon ICON_UP = new ImageIcon("imagen\\uparrow.gif");
        static private ImageIcon ICON_DOWN = new
ImageIcon("imagen\\downarrow.gif");
        static private ImageIcon ICON_BLANK = new ImageIcon("imagen\\blank.gif");

    public DatoValor(String symbol, String name, double last,
        double open, double change, double changePr, long volume) {
        m_abrevia = new DatoIcono( getIcon(change), symbol);
        m_nombre = name;
        m_last = last;
        m_open = open;
        m_cambio = new DatoColor( change );
        m_cambioPct = new DatoColor( changePr );
        m_volumen = volume;
    }

   public ImageIcon getIcon(double cambio) {
        return (cambio > 0 ? ICON_UP : (cambio < 0 ? ICON_DOWN:ICON_BLANK));
   }
}

class DatoColumna {
   public String m_titulo;
   public int m_ancho;
   public int m_alineacion;

   public DatoColumna(String titulo, int ancho, int alineacion) {
        m_titulo = titulo;
        m_ancho = ancho;
        m_alineacion = alineacion;
   }
}
```

```
class DatoTablaValores extends AbstractTableModel {
    static final public DatoColumna m_columnas[] = {
                new DatoColumna( "Símbolo", 100, JLabel.LEFT ),
                new DatoColumna( "Nombre", 160, JLabel.LEFT ),
                new DatoColumna( "Last", 100, JLabel.RIGHT ),
                new DatoColumna( "Open", 100, JLabel.RIGHT ),
                new DatoColumna( "Cambio", 100, JLabel.RIGHT ),
                new DatoColumna( "Cambio %", 100, JLabel.RIGHT ),
                new DatoColumna( "Volumen", 100, JLabel.RIGHT )
    };

    public static ImageIcon COLUMNA_UP = new ImageIcon("imagen\\sortup.gif");
    public static ImageIcon COLUMNA_DOWN = new ImageIcon("imagen\\sortdown.gif");
    protected SimpleDateFormat m_frm;
    protected NumberFormat m_volumeFormat;
    protected Vector m_vector;
    protected Date m_fecha;
    public int m_sortCol = 0;
    public boolean m_sortAsc = true;

    static final String QUERY = "SELECT d.simbolo, s.nombre, " +
                    "d.last, d.open, d.cambio, d.cambioPct, " +
       "d.volumen FROM datos d INNER JOIN simbolos s ON d.simbolo = s.simbolo" +
       " WHERE month(d.fecha) = ? AND day(d.fecha)= ? AND year(d.fecha)= ?";

    public void recuperaDatos(Date fecha) throws SQLException,
ClassNotFoundException {
        GregorianCalendar calendar = new GregorianCalendar();
        calendar.setTime(fecha);
        int mes =  calendar.get( Calendar.MONTH ) + 1;
        int dia =  calendar.get( Calendar.DAY_OF_MONTH );
        int year = calendar.get( Calendar.YEAR );
        m_fecha = fecha;
        m_vector = new Vector();

        Connection conex = null;
        PreparedStatement pst = null;
            String jdbcURL= "jdbc:mysql://localhost:3306/valores" +
                        "?useUnicode=true&useJDBCCompliantTimezoneShift=true" +
                        "&useLegacyDatetimeCode=false&serverTimezone=UTC";
        try {
            // Mala decisión dejar password en programa!!
            conex= DriverManager.getConnection( jdbcURL, "root", "1234" );
            // Mejor opción: declarar un miembro estático en la clase principal,
            // y preguntarla la primera vez si es null, sino, usarla si es OK.
            pst = conex.prepareStatement(QUERY);
            pst.setInt(1, mes);
            pst.setInt(2, dia);
```

```
            pst.setInt(3, year);
            ResultSet results = pst.executeQuery();
            while ( results.next() ) {
                String simbolo = results.getString(1);
                String nombre = results.getString(2);
                double last = results.getDouble(3);
                double open = results.getDouble(4);
                double cambio = results.getDouble(5);
                double cambioPct = results.getDouble(6);
                long volumen = results.getLong(7);
                m_vector.addElement( new DatoValor(simbolo, nombre, last, open,
                                                   cambio, cambioPct, volumen) );
            }
            sortDatos();
        }
        finally {
            if(pst != null) pst.close();
            if(conex != null) conex.close();
        }
    }

    public DatoTablaValores() {
        m_frm = new SimpleDateFormat("MM/dd/yyyy");
        m_volumeFormat = NumberFormat.getInstance();
        m_volumeFormat.setGroupingUsed(true);
        m_volumeFormat.setMaximumFractionDigits(0);
        m_vector = new Vector();
        sortDatos();
    }

    public int getRowCount() { return m_vector==null ? 0 : m_vector.size(); }

    public int getColumnCount() { return m_columnas.length; }

    public String getColumnName(int column) {
             return m_columnas[column].m_titulo; }

    public boolean isCellEditable(int nRow, int nCol) { return false; }

    public Object getValueAt(int nRow, int nCol) {
        if (nRow < 0 || nRow >= getRowCount()) return "";
        DatoValor row = (DatoValor)m_vector.elementAt(nRow);
        switch (nCol) {
        case 0: return row.m_abrevia;
        case 1: return row.m_nombre;
        case 2: return row.m_last;
        case 3: return row.m_open;
        case 4: return row.m_cambio;
        case 5: return row.m_cambioPct;
```

```
        case 6: return row.m_volumen;
        }
        return "";
    }

    public String getTitulo() {
        if (m_fecha==null) return "Cuotas de Valores";
        return "Cuotas de valores el " + m_frm.format(m_fecha);
    }

    public Icon getColumnIcon(int column) {
        if (column==m_sortCol) return m_sortAsc ? COLUMNA_UP : COLUMNA_DOWN;
        return null;
    }

    public void sortDatos() {
        Collections.sort(m_vector, new StockComparator(m_sortCol, m_sortAsc) );
    }
}

class StockComparator implements Comparator {
    protected int m_sortCol;
    protected boolean m_sortAsc;

    public StockComparator(int sortCol, boolean sortAsc) {
        m_sortCol = sortCol;
        m_sortAsc = sortAsc;
    }

    public int compare(Object o1, Object o2) {
        if( !(o1 instanceof DatoValor) || !(o2 instanceof DatoValor)) return 0;
        DatoValor s1 = (DatoValor) o1;
        DatoValor s2 = (DatoValor) o2;
        int result = 0;
        double d1, d2;
        switch (m_sortCol) {
        case 0: // abreviatura
                String str1 = (String)s1.m_abrevia.m_dato;
                String str2 = (String)s2.m_abrevia.m_dato;
                result = str1.compareTo(str2);
                break;
        case 1:// nombre
               result = s1.m_nombre.compareTo(s2.m_nombre);
               break;
        case 2: // last
                d1 = s1.m_last.doubleValue();
                d2 = s2.m_last.doubleValue();
                result = d1 < d2 ? -1 : (d1 > d2 ? 1 : 0);
                break;
```

```
        case 3:// open
               d1 = s1.m_open.doubleValue();
               d2 = s2.m_open.doubleValue();
               result = d1 < d2 ? -1: (d1 > d2 ? 1 : 0);
               break;
        case 4:// cambio
               d1 = ((Double)s1.m_cambio.m_data).doubleValue();
               d2 = ((Double)s2.m_cambio.m_data).doubleValue();
               result = d1 < d2 ? -1 : (d1 > d2 ? 1 : 0);
               break;
        case 5:// % cambio
               d1 = ((Double)s1.m_cambioPct.m_data).doubleValue();
               d2 = ((Double)s2.m_cambioPct.m_data).doubleValue();
               result = d1 < d2 ? -1 : (d1 > d2 ? 1 : 0);
               break;
        case 6:// volumen
                    long l1 = s1.m_volumen.longValue();
                    long l2 = s2.m_volumen.longValue();
                    result = l1<l2 ? -1 : (l1>l2 ? 1 : 0);
                    break;
        }
        if (!m_sortAsc) result = -result;
        return result;
    }

   public boolean equals(Object obj) {
        if (obj instanceof StockComparator) {
        StockComparator compObj = (StockComparator)obj;
        return (compObj.m_sortCol == m_sortCol) && (compObj.m_sortAsc==m_sortAsc);
      }
      return false;
   }
}
```

10.5 ASPECTOS AVANZADOS.

10.5.1 TRABAJAR CON METADATOS.

La interfaz ***DatabaseMetadata*** ofrece información sobre la BD a la que nos hemos conectado. Aporta un gran número de métodos. Muchos de ellos devuelven objetos *ResultSet* y por tanto debemos usar métodos *getXXX()* para recuperar la información.

Algunas de las cabeceras de estos métodos toman como parámetro

patrones de cadenas, en los que se pueden utilizar caracteres comodín. "%" identifica una cadena de 0 o más caracteres y "_" se identifica con un sólo carácter, de esta forma, sólo los datos que se correspondan con el patrón serán devueltos por el método.

Si un driver no soporta un método de la interfaz *DatabaseMetaData* se lanza una excepción ***SQLException***, y en el caso de que el método devuelva un objeto *ResultSet*, se obtendrá un *ResultSet* vacío.

Para obtener un objeto ***DatabaseMetaData*** sobre el que lanzar los métodos que nos darán la información sobre el SGBD se ejecuta el método ***getMetaData()*** del objeto ***Connection***:

```
Connection con = DriverManager.getConnection(jdbcUrl,"root","");
DatabaseMetadata dbmd = con.getMetaData();
```

OBTENER INFORMACIÓN DE UNA BD

Vamos a realizar una aplicación Java que se conecte a una BD MySQL. Una vez conectados vamos a obtener algunos datos de interés sobre ella. Los métodos que se utilizan de esta interfaz se han dividido en 5 métodos diferentes, atendiendo a la información que obtenemos: producto, driver, funciones, tablas y procedimientos. En el siguiente fragmento de código podemos ver alguno de estos métodos:

```
Connection con = null;
DatabaseMetaData dbmd = null;
try {
    //Registrando el Driver, quizás debas actualizarlo
    String driver = "com.mysql.jdbc.Driver";
    Class.forName(driver).newInstance();
    System.out.println("Driver "+ driver + " Registrado");
    //Abrir la conexión con la Base de Datos
    System.out.println("Conectando con la Base de datos...");
    String jdbcUrl = "jdbc:mysql://localhost:3306/empresa";
    con = DriverManager.getConnection(jdbcUrl,"root","");
    System.out.println("Conexión establecida con SGBD...");
```

```
            dbmd = con.getMetaData();
            infoProducto(dbmd);        // Info. producto SGBD
            infoDriver(dbmd);          // Información del driver JDBC
            infoFunciones(dbmd);       // Info funciones de la BD
            infoTablas(dbmd);          // Tablas existentes
            infoProcedimientos(dbmd); // Procedimientos existentes
        }
        catch(SQLException se) {
            se.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if(conn!=null) conn.close();
            }
            catch(SQLException se) {
                se.printStackTrace();
            }
        }
```

El primero de los métodos es el método **infoProducto()**. Este método ofrece información general sobre el SGBD: nombre, versión, URL de JDBC, usuario conectado, características que soporta, etc.

```
        public static void infoProducto(DatabaseMetaData dbmd) throws
        SQLException {
            System.out.println(">>>Información sobre el SGBD:");
            String producto= dbmd.getDatabaseProductName();
            String version= dbmd.getDatabaseProductVersion();
            boolean soportaSQL= dbmd.supportsANSI92EntryLevelSQL();
            boolean soportaConvert= dbmd.supportsConvert();
            boolean usaFich= dbmd.usesLocalFiles();
            boolean soportaGRBY= dbmd.supportsGroupBy();
            boolean soportaMinSQL= dbmd.supportsMinimumSQLGrammar();
            String nombre= dbmd.getUserName();
            String url= dbmd.getURL();
            System.out.println(" Producto: " + producto + " " + version);
            System.out.println(" Soporta el SQL ANSI92: " + soportaSQL);
            System.out.println(" Soporta función CONVERT en tipos SQL: " +
                               soportaConvert);
            System.out.println(" Almacena tablas en ficheros locales: " +
                               usaFich);
```

```
        System.out.println(" Nombre de usuario conectado: " + nombre);
        System.out.println(" URL de la Base de Datos: " + url);
        System.out.println(" Soporta GROUP BY: " + soportaGRBY);
        System.out.println(" Soporta la mínima gramática SQL: " +
                           soportaMinSQL);
        System.out.println();
    }
```

El siguiente método es **infoDriver()**. Dentro se obtiene información sobre el driver: nombre, versión, versión inferior y superior:

```
    public static void infoDriver(DatabaseMetaData dbmd) throws
    SQLException {
        System.out.println(">>>Información sobre el driver:");
        String driver= dbmd.getDriverName();
        String driVersion= dbmd.getDriverVersion();
        int verMayor= dbmd.getDriverMajorVersion();
        int verMenor= dbmd.getDriverMinorVersion();
        System.out.println(" Driver: " + driver + " " + driVersion);
        System.out.println(" Versión superior del driver: "+verMayor);
        System.out.println(" Versión inferior del driver: "+verMenor);
        System.out.println();
    }
```

La información acerca de las funciones que ofrece nuestra BD la obtenemos a partir del método **infoFunciones()**. Este método nos indicará las funciones que soporta el SGBD en lo que respecta a cadenas de caracteres, cálculos, fechas y sistema.

```
    public static void infoFunciones(DatabaseMetaData dbmd) throws
    SQLException {
        System.out.println(">>>Funciones del DBMS:");
        String fCadenas= dbmd.getStringFunctions();
        String fSistema= dbmd.getSystemFunctions();
        String fTiempo= dbmd.getTimeDateFunctions();
        String fNumericas= dbmd.getNumericFunctions();
        System.out.println(" Funciones de Cadenas:" + fCadenas);
        System.out.println(" Funciones Numéricas: " + fNumericas);
        System.out.println(" Funciones del Sistema: " + fSistema);
        System.out.println(" Funciones de Fecha y Hora: " + fTiempo);
        System.out.println();
    }
```

La llamada al método **getTables()** usa los dos primeros parámetros para obtener las tablas de un catálogo, no vamos a mostrar el listado pero indicamos como sería el método. Le pasamos un null en los dos primeros parámetros. El tercer parámetro es el patrón de búsqueda que se aplicar al nombre de las tablas, si utilizamos el carácter % obtendremos todas las tablas. El último parámetro indica el tipo de tabla que queremos obtener y es un array con todos los nombres de tipos de tablas de las que queremos obtener información, los tipos de tabla son: TABLE, VIEW, SYSTEMTABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS y SYNONYM. Si queremos recuperar información sobre las tablas de usuario y de sistema, tendremos un array con dos elementos, el primero de ellos contendrá la cadena TABLE y el segundo la cadena SYSTEMTABLE. Los nombres de las columnas que tendrá el Resultset devuelto son:

```
TABLE_CAT: catálogo de la tabla.
TABLE_SCHEM: esquema de la tabla.
TABLE_NAME: nombre de la tabla.
TABLE_TYPE: tipo de la tabla.
REMARKS: comentarios acerca de la tabla.
```

Para recuperar cualquier columna de este ResultSet utilizamos el método getString() del ResultSet.

El método **infoProcedimientos()** tampoco lo vamos a listar y sería muy similar al anterior, en lugar de recuperar información sobre las tablas, obtiene información sobre los procedimientos almacenados usando **getProcedures(catalogo, patronEsquema, patronNombres)**. Los dos primeros pueden ir a null y hay que pasarle el patrón de búsqueda que se aplica al nombre de los procedimientos almacenados, si queremos recuperar todos, utilizaremos el carácter %. Los nombres de las columnas del ResultSet devuelto son:

```
PROCEDURE_CAT: catálogo del procedimiento.
PROCEDURE_SCHEM: esquema del procedimiento.
PROCEDURE_NAME: nombre del procedimiento.
REMARKS: comentarios acerca del procedimiento.
PROCEDURE_TYPE: tipo de procedimiento.
```

OBTENER INFORMACIÓN DE UN RESULTSET

El interfaz ResultSetMetaData también proporciona métodos para recuperar información de un ResultSet. A continuación mostramos una tabla con los más usados:

- **getTableName()** String con nombre de tabla usada.
- **getColumnCount()** int con el número de columnas
- **getCatalogName(column: int)**: String
- **getColumnClassName(column: int)**: String
- **getColumnDisplaySize(column:int)**: int
- **getColumnLabel(column:int)**:String
- **getColumnName**(c:int):String nombre de columna en posición c
- **getColumnType**(column:int):int tipo JDBC de la columna c
- **getColumnTypeName**(column:int):String Nombre del tipo JDBC
- **getSchemaName** (column: int): String
- **getTableName**(column:int): String
- **isNullable**(column:int):int

A continuación vamos a mostrar un ejemplo de uso de la interface **ResultSetMetaData**, en este caso se va a consultar la tabla de sucursales y se va a mostrar información de cada una de las columnas como el nombre de la columna y el tipo.

```
public static void infoColumnas(ResultSetMetaData rsmd) throws
SQLException {
    System.out.println(">> Consultado la tabla [" +
                       rsmd.getTableName(1) + "]");
    int nCols = rsmd.getColumnCount();
    System.out.println("Numero de columnas: " + nCols);
    for (int col= 1; col <= nCols; col++) {
        System.out.println("Nombre: " + rsmd.getColumnName(col) +
                           " Tipo: " + rsmd.getColumnTypeName(col)
                          );
```

```
        }
    }
```

EJERCICIO 10: Realiza un formulario donde muestre todas las tablas de la BD mediante un ComboBox obtenidas a partir del DatabaseMetaData e indique para cada una de ellas el número de columnas, su nombre y su tipo.

10.5.2. DATASOURCES Y CONNECTION POOLING.

El paquete javax.sql y se denomina "extensión estándar del API JDBC" y añade una funcionalidad significativa al API que se puede resumir en cuatro categorías generales:

- **La interface DataSource**: para trabajar con el servicio de nombres Java Naming and Directory Interface (**JNDI**), permite establecer conexiones con una fuente de datos.
- **Pooling de conexiones**: mecanismo mediante el cual se pueden reutilizar las conexiones en lugar de crear una nueva conexión cada vez que se necesite.
- **Transacciones distribuidas**: mecanismo que permite utilizar en una misma transacción diferentes servidores de BD.
- **La interface RowSet**: un componente JavaBean que contiene un conjunto de filas y se utiliza sobre todo para ofrecer datos a un cliente.

DATASOURCES

Un objeto **DataSource** es la representación de una fuente de datos en Java. Permite almacenar datos. Puede ser tan sofisticada como una compleja BD o tan simple como un fichero de texto con filas y columnas. Se puede almacenar en un servidor remoto o en local.

DataSource es una alternativa a la utilización de **DriverManager** a la

hora de establecer una conexión con una fuente de datos. Tiene métodos para establecer una conexión y una serie de propiedades que identifican y describen la fuente de datos a la que representa. El nombre de estas propiedades la podemos ver en la siguiente tabla, aunque cada distribuidor puede añadir nuevas propiedades específicas de su driver:

Nombre de la propiedad	Tipo de dato	Descripción
databaseName	String	Nombre de la base de datos
dataSourceName	String	Nombre lógico de los objetos XADataSource y PooledConnection
description	String	Descripción de la fuente de datos.
networkProtocol	String	El protocolo de red utilizado para la comunicación con el servidor.
password	String	La contraseña del usuario de la base de datos.
portNumber	int	El número de puerto en el que el servidor estar escuhando las peticiones.
roleName	String	El rol inicial de SQL.
serverName	String	El nombre del servidor de la base de datos.
user	String	El nombre de la cuenta del usuario de la base de datos.

Además trabaja junto con el servicio de nombres **JNDI** (Java Naming and Directory Interface). Si se registra el objeto DataSource con JNDI, obtenemos las siguientes ventajas:

- La aplicación no tendrá que codificar la información referente al driver, a diferencia de cómo se hace con DriverManager.
- Un programador puede elegir un nombre lógico para una fuente de datos y registrar el nombre lógico con el servicio de nombres JNDI. La aplicación utilizará el nombre lógico y el servicio de nombre JNDI ofrecerá el objeto DataSource asociado con ese nombre lógico.

JNDI ofrece un mecanismo uniforme para que una aplicación pueda encontrar y acceder a servicios remotos a través de la red. Este servicio remoto puede ser una BD. Una vez que el objeto DataSource correspondiente se encuentra registrado con el servicio de nombres JNDI, una aplicación puede utilizar el API JNDI para acceder al objeto DataSource y así poder conectarse a la fuente de datos que este objeto representa.

EJEMPLO 31: Crea un objeto DataSource de MySQL, se establecen algunas propiedades, se conecta y se desconecta.

```
module m10 {
    requires java.sql;
    requires java.desktop;
    requires mysql.connector.java;
}
```

```
package p10;

import java.sql.Connection;
import java.sql.SQLException;
import com.mysql.cj.jdbc.MysqlDataSource;

public class PruebaDatasource {
    public static void main(String[] args) throws SQLException {
        MysqlDataSource mds = new MysqlDataSource();
        mds.setUser("userdao");
        mds.setPassword("UserDAO_1");
        mds.setDatabaseName("demodao");
        mds.setPortNumber(3306);
        mds.setServerName("localhost");
        //Abrimos la conexión
        Connection con = mds.getConnection();
        System.out.println("Conexión establecida!");
        con.close();
       }
}
```

USAR DATASOURCE CON JNDI

JNDI es una interface de programación (API) que ofrece una forma estandar para almacenar y localizar objetos en un servicio de directorio. Está definido para ser independiente de cualquier implementación de este servicio de directorio. Así se puede acceder a una gran variedad de directorios -nuevos, emergentes y ya desarrollados- de una forma similar desde programas Java.

La arquitectura JNDI consiste en un API y un "service provider interface (SPI)". El SPI permite conectar de forma transparente una gran variedad de servicios de nombres y directorios. El JNDI está dividido en 5 paquetes:

- `javax.naming`
- `javax.naming.directory`
- `javax.naming.event`
- `javax.naming.ldap`
- `javax.naming.spi`

En nuestro caso vamos a utilizar JNDI para registrar y almacenar el objeto DataSource, que queda accesible desde cualquier parte que use ese servicio de directorio.

En el método main() del programa, creamos un contexto inicial. Indicamos que estamos usando el proveedor de servicios del sistema de ficheros representado por la propiedad `Context.PROVIDER_URL`, que en nuestro caso usa el repositorio (almacén) de una carpeta del sistema de ficheros que previamente debemos crear `file://C:/jndi/space`. Después llamamos al constructor del InitialContext, de esta forma:

```
//Creamos el contexto inicial usando el sistema de ficheros
System.out.println("Creamos el contexto Inicial...");
Hashtable env = new Hashtable();
String sp = "com.sun.jndi.fscontext.RefFSContextFactory";
String urlProvider = "file:/ejercicios/jndi/space";
env.put(Context.INITIAL_CONTEXT_FACTORY,sp);
env.put(Context.PROVIDER_URL, urlProvider);
```

```
Context ctx = new InitialContext(env);
```

Para poder utilizar el proveedor de servicios del sistema de ficheros hay que descargar las librerías JNDI. Para registrar el datasource utilizamos la interfaz `javax.naming.Context` e invocamos al método **rebind()**, de esta forma si el objeto ya está registrado lo volverá a registrar y en caso de no estar, lo registrará. Si utilizamos el método **bind()** que también permite registrar el objeto, en caso de que ya este registrado producirá una **javax.naming.NameAlreadyBoundException**, indicando que el objeto ya se encuentra registrado.

En el siguiente fragmento de código vemos como registramos un objeto `MysqlDataSource` dentro del contexto.

```
public static void registrarDataSource(Context ctx, String
dataSourceName) throws NamingException {
    //Instanciamos el objeto DataSource
    MysqlDataSource mds = new MysqlDataSource();
    mds.setUser("root");
    mds.setPassword("");
    mds.setDatabaseName("empresa");
    mds.setPortNumber(3306);
    mds.setServerName("localhost");
    //Registramos el objeto
    ctx.rebind(dataSourceName, mds);
}
```

Localizar un objeto

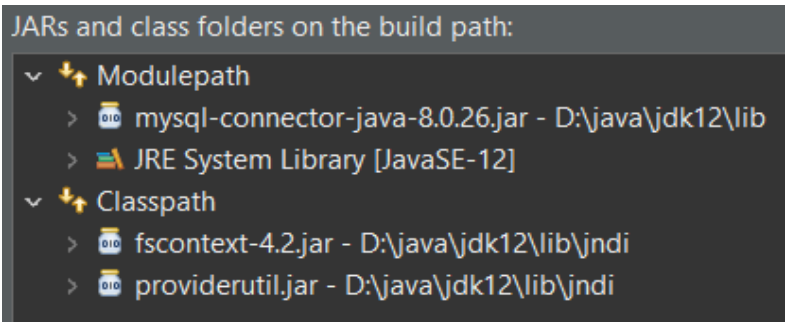
Luego, usamos **Context.lookup()** para localizar un objeto. El siguiente código localiza el objeto cuyo contenido es la referencia del objeto dataSourceName y devuelve el objeto que hemos registrado con ese mismo nombre que en este caso es un DataSource (MysqlDataSource).

```
//Buscamos el DataSource
System.out.println("Buscamos el DataSource...");
DataSource ds = null;
ds = (DataSource) ctx.lookup(dataSourceName);
```

Para ejecutar el programa, necesitamos acceso a las clases JNDI y al proveedor de servicios del sistema de ficheros. Para incluir las clases del proveedor de servicios del sistema de ficheros (`fscontext.jar` y `providerutil.jar`), o las incluimos en nuestra variable CLASSPATH o las añadimos como librerias al proyecto.

EJEMPLO 32: listado del uso de un Datasource con JNDI.

```
module m10 {
	requires java.sql;
	requires java.desktop;
	requires mysql.connector.java;
	requires java.naming;
}
```




```
package p10;

import java.sql.Connection;
import java.sql.SQLException;
import com.mysql.cj.jdbc.MysqlDataSource;

import java.util.Hashtable;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Prueba1 {
```

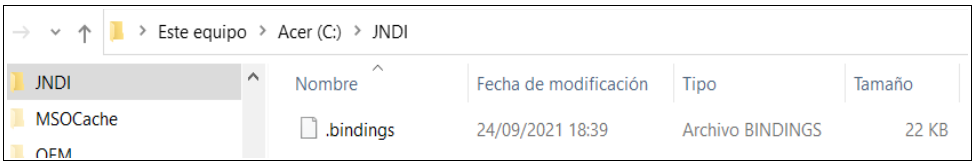
```
    public static void main(String[] args) throws SQLException,
                                                   NamingException {
        //Creamos el contexto inicial usando el sistema de ficheros
        System.out.println("Creamos el contexto Inicial...");
        Hashtable env = new Hashtable();
        String sp = "com.sun.jndi.fscontext.RefFSContextFactory";
        String urlProvider = "file:/C:/JNDI";
        env.put(Context.INITIAL_CONTEXT_FACTORY, sp);
        env.put(Context.PROVIDER_URL, urlProvider);
        Context ctx = new InitialContext(env);
        registrarDataSource(ctx, "BD_de_DAO");
        // Para encontrar la BD, la buscamos en el contexto
        System.out.println("Busca datasource con nombre BD_de_DAO.");
        MysqlDataSource mds = (MysqlDataSource)ctx.lookup("BD_de_DAO");
        // Abrimos la conexión
        Connection con = mds.getConnection();
        System.out.println("Conexión establecida!");
        con.close();
    }

    public static void registrarDataSource(Context ctx,
                                           String dsNombre)
    throws NamingException {
           // Instanciamos el objeto DataSource
           MysqlDataSource mds = new MysqlDataSource();
           mds.setUser("userdao");         // Podríamos dejar sin asignar
           mds.setPassword("UserDAO_1");   // Podríamos dejar sin asignar
           mds.setDatabaseName("demodao");
           mds.setPortNumber(3306);
           mds.setServerName("localhost");
           System.out.println("Registramos el objeto...");
           ctx.rebind(dataSourceName, mds);
       }
}
```

Al ejecutarlo:

```
Creamos el contexto Inicial...
Registramos el objeto...
Busca datasource con nombre BD_de_DAO.
Conexión establecida!
```

Y si miramos en el repositorio usado, veremos que está serializado el

objeto DataSource:



POOL DE CONEXIONES

Un pool de conexiones es una caché de objetos conexión a BD. Cada objeto representa una conexión física a una BD que puede usar una aplicación para conectarse de esta forma: la aplicación pide conectarse al pool. Si el pool tiene una conexión que satisface la petición se la pasa a la aplicación. Si no la encuentra, le crea una nueva. Cuando la aplicación ya no la necesite, devuelve la conexión al pool, de forma que queda disponible para otra aplicación.

Tener un pool de conexiones permite reutilizar objetos conexión y reducir la sobrecarga que supone crear, establecer y cerrar conexiones cada vez que se necesie una, mejorando el rendimiento de las aplicaciones que hacen un uso intensivo de las BD.

El componente clave dentro del pool de conexiones es el interfaz **javax.sql.PooledConnection**. Un objeto DataSource se implementa para que soporte la característica de pool de conexiones, es decir, para que sus conexiones sean reutilizables.

Cuando la clase (DataSource) ofrecida por el distribuidor implementa el pool de conexiones, cada objeto Connection que devuelva el método **getConnection()** del **DataSource**, podrá ser reutilizado. Cuando la aplicación cierra una conexión perteneciente a un datasource, la conexión no se destruye sino que se devuelve al pool. La próxima vez que la aplicación llame al método **getConnection()** de DataSource, se

utilizará la conexión existente.

Para utilizar un objeto DataSource con pool de conexiones, se crea de la misma forma, pero hay que asegurar que la conexión que se utiliza se cierra siempre, de esta forma se garantiza que la conexión pueda reutilizarse. Es necesario cerrar siempre las conexiones de una manera explícita. En lo demás, que la conexión sea pooled es transparente para el programador. El datasource puede almacenarse en un servicio de directorio para que las aplicaciones puedan usar el mismo y pedirle las conexiones pooled.

EJEMPLO 32. Usar Datasource con pool de conexiones.

```
package p10;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.mysql.cj.jdbc.MysqlConnectionPoolDataSource;
import com.mysql.cj.jdbc.MysqlDataSource;

import java.util.Hashtable;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.PooledConnection;

public class Prueba1 {

    public static void main(String[] args)
    throws SQLException, NamingException {
        // Crea el contexto inicial usando el sistema de ficheros
        System.out.println("Creamos el contexto Inicial...");
        Hashtable env = new Hashtable();
        String sp = "com.sun.jndi.fscontext.RefFSContextFactory";
        String urlProvider = "file:/C:/JNDI";
```

```
        env.put(Context.INITIAL_CONTEXT_FACTORY, sp);
        env.put(Context.PROVIDER_URL, urlProvider);
        Context ctx = new InitialContext(env);
        registrarPool(ctx, "POOL_de_DAO");
        // Buscar el pool en el contexto
        System.out.println("Busca pool POOL_de_DAO.");
        MysqlConnectionPoolDataSource mds =
              (MysqlConnectionPoolDataSource)ctx.lookup("POOL_de_DAO");
        PooledConnection pc = null;
        Connection con1 = null;
        Statement s1 = null;
        Connection con2 = null;
        Statement s2 = null;
        ResultSet rs = null;
        try {
            pc = mds.getPooledConnection(); // Crea el PooledConnection
            con1 = pc.getConnection();    // Abre conex. y crea sentencia
            s1 = con1.createStatement();
            System.out.println("Conexion 1 creada...");
            rs = s1.executeQuery("select count(*) from empleado");
            rs.next();
            System.out.println("[CON1] Nº empleados: " + rs.getInt(1) );
            rs.close();
            con2 = pc.getConnection(); // Nueva conex. y sentencia
            s2 = con2.createStatement();
            System.out.println("Conexion 2 creada...");
            rs = s2.executeQuery("select count(*) from empleado");
            rs.next();
            System.out.println("[CON2] Nº empleados: " + rs.getInt(1) );
            rs.close();
        } catch (SQLException se ){
            se.printStackTrace();
        } catch (Exception e ){
            e.printStackTrace();
        } finally {
            try {
                if (s1 != null ) { s1.close(); }
                if (con1 != null ) { con1.close(); }
                if (s2 != null ) { s2.close(); }
                if (con2 != null ) { con2.close(); }
                if (pc != null ) { pc.close(); }
            } catch (Exception fe) {
                fe.printStackTrace();
```

```
            }
        }
    }

    public static void registrarPool(Context ctx, String nombrePool)
    throws NamingException {
        // Instanciar el DataSource Pooled
        MysqlConnectionPoolDataSource mds =
                                    new MysqlConnectionPoolDataSource();
        mds.setUser("userdao");
        mds.setPassword("UserDAO_1"); // Tb. no asignar y preguntar
        mds.setDatabaseName("demodao");
        mds.setPortNumber(3306);
        mds.setServerName("localhost");
        System.out.println("Registrar objeto pool de conexiones...");
        ctx.rebind(nombrePool, mds);
    }
}
```

Al ejecutarlo:

```
Creamos el contexto Inicial...
Registramos el objeto pool de conexiones...
Busca pool POOL_de_DAO.
Conexion 1 creada...
[CON1] Nº empleados: 2
Conexion 2 creada...
[CON2] Nº empleados: 2
```

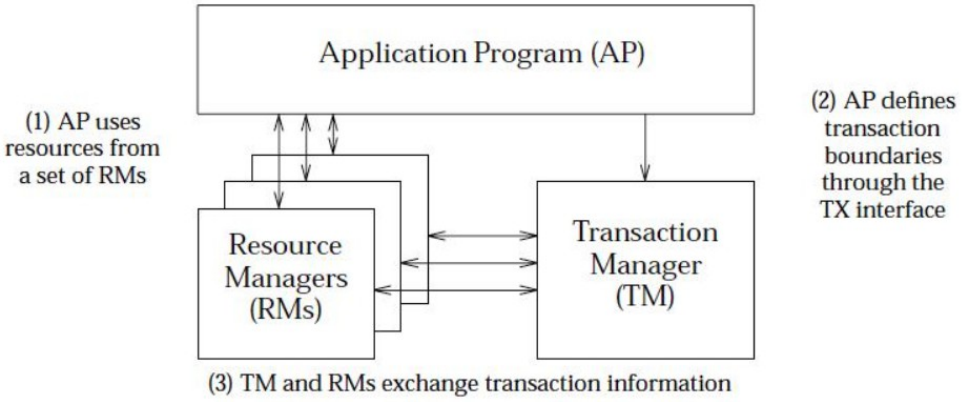
TRANSACCIONES DISTRIBUIDAS

Una transacción distribuida es aquella en la que participa más de un SGBD. Podemos definirla con esta figura.

La transacción distribuida en la especificación XA consiste principalmente en la interacción entre RM y TM:

- **AP**: el programa de aplicación define el límite de la transacción (el inicio y el final de la transacción) y accede a los recursos.

- **RM**: Administrador de recursos (Resource Manager) administra los recursos compartidos por la computadora y varios softwares pueden acceder a estos recursos, como BD, sistemas de archivos, servidores de impresoras, etc.
- **TM**: Transaction Manager es responsable de administrar transacciones globales, asignar identificadores únicos para transacciones, monitorear el progreso de las transacciones y responsable del envío de transacciones, reversión y recuperación de fallos.



MYSQL con el motor de almacenamiento InnoDB pueden garantizar que ACID se implemente a nivel de motor de almacenamiento, mientras que las transacciones distribuidas permiten extenderlas a nivel de BD o incluso a nivel de múltiples bases de datos, lo que se logra a través de un protocolo de confirmación de dos fases que está implementado en MySQL 5.0 o superiores (pero solo con InnoDB).

En una transacción distribuida de BD MySQL, cada BD MySQL es un administrador de recursos (RM) en XA, y el TM es el cliente que se conecta al servidor MySQL. Cabe señalar que en MySQL, las

transacciones distribuidas solo se pueden usar cuando el nivel de aislamiento es serializable, por lo que debe usar:

```
set global tx_isolation='serializable',
    session tx_isolation='serializable';
```

Y se realiza en un protocolo de dos fases:

- En la primera fase, TM pide a todos los RM que se preparen para realizar un trabajo como parte (rama) de la transacción correspondiente, y espera a que RM le indique si lo ha realizado. Si RM juzga que el trabajo que realiza puede enviarse, el trabajo se lleva a cabo y envía a TM un recibo OK. De lo contrario, envía a TM un recibo NO. Después de que RM envía una respuesta negativa y revierte el trabajo, puede descartar la información de la transacción.
- La segunda fase: TM decide si comitear o revertir la transacción en función de los resultados de cada RM en la fase 1. Si todos los RM tienen éxito, TM notifica a todos los RM que deben enviar resultados y confirmar cambios. Si hay un recibo de un RM a NO, TM notifica a todos los RM que apliquen rollback a su rama de la transacción.

Un objeto DataSource puede desplegarse para crear conexiones que puedan utilizarse en transacciones distribuidas. Hay que instanciar dos clases diferentes:

- Un objeto de la clase XADataSource
- Un objeto de la clase DataSource.

A nivel de programación hay que tener en cuenta que el programador dentro de una transacción distribuida nunca debería de hacer un commit ni un rollback, ni podría trabajar en modo autocommit activado.

El motivo es que en ese caso interferiría con el TM encargado de controlar la transacción distribuida.

EJEMPLO 33: En este ejemplo vamos a suponer que hay un negocio que tiene dos tiendas (tienda1 y tienda2) y que una de ellas va a llevar 5 unidades de producto a la otra. Cada tienda tendrá su propio SGBDR (llamados BD1 y BD2 respectivamente). Y en su esquema test tienen una tabla inventario donde para cada producto tienen el stock total.

```
mysql> create table inventario(
    -> productoID int not null,
    -> stock int default 0,
    -> primary key(productoID)
    -> );
Query OK, 0 rows affected (0.11 sec)
```

Nota: *En el código la ubicación de los servidores está **hardcoded** (define de forma fija la referencia a estos servidores). Esto no es buena idea porque haces dependiente al código de la infraestructura física de la empresa, lo hago por sencillez. Una mejor solución es leer esa información de un fichero o usar un servicio de directorio como JNDI.*

```
module m10 {
        requires java.sql;
        requires java.desktop;
        requires mysql.connector.java;
        requires java.naming;
        requires java.transaction.xa;
}
```

```
package p10;

import java.sql.Connection;
```

```java
import java.sql.Statement;

import javax.sql.XAConnection;
import javax.transaction.xa.XAResource;
import javax.transaction.xa.Xid;

import com.mysql.cj.jdbc.MysqlXADataSource;
import com.mysql.cj.jdbc.MysqlXid;

public class DemoTransacDistri {

    public static MysqlXADataSource getDataSource(String nombre,
                                        String user, String pwd) {
        try {
            MysqlXADataSource ds = new MysqlXADataSource();
            ds.setUrl(nombre);
            ds.setUser(user);
            ds.setPassword(pwd);
            return ds;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    public static void main(String[] arg) {
        String bd1 = "jdbc:mysql://192.168.0.1:3306/test"; // hardcoded
        String bd2 = "jdbc:mysql://192.168.0.2:3306/test";
        try {
            MysqlXADataSource ds1 = getDataSource(bd1, "root", "123456");
            MysqlXADataSource ds2 = getDataSource(bd2, "root", "123456");
            // BD1 obtiene la conexión
            XAConnection xaC1 = ds1.getXAConnection();
            XAResource xaR1 = xaC1.getXAResource();
            Connection c1 = xaC1.getConnection();
            Statement s1 = c1.createStatement();
            // BD2 obtiene la conexión
            XAConnection xaC2 = ds2.getXAConnection();
            XAResource xaR2 = xaC2.getXAResource();
            Connection c2 = xaC2.getConnection();
            Statement s2 = c2.createStatement();
            // Crea el xid de cada rama de transacción
            Xid xid1=new MysqlXid(new byte[]{0x01},new byte[]{0x02},100);
```

```
            Xid xid2=new MysqlXid(new byte[]{0x11},new byte[]{0x12},100);
            System.out.println("Tienda1 mueve 5 productos a tienda2.");
            try {
                // Rama 1 de Transacción: resta 5 unidades al stock
                xaR1.start(xid1, XAResource.TMNOFLAGS);
                int update1 = s1.executeUpdate("update inventario set
stock= stock - 5 where productoID = 1");
                xaR1.end(xid1, XAResource.TMSUCCESS);
                // Rama 2 de Transacción: suma 5 unidades al stock
                xaR2.start(xid2, XAResource.TMNOFLAGS);
                int update2 = s2.executeUpdate("update inventario set
stock= stock + 5 where productoID = 1");
                xaR2.end(xid2, XAResource.TMSUCCESS);
                System.out.println("Comienza FASE1 de transacción...");
                int ret1 = xaR1.prepare(xid1);
                int ret2 = xaR2.prepare(xid2);
                System.out.println("Comienza FASE2 de transacción...");
                if (XAResource.XA_OK==ret1 && XAResource.XA_OK == ret2) {
                    xaR1.commit(xid1, false);
                    xaR2.commit(xid2, false);
                    System.out.println("BD1:"+update1+", BD2:"+update2);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

10.5.3. UTILIZANDO ROWSETS.

Así como los DataSources son una alternativa a DriverManager los objetos **RowSets** son una alternativa a los objetos ResultSets.

La interface **RowSet** hereda el comportamiento del interfaz ResultSet y por tanto aporta la misma funcionalidad. **La diferencia con respecto a los ResultSet es que es un componente JavaBean** (tiene propiedades y un mecanismo de notificaciones) y por lo tanto tiene

métodos para añadir y eliminar oyentes y para asignar y obtener sus propiedades. Una de estas propiedades es un comando en forma de cadena que será una consulta SQL que podremos ejecutar sobre el objeto RowSet. Además nos ofrece la posibilidad de implementar navegación (scrollability) y modificar los datos (updatability),

NOTIFICACIONES.

Todos los objetos RowSet tienen 3 eventos:

- Movimiento del cursor (fila actual).
- Operaciones a nivel de fila: update, insert o delete.
- Cambios completos del RowSet.

Estas notificaciones se envían a todos los *listeners* que implementen la interface **RowSetListener** y que el objeto haya registrado. Un listener podría ser un componente GUI como una gráfica de barras. Por ejemplo la siguiente laínea de código registra una barra (bg) en el objeto rs.

```
rs.addListener(bg);
```

SCROLLABILITY O UPDATABILITY

Algunos SGBD no soportan result sets que sean navegables y otros no soportan que sean modificables. Si un driver para ese SGBD no añade esta funcionalidad, pudes usar un objeto RowSet para tenerla.

Se puede implementar un objeto RowSet para poblarlo con datos de cualquier tipo de fuente de datos, sin tener que ser necesariamente una base de datos relacional. Una vez que se han obtenido los datos desde la fuente de datos se puede desconectar el objeto RowSet para trabajar con sus datos. Los RowSet desconectados tienen datos para solo lectura, pero como son serializables y más ligeros, los hace ideales para enviar datos a

dispositivos móviles a través de la red.

La interface RowSet tiene estas clases implementadas:

- **JdbcRowSet**: de tipo conectado.
- **CachedRowSet**: desconectado. Define lo básico, el resto son extensiones de este. Puede hacer:
    - Obtener una conexión a un datasource y ejecutar una consulta.
    - Leer datos del ResultSet y poblarse con ellos.
    - Manipular y hacer cambios mientras está desconectado.
    - Reconectarse y volcar los cambios.
    - Comprobar conflictos y resolverlos.
- **WebRowSet**: extiende a CachedRowSet. Puede además:
    - Escribirse como un documento XML.
    - Leer un documento XML que describe un WebRowSet.
- **JoinRowSet**: extiende a un WebRowSet. Puede además:
    - Hacer la operación equivalente a un JOIN de SQL sin conectarse.
- **FilteredRowSet**: extiende a un WebRowSet. Puede además:
    - Aplicar un criterio de filtrado a los datos.

USAR LA CLASE JdbcRowSet

Trabajar con objetos RowSet requiere un enfoque diferente del que estamos acostumbrados con los objetos ResultSet del JDBC estándar. De hecho, cuando comenzamos a trabajar con RowSets veremos que es mucho más sencillo. Por ejemplo, cuando trabajamos con una aplicación que utiliza los objetos estándar de JDBC necesitamos como mínimo 3 objetos para poder interactuar con la BD, sin embargo ahora veremos que con un objeto RowSet es suficiente para hacer la misma tarea.

Crear Objetos JdbcRowSet

Usaremos una instancia de la clase **RowSetFactory** creada a partir de **RowSetProvider**.

```
RowSetFactory factory = RowSetProvider.newFactory();
try (JdbcRowSet jdbcRs = factory.createJdbcRowSet()) {
  jdbcRs.setUrl(this.settings.urlString);
  jdbcRs.setUsername(this.settings.userName);
  jdbcRs.setPassword(this.settings.password);
  jdbcRs.setCommand("select * from COFFEES");
  jdbcRs.execute();
  // ...
```

La instancia recién creada ya tiene estas propiedades:

- type: `ResultSet.TYPE_SCROLL_INSENSITIVE` (es scrollable)
- concurrency: `ResultSet.CONCUR_UPDATABLE` (es updatable)
- escapeProcessing: true (el driver realizará procesamiento de escape, traduce símbolos de escape a código que comprenda el motor de la base de datos)
- maxRows: 0 (no limita el número de filas)
- maxFieldSize: 0 (no limita los bytes de una columna de tipo `BINARY`, `VARBINARY`, `LONGVARBINARY`, `CHAR`, `VARCHAR`, y `LONGVARCHAR`)
- queryTimeout: 0 (no limita el tiempo que permite ejecutar una sentencia)
- showDeleted: false (las filas borradas no son visibles)
- transactionIsolation: `Connection.TRANSACTION_READ_COMMITTED` (lee solamente datos que hayan sido commiteados)
- typeMap: null (el mapa asociado con la conexión)

Estableciendo propiedades de conexión del RowSet

A diferencia de losResultSet, losRowSets se conectan al origen de los datos cuando éste necesita consultar o actualizar algún valor de los mismos. No es necesario crear un objeto Connection para realizar estas tareas ya que el RowSet las gestiona por ti. Esta principal

diferencia es lo que lo hace más eficiente que un ResultSet.

Sin embargo, se deben de satisfacer dos condiciones para usar el objeto RowSet. La primera, es que se debe de registrar el driver utilizando alguno de las dos técnicas que ya hemos visto, bien DriverManager.registerDriver() o Class.forName(). Y segundo, se deben establecer las propiedades típicas de una conexión como URL de la base de datos, usuario y contraseña. El siguiente fragmento de código muestra cómo configuramos las propiedades de conexión del objetoRowSet.

```
//Cargamos el driver
Class.forName("com.mysql.cj.jdbc.Driver");
//Creamos el objeto JdbcRowSet e
//inicializamos las propiedades de conexión
JdbcRowSetImpl jrs = new JdbcRowSetImpl();
jrs.setUrl(jdbcUrl);
jrs.setUsername(user);
jrs.setPassword(pass);
```

Para obtener datos el objeto JdbcRowSet debe conectarse a la BD. Y para ello debes proporcionar valor paa estas 4 propiedades:

- **username**: la cuenta de usuario que te permite acceder a la BD.
- **password**: el password de esa cuenta.
- **url**: la JDBC URL para la BD.
- **datasourceName**: el nombre usado paa recuperar el objeto DataSource registrado en un servicio JNDI.

Como el registro de un nombre de DataSource en un servicio JNDI es una tarea que normalmente recae en un administrador de sistemas, así que en vez de utilizar esa propiedad usaremos una conexión realizada con DriverManager y la propiedad **url** y no **datasourceName**.

Ejecutando sentencias SQL con RowSets

Se pueden ejecutar tanto sentencias DDL como DML con RowSets. Sin embargo, la forma con la que ejecutamos estas sentencias es un poco diferente. Una de las diferencias es que no es necesario instanciar ningún objeto Statement, PreparedStatement o CallableStatement. El RowSet gestiona internamente el acceso a los datos dependiendo de la query que se le ha pasado.

Si se ejecuta una sentencia SELECT, el RowSet contiene los datos devueltos por la consulta y hará caso omiso a los métodos que se producen debido a INSERTS, UPDATES, DELETES o sentencias DML.

Sin embargo, siempre se lanzará un SQLException en caso de que haya habido algún problema con alguna de las sentencias que se han ejecutado. Para poder ejecutar una sentencia se debe de llamar al método **execute(String sql)**, que recibe como parámetro un String que indica cual es la sentencia que se quiere ejecutar.

Si se utilizan consultas parametrizadas se debe de rellenar cada parámetro invocando al método **setXXX()** tal y como hacíamos cuando utilizábamos los PreparedStatements.

Navegar por los datos

Si no es scrollable solo puedes utilizar el método **next()** para mover el cursor hacia adelante y cambiar de fila. Pero si es scrollable puede usar los mismos métodos de un ResultSet. Por ejemplo:

```
jdbcRs.absolute(4);
jdbcRs.previous();
```

Operaciones con los datos.

Puedes modificar los datos del objeto ResultSet. Por ejemplo el precio de un producto.

```
jdbcRs.absolute(3);
```

```
jdbcRs.updateFloat("precio", 10.99f);
jdbcRs.updateRow();
```

El método updateFloat() modifica el dato del RowSet y el método updateRow() modifica la base de datos si está conectado.

Insertar una fila nueva es equivalente a insrtarla en la tabla (si la consulta es de una sola tabla y está conectado). Puedes mover el cursor a la fila insertada, dar valor a cada columna y llamar al método insertRow(). Podemos insertar varias filas a la vez.

```
jdbcRs.moveToInsertRow();
jdbcRs.updateString("nombre", "Nieve");
jdbcRs.moveToInsertRow();
jdbcRs.updateString("nombre", "Hielo");
jdbcRs.insertRow();
```

Borrar filas es similar. Por ejemplo, vamos a borrar la última fila del RowSet y de la BD (está conectado).

```
jdbcRs.last();
jdbcRs.deleteRow();
```

EJEMPLO 34: cómo ejecutar una sentencia SQL utilizando el RowSet.

```
package p10;

import java.sql.SQLException;

import javax.sql.rowset.JdbcRowSet;
import javax.sql.rowset.RowSetFactory;
import javax.sql.rowset.RowSetProvider;

public class DemoJdbcRowSet {

    public static void main(String[] args) {
        RowSetFactory factory = null;
        try {
            factory = RowSetProvider.newFactory();
        } catch(SQLException e) {
```

```
            e.printStackTrace();
        }
        try(JdbcRowSet jRs = factory.createJdbcRowSet()) {
            // Cargamos el driver
            Class.forName("com.mysql.cj.jdbc.Driver");
            // Definir propiedades de conexión
            jRs.setUrl(
               "jdbc:mysql://localhost:3306/demodao?serverTimezone=UTC");
            jRs.setUsername("userdao");
            jRs.setPassword("UserDAO_1");
            // definir sentencia
            jRs.setCommand("select * from empleado where id > ?");
            jRs.setInt(1, 1);
            jRs.execute();
            // Mostrar resultados
            while( jRs.next() ) {
               System.out.printf(" Empleado (Id: %5d, Nombre: %20s)\n",
                           jRs.getLong("id"), jRs.getString("nombre")  );
            }
        }
        catch(SQLException e) {
            e.printStackTrace();
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
         }
    }
}
```

USAR LA CLASE CachedRowSet

La clase **JdbcRowSet** proporciona algunos beneficios pero mantiene algunas limitaciones como (igual que ResultSet) necesitar una conexión permanente con la fuente de datos. Otro de los inconvenientes es que no se puede serializar el objeto, y eso limita bastante la capacidad de poder distribuir los objetos.

La clase **CachedRowSet** soluciona esas limitaciones y proporciona una implementación offline y serializable de RowSet. Para operar en un estado sin conexión el objeto CachedRowSet crea una caché de la

información que se ha consultado y permite trabajar en un estado sin conexión. Además es capaz de obtener datos desde caualquier fuente de datos, por un ejemplo un fichero de hoja de cálculo.

Una vez se ha rellenado el objeto con la información solicitada se puede serializar el objeto para compartir los datos con otros clientes sin necesidad de tener una conexión permanente con la BD.

Crear objetos de tipo CachedRowSet

Se usa una instancia de RowSetFactory que es creada desde RowSetProvider. Ejemplo:

```
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
```

Tendrá las misma propieaddes por defecto que tenía un JdbcRowSet y además tiene una instancia por defecto de **SyncProvider** de la clase **RIOptimisticProvider**. El objeto SyncProvider hace el papel de **RowSetReader** (un reader) y un **RowSetWriter** (un writer), para leer y escribir datos de su fuente de datos. Estos objetos trabajan en el fondo.

Definir propiedades de conexión

En principio, todo lo comentado para JdbcRowSet es aplicable para CachedRowSet. Ejemplo:

```
public void setPropiedadesdeConexion(String u, String passwd) {
    crs.setUsername(u);
    crs.setPassword(passwd);
    crs.setUrl("jdbc:mySubprotocol:mySubname");
    // ...
```

Y si la fuente es una BD puedes usar la propiedad command para indicar la sentencia SQL a ejecutar. Ejemplo:

```
crs.setCommand("select * from empleado");
```

Columnas clave

Si actualizas los datos y quieres volcarlos luego a la BD, deberís definir las columnas que forman la clave de las filas para mantener la capacidad de idenficación de cada fila. La siguiente línea de código fija la clave primaria a la primera columna (aunque pueden ser varias):

```
int[] keys = {1};
crs.setKeyColumns(keys);
```

El papel de los Reader

Cuando se invoca al método execute(), un objeto Reader del RowSet disconectado puebla con datos al objeto volviéndose a conectar con la fuente de datos. El objeto SyncProvider (de tipo RIOptimisticProvider) aporta por defecto un lector que obtiene la conexión usando las propieades username, password y JDBC URL o el datasourcename. Ejecuta la consulta, rellena los datos del objeto y vuelve a cerrar la conexión.

Operaciones sobre los datos

Actualizar datos en el CachedRowSet es similar a hacerlo en un JdbcRowSet. Por ejemplo, este trozo de código incrementa en 1 el stock del producto 1 en la tabla inventario.

```
while ( crs.next() ) {
   System.out.println("Encontrado producto " + crs.getInt("id") );
   if (crs.getInt("id") == 1) {
      int cantidad = crs.getInt("stock") + 1;
      System.out.println("Modificando stock a " + cantidad );
      crs.updateInt("stock", cantidad );
      crs.updateRow();
      crs.acceptChanges(con); // Syncing fila a BD -> commit
   }
} // fin while
```

Para insertar filas ocurre lo mismo que en el JdbcRowwSet:

```
crs.moveToInsertRow();
crs.updateInt("id", 9);
crs.updateString("stock", 7);
crs.insertRow();
```

Y para borrar es similar. Ejemplo, borrar el id 9:

```
while (crs.next()) {
    if (crs.getInt("id") == 9) {
        crs.deleteRow();
        break;
    }
}
```

Actualizar los cambios en la fuente de datos desconectada

Un objeto CachedRowSet al que se modifican sus datos sin estar conectada, hay que indicarle si los cambios queremos aplicarlos a la fuente de datos (como hacer un commit) o descartarlos (como hacer un rollback). El CachedRowSet mantiene ambos datos (los originales y los modificados) en memoria. Para aplicar los cambios en la fuente de datos hay que invocar al método **acceptChanges()**. Este método provoca que el CachedRowSet se contecte a la fuente de datos y envíe los cambios realizados. Si los cambios fallan por algún motivo se produce una SQLException.

Además el CachedRowSet proporciona dos métodos que permiten recuperar el valor original de los datos. El primero de ellos es **restoreOriginal()**, que devuelve al rowset los valores que había antes de hacer los cambios. El segundo método, **cancelRowUpdates()**, deshace los cambios que se han realizado sobre el registro actual.

Al aplicar cambios puede ocurrir que aparezcan conflictos (intentar guardar una fila en la BD que otro usuario modificó o borró por ejemplo). Hay escritores que no comprueban los conflictos, solo escriben los cambios (el caso del RIXMLProvider usado en un

WebRowSet) y hay otros que ponen bloqueos en la BD para evitar que otros cambien las filas. La implementación del CachedRowSet es RIOptimisticProvider porque utiliza un modelo optimista de concurrencia y por tanto es el escritor el que debe hacerse cargo de los posibles conflictos, dado que si hay conflictos, no escribirá ningún cambio.

El **RIOPtimisticProvider** ofrece la opción de que examines los valores en conflicto y decidas cuales deberían hacerse persistentes usando un objeto **SyncResolver**. Si el escritor encuentra conflictos, crea un SyncResolver con los datos que los han causado. El método acceptChanges() lanza una excepción **SyncProviderException**, que la aplicación puede atrapar y usar el SyncResolver. Ejemplo:

```
try {
    crs.acceptChanges();
} catch (SyncProviderException spe) {
    SyncResolver resolver = spe.getSyncResolver();
}
```

El objeto resolver es un RowSet que copia los datos del crs pero solo contiene los valores que han causado problemas dejando a null el resto de columnas. Puedes iterar por el resolver y el crs y compararlos. Por ejemplo:

```
try {
    // ...
    // Syncing nueva fila a la BD
    System.out.println("Tras añadir una fila...");
    crs.acceptChanges(con);
    System.out.println("Fila añadida...");
    this.viewTable(con);
    // ...
} catch (SyncProviderException spe) {
    SyncResolver r = spe.getSyncResolver();
    Object crsV;      // valor en RowSet
    Object resolverV; // valor en SyncResolver
```

```
                Object resolvedV; // valor que persiste
                while ( r.nextConflict() ) {
                    if(r.getStatus() == SyncResolver.INSERT_ROW_CONFLICT){
                        int fila = r.getRow();
                        crs.absolute(fila);
                        int colCont = crs.getMetaData().getColumnCount();
                        for(int j = 1; j <= colCont; j++) {
                            if(r.getConflictValue(j) != null) {
                                crsVal = crs.getObject(j);
                                resolverV = resolver.getConflictValue(j);
                                // Compara crsV y resolverV para decidir
                                // cual debe quedar persistente
                                resolvedV = crsV;
                                r.setResolvedValue(j, resolvedV);
                            }
                        }
                    }
                }
            }
```

Notificando a los Listeners

Ser un componente JavaBeans significa que un objeto RowSet puede notificar a otros componentes cuando ocurran ciertas cosas. El programador debe añadir o eliminar los componentes a los que se avisa. Un listener de un RowSet debe implementar los métodos de la interface **RowSetListener**:

- **cursorMoved**: el cursor de datos se ha movido de fila.
- **rowChanged**: el valor de una columna de una fila ha cambiado.
- **rowSetChanged**: Indica que el RowSet se ha poblado con datos.

EJEMPLO 35: Las aplicaciones cliente (en un móvil/Tablet) recuperan datos de una BD en lotes de 4 filas, trabajan off-line y modifican e insertan una nueva fila y vuelcan los datos de nuevo a la BD. Además enganchamos u RowSetListener para ver los eventos que se producen.

```
package p10;

import java.net.MalformedURLException;
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import java.sql.Statement;

import javax.sql.RowSetEvent;
import javax.sql.RowSetListener;
import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.RowSetFactory;
import javax.sql.rowset.RowSetProvider;
import javax.sql.rowset.spi.SyncProviderException;
import javax.sql.rowset.spi.SyncResolver;

public class DemoCRS {
    private String dbNombre;
    private Connection con;
    private String dbms;

    public DemoCRS(Connection c, String sgbd, String nombreBD) {
        super();
        this.con = c;
        this.dbNombre = nombreBD;
        this.dbms = sgbd;
    }

    public void testPaginacion() throws SQLException,
MalformedURLException {
        this.con.setAutoCommit(false);
        RowSetFactory factory = RowSetProvider.newFactory();
        CachedRowSet crs = factory.createCachedRowSet();
        try {
            crs.setUsername("userdao");
            crs.setPassword("UserDAO_1");
            crs.setUrl("jdbc:" + dbms + "://localhost:3306/"+ dbNombre +
                       "?serverTimezone=UTC");
            crs.setCommand("select * from inventario");
            // Poner tamaño de pagina a 4 para obtener 4 filas cada vez
            crs.setPageSize(4);
            crs.execute(); // Obtener el primer conjunto de datos
```

```
            crs.addRowSetListener( new DemoRSL() );
            // Keep on getting data in chunks until done.
            int i = 1;
            do {
                System.out.println("Página número: " + i);
                while (crs.next()) {
                    System.out.println("Elemento "
                                       + crs.getInt("productoID")
                                       +  ": " + crs.getInt("stock"));
                    if (crs.getInt("productoID") == 1) {
                        int actual = crs.getInt("stock") + 1;
                        System.out.println("Modificando cantidad a " +
                                           actual);
                        crs.updateInt("stock", actual);
                        crs.updateRow();
                        crs.acceptChanges(con); // Syncing con la BD
                    }
                } // fin while interno
                i++;
            } while (crs.nextPage()); // fin bucle externo
            // Insertar nueva fila y volver a página anterior
            int nuevoId = (int)(Math.random() * 100);
            if (this.existeFila(nuevoId)) {
                System.out.println("Elemento ID " + nuevoId +
                                   " ya existe");
            } else {
                crs.previousPage();
                crs.moveToInsertRow();
                crs.updateInt("productoID", nuevoId);
                crs.updateInt("stock", 1);
                crs.insertRow();
                crs.moveToCurrentRow();
                System.out.println("Tras insertar la fila...");
                crs.acceptChanges(con);     // Syncing
                System.out.println("Fila añadida...");
                verTabla(con);
            }
        } catch (SyncProviderException spe) {
            SyncResolver resolver = spe.getSyncResolver();
            Object crsV; // value in the RowSet object
            Object resolverV; // value in the SyncResolver object
            Object resolvedV; // value to be persisted
            while (resolver.nextConflict()) {
```

```
            if (resolver.getStatus()== SyncResolver.INSERT_ROW_CONFLICT){
                int row = resolver.getRow();
                crs.absolute(row);
                int colCount = crs.getMetaData().getColumnCount();
                for (int j = 1; j <= colCount; j++) {
                    if (resolver.getConflictValue(j) != null) {
                        crsV = crs.getObject(j);
                        resolverV = resolver.getConflictValue(j);
                        // Compara crsV y resolverV
                        resolvedV = crsV; // Se elige el del CachedRowSet
                        resolver.setResolvedValue(j, resolvedV);
                    } // if
                }  // for
            } // if
        } // while
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    } finally {
            if (crs != null) crs.close();
        con.setAutoCommit(true);
    }
}

private boolean existeFila(int id) throws SQLException {
 String query="select productoID from inventario where productoID = ?";
    try (PreparedStatement ps = con.prepareStatement(query)){
        ps.setLong(1, id);
        if (ps.execute(query)) return true;
    } catch (SQLException e) {
       // Si hay error es porque no devuelve fila
    }
    return false;
}

public static void verTabla(Connection con) throws SQLException {
    String query = "select * from inventario";
    try (Statement s = con.createStatement()){
        ResultSet rs = s.executeQuery(query);
        while (rs.next()) {
            System.out.println("Inventario: (id: " +
                               rs.getInt("productoID") + ", stock: " +
                               rs.getInt("stock") + ")");
        }
```

```
      } catch (SQLException e) {
          e.printStackTrace();
      }
  }

  public static void main(String[] args) {
      Connection c = null;
      try {
            c = DriverManager.getConnection(
               "jdbc:mysql://localhost:3306/demodao?serverTimezone=UTC",
               "userdao", "UserDAO_1");
          DemoCRS dcrs = new DemoCRS(c, "mysql", "demodao");
          verTabla(c);
          dcrs.testPaginacion();
      } catch (SQLException e) {
          e.printStackTrace();
      } catch (Exception ex) {
          System.out.println("Excepción desconocida...");
          ex.printStackTrace();
      }
      finally {
         try { c.close(); } catch(Exception e1) {}
      }
   }
}

class DemoRSL implements RowSetListener {

         public void rowSetChanged(RowSetEvent event) {
           System.out.println("Cambian filas del objeto
(rowSetChanged)");
         }

         public void rowChanged(RowSetEvent event) {
           System.out.println("Cambian datos en la fila (rowChanged)");
         }

         public void cursorMoved(RowSetEvent event) {
           System.out.println("Se mueve de fila (cursorMoved)");
         }
}
```

USAR LA CLASE WebRowSet

Así como su clase padre, CachedRowSet, los objetos instanciados a partir de una clase WebRowSet pueden ser serializados y enviados a través de la red hacia los clientes. También puede operar con los datos sin necesidad de tener una conexión permanente. Sin embargo, la clase WebRowSet añade otras características de gran utilidad como por ejemplo la capacidad de representar un objeto WebRowSet con XML.

Además de poder generar documentos XML, un objeto WebRowSet puede rellenarse a partir de un fichero XML. Por ejemplo un cliente puede actualizar los datos, generar el fichero XML con los cambios pertinentes y enviarlos a un JSP o un servlet para que actualice los datos con la fuente original.

SOBRE LOS DOCUMENTOS XML DE UN WEBROWSET
Contienen los datos originales, los modificados y descripciones de las columnas. El esquema XML del WebRowSet define lo que contiene el objeto. Tanto el emisor como el recipiente usan este esquema. Como writeXml y readXml hacen el tran¡bajo, el programador no necesita interactuar con este esquema. Los documentos XML contienen elementos y subelementos organizados de forma jerárquica. Los 3 elementos principales del documento XML de un objeto WebRowSet son: properties, metadata y Data.

Los tags de elementos señalizan el inicio y fin de un elemento. Por ejemplo, <properties> indica el comienzo y </properties> indica el final.

Properties
Cuando llamas al método writeXml() producirá un documento XML que describe los datos. Si fuese la lista de stock de los productos:

```
<properties>
  <command>
    select productoID, stock from inventario
  </command>
  <concurrency>1008</concurrency>
  <datasource><null/></datasource>
  <escape-processing>true</escape-processing>
  <fetch-direction>1000</fetch-direction>
```

```
  <fetch-size>0</fetch-size>
  <isolation-level>2</isolation-level>
  <key-columns>
    <column>1</column>
  </key-columns>
  <map>
  </map>
  <max-field-size>0</max-field-size>
  <max-rows>0</max-rows>
  <query-timeout>0</query-timeout>
  <read-only>true</read-only>
  <rowset-type>
    ResultSet.TYPE_SCROLL_INSENSITIVE
  </rowset-type>
  <show-deleted>false</show-deleted>
  <table-name>inventario</table-name>
  <url>jdbc:mysql://localhost:3306/demodao?serverTimezone=UTC</url>
  <sync-provider>
    <sync-provider-name>
      com.sun.rowset.providers.RIOptimisticProvider
    </sync-provider-name>
    <sync-provider-vendor>
      Sun Microsystems Inc.
    </sync-provider-vendor>
    <sync-provider-version>
      1.0
    </sync-provider-version>
    <sync-provider-grade>
      2
    </sync-provider-grade>
    <data-source-lock>1</data-source-lock>
  </sync-provider>
</properties>
```

Si algunas propiedades no tienen valor como el tag <datasource/> (que es una abreviatura de <datasource></datasource>) significa que la propiedad url está fijada pero no la de datasourcename. Ni el nombre de usuario ni el password aparecen porque deben permanecer en secreto.

Metadata

Describen las columnas que definen el contenido del objeto WebRowSet.En el ejemplo:

```
<metadata>
  <column-count>2</column-count>
  <column-definition>
    <column-index>1</column-index>
    <auto-increment>false</auto-increment>
    <case-sensitive>false</case-sensitive>
    <currency>false</currency>
    <nullable>0</nullable>
    <signed>false</signed>
    <searchable>true</searchable>
    <column-display-size>32</column-display-size>
    <column-label>productoID</column-label>
    <column-name>productoID</column-name>
    <schema-name></schema-name>
    <column-precision>32</column-precision>
    <column-scale>0</column-scale>
    <table-name>inventario</table-name>
    <catalog-name>demodao</catalog-name>
    <column-type>12</column-type>
    <column-type-name>INT</column-type-name>
  </column-definition>
  <column-definition>
    <column-index>2</column-index>
    <auto-increment>false</auto-increment>
    <case-sensitive>true</case-sensitive>
    <currency>false</currency>
    <nullable>0</nullable>
    <signed>true</signed>
    <searchable>true</searchable>
    <column-display-size>
      12
    </column-display-size>
    <column-label>STOCK</column-label>
    <column-name>STOCK</column-name>
    <schema-name></schema-name>
    <column-precision>10</column-precision>
    <column-scale>2</column-scale>
    <table-name>inventario</table-name>
    <catalog-name>demodao</catalog-name>
    <column-type>2</column-type>
    <column-type-name>INT</column-type-name>
  </column-definition>
```

```
</metadata>

Data
La sección de datos contine los valores de cada columna del objeto
WebRowSet. Si has hecho cambios en los datos será ligeramente
distinto porque contendrá los valores cambiados

<data>
  <currentRow>
    <columnValue>1</columnValue>
    <columnValue>15</columnValue>
  </currentRow>
  <currentRow>
    <columnValue>2</columnValue>
    <columnValue>10</columnValue>
  </currentRow>
</data>
```

Tener los datos en formato XML ofrece bastante flexibilidad cuando necesitamos representarlos con diferentes dispositivos y navegadores. Se puede utilizar una plantilla XSL para formatear el documento XML y representar los datos en clientes ligeros como pueden ser navegadores o dispositivos móviles.

Normalmente una clase WebRowSet trabaja en entornos HTTP. Los clientes y el servidor intercambian información en formato XML por medio de objetos WebRowSet. En esta arquitectura los clientes solicitan a un servlet la información que este utiliza, una clase WebRowSet para obtener los datos y generar un documento XML que se le envía al cliente. El cliente utiliza un WebRowSet local para rellenar los datos. Si se modifica algún valor el cliente vuelve a generar un documento XML con los cambios y se lo envía al servlet que este aplica los cambios con la fuente de datos.

Crear y poblar Objetos WebRowSet

Se crean usando una instancia de RowSetFactory que se crea a su vez

desde la clase RowSetProvider.

```
RowSetFactory f = RowSetProvider.newFactory();
try(WebRowSet listaStock = f.createWebRowSet()) {
```

Aunque el objeto listaStock no tiene datos, usamos sus propiedades por defecto como el objeto **BaseRowSet**. Su **SyncProvider** tiene una implementación de **RIOptimisticProvider** aunque lo resetea a un **RIXMLProvider**.

Para escribir un WebRowSet como un documento XML se usa el método **writeXml()** y para leerlo **readXml()**. Ambos realizan el trabajo en el fondo, son inaccesibles para el programador. Reciben un stream que si le puedes pasar y puede ser un OutputStream (como un FileOutputStream, FileWriter, etc.).

Hacer cambios en los datos

Las operaciones de insertar, modificar y borrar filas son similares a como se hacían en el CachedRowSet. Por ejemplo para insertar una fila:

```
listaStock.absolute(3);
listaStock.moveToInsertRow();
listaStock.updateInt("productoID", 9);
listaStock.updateInt("stock", 5);
listaStock.insertRow();
listaStock.moveToCurrentRow();
```

EJEMPLO 36: crear una lista de la tabla inventario, insertar, modificar y borrar una fila en el WebRowSet y volcarla a un fichero XML. Luego leer el fichero en el mismo (u otro) WebRowSet.

```
package p10;

import javax.sql.rowset.RowSetFactory;
import javax.sql.rowset.RowSetProvider;
import javax.sql.rowset.WebRowSet;
```

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.sql.SQLException;

public class DemoWRS {
    private String dbNombre;
    private String dbms;

    public DemoWRS(String sgbd, String nombreBD) {
        super();
        this.dbNombre = nombreBD;
        this.dbms = sgbd;
    }

    public void testWebRowSet() throws SQLException, IOException {
       String ficheroInventario = "inventario.xml";
       RowSetFactory factory = RowSetProvider.newFactory();
       try (WebRowSet wrs = factory.createWebRowSet();
          FileWriter fWriter = new FileWriter(ficheroInventario);
          FileReader fReader = new FileReader(ficheroInventario);) {
         int[] keyCols = {1};
         wrs.setUsername("userdao");
         wrs.setPassword("UserDAO_1");
         wrs.setUrl("jdbc:" + dbms + "://localhost:3306/"+
                    dbNombre + "?serverTimezone=UTC");
         wrs.setCommand("select productoId, stock from inventario");
         wrs.setKeyColumns(keyCols);
         wrs.execute(); // Poblar de datos
         System.out.println("Filas en el WebRowSet: " + wrs.size());
         // Insertar fila
         wrs.moveToInsertRow();
         wrs.updateInt("productoId", (int)(Math.random()*100));
         wrs.updateInt("stock", 1);
         wrs.insertRow();
         wrs.moveToCurrentRow();
         System.out.println("Insertado nuevo producto");
         System.out.println("Filas en el WebRowSet: " + wrs.size());
         wrs.beforeFirst();
         while (wrs.next()) {
             if (wrs.getInt(1) == 1) {
                 System.out.println( "Borrando fila del producto 1..." );
                 wrs.deleteRow();
```

```
                break;
            }
        }
        // Modificar stock del producto 2
        wrs.beforeFirst();
        while (wrs.next()) {
            if (wrs.getInt(1) == 2) {
                System.out.println("Modificando producto 2...");
                wrs.updateInt(2, 3);
                wrs.updateRow();
                break;
            }
        }
        int size1 = wrs.size();
        wrs.writeXml(fWriter);
        fWriter.flush();
        fWriter.close();
        // Crear otro objeto WebRowSet destino
        // Leería del fichero y vuelca datos en BD
        // Aquí aprovechamos el mismo (es un ejemplo)
        // wrs.setUrl(""); wrs.setUsername(""); wrs.setPassword("");
        // Leer el fichero XML
        wrs.close();            // Borramos los datos del RowSet
        wrs.readXml(fReader);  // Los cargamos del fichero
        int size2 = wrs.size();
        if (size1 == size2) {
      System.out.println( "WebRowSet serializado y deserialiazado OK.");
            // Podría volcarlo a una BD...
       } else {
           System.out.println("Error al serializar/deserializar");
       }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    try {
        DemoWRS dwrs = new DemoWRS("mysql", "demodao");
        dwrs.testWebRowSet();
    } catch (SQLException e) {
        e.printStackTrace();
    } catch (Exception ex) {
```

```
            System.out.println("Error inesperado...");
            ex.printStackTrace();
        }
    }
}
```

Al ejecutarse y el fichero XML donde se serializa el WebRowSet:

```
Filas en el WebRowSet: 9
Insertado nuevo producto
Filas en el WebRowSet: 10
Borrando fila del producto 1...
Modificando producto 2...
WebRowSet serializado y deserialiazado OK.
```

Node	Content
xml	version="1.0"
webRowSet	(properties, metadata, data)
xmlns	http://java.sun.com/xml/ns/jdbc
xmlns:xsi	http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation	http://java.sun.com/xml/ns/jdbc http://java.sun.c
properties	(command, concurrency, datasource, escape-pro
metadata	(column-count, (column-definition*))
data	(currentRow*, insertRow*, deleteRow*, modifyRo
insertRow	(columnValue | updateValue)*
deleteRow	(columnValue, updateValue)*
currentRow	(columnValue)*
columnValue	2
columnValue	15
updateRow	3
currentRow	(columnValue)*

Design Source

USANDO OBJETOS JoinRowSet

Un JoinRowSet permite crear un SQL JOIN entre objetos RowSet

cuando no estén conectados a la fuente de datos. Esto evita la sobrecarga de tener varias conexiones y descarga de trabajo al servidor de bases de datos. La interface JoinRowSet es una subinterface de CachedRowSet, por tanto desconectado.

Creaando Objetos JoinRowSet

El siguiente ejemplo crea un JoinRowSet usando la tabla empleado de la que vemos sus datos actuales y una nueva tabla tareas que creamos y poblamos en el sgbd con estas sentencias:

```
mysql> select * from empleado;
+-----+------------+
| id  | nombre     |
+-----+------------+
| 601 | Pepe Lopez |
| 603 | Pepe Lopez |
+-----+------------+
2 rows in set (0.00 sec)
```

```
mysql> select * from tareas;
+---------+-------------+--------+-------+
| tareaID | responsable | inicio | horas |
+---------+-------------+--------+-------+
|       4 |         601 | NULL   |    30 |
|       5 |         601 | NULL   |    20 |
|       6 |         603 | NULL   |    15 |
+---------+-------------+--------+-------+
3 rows in set (0.00 sec)
```

```
create table tareas(
tareaID int not null auto_increment,
responsable int not null,
inicio datetime,
horas double default (curdate()),
constraint tareas_pk primary key(tareaID),
constraint tareas_fk foreign key(responsable) references empleado(id)
);

insert into tareas(responsable, horas)
values (601, 30), (601, 20), (603, 15);
```

El código que crea el JoinRowSet:

```
RowSetFactory factory = RowSetProvider.newFactory();
try(CachedRowSet empCRS = factory.createCachedRowSet();
    CachedRowSet tarCRS = factory.createCachedRowSet();
    JoinRowSet jrs = factory.createJoinRowSet()) {
     empCRS.setCommand("select * from empleado");
     tarCRS.setCommand("select * from tareas");
     // Fijas parámetros de los CachedRowSet url, username y password
```

```
empCRS.execute();
tarCRS.execute();
// ...
```

Añadir Objetos RowSet

Se añaden al JoinRowSet los RowSet que participen en la operación JOIN. Aunque se pueden usar JdbcSet, no suelen utilizarse al estar conectado. Para hacer unir el nombre del empleado a las tareas que teiene encargadas haríamos esta sentencia en el sgbd:

```
String query =
    "select empleado.nombre, tareas.tareaID, tareas.horas " +
    "from empleado join tareas on id = responsable";
```

```
mysql> select nombre, tareaID, horas
    -> from empleado join tareas on id = responsable;
+------------+---------+-------+
| nombre     | tareaID | horas |
+------------+---------+-------+
| Pepe Lopez |       4 |    30 |
| Pepe Lopez |       5 |    20 |
| Pepe Lopez |       6 |    15 |
+------------+---------+-------+
3 rows in set (0.00 sec)
```

Añadimos los RowSets que contienen los datos que queremos unir al JoinRowSet y hacemos la operación desde la aplicación sin neccesidad de reconectarse.

```
jrs.addRowSet(empCRS, "id");
jrs.addRowSet(tarCRS, "responsable");
```

Definir las Columnas que Deben Coincidir

Cuando añades el RowSet, la columna que se usa para realizar la reunión (se llama ***match column***) *se puede indicar en el momento de añadirlo como hemos hecho en el ejemplo anterior*. Ahora solo habría que usar los datos:

```
      System.out.println("Tareas de más de 30 horas y empleado");
      while( jrs.next() ) {
        if(jrs.getDouble("horas") > 30) {
          System.out.println(" Responsable: " + jrs.getString("nombre") +
                             "\n tarea: " + jrs.getInt("tareaID") +
                             "\n horas: " + jrs.getDouble("horas"));
        }
      }
```

La interface JoinRowSet define constantes para indicar el tipo de JOIN que necesitas realizar: `JoinRowSet.INNER_JOIN`.

EJEMPLO 37: realizar un JOIN en la aplicación de las tablas empleado y tareas para mostrar la tarea, el nombre del empleado responsable y las horas de la tarea, de aquellas tareas mayores de 30 horas:

```
package p10;

import java.util.Scanner;

import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.JoinRowSet;
import javax.sql.rowset.RowSetFactory;
import javax.sql.rowset.RowSetProvider;
import java.sql.SQLException;

public class DemoJRS{
  private String dbNombre;
  private String dbms;

  public DemoJRS(String sgbd, String nombreBD) {
      super();
      this.dbNombre = nombreBD;
      this.dbms = sgbd;
  }

  public void testJRS(double horas) throws SQLException {
    RowSetFactory factory = RowSetProvider.newFactory();
    try(CachedRowSet empCRS = factory.createCachedRowSet();
        CachedRowSet tarCRS = factory.createCachedRowSet();
        JoinRowSet jrs = factory.createJoinRowSet()) {
        empCRS.setCommand("select id, nombre from empleado");
```

```
        empCRS.setUsername("userdao");
        empCRS.setPassword("UserDAO_1");
        empCRS.setUrl("jdbc:" + dbms + "://localhost:3306/"+ dbNombre +
                      "?serverTimezone=UTC");
        empCRS.execute();
        tarCRS.setCommand("select tareaID, responsable, horas from
tareas");
        tarCRS.setUsername("userdao");
        tarCRS.setPassword("UserDAO_1");
        tarCRS.setUrl("jdbc:" + dbms + "://localhost:3306/"+ dbNombre +
                    "?serverTimezone=UTC");
        tarCRS.execute();
        // Añadir RowSet que participan en el JOIN
        jrs.addRowSet(empCRS, "id");
        jrs.addRowSet(tarCRS, "responsable");
        // Mostrar resultados
    System.out.println("Tareas de más de " + horas + " y responsable:");
        while(jrs.next()) {
            double h = jrs.getDouble("horas");
            if(h >= horas) {
                System.out.println("\n Tarea: " + jrs.getInt("tareaID") +
                                   "\n Responsable: " + jrs.getString("nombre") +
                                   "\n horas: " + h);
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
  }

  public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      try {
          System.out.print("Tareas de duración superior a (horas): ");
          double horas = sc.nextDouble();
          DemoJRS djrs = new DemoJRS("mysql", "demodao");
          djrs.testJRS(horas);
      } catch (SQLException e) {
         e.printStackTrace();
      } finally {
         sc.close();
      }
  } // fin main
```

```
} // fin clase
```

```
Tareas de duración superior a (horas): 20
Tareas de más de 20.0 y responsable:

 Tarea: 5
 Responsable: Pepe Lopez
 horas: 20.0

 Tarea: 4
 Responsable: Pepe Lopez
 horas: 30.0
```

USANDO OBJETOS FilteredRowSet

Un objeto FilteredRowSet permite filtrar la cantidad de filas visibles en el RowSet. Debes definir el filtro y aplicarlo y se realiza desconectado de la fuenta de datos. Es como usar la cláusula WHERE de una consulta SQL.

Definir Criterio de Filtrado en Objetos Predicado

Para fijar el criterio de filtrado de filas debes definir un objeto que implemente la interface **Predicate**. Debe tener:

- El valor que hace de límite superior.
- El valor que hace de límite inferior.
- El nombre o número de la columna a los que se aplica los límites.

El rango de valores es inclusivo. Ejemplo de implementación de un Predicate:

```
public class FiltroHoras implements Predicate {
    private int bajo;
    private int alto;
    private String colNombre = null;
    private int colNumero = -1;

    public FiltroHoras(int b, int a, int cn) {
        this.bajo = b;
```

```
        this.alto = a;
        this.colNumero = cn;
    }

    public FiltroHoras(int b, int a, String cn) {
        this.bajo = b;
        this.alto = a;
        this.colNombre = cn;
    }

    public boolean evaluate(Object valor, String cn) {
        if (cn.equalsIgnoreCase(this.colNombre)) {
            int colValor = ((Integer)valor).intValue();
            return colValor >= this.bajo && colValor <= this.alto;
        }
        return true;
    }

    public boolean evaluate(Object valor, int cn) {
        if (this.colNumero == cn) {
            int colValor = ((Integer)valor).intValue();
            return colValor >= this.bajo && colValor <= this.alto;
        }
        return true;
    }

    public boolean evaluate(RowSet rs) {
        CachedRowSet frs = (CachedRowSet)rs;
        boolean e = false;
        try {
            int colValor = -1;
            if (this.colNumero > 0) {
                colValor = frs.getInt(this.colNumero);
            } else if (this.colNombre != null) {
                colValor = frs.getInt(this.colNombre);
            } else {
                return false;
            }
            if (colValor >= this.bajo && colValor <= this.alto) {
                e = true;
            }
        } catch (SQLException e) {
            e.printStack();
```

```
            return false;
        } catch (NullPointerException npe) {
            System.err.println("NullPointerException");
            return false;
        }
        return e;
    }
}
```

Para instanciarla:

```
       FiltroHoras fh = new FiltroHoras(10, 20, 3);
```

Observa que solo se aplica a una columna (la número 3 en este caso) pero es posible aplicarlo a varias creando un arrays:

```
public Filtro2(Object[] bajo, Object[] alto, Object[] cn) {
    this.bajo = bajo;
    this.alto = alto;
    this.colNumero = colNumber;
}
```

El número de elementos en los 3 arrays debería ser el mismo. En este caso la implementación del método evaluate debería ser algo así:

```
public boolean evaluate(RowSet rs) {
    CachedRowSet crs = (CachedRowSet)rs;
    boolean bool1;
    boolean bool2;
    for (int i = 0; i < colNumero.length; i++) {
        if (rs.getObject(colNumero[i] >= bajo[i] &&
            rs.getObject(colNumero[i] <= alto[i]) {
            bool1 = true;
        } else {
            bool2 = true;
        }
        if (bool2) {
            return false;
        } else {
            return true;
        }
    }
```

```
}
```

Crear Objetos FilteredRowSet

De forma similar a los ejemplos anteriores:

```
RowSetFactory factory = RowSetProvider.newFactory();
try (FilteredRowSet frs = factory.createFilteredRowSet()) {
  // ...
```

Como el resto de RowSet desconectados hay que poblarlos a partir de la fuente de datos definiendo los parámetros de conexión:

```
frs.setCommand("select...");
frs.setUsername();
frs.setPassword();
frs.setUrl();
frs.execute();
// ...
```

Crear y Fijar el Predicado

Ahora hay que crear y configurar un Predicate:

```
Filtro1 f1 = new Filtro1(10, 20, 3);
frs.setFilter(f1);
```

Cuando se ejecute el método next() se llamará al método evaluate() del predicado que hayas fijado. Si devuelve true, la fila será visible, en caso contrario continuará el proceso con la siguiente fila.

Puedes configurar varios predicados de forma lineal. Es decir, aplicas el primero y llamas a next(). Luego aplicas el segundo y vuelves a llamar a next().

Modificar datos de filas

Puedes hacer inserciones, modificaciones y borrados de las filas siempre y cuando no violen los criterios de filtrado.

10.5.4 SERIALIZAR Y DESERIALIZAR OBJETOS EN SQL.

Serializar objetos aporta al programa la capacidad de leer y escribir objetos desde/hacia un stream de bytes. Eso hace posible que puedan ir a un disco duro o viajar por una red de unos programas a otros. Para conseguirlo, la clase del objeto debe implementar la interface java.io.Serializable, o Externalizable (tienes más opcionaes para personalizar el proceso). La interface Externalizable define dos métodos:

- **void readExternal(ObjectInput in)**: restaura el contenido de un objeto llamando a los métodos DataInput para tipos primitivos y readObject para objetos.
- **void writeExternal(ObjectOutput out)**: guarda los contenidos de un objeto llamando a los métodos DataOutput para sus tipos primitivos y ObjectOutput para objetos.

En el JDK, la serialización se puede realizar usando 3 cosas:

- Implementando java.io.Serializable.
- Usando ObjectOutputStream.
- Usando ObjectInputStream.

EJEMPLO 37: Serializar y deserializar Objectos Java a Ficheros.

```
FileOutputStream out = new FileOutputStream("myFile.ser");
ObjectOutputStream stream = new ObjectOutputStream(out);
stream.writeObject("my string");
stream.writeObject(new Date());
stream.flush();
// Deserializar
FileInputStream in = new FileInputStream("myFile.ser");
ObjectInputStream stream = new ObjectInputStream(in);
String myString = (String) stream.readObject();
Date date = (Date) stream.readObject();
```

También tienes que tener en cuenta que si quieres usar el modelo

relacional deberás implementar funciones que transformen uno de tus objetos en filas de una o varias tablas del modelo relacional y viceversa.

Si la BD soporta el modelo lógico objeto-relacional, el panorama puede cambiar. Mira este ejemplo para Oracle:

EJEMPLO 38 EN ORACLE: La serialización y deserialización de un objeto tiene lugar automáticamente con los métodos writeSQL() y readSQL().

```
       CREATE TYPE empleado AS OBJECT (
            nombre VARCHAR2(50),
            num    INTEGER,
       );
```

```
import java.sql.*;
import oracle.jdbc2.*;

public class Empleado implements SQLData {
    private String sql_type;
    public String empName;
    public int num;

    public Empleado(){  }

    public Empleado(String sql_tipo, String nombre, int num){
           this.sql_tipo = sql_tipo;
           this.nombre = nombre;
           this.num = num;
    }

  public String getSQLTipo() throws SQLException { return sql_tipo; }
  public void readSQL(SQLInput stream, String tipo) throws SQLException {
      sql_tipo = tipo;
           nombre = stream.readString();
           num = stream.readInt();
         }
         public void writeSQL(SQLOutput stream) throws SQLException   {
           stream.writeString(nombre);
```

```
            stream.writeInt(num);
        }
    }
```

9.6. PERSITENCIA DE OBJETOS CON ORM.

Muchas aplicaciones Java necesitan tener datos persistentes. En la mayoría de los casos esto significa usar una BD relacional.

El API JDBC y los drivers para la mayoría de los SGBD proporcionan una forma estándar de ejecutar sentencias SQL desde un programa Java. Sin embargo, si tu programa manipula objetos Java se complica conseguir su persitencia por las "**diferencias conceptuales**" entre el modelo de objetos (dominio de la aplicación) y el modelo relacional (usado en la BD relacional).

El modelo de objetos está basado en principios de ingeniería del software y modela los objetos en el dominio del problema, mientras que el modelo relacional está basado en principios matemáticos y organiza los datos para un almacenamiento y recuperación eficientes.

Ninguno de estos modelos es mejor que el otro, el problema es que son diferentes y no siempre se acoplan de forma confortable en la misma aplicación.

El término ORM traducido sería **mapeo objeto relacional** y se refiere a la técnica de mapear (transformar) datos de una representación de modelo de objetos (objetos Java) a una representación de modelo de datos relacional y viceversa.

Hibernate es uno de los frameworks ORM que implementa la especificación JPA y el más popular usado en aplicaciones de tipo empresarial Java.

Paquete javax.persistence

La API de persistencia de Java proporciona a los desarrolladores de Java una función de mapeo de objetos / relacionales para gestionar datos relacionales en aplicaciones Java. Documento oficial de Java:

- https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html

Documento JPA API:

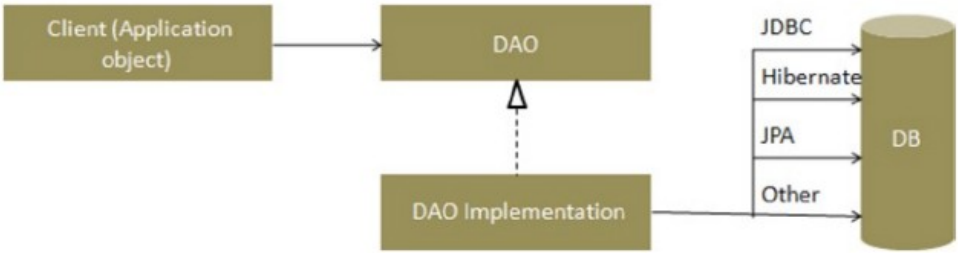
- http://docs.jboss.org/hibernate/jpa/2.2/api/overview-summary.html

9.6.1. USANDO EL PATRÓN DAO Y POJOS.

Cuando tengamos una aplicación que necesite dar persistencia a sus objetos usando una BDR podemos crear una capa dedicada a proporcionarla a la que se denomina **Data Access Layer**.

Será responsable de hacer transparente al código de la aplicación (su cliente) del mecanismo de persistencia que se utilice para almacenar y recuperar objetos entre la aplicación y la BD. Es decir, hace de interfaz.

Esta transparencia implica que esta capa podrá cambiar el mecanismo de almacenamiento, usando por ejemplo JDBC o tecnologías ORM como Hibernate, etc. sin afectar al cliente (el resto del código de la aplicación). Esta transparencia se consigue siguiendo el patrón de diseño **Data Access Object (DAO)** que se muestra en la figura.



El objeto DAO aporta una interface con la BD ocultando a la aplicación el mecanismo de persistencia que está utilizando, abstrayendo e independizando el código del cliente del mecanismo de persistencia.

Aquí tienes varios enlaces que puedes consultar:

- http://en.wikipedia.org/wiki/Object-relational_mapping
- www.hibernate.org/
- www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

Vamos a crear un proyecto que utilice este patrón a modo de ejemplo.

Creamos esta BD conectándonos a mysql como root y un usuario llamado userdao para trabajar en ella. Puedes copiar las sentencias que crean la BD, el usuario, le dan permisos, crea las tablas y las puebla con datos:

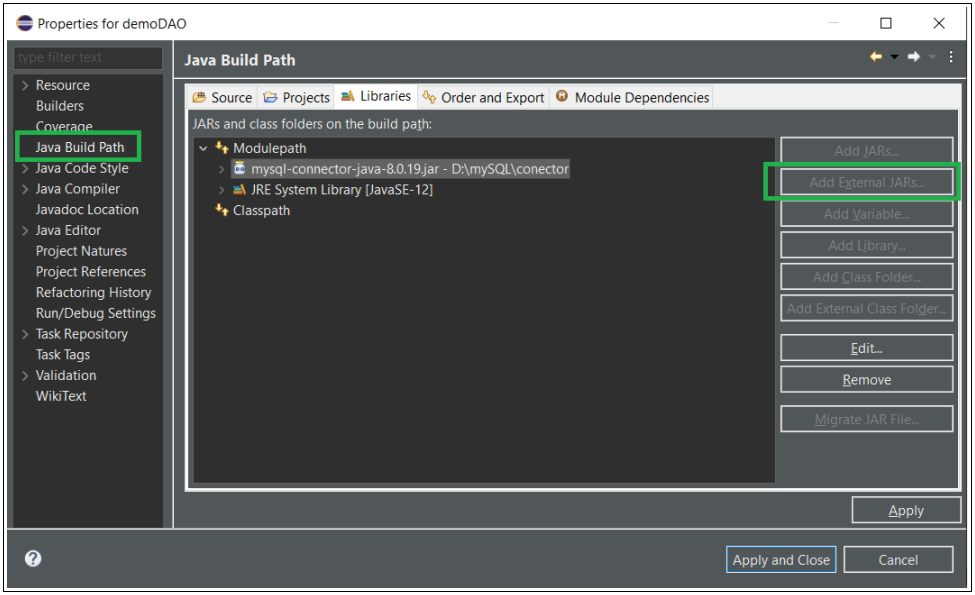
```
mysql -u root -p
create database demodao;
create user 'userdao'@'%' IDENTIFIED BY 'UserDAO_1';
grant all privileges on demodao.* TO 'userdao'@'%';
flush privileges;
use demodao;

create table categorias (
categoriaID int not null auto_increment,
descripcion varchar(20) not null,
primary key(categoriaID)
);

create table libros (
libroID int not null auto_increment,
```

```
categoria int not null,
titulo varchar(60) not null,
editorial varchar(60),
autores varchar(60),
precio numeric(8,2),
primary key(libroID),
constraint fk_libCat foreign key(categoria) references
categorias(categoriaID)
);

insert into categorias(descripcion) values ('Clojure');
insert into categorias(descripcion) values ('Groovy');
insert into categorias(descripcion) values ('Java');
insert into categorias(descripcion) values ('Kotlin');

insert into libros(categoria, titulo, editorial, autores, precio)
values
(1, 'Practical Clojure', 'Apress', 'Luke VanderHart', 44.67),
(2, 'Beginning Groovy, Grails&Griffon', 'Apress', 'Vishal Laika', 39.47),
(3, 'Java EE 8 Recipes'2Ed, 'Apress', 'Josh Juneau', 37.5),
(4, 'Kotlin for Android Development', '', 'Antonio Leiva', 29.11),
(4, 'Kotlin in Action', 'Manning', 'Dimity jemerov, Svetlana Isakova',
27.97);
commit;
```

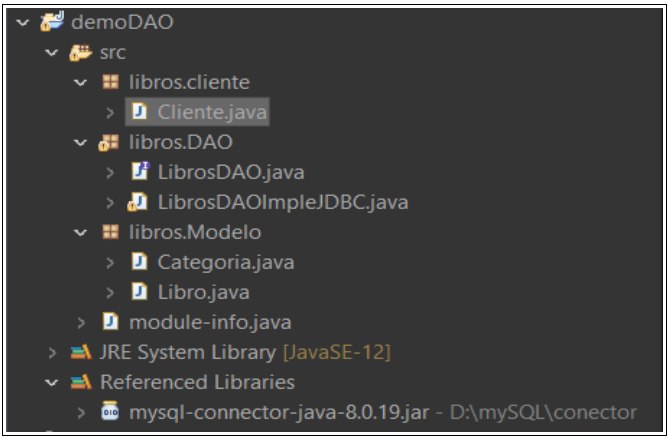
Construimos el proyecto Java en eclipse, creamos un paquete llamada libros bajo la carpeta src. Dentro de él creamos 3 nuevos paquetes:

- libros.Modelo: Contiene las clases de los objetos de nuestra aplicación (los POJOS). En este caso las clases Libro y Categoria.
- libros.DAO que contendrá la capa de acceso a datos, es decir, una interface llamada LibrosDAO y una clase que la implementa usando JDBC que se llama LibrosDAOImpleJDCB.
- libros.Cliente: contiene la aplicación que queremos desacoplar de la capa de acceso a datos.

Añadimos el jar del conector JDBC de mySQl al proyecto en mi caso modificando las opciones del proyecto e indicando un jar externo en la opción Java Build Path -> Libraries



La estructura final será esta.



Vamos a crear una pequeña aplicación cliente que simplemente muestre todos los libros y luego muestre los libros cuya categoría contiene en su descripción unas palabras que indica el usuario. No queremos ver

SQL en este código (desacoplado de la capa de acceso). El código podría ser este:

```java
package libros.cliente;

import java.util.Scanner;
import java.util.List;
import libros.DAO.LibrosDAO;
import libros.DAO.LibrosDAOImpleJDBC;
import libros.Modelo.Libro;

public class Cliente {
    // Cambiando esta sentencia puedes cambiar la tecnología usada
    private static LibrosDAO ld = new LibrosDAOImpleJDBC();

    public static void main(String[] args) {
        Scanner sc = new Scanner( System.in );
        // Listar todos los libros
        System.out.println("Listado de todos los libros:");
        muestraTodosLosLibros();
        // Busca libros cuya categoría contiene una cadena
        System.out.print("Teclee cadena categoría: ");
        String cadena = sc.nextLine();
        System.err.println("Libros cuya categoría contiene <<" + cadena +">> :");
        muestraLibrosCategoria(cadena);
        sc.close();
    }

    private static void muestraTodosLosLibros() {
        List<Libro> lista = ld.todosLosLibros();
        for (Libro libro : lista )
            System.out.println(libro);
    }

    private static void muestraLibrosCategoria(String cadena) {
        List<Libro> lista = ld.librosdeCategoria(cadena);
        for (Libro libro : lista )
            System.out.println(libro);
    }
}
```

Dentro del paquete Modelo creamos una clase por cada objeto que vaya a necesitar nuestra aplicación (los POJOS). En nuestro ejemplo Categoria y Libro cuyo código aparece a continuación.

```java
package libros.Modelo;

public class Categoria {
     private Long id;
     private String descripcion;

    public Long getId() { return id;  }
    public void setId(Long id) { this.id = id; }
    public String getDescripcion() { return descripcion; }
    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }
    @Override
    public String toString() {
        return "Categoria = (Id: " + id + ", Descripcion: "
                + descripcion + ")";
    }
}
```

```java
package libros.Modelo;

public class Libro {
     private Long id;
     private Long categoria;
     private String titulo;
     private String autores;
     private String editorial;
     private double precio;

     public Long getId() { return id; }
     public void setId(Long id) { this.id = id; }
     public Long getCategoria() { return categoria;  }
     public void setCategoria(Long categoria) { this.categoria = categoria; }
     public String getTitulo() { return titulo; }
     public void setTitulo(String titulo) { this.titulo = titulo; }
     public String getAutores() { return autores;  }
     public void setAutores(String autores) { this.autores = autores;  }
     public String getEditorial() { return editorial;  }
     public void setEditorial(String editorial) { this.editorial = editorial; }
     public double getPrecio() { return precio; }
     public void setPrecio(double precio) { this.precio = precio; }
     @Override
     public String toString() {
         return "Libro= (Id: " + id  + ", Categoria: " + categoria + ", Titulo: "
                 + titulo + ", Autores: " + autores
                 + ", Editorial: " + editorial + ", Precio: " + precio + ")";
     }
}
```

Ahora creamos dentro del paquete libros.DAO vamos a implementar la interface LibrosDAO usando JDBC.

```java
package libros.DAO;

import java.util.List;
import libros.Modelo.Libro;
import libros.Modelo.Categoria;

public interface LibrosDAO {
    public List<Libro>todosLosLibros();
    public List<Libro>librosdeCategoria(String categoria);
    public List<Categoria>todasLasCategorias();
    public void insert(Libro libro);
    public void update(Libro libro);
    public void delete(Long libroId);
}
```

Como nuestra aplicación usará esta interface, si queremos cambiar la clase que se encarga de interactuar con la BDR a otra que lo haga con otra tecnología, nos será más sencillo hacerlo, porque el código tiene un acoplamiento muy bajo.

Como vamos a usar JDBC debemos modificar el módulo para indicar que necesitamos el módulo sql:

```
module demoDAO {
    requires java.sql;
}
```

Ahora tenemos que implementar en libros.DAO la interface. Voy a dividir el código en varias partes para ir comentándolo.

En el primer fragmento aparecen los imports, la definición de la clase y un código estático que carga el driver cuando se carga la clase. Además crea el método para obtener una conexión con la BD y otro

para cerrarlo.

```java
package libros.DAO;

import java.util.List;

import libros.Modelo.Categoria;
import libros.Modelo.Libro;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;

public class LibrosDAOImpleJDBC implements LibrosDAO {

    static {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException ex) {}
    }

    private Connection getConexion() throws SQLException {
        System. out.println( "Conectando con la Base de datos..."  );
        String jdbcURL= "jdbc:mysql://localhost:3306/demodao"  +
                        "?useUnicode=true" +
                        "&useJDBCCompliantTimezoneShift=true" +
                        "&useLegacyDatetimeCode=false" + "&serverTimezone=UTC";
         return DriverManager.getConnection(jdbcURL,"userdao", "UserDAO_1");

    }

    private void cierraConexion(Connection conexion) {
        if (conexion == null) return;
        try {
            conexion.close();
        } catch (SQLException ex) { }
    }
```

En el siguiente fragmento aparece el método que hace el listado de

todos los libros:

```java
@Override
public List<Libro> todosLosLibros() {
    List<Libro> listaLibros = new ArrayList<>();
    String sql = "select libroID, titulo, autores, categoria, editorial, precio "
                + "from libros order by titulo";
    Connection conexion = null;
    try {
        conexion = getConexion();
        PreparedStatement ps = conexion.prepareStatement(sql);
        ResultSet rs = ps.executeQuery();
        while ( rs.next() ) {
            Libro libro = new Libro();
            libro.setId( rs.getLong("libroID") );
            libro.setTitulo( rs.getString("titulo") );
            libro.setCategoria( rs.getLong("categoria") );
            libro.setAutores( rs.getString("autores") );
            libro.setEditorial( rs.getString("editorial") );
            libro.setPrecio( rs.getDouble("precio") );
            listaLibros.add(libro);
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
    } finally {
        cierraConexion(conexion);
    }
    return listaLibros;
}
```

Y en los siguientes, el método que crea la lista de libros cuya categoría contiene una cadena de texto en su descripción, el que crea una lista con todas las categorías (no lo usamos) y los de añadir, emodificar y borrar libros que no los implementamos por no hacer más largo el ejemplo.

Después aparecerá una figura con un ejemplo de ejecución.

```java
@Override
public List<Libro> librosdeCategoria(String categoria) {
    List<Libro> listaLibros = new ArrayList<>();
    String sql = "select libroID, titulo, autores, categoria, editorial, precio"
                 + " from categorias join libros on categoriaID = categoria"
                 + " where instr(descripcion, ?) > 0 ";
    Connection conexion = null;
    try {
        conexion = getConexion();
        PreparedStatement ps = conexion.prepareStatement(sql);
        ps.setString(1, categoria);
        ResultSet rs = ps.executeQuery();
        while ( rs.next() ) {
            Libro libro = new Libro();
            libro.setId( rs.getLong("libroID") );
            libro.setTitulo( rs.getString("titulo") );
            libro.setEditorial( rs.getString("editorial") );
            libro.setAutores( rs.getString("autores") );
            libro.setCategoria( rs.getLong("categoria") );
            libro.setPrecio( rs.getDouble("precio") );
            listaLibros.add(libro);
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
    } finally {
        cierraConexion(conexion);
    }
    return listaLibros;
}
```

```java
@Override
public List<Categoria> todasLasCategorias() {
    List<Categoria> lista = new ArrayList<>();
    String sql = "select * from categorias";
    Connection conexion = null;
    try {
        conexion = getConexion();
        PreparedStatement ps = conexion.prepareStatement(sql);
        ResultSet rs = ps.executeQuery();
        while ( rs.next() ) {
            Categoria categoria = new Categoria();
            categoria.setId( rs.getLong("id") );
            categoria.setDescripcion( rs.getString("descripcion") );
            lista.add( categoria );
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
    } finally {
        cierraConexion(conexion);
    }
    return lista;
}
```

```java
    @Override
    public void insert(Libro libro) {
        // TODO Auto-generated method stub

    }

    @Override
    public void update(Libro libro) {
        // TODO Auto-generated method stub

    }

    @Override
    public void delete(Long libroId) {
        // TODO Auto-generated method stub

    }
}
```

```
Listado de todos los libros:
Conectando con la Base de datos...
Libro= (Id: 2, Categoria: 2, Titulo: Beginning Groovy, Grails&Griffon, Autores: Vishal Laika, Editorial: Apress, Precio: 39.47)
Libro= (Id: 3, Categoria: 3, Titulo: Java EE 8 Recipes 2Ed, Autores: Josh Juneau, Editorial: Apress, Precio: 37.5)
Libro= (Id: 4, Categoria: 4, Titulo: Kotlin for Android Development, Autores: Antonio Leiva, Editorial: null, Precio: 29.11)
Libro= (Id: 5, Categoria: 4, Titulo: Kotlin in Action, Autores: Dimity jemerov, Svetlana Isakova, Editorial: Manning, Precio: 27.97)
Libro= (Id: 1, Categoria: 1, Titulo: Practical Clojure, Autores: Luke VanderHart, Editorial: Apress, Precio: 44.67)
Teclee cadena categoría: ava
Libros cuya categoria contiene <<ava>> :
Conectando con la Base de datos...
Libro= (Id: 3, Categoria: 3, Titulo: Java EE 8 Recipes 2Ed, Autores: Josh Juneau, Editorial: Apress, Precio: 37.5)
```

9.6.2. USANDO JPA CON ECLIPSELINK.

A través de JPA, el desarrollador puede almacenar, eliminar, actualizar y recuperar datos de BDR a objetos Java y viceversa. JPA se puede utilizar en aplicaciones Java EE y Java SE, aunque donde se saca más partido es en aplicaciones de tipo EE, vamos a desplegar un ejemplo con SE que es el java que estamos usando.

Algo que se debe tener en cuenta es que JPA define sólo especificaciones (es decir únicamente nos dice lo que se tiene que hacer), pero no proporciona una implementación. La implementación de JPA la proporcionan proveedores como Hibernate, EclipseLink, Apache OpenJPA y frameworks como Spring, etc.

JPA permite al desarrollador trabajar directamente con objetos en lugar de sentencias SQL por lo que no tendremos que escribir absolutamente ninguna en nuestros programas, esto equivale también a rapidez en el desarrollo de aplicaciones Java.

Proveedores de JPA

JPA es una API de código abierto, por lo tanto, varios proveedores como Oracle, Redhat, Eclipse, etc. proporcionan nuevos productos implementando JPA. Algunos de estos productos: Hibernate, EclipseLink, TopLink, Spring Data JPA, etc.

Versiones API JPA

La primera versión de Java Persistence API, JPA 1.0, se lanzó en 2006 como parte de la especificación EJB 3.0. Las siguientes versiones de desarrollo lanzadas bajo la especificación JPA:

- JPA 2.0: Esta versión fue lanzada en el año 2009.
- JPA 2.1: JPA 2.1 se lanzó en el 2013.
- JPA 2.2: JPA 2.2 se lanzó en el 2017.
- JPA 3.0: lanzada en 2019.

¿Ventajas de JPA?

Existen innumerables ventajas de usar JPA en el desarrollo de aplicaciones Java, algunas:

- La carga de interactuar con la BD se reduce significativamente.
- La programación se simplifica al ocultar el mapeo directo desde un modelo Java.
- El coste de crear el archivo de definición (xml) se reduce mediante el uso de anotaciones.
- Podemos fusionar las aplicaciones utilizadas con otros proveedores de JPA.

JPA es la propuesta estándar que ofrece Java para implementar un Framework Object Relational Mapping (ORM), que permite interactuar con la BD por medio de objetos, de esta forma, JPA es el encargado de convertir los objetos Java en instrucciones del SGBD.
Cuando empezamos a trabajar con BD en Java lo primero que aprendemos es a utiliza el API JDBC que nos permite realizar consultas directas a la BD usando SQL nativo. Pero tenemos el problema de que Java es un lenguaje orientado a objetos y se deben convertir los atributos de las clases en una consulta SQL como SELECT, INSERT, UPDATE, DELTE, etc. lo que ocasiona un gran esfuerzo de trabajo y provoca muchos errores en tiempo de ejecución, debido principalmente a que las sentencias SQL se deben generar frecuentemente al vuelo. Puedes ver en este post la diferencia entre JPA y JDBC.

Dentro de las implementaciones más utilizadas están Hibernate, EclipseLink & TopLink, las dos primeras son las más utilizadas en el mundo open source y TopLink es muy utilizada en desarrollos y productos relacionados con Oracle.

En aplicaciones web (que suelen utilizar frameworks para ganar productividad está Spring (https://spring.io/projects/spring-data-jpa), clojure, FX, etc.

Antes de preguntarnos cuál es la diferencia entre todas estas implementaciones tenemos que comprender que en teoría todas debería de ofrecer la misma funcionalidad y el mismo comportamiento, lo que nos permitiría migrar entre una implementación a otra sin afectar en nada nuestra aplicación. Desde luego esto es solo teoría, ya que en la actualidad no todas las implementaciones implementan al 100% la especificación de JPA, además en escenarios muy concretos

puede que se comporten ligeramente diferentes, por lo que puede requerir realizar algunos ajustes antes de migrar correctamente de proveedor.

Java define la API de persistencia en la librería **persistence-api.jar**. Podemos descargarla, por ejemplo, del repositorio de maven http://mvnrepository.com/artifact/javax.persistence/persistence-api. Sin embargo, esta versión de oracle/maven sólo llega a la versión JPA 1.0.2. Si queremos usar JPA versión 2 o superior, debemos bajar alguna de las implementaciones de Hibernate, EclipseLink, etc. En nuestro ejemplo pondremos la de eclipselink, así que podemos bajarnos esta http://mvnrepository.com/artifact/org.eclipse.persistence/javax.persistence

Eclipse SDK SE no tiene soporte para aplicaciones empresariales (Java EE). Para conectarse con MySQL se necesita una API de java que soporte operaciones con datos persistentes y un entorno de trabajo que se encargue de la conexión con BD.

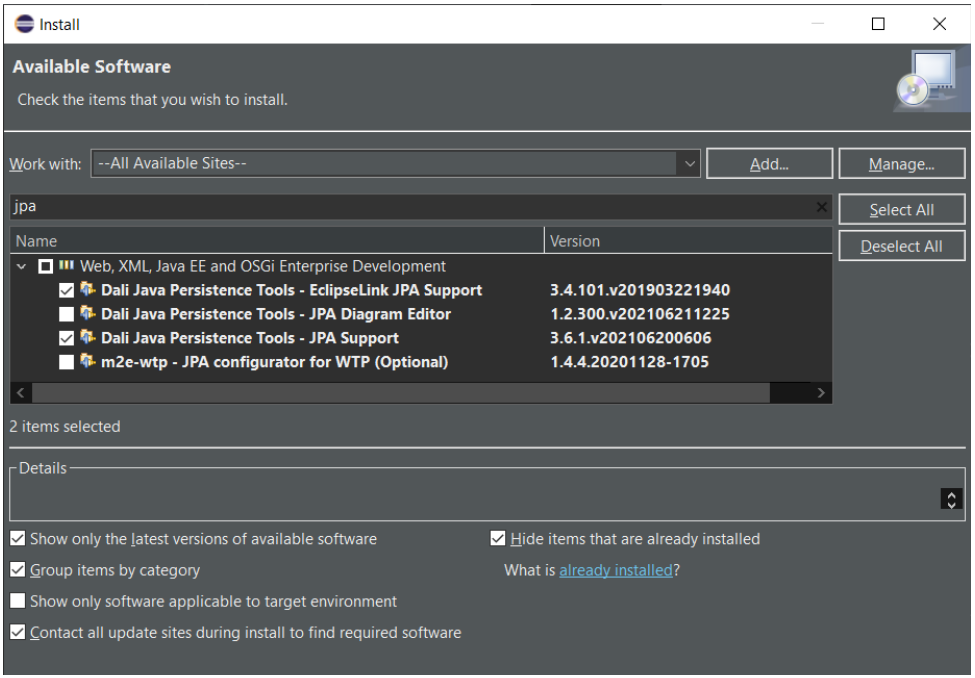
INSTALAR SOPORTE DEL IDE PARA JPA

En este caso instalaremos los paquetes **JPA Support** y el modulo **EclipseLink JPA Support** usando la opción ¨**Help -> Install New Software…**¨. Buscamos **jpa** en todos los sitios y todas las caregorías (puede que tarde un poco sino acotamos más la búsqueda) y seleccionamos **JPA Support**.

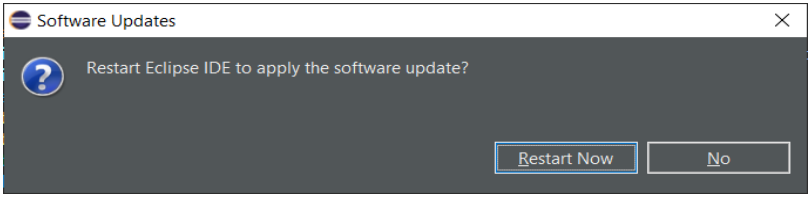
Nos aparecerá la sección "Web, XML, Java EE and OSGi Enterprise Development" porque como ya se ha comentado antes, este tipo de arquitecturas es más ventajosa para grandes aplicaciones empresariales.

Marcamos los paquetes "Dali Java Persistence Tools-EclipseLink JPA

Support" y "Dali Java Persistence Tools-JPA Support" como se ve en la siguiente figura. Aceptamos licencias si es necesario e instalamos.



Ahora habrá que reiniciar el IDE.

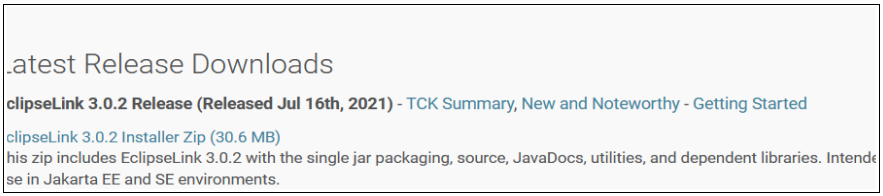


Después de la instalación queda la habilitada la creación de proyectos JPA y se podrá usar el entorno EclipseLink. La configuración no ha terminado porque las librerías propias de EclipseLink no se han

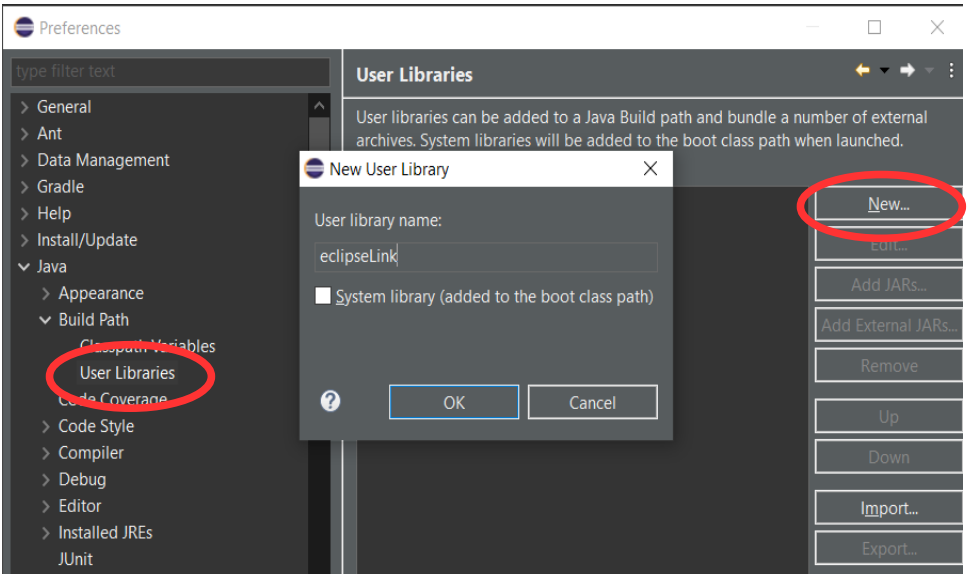
instalado aún.

INSTALAR SOPORTE DEL IDE PARA JPA

Desde el proyecto EclipseLink ofrecen un plugin para Eclipse, en este caso usaremos el paquete ZIP (al momento EclipseLink 3.0.2). Se descarga y descomprime el paquete, puede ser en el workspace de Eclipse o en otra carpeta. El paquete contiene librerías de Java (JAR Files) y para facilitar la inclusión de estas librerías en nuevos proyectos crearemos una **Librería de Usuario** en Eclipse.

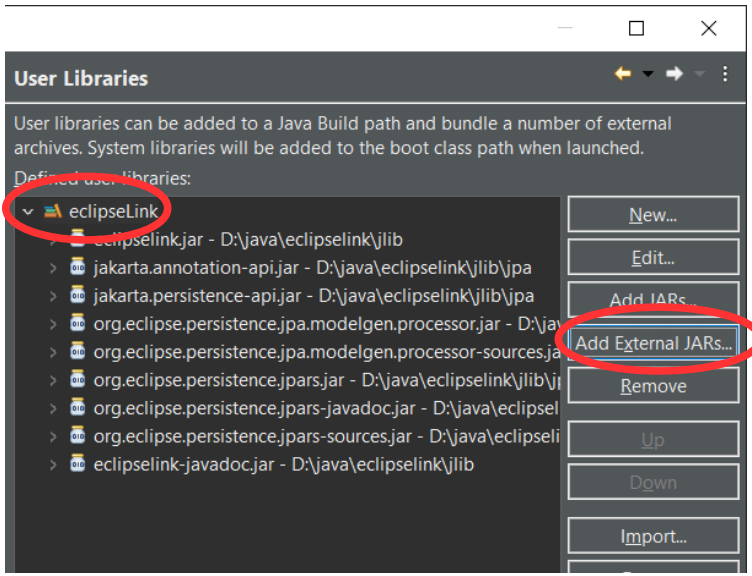


Para ello, como se aprecia en la imagen:



Vamos a la opción ¨**Window -> Preferences**", aparece una ventana con las opciones de Eclipse. En el árbol de secciones vamos a **"Java -> Build Path -> User Libraries"** escogemos **New** y le damos nombre a la nueva librería, por ejemplo "eclipseLink" como aparece en la figura..

En la librería creada se agregaran los ficheros JAR descargados dey descomprimidos de EclipseLink. Usamos la opción **Add External JARs**... del diálogo y navegamos hasta los archivos `./eclipselink/jlib/eclipselink.jar`, y seleccionamos todos los ficheros de `./eclipselink/jlib/jpa/javax.persistence*.jar` o `jakarta.presistence` (en las nuevas versiones) y opcionalmente añadimos la documentacion `./eclipselink/eclipselink-javadocs.zip`.

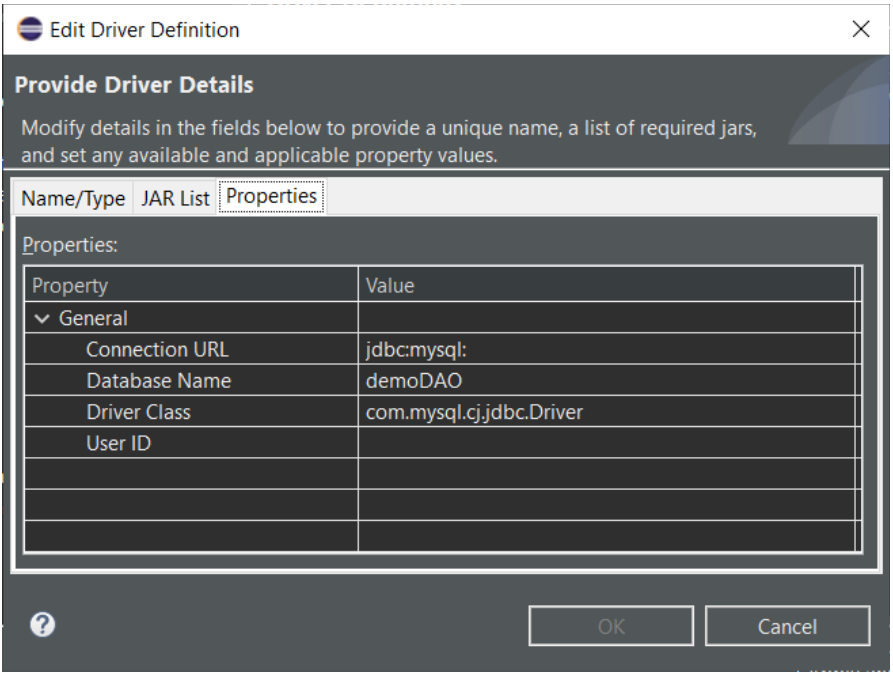


Creamos otra librería de usuario llamada "persistencia" con el paquete jar la librería **persistence-api.jar** que también descargamos desde eclipseLink.

CREAR OTRA LIBRERIA PARA USAR MYSQL CON JDBC

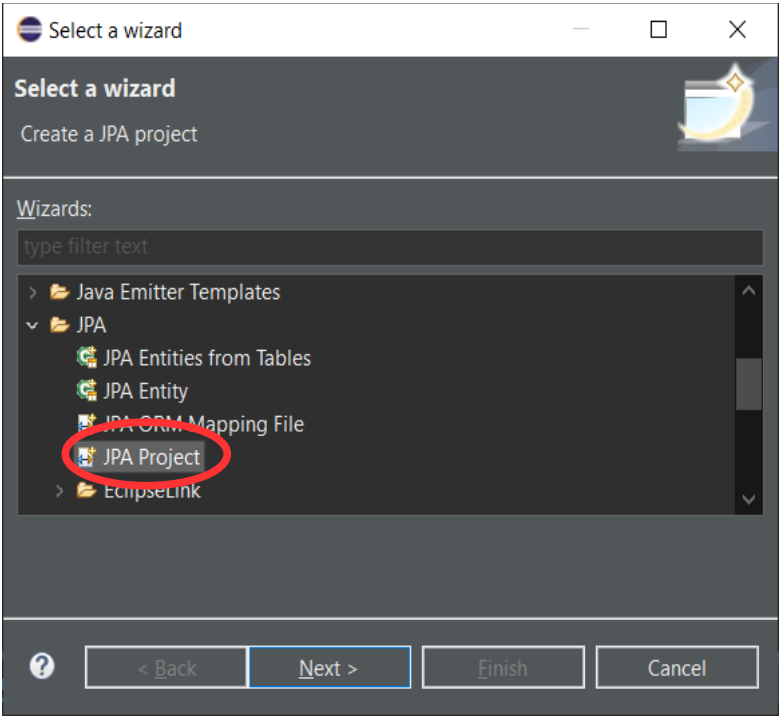
Para poder conectarse a un servidor MySQL hace falta el driver de conexión. Creamos otra librería de usuario y la llamamos mysql, donde añadimos el jar del driver.

Para poder usar este driver hay que configurar una Conexión a BD. Para ello usamos la opción ¨**Window > Preferences**¨ y en el árbol de secciones seleccionamos ¨**Data Management > Connectivity > Driver Definitions**¨ damos click en **Add**.

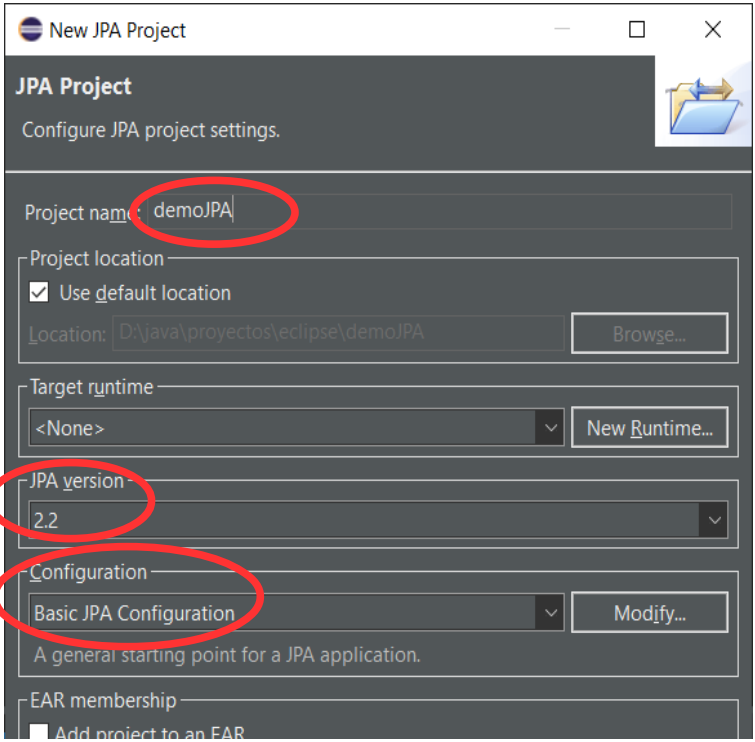


CREAR NUEVO PROYECTO JPA

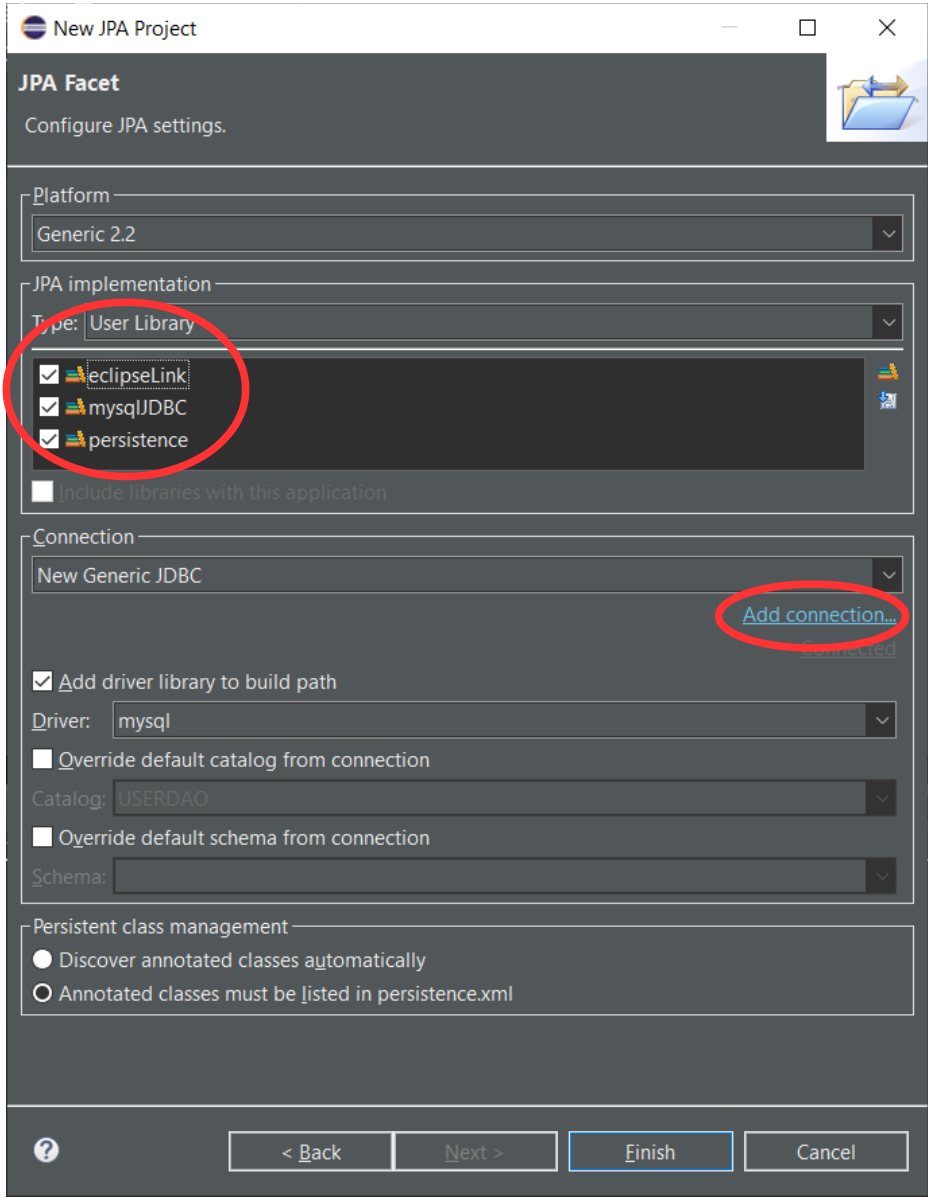
Escogemos la opción **"File -> New -> Others"** y **"JPA Project"** lo escogemos. De lo contrario escogemos **"Project"** para lanzar el asistente de nuevo proyecto. En el árbol de opciones escogemos **"JPA -> JPA Project"**.



Se abre el asistente para un nuevo Proyecto JPA. En el primer dialogo damos nombre al proyecto y clic en **"Next"** hasta el último diálogo donde definiremos algunas opciones.

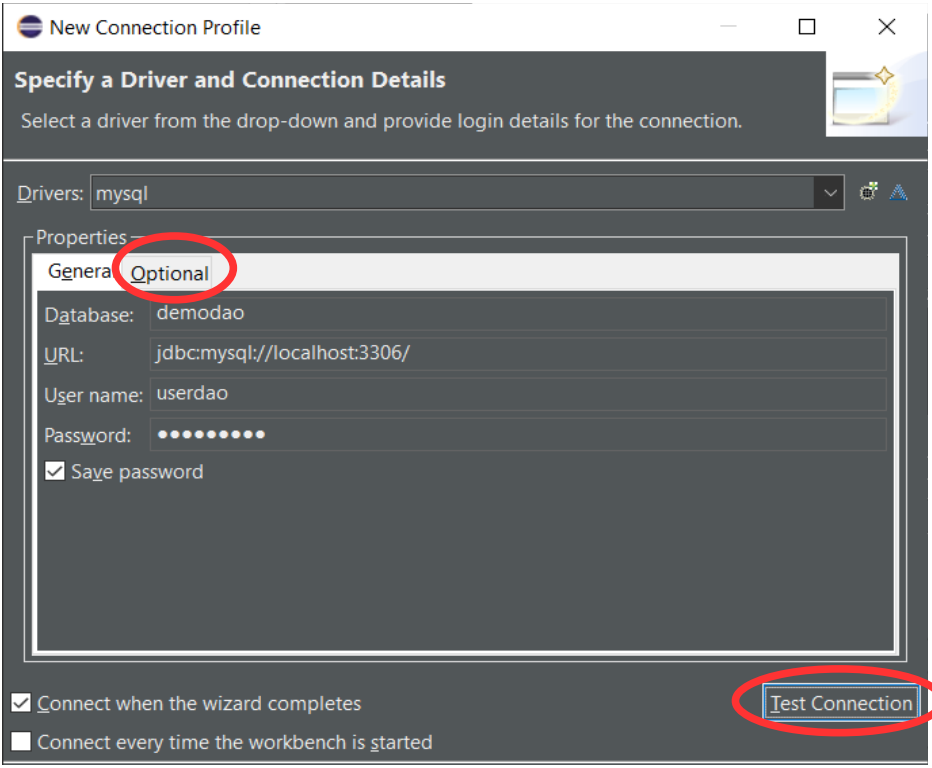


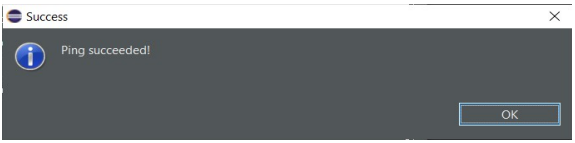
En la siguiente figura vemos el último diálogo donde definimos la versión de JPA que queremos usar, añadimos las librerías de usuario que creamos con anterioridad.


New JPA Project
JPA Facet
Configure JPA settings.
Platform
Generic 2.2
JPA implementation
Type: User Library
eclipseLink
mysqlJDBC
persistence
Include libraries with this application
Connection
New Generic JDBC
Add connection...
Add driver library to build path
Driver: mysql
Override default catalog from connection
Catalog: USERDAO
Override default schema from connection
Schema:
Persistent class management
Discover annotated classes automatically
Annotated classes must be listed in persistence.xml
< Back
Next >
Finish
Cancel

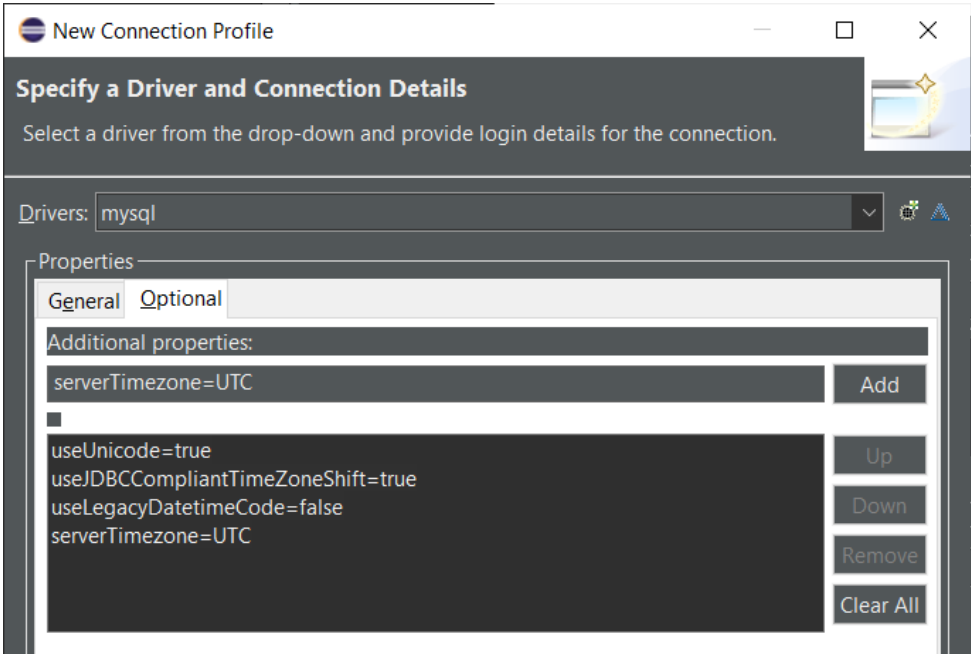
En Connection añadimos una conexión a la BD con la que queremos conectar y que también definimos previamente. Podemos testear la conexión una vez definida para comprobar que efectivamente no hay problemas de conexión con la BD.

En el caso de MySQL hay varias opciones (en la pestaña Optional) que debemos indicar, sobre todo "serverTimezone=UTC".

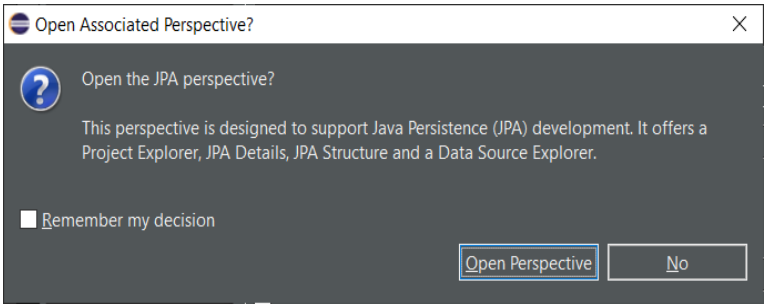




Aquí se observan las opciones para MySQL:



Cuando completamos la definición del proyecto nos preguntan si queremos activar la perspectiva de JPA, le indicaremos que si porque tendremos herramientas de soporte que nos facilitará la tarea.



Ahora, antes de mostrar el código de ejemplo, vamos a repasar algunos elementos de JPA para que entendamos que estamos haciendo.

CLASE ENTITYMANAGER

JPA tiene como interface principal al **EntityManager**, el componente que se encarga de controlar el ciclo de vida de todas las entidades definidas en la unidad de persistencia y es mediante esta interface como se realizan las operaciones básicas de una BD: consultar, actualizar, borrar, crear (CRUD). También es la que controla las transacciones.

Los EntityManager se configuran siempre a partir de las unidades de persistencia definidas en el archivo **persistence.xml** y es posible crearlas de dos formas según el tipo de aplicación:

- **Aplicación de escritorio (nuestro caso)**: será necesario instanciar el EntityManager a través de la clase **EntityManagerFactory**. El programador es el responsable de abrir y cerrar las transacciones.
- **Aplicaciones Empresariales**: Si nuestra aplicación vive dentro de una Application Server o en un EJB Container, los EntityManager se referencian por medio de Inyección de dependencias (CDI), utilizando la anotación @PersistenceContext, de este modo, el Application Server es el encargado de la creación de los EntityManager para su inyección. El Application Server se puede encargar de forma automática de las transacciones.

JPA tiene más sentido cuando lo utilizamos en entornos empresariales, sin embargo, a modo de ejemplo, nosotros lo usaremos en un sencillo ejemplo para ver su funcionamiento.

Para instanciar el EntityManager usaremos un código similar a este:

```
// En versiones JPA previas los paquetes están en javax
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;

    EntityManagerFactory factory =
                Persistence.createEntityManagerFactory("demoJPA");
    EntityManager m = factory.createEntityManager();
```

Se obtiene una instancia de la Interface EntityManagerFactory, mediante la clase Persistence, esta última recibe como parámetro el nombre de la unidad de persistencia que definimos en el archivo persistence.xml que en el ejemplo llamaremos "demoJPA". Con el EntityManagerFactory se obtiene una instancia de EntityManager. El EntityManagerFactory tiene la responsabilidad de obtener la implementación concreta del EntityManager del proveedor de JPA que dependerá completamente de la implementación de JPA que estemos utilizando en nuestra unidad de persistencia.

Volviendo al archivo **persistence.xml** podemos ver que existe la siguiente línea:

```
<persistence-unit name="demoJPA" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
```

La última línea le dice al EntityManagerFactory con que proveedor estamos trabajando de tal manera que, si el día de mañana decidimos trabajar con un proveedor distinto a EclipseLink, tan solo tendremos que definir la clase de la nueva implementación de JPA.

EntityManager será la clase que nos permitirá hacer transacciones con la BD, es decir, guardar entidades, modificarlas, consultarlas, etc. Tiene métodos, entre otros muchos: **persist()** para guardar una de nuestras entidades en la BD, **createQuery()** para crear una consulta y

obtener datos de la BD y **getTransaction()**, para obtener una "transaction". Para iniciar una "transaction" se llama a begin()

```
EntityManager manager = ...;
EntityTransaction tx = manager.getTransaction();
tx.begin();
// varias operaciones en base de datos.
// si todas van bien
tx.commit();
// si alguna va mal, en cuanto detectemos el fallo
tx.rollback();
```

EL FICHERO PERSISTENCE.XML

En JPA el fichero *persistence.xml* estará dentro de un directorio llamado META-INF. En este fichero se indican los parámetros de conexión a la BD y las clases java que son persistentes, es decir, las clases java a las que hemos puesto anotaciones propias de JPA.

En este fichero se define una *persistence-unit* con un nombre que debe ser único. A esta *persistence-unit* se le puede poner un atributo *transaction-type*, cuyos valores pueden ser:

- *RESOURCE_LOCAL:* nosotros en nuestro código nos debemos encargar de crear el *EntityManager*.
- JTA: nosotros no debemos encargarnos de crear ese *EntityManager*, alguien nos lo tiene que pasar. Esta opción sólo tiene sentido si nuestra aplicación corre en un contenedor de aplicaciones, estilo *TomEE*, *Glassfish* o *JBoss* (*Apache Tomcat* no vale). Ese servidor de aplicaciones será el que cree la clase *EntityManager* y nos la pase. En nuestro ejemplo, como utilizamos *Java SE* no vamos a usar contenedor de aplicaciones, así que pondremos *RESOURCE_LOCAL*

Nuestro fichero tendrá este contenido:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
<persistence-unit name="demoJPA" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <class>demoJPA.modelo.Empleado</class>
  <exclude-unlisted-classes>true</exclude-unlisted-classes>
  <properties>
    <property name="eclipselink.ddl-generation" value="create-tables" />
    <property name="eclipselink.ddl-generation.output-mode"
value="database" />
    <property name="jakarta.persistence.schema-
generation.database.action" value="create"/>
    <property name="jakarta.persistence.schema-generation.scripts.action"
value="create"/>
    <property name="jakarta.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>
    <property name="jakarta.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/demoDAO?serverTimezone=UTC"/>
    <property name="jakarta.persistence.jdbc.user" value="userdao"/>
    <property name="jakarta.persistence.jdbc.password"
value="UserDAO_1"/>
    <property name="eclipselink.jdbc.read-connections.min" value="1"/>
    <property name="eclipselink.jdbc.write-connections.min" value="1"/>
  </properties>
  </persistence-unit>
</persistence>
```

Hemos indicado que excluya las clases no listadas y a continuación, hemos puesto las clases una a una (en el ejemplom solo usaremos persistencia en la clase Empleado). La otra opción es poner la opción a false y ya no haría falta poner las clases.

En cualquier caso, para el caso de Java SE, la opción *exclude-unlisted-classes* puede no estar soportada por el proveedor de persistencia, así que lo mejor es simplemente poner el listado de clases, sin poner esta opción.

Finalmente, aparecen los parámetros de conexión a la base de datos. Para ello, podemos usar bien propiedades generales definidas por *JPA*, bien propiedades específicas del proveedor de persistencia que usemos, es decir, propiedades específicas de *hibernate*, de *eclipseLync*, etc.

ANOTACIONES

Nuestro código se va a comunicar con JPA a través de anotaciones. Una de las grandes ventajas de JPA es que nos permite manipular la base de datos a través de objetos, estos objetos son conocidos como Entity, las cuales son clases comunes y corrientes también llamadas POJO's (Plain Old Java Objects), estas clases tiene la particularidad de ser clases que están mapeadas contra una tabla de la BD, dicho mapeo se lleva a cabo generalmente mediante Anotaciones. Dichas anotaciones brindan los suficientes metadatos como para poder relacionar las clases con las tablas y sus propiedades con las columnas. Es de esta forma que JPA es capaz de interactuar con la BD a través de las clases. Vamos a comentar algunas anotaciones:

ENTIDAD: @Entity

Esta anotación se debe definir a nivel de clase (antes de la clase a la que afecta) y sirve para indicarle a JPA que esa clase es una Entity (una clase que tiene persistencia). Ejemplo:

```
@Entity
public class Empleado {
    private Long id;
    private String nombre;
    // Getter y setters y más elementos...
}
```

Faltaría un paso más para que JPA cargue esta clase como Entidad,

debe de estar contenida en la unidad de persistencia en el archivo persistence.xml.

TABLA: @Table

La anotación se utiliza para indicarle a JPA contra que tabla debe de mapear una entidad, de esta manera cuando se realice una persistencia, borrado o select de la entidad, JPA sabrá contra que tabla de la BD debe interactuar. Tiene otras propiedades:

- **name:** describe el nombre real de la tabla en la BD, es recomendable que el nombre sea exacto respetando mayúsculas y minúsculas, sobre todo en Linux.
- **schema:** indica el schema en el que se encuentra la tabla. Esta propiedad por lo general no es necesaria, a menos que la tabla se encuentre en un schema diferente al que utilizamos para logearnos.
- **Indexes:** JPA permite indicar los índex que tiene nuestra tabla, esta opción toma relevancia cuando indicamos a JPA que cree las tablas por nosotros (nosotros no usamos esta característica).

```
@Entity
@Table(
    name = "empleados" ,
    schema = "base",
    indexes = {@Index(name = "emp_pk", columnList = "id",unique =
true)}
)
public class Empleado {
    private Long id;
    private String nombre;
    // Getter y setters y más elementos...
}
```

CLAVES PRIMARIAS: @Id

l igual que en las tablas, las entidades también requieren un

identificador, dicho identificador deberá diferenciar cada entidad del resto. Como regla general, todas las entidades deberán definir un ID, de lo contrario provocaremos que el EntityManager marque error a la hora de instanciarlo.

El ID es importante porque será utilizando por EntityManager a la hora de persistir un objeto, y es por este que puede determinar sobre que fila hacer el select, update o delete. JPA soporta ID simples de un solo campo o ID complejos, formados por más de un campo, sin embargo, veremos únicamente los ID simples.

Para determinar el ID de una entidad es tan simple como poner la anotación @Id sobre la propiedad que sería el ID de la entidad. Ejemplo:

```
public class Empleado {
    @Id
    private Long id;
    private String nombre;
    //...
}
```

El ID puede ser cualquier tipo de dato soportado por JPA, como todos los tipos primitivos y clases wraper, enumeraciones y Calendar. En general puedes utilizar el que más se adapte a tus necesidades, pero te recomiendo que no uses los tipos primitivos para las columnas, en su lugar usa las clases wraper, por ejemplo, en lugar de int usa Integer, en lugar double usa Double y así para cada tipo de dato. Y la razón es muy simple, los tipos primitivos no aceptan valores Nulos lo que puede provocar conflictos con las BD.

CLAVES PRIMARIAS: @GeneratedValue

Si el ID es autogenerado (Identity) como en el caso de MySQL y MS SQL Server, o si es calculado a través de una secuencia como en el

caso de Oracle y Postgres. Pues bien, JPA cuenta con la anotación @GeneratedValue para indicarle a JPA que regla de autogeneración de la llave primaria vamos a utilizar. JPA soporta 4 estrategias de autogeneración pero las más interesantes son las nativas:

- **Identity:** es la estrategia más fácil de utilizar pues solo hay que indicarla. JPA no enviará este valor, pues asume que la columna es auto generada. Esto provoca que el contador de la columna incremente en 1 cada vez que un nuevo objeto es insertado.

```
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    // ...
}
```

- **Sequence:** indica a JPA que el ID se genera a través de una secuencia de la BD. De esta manera, cuando se realice un insert, esta agregara la instrucción para que en el ID se inserte el siguiente valor de la secuencia. La anotación **@SequenceGenerator** se define a nivel de clase y se utiliza para indicar a JPA que secuencia debe de utilizar para insertar en la BD.

```
@SequenceGenerator(
    name="EmpSeq",
    sequenceName = "EMPLE_SEQ",
    initialValue = 1,
    allocationSize = 10
)
public class Empleado {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;
    // ...
```

```
    }
```

Volviendo a nuestro proyecto de ejemplo, te muestro el código que ya entenderás. Bajo el directorio `/src/main/Java` del proyecto creamos el package `demoJPA.modelo` y dentro creamos la clase Empleado (el POJO al que queremos dar persistencia).

```
package demoJPA.modelo;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Table;
import jakarta.persistence.Id;

@Entity
@Table(name="empleado")
public class Empleado {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    private String nombre;

    public Empleado() {}
    public Empleado(String nombre) { this.nombre = nombre; }
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getNombre() { return nombre; }
    public void setNombre(String nombre){ this.nombre = nombre; }
    @Override
    public String toString() {
       return "Empleado (id=" + id + ", nombre=" + nombre + ")";
    }
}
```

ALGUNAS OPERACIONES CON JPA

Ahora creamos el package demoJPA.cliente y dentro la clase Prueba.java que realizará operaciones con JPA.

El código comienza con imports y la instanciación del EntityManager y en el método main realizará las operaciones que estarán encapsuladas en métodos.

```
package demoJPA.cliente;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;
import jakarta.persistence.Query;
import demoJPA.modelo.Empleado;
import java.util.List;

public class Prueba {
    private EntityManager manager;

    public Prueba(EntityManager em) {
       manager = em;
    }

    public static void main(String[] args) {
       EntityManagerFactory factory =
Persistence.createEntityManagerFactory("demoJPA");
       EntityManager m = factory.createEntityManager();
       Prueba p = new Prueba(m);
       p.creaEmpleados();
       p.muestraEmpleados();
       System.out.println();
       Empleado e = p.empleadoPorNombre("Maria");
       System.out.println();
       p.modificaEmpleado(e.getId(), "Maria Lopez");
       p.muestraEmpleados();
       System.out.println();
       p.borraEmpleado(e.getId());
       p.muestraEmpleados();
       System.out.println();
       System.out.println("... hecho");
       m.close();
       factory.close();
    }
```

Inserción

Para crear algunas instancias y guardarlas usaremos el método siguiente:

```
    private void creaEmpleados() {
        System.out.println("CREANDO EMPLEADOS:");
        // manager es el EntityManager obtenido anteriormente
       EntityTransaction tx = manager.getTransaction();
       tx.begin();
       try {
             manager.persist( new Empleado("Pepe Lopez") );
             manager.persist( new Empleado("Maria") );
             tx.commit();
        } catch (Exception e) {
             e.printStackTrace();
             tx.rollback();
        }
    }
```

Obtenemos a partir del *EntityManager* una *EntityTransaction* usando el método *getTransaction()*. Comenzamos la transacción llamando a *begin()* y empezamos con la creación de elementos. Basta con instanciar y rellenar los campos de *Empleado* (usamos el constructor que admite un nombre como parámetro) y llamando a *manager.persist()* pasando como parámetro el *Empleado* recién creado, ya lo tenemos guardado en la BD. Repetimos la operación para tener dos empleados y terminamos la transacción con *tx.commit()*. Si hubiera cualquier error y saltara una excepción, aparte de imprimirla se hace un *tx.rollback()*.

Consulta de todos los empleados

Para ver los *Empleados* recién creados, hacemos una consulta a BD:

```
   private void muestraEmpleados() {
        System.out.println("Lista de empleados:");
       List<Empleado> lista = manager.createQuery("Select a From
Empleado a", Empleado.class).getResultList();
       System.out.println("  Num de empleados:" + lista.size() );
```

```
        for (Empleado e: lista) {
            System.out.println(e);
        }
    }
```

Para las consultas, en principio no es necesario iniciar ni terminar ninguna transacción. Una consulta no es una operación que queramos deshacer si falla. Así que simplemente llamamos al método *createQuery()* de *EntityManager* pasando dos parámetros.

- El primero es la consulta que queremos realizar, en un lenguaje similar pero no igual a *SQL*, que se llama *JPQL* (*Java Persistent Query Language*).
- El segundo es la clase que esperamos que nos devuelva la consulta, en nuestro caso una lista de *Employee.class*.

A la *query* obtenida de esta forma, llamamos a su método *getResultList()*, que nos devolverá directamente una lista de *List<Empleado>*. Ya sólo nos queda hacer un bucle ir sacando por pantalla los resultados o lo que queramos hacer con ellos.

Si queremos consultar un *Empleado* concreto, por ejemplo, por su nombre, sólo tenemos que añadir la cláusula *where* en la sentencia *JPQL*, como se muestra en el siguiente método

```
private Empleado empleadoPorNombre(String nombre) {
        System.out.println("BUSCAR " + nombre );
       Query q = manager.createQuery("Select a from Empleado a
where a.nombre =:unNombre");
       q.setParameter("unNombre", nombre);
       Empleado e = (Empleado) q.getSingleResult();
        System.out.println("Resultado : " + e );
       return e;
    }
```

En la sentencia *JPQL* hemos puesto ***select a from Empleado a***. Esto guarda en la variable *a* cada empleado, de forma que con *a.nombre* podemos acceder al nombre de cada empleado. Podríamos usar cualquier otro atributo de la clase *Empleado*, pero en este ejemplo buscaremos por nombre y por ello la cláusula *where* pone *a.nombre= :nombre*.

La primera parte, *a.nombre es* el atributo de cada uno de los *Empleado* en la BD y queremos que sea igual a un nombre concreto. Como el nombre nos lo pasan como parámetro del método empleadoPorNombre, en la consulta *JPQL* ponemos una variable que nos inventemos, precedida de un *:* para que *JPQL* sepa que eso es una variable. La hemos llamado *:nombre*, aunque podríamos usar cualquier otro nombre.
Una vez obtenida la consulta, debemos dar valor a las variables que hayamos definido, *:nombre* en nuestro caso. Para ello llamamos al método *setParameter()* de la *query*, pasando como primer parámetro en nombre de la variable, sin los dos puntos, y como segundo parámetro, el valor que queramos para ella.

Si somos optimistas y presumimos que no va a haber dos empleados con el mismo nombre, en vez de obtener una lista llamamos al método *getSingleResult()*, que nos devolverá el primer empleado que encuentre en la BD y cumpla la condición. Como *getSingleResult()* devuelve un *Object*, debemos hacer el *cast* a *Empleado*. Pero saltará una excepción si no hay resultados en la BD que cumplan la condición, o si hay más de uno. *getSingleResult()* es más adecuado si consultamos por clave o por alguna columna con la restricción de que no tenga valores repetidos.

Modificación

La modificación de un dato también es sencilla. Aparte de poder usar una sentencia *JPQL* de *update*, podemos consultar la BD, obtener la

clase, modificar sus datos directamente con los métodos *set* y luego salvar. Esto es lo que hacemos en el siguiente trozo de código

```
private void modificaEmpleado(Long id, String nuevoNombre) {
        System.out.println("MODIFICANDO EMPLEADO " + id);
        EntityTransaction tx = manager.getTransaction();
        tx.begin();
        try {
           Empleado e = manager.find(Empleado.class, id);
           e.setNombre(nuevoNombre);
           manager.persist(e);
           tx.commit();
        } catch (Exception e) {
           e.printStackTrace();
           tx.rollback();
        }
     }
```

Buscamos el *Employee* en base de datos por medio del método *find()* de *EntityManager*. A este método le pasamos el tipo de clase que esperamos como respuesta *Employee.class* y el *id* en base de datos. Si lo encuntra, nos devolverá el *Employee* en cuestión.
Cambiamos el nombre del *Employee* con el método *setName()* y finalmente, guardamos los cambios llamando a *persist()*. Un *commit()* y listo.

Borrado

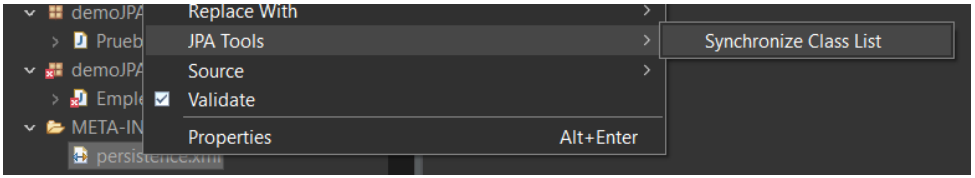
Para borrar, nuevamente tendríamos la opción de hacer un *delete* usando el lenguaje *JPQL*, o bien si tenemos la clase *Employee*, bastaría con llamar al método *remove()* de *EntityManager*, que es justo lo que vamos a hacer

```
private void borraEmpleado(Long id) {
        System.out.println("BORRANDO EMPLEADO " + id);
        EntityTransaction tx = manager.getTransaction();
        tx.begin();
        try {
           Empleado e = manager.find(Empleado.class, id);
```

```
            manager.remove(e);
            tx.commit();
        } catch (Exception e){
            e.printStackTrace();
            tx.rollback();
        }
    }
```

Poco hay que comentar, se busca el *Employee* en base de datos igual que antes, y se borra.

> `Nota:` Me aparecía un error en la clase @Entity y al final sobre el fichero persistence, botón derecho, JPA Tools -> Synchronize Class List o bien dejas que busque JPA las clases a hacer persistentes (así te ahorras la faena).



Y el resultado de la ejecución:

```
[EL Info]: 2021-09-22 09:35:20.234--ServerSession(573673894)--EclipseLink, version: Eclipse Persistence
CREANDO EMPLEADOS:
Lista de empleados:
  Num de empleados:2
Empleado (id=601, nombre=Pepe Lopez)
Empleado (id=602, nombre=Maria)

BUSCAR Maria
Resultado : Empleado (id=602, nombre=Maria)

MODIFICANDO EMPLEADO 602
Lista de empleados:
  Num de empleados:2
Empleado (id=601, nombre=Pepe Lopez)
Empleado (id=602, nombre=Maria Lopez)

BORRANDO EMPLEADO 602
Lista de empleados:
  Num de empleados:1
Empleado (id=601, nombre=Pepe Lopez)

... hecho
```

10.6. EJERCICIOS.

EJERCICIO 1. Instalar Mysql server 8.0 community edition.

EJERCICIO 2. Crea una BD llamada "prueba" y define esta tabla:

```
EMPLEADOS = id + nombre + apellido + cargo + salario
       PK: id
```

Y añade estas filas:

```
60 Miguel Ros CTO 98000
70 Alejandro Bernal Ing. del software 88000
50 Jazmin Sola  Administradora 88000
30 Maria Comins CEO 100000
10 Antonio Salcedo Programador 48000
```

EJERCICIO 3. Crea un programa en Java que conecte con el servidor MySQL a la base de datos prueba usando el driver de tipo 4 "mysql-connector-java-8.0.1.jar" o una versión similar y una cadena de conexión que contenga el usuario y el password y que indique si lo consigue o la cadena de errores si hay algún problema.

EJERCICIO 4. Modifica el programa del ejercicio anterior y muestra un listado de los datos de la tabla empleados.

```
Conectando con la Base de datos...
Conexión establecida con la BD...
10 – Antonio – Salcedo _ Programador _ 48000
30 – Maria – Comins _ CEO _ 100000
50 – Jazmin – Sola _ Administradora _ 88000
60 – Miguel – Ros _ CTO _ 98000
70 – Alejandro – Bernal _ Ing del software _ 88000
Conexión cerrada con la BD.
```

EJERCICIO 5. Modifica el programa anterior para que pida al usuario una cantidad y solo muestre los datos de aquellos empleados que tengan un salario superior a la cantidad leída.

```
Conectando con la Base de datos...
Conexión establecida con la BD...
Mostrar empleados con sueldo mayor de: 90000
30 – Maria – Comins _ CEO _ 100000
60 – Miguel – Ros _ CTO _ 98000
Conexión cerrada con la BD.
```

EJERCICIO 6. Averiguar todas las opciones de conexión de un driver utilizando la clase Driver y su método getPropertyInfo() generando un string con formato XML que se ve parcialmente en la siguiente figura:

```
--- driver Propiedades Info  ---
<?xml version='1.0'>
<PropiedadesDriver driver="com.mysql.cj.jdbc.Driver" url="jdbc:mysql://localhost/prueba">
 <PropiedadDriver>
    <name>host</name>
    <required>true</required>
    <value>localhost</value>
    <description>Hostname of MySQL Server</description>
  <opciones>
  </opciones>
 </PropiedadDriver>
 <PropiedadDriver>
    <name>port</name>
    <required>false</required>
    <value>3306</value>
    <description>Port number of MySQL Server</description>
  <opciones>
  </opciones>
 </PropiedadDriver>
```

EJERCICIO 7. Averigua si una BD soporta transacciones preguntando la marca, host, puerto y nombre de la BD, creando una conexión y usando el objeto DataBaseMetaData.

```
Conectar al SGBD de marca (mysql, oracle, postgreesql, sqlserver...): mysql
Host del SGBD: localhost
Puerto del SGBD: 3306
Nombre de la BD: prueba
conexion =com.mysql.cj.jdbc.ConnectionImpl@694abbdc
Soporta transacciones: true
```

EJERCICIO 8. Averigua y cambia el límite máximo de la cantidad de filas intercambiadas en las consultas con el servidor.

EJERCICIO 9. Pon el apellido del empleado 70 a NULL desde mysql y haz un programa que imprima el apellido y el id de los empleados y detecte si el valor del apellido es NULL.

EJERCICIO 10. Haz un método llamado getNombresCols que acepte de parámetro un objeto resultSet y devuelva un string con los nombres de sus columnas y a qué tabla corresponden en formato XML. Prueba el resultado con esta consulta:

```
Salcedo 10
Comins 30
Sola 50
Ros 60
apellido a null del empleado 70
```

```
select id,
      concat(nombre, ' ', apellido) as nombre,
      cargo,
      salario
from empleados;
```

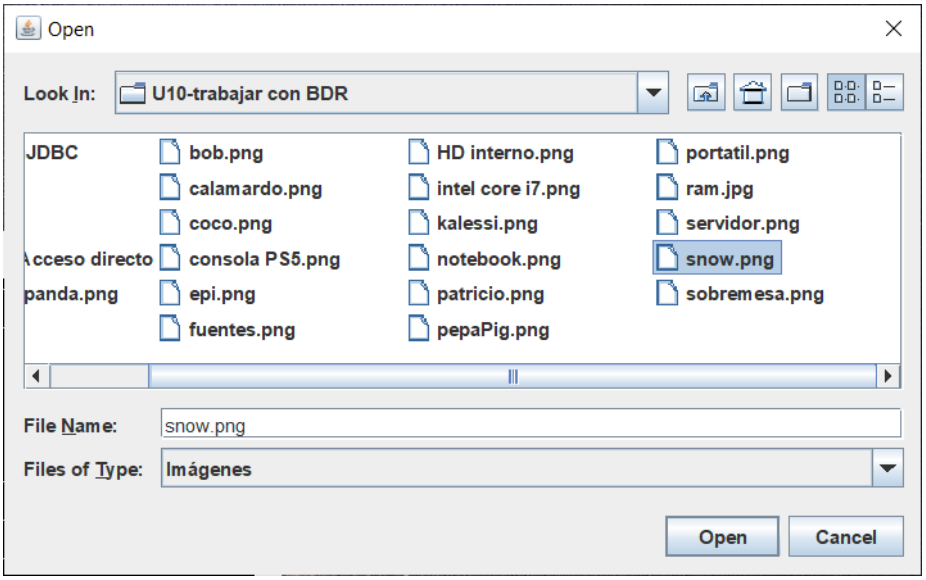
```
<NombreColumnas>
<columna="id"  tabla="empleados"/>
<columna="nombre"  tabla=""/>
<columna="cargo"  tabla="empleados"/>
<columna="salario"  tabla="empleados"/>
</NombreColumna>
```

EJERCICIO 11. Crea una tabla llamada fotos con esta columnas:

```
FICHEROS = id (int) + nombre(varchar2) + bytes (blob)
PK: id
```

El id es la clave primaria, nombre es el nombre de un fichero que contiene una imagen y bytes son sus bytes.

Haz un programa que permita elegir un fichero de tipo imagen con un JfileChooser y un método llamado insertaFichero(String id, String nombre, String pathName) y que inserte la foto en una fila de la tabla FICHEROS usando una sentencia preparada (con parámetros) y `setBinaryStream()`.




```
mysql> select id, nombre from ficheros;
+----+----------+
| id | nombre   |
+----+----------+
| 10 | snow.png |
+----+----------+
1 row in set (0.00 sec)
```

EJERCICIO 12. Haz un programa con GUI que se conecte a la BD y muestre las imágenes de tipo .png, .jpg y .gif almacenadas en la tabla ficheros.


10 snow.png
Anterior
Siguiente