JDBC - Acceso a Bases de Datos

Este tutor está basado en una traducción-adaptación del tutorial de **Sun.**

JDB*Ctm* fue diseñado para mantener sencillas las cosas sencillas. Esto significa que el API JDBC hace muy sencillas las tareas diarias de una base de datos, como una simple sentencia SELECT. Esta sección nos llevará a través de ejemplos que utilizan el JDBC para ejecutar sentencias SQL comunes, para que podamos ver lo sencilla que es la utilización del API JDBC básico.

JDBC Basico que cubre el API JDBC 1.0, que está incluido en el JDKtm 1.1. La segunda parte cubre el API JDBC 2.0 API, que forma parte de la versión 1.2 del JDK. También describe brevemente las extensiones del API JDBC, que, al igual que otras extensiones estándard, serán liberadas independientemente.
Al final de esta primera sección, sabremos como utilizar el API básico del JDBC para crear tablas, insertar valores en ellas, pedir tablas, recuperar los resultados de las peticiones y actualizar las tablas. En este proceso, aprenderemos como utilizar las sentencias sencillas y sentencias preparadas, y veremos un ejemplo de un procedimiento almacenado. También aprenderemos como realizar transacciones y como capturar excepciones y avisos. En la última parte de este sección veremos como crear un Applet.

Para empezar...
Lo primero que tenemos que hacer es asegurarnos de que disponemos de la configuración apropiada. Esto incluye los siguientes pasos:
Instalar Java y el JDBC en nuestra máquina.
Para instalar tanto la plataforma JAVA como el API JDBC, simplemente tenemos que seguir las instrucciones de descarga de la última versión del JDK (Java Development Kit). Junto con el JDK también viene el JDBC.. El código de ejemplo de desmostración del API del JDBC 1.0 fue escrito para el JDK 1.1 y se ejecutará en cualquier versión de la plataforma Java compatible con el JDK 1.1, incluyendo el JDK1.2. Teniendo en cuenta que los ejemplos del API del JDBC 2.0 requieren el JDK 1.2 y no se podrán ejecutar sobe el JDK 1.1. Podrás encontrar la última versión del JDK en la siguiente dirección::
http://java.sun.com/products/JDK/CurrentRelease

1. Instalar un driver en nuestra máquina.
Nuestro Driver debe incluir instrucciones para su instalación. Para los drivers JDBC escritos para controladores de bases de datos específicos la instalación consiste sólo en copiar el driver en nuesta máquina; no se necesita ninguna configuración especial.
El driver "puente JDBC-ODBC" no es tan sencillo de configurar. Si descargamos las versiones Solaris o Windows de JDK 1.1, automáticamente obtendremos una versión del driver Bridge JDBC-ODBC, que tampoco requiere una configuración especial. Si embargo, ODBC, si lo necesita. Si no tenemos ODBC en nuestra máquina, necesitaremos preguntarle al vendedor del driver ODBC sobre su instalación y configuración.

2. Instalar nuestro Controlador de Base de Datos si es necesario.
Si no tenemos instalado un controlador de base de datos, necesitaremos seguir las instrucciones de instalación del vendedor. La mayoría de los usuarios tienen un controlador de base de datos instalado y trabajarán con un base de datos establecida.

3.Seleccionar una Base de Datos
A lo largo de la sección asumiremos que la base de datos **COFFEEBREAK** ya existe. (crear una base de datos no es nada díficil, pero requiere permisos especiales y normalmente lo hace un administrador de bases de datos). Cuando creemos las tablas utilizadas como ejemplos en este tutorial, serán la base de datos por defecto. Hemos mantenido un número pequeño de tablas para mantener las cosas manejables.

Supongamos que nuestra base de datos está siendo utilizada por el propietario de un pequeño café llamado "The Coffee Break", donde los granos de café se venden por kilos y el café liquido se vende por tazas. Para mantener las cosas sencillas, también supondremos que el propietario sólo necesita dos tablas, una para los tipos de café y otra para los suministradores.
Primero veremos como abrir una conexión con nuestro controlador de base de datos, y luego, ya que JDBC puede enviar codigo SQL a nuestro controlador, demostraremos algún código SQL. Después, veremos lo sencillo que es utilizar JDBC para pasar esas sentencias SQL a nuestro controlador de bases de datos y procesar los resultados devueltos.

Este código ha sido probado en la mayoría de los controladores de base de datos.
Sin embargo, podríamos encontrar algunos problemas de compatibilidad su utilizamos antiguos drivers ODB con el puente JDBC.ODBC.

Establecer una Conexión
Lo primero que tenemos que hacer es establecer una conexión con el controlador de base de datos que queremos utilizar. Esto implica dos pasos: (1) cargar el driver y (2) hacer la conexión.

Cargar los Drivers
Cargar el driver o drivers que queremos utilizar es muy sencillo y sólo implica una línea de código. Si, por ejemplo, queremos utilizar el puente JDBC-ODBC, se cargaría la siguiente línea de código:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

La documentación del driver nos dará el nombre de la clase a utilizar. Por ejemplo, si el nombre de la clase es **jdbc.DriverXYZ**, cargaríamos el driver con esta línea de código:

```
Class.forName("jdbc.DriverXYZ");
```

No necesitamos crear un ejemplar de un driver y registrarlo con el **DriverManager** porque la llamada a **Class.forName** lo hace automáticamente. Si hubiéramos creado nuestro propio ejemplar, crearíamos un duplicado innecesario, pero no pasaría nada.
Una vez cargado el driver, es posible hacer una conexión con un controlador de base de datos.

Hacer la Conexión
El segundo paso para establecer una conexión es tener el driver apropiado conectado al controlador de base de datos. La siguiente línea de código ilustra la idea general:

```
Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");
```

Este paso también es sencillo, lo más duro es saber qué suministrar para **url**. Si estamos utilizando el puente JDBC-ODBC, el JDBC URL empezará con **jdbc:odbc**:.
el resto de la URL normalmente es la fuente de nuestros datos o el sistema de base de datos. Por eso, si estamos utilizando ODBC para acceder a una fuente de datos ODBC llamada **"Fred**," por ejemplo, nuestro URL podría ser **jdbc:odbc:Fre**d. En lugar de **"myLogi**n" pondríamos el nombre utilizado para entrar en el controlador de la base de datos; en lugar de **"myPasswor**d" pondríamos nuestra password para el controlador de la base de datos. Por eso si entramos en el controlador con el nombre **"Fernand**o" y la password of **"J8**," estas dos líneas de código
estableceran una conexión:

```
String url = "jdbc:odbc:Fred";
Connection con = DriverManager.getConnection(url, "Fernando", "J8");
```

Si estamos utilizando un puente JDBC desarrollado por una tercera parte, la documentación nos dirá el subprotocolo a utilizar, es decir, qué poner despues de **jdbc:** en la URL. Por ejemplo, si el desarrollador ha registrado el nombre "acme" como el subprotocolo, la primera y segunda parte de la URL de JDBC serán **jdbc:acme**:. La documentación del driver también nos dará las guías para el resto de la URL del JDBC. Esta última parte de la URL suministra información para la
identificación de los datos fuente.
Si uno de los drivers que hemos cargado reconoce la URL suministada por el método **DriverManager.getConnectio**n, dicho driver establecerá una conexión con el controlador de base de datos especificado en la URL del JDBC. La clase **DriverManage**r, como su nombre indica, maneja todos los detalles del establecimiento de la conexión detrás de la escena. A menos que estemos escribiendo un driver, posiblemente nunca utilizaremos ningún método del interface
Driver, y el único método de **DriverManager** que realmente necesitaremos conocer es **DriverManager.getConnectio**n.
La conexión devuelta por el método **DriverManager.getConnection** es una conexión abierta que se puede utilizar para crear sentencias JDBC que pasen nuestras sentencias SQL al controlador de la base de datos. En el ejemplo anterior, **con** es una conexión abierta, y se utilizará en los ejemplos posteriores.

Seleccionar Tablas

Crear una Tabla
Primero, crearemos una de las tablas de nuestro ejemplo. Esta tabla, **COFFEE**S, contiene la información esencial sobre los cafés vendidos en "The Coffee Break", incluyendo los nombres de los cafés, sus precios, el número de libras vendidas la semana actual, y el número de libras vendidas hasta la fecha. Aquí puedes ver la tabla **COFFEE**S, que describiremos más adelante:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

La columna que almacena el nombre del café es **COF_NAME,** y contiene valores con el tipo **VARCHAR** de SQL y una longitud máxima de 32 caracteres. Como utilizamos nombres diferentes para cada tipo de café vendido, el nombre será un único identificador para un café particular y por lo tanto puede servir como clave primaria.
La segunda colunma, llamada **SUP_I**D, contiene un número que identifica al suministrador del café; este número será un tipo **INTEGER** de SQL. La tercera columna, llamada **PRICE,** almacena valores del tipo **FLOAT** de SQL porque necesita contener valores decimales. (Observa que el dinero normalmente se almacena en un tipo **DECIMAL** o **NUMERIC** de SQL, pero debido a las diferencias entre controladores de bases de datos y para evitar la incompatibilidad con viejas
versiones de JDBC, utilizamos el tipo más estándard **FLOA**T.) La columna llamada **SALES** almacena valores del tipo **INTEGER** de SQL e indica el número de libras vendidas durante la semana actual. La columna final, **TOTA**L,contiene otro valor **INTEGER** de SQL que contiene el número total de libras vendidas hasta la fecha.

SUPPLIERS, la segunda tabla de nuesta base de datos, tiene información sobre cada uno de los suministradores:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Las tablas **COFFEES** y **SUPPLIERS** contienen la columna **SUP_I**D, lo que significa que estas dos tablas pueden utilizarse en sentencias **SELECT** para obtener datos basados en la información de ambas tablas. La columna **SUP_ID** es la clave primaria de la tabla **SUPPLIER**S, y por lo tanto, es un identificador único para cada uno de los suministradores de café. En la tabla **COFFEE**S, **SUP_ID** es llamada clave extranjera. (Se puede pensar en una clave extranjera en el sentido en que es importada desde otra tabla). Observa que cada número **SUP_ID** aparece sólo una vez en la tabla **SUPPLIER**S; esto es necesario para ser una clave primaria. Sin embargo, en la tabla **COFFEE**S, donde es una clave extranjera, es perfectamente correcto que haya números duplicados de **SUP_ID** porque un suministrador puede vender varios tipos de café. Más adelante en este capítulo podremos ver cómo utilizar claves primarias y extranjeras en una sentencia **SELEC**T.

La siguiente sentencia SQL crea la tabla **COFFEE**S. Las entradas dentro de los paréntesis exteriores consisten en el nombre de una columna seguido por un espacio y el tipo SQL que se va a almacenar en esa columna. Una coma separa la entrada de una columna (que consiste en el nombre de la columna y el tipo SQL) de otra. El tipo **VARCHAR** se crea con una longitud máxima, por eso toma un parámetro que indica la longitud máxima. El parámetro debe estar entre paréntesis siguiendo al tipo. La sentencia SQL mostrada aquí, por ejemplo, específica que los nombres de la columna COF-NAME pueden tener hasta 32 caracteres de longitud:

```
CREATE TABLE COFFEES
(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, SALES INTEGER,
TOTAL INTEGER)
```

Este código no termina con un terminador de sentecia de un controlador de base de datos, que puede variar de un controlador a otro. Por ejemplo, Oracle utiliza un punto y coma (;) para finalizar una sentencia, y Sybase utiliza la plabra **g**o. El driver que estamos utilizado proporcionará automáticamente el terminador de setencia apropiado, y no necesitaremos introducirlo en nuestro código JDBC.
Otra cosa que debíamos apuntar sobre las sentencias SQL es su forma. En la sentencia **CREATE TABL**E, las palabras clave se han imprimido en letras máyusculas, y cada ítem en una línea separada. SQL no requiere nada de esto, estas convenciones son sólo para una fácil lectura. El estándard SQL dice que la palabras claves no son sensibles a las mayúsculas, así, por ejemplo, la anterior sentencia **SELECT** pude escribirse de varias formas. Y como ejemplo, estas dos versiones son equivalentes en lo que concierne a SQL:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE "Washington"
```

Sin embargo, el material entre comillas si es sensible a las maýusculas: en el nombre **"Washingto**n", **"**W" debe estar en maýuscula y el resto de las letras en minúscula.
Los requerimientos pueden variar de un controlador de base de datos a otro cuando se trada de nombres de identificadores. Por ejemplo, algunos controladores, requieren que los nombres de columna y de tabla seán exactamente los mismos que se crearon en las sentencias **CREATE** y **TABL**E, mientras que otros controladores no lo necesitan. Para asegurarnos, utilizaremos mayúsculas para identificadores como **COFFEES** y **SUPPLIERS** porque así es como los definimos.

Hasta ahora hemos escrito la sentencia SQL que crea la tabla **COFFEE**S. Ahora le pondremos comillas (crearemos un string) y asignaremos el string a la variable **createTableCoffees** para poder utilizarla en nuestro código JDBC más adelante. Como hemos visto, al controlador de base de datos no le importa si las líneas están divididas, pero en el lenguaje Java, un objeto **String** que se extienda más allá de una línea no será compilado. Consecuentemente, cuando estamos entregando cadenas, necesitamos encerrar cada línea entre comillas y utilizar el signo más (+)
para concatenarlas:

```
String createTableCoffees = "CREATE TABLE COFFEES " + "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " + "SALES INTEGER, TOTAL INTEGER)";
```

Los tipos de datos que hemos utilizado en nuestras sentencias **CREATE** y **TABLE** son tipos genéricos SQL (también llamados tipos JDBC) que están definidos en la clase **java.sql.Type**s. Los controladores de bases de datos generalmente utilizan estos tipos estándards, por eso cuando llegue el momento de probar alguna aplicación, sólo podremos utilizar la aplicación **CreateCoffees.jav**a, que utiliza las sentencias **CREATE** y **TABLE**. Si tu controlador utiliza sus propios nombres de tipos, te suministrarermos más adelante una aplicación que hace eso.
Sin embargo, antes de ejecutar alguna aplicación, veremos lo más básico sobre el JDBC.

Crear sentencias JDBC
Un objeto **Statement** es el que envía nuestras sentencias SQL al controlador de la base de datos. Simplemente creamos un objeto **Statement** y lo ejecutamos, suministando el método SQL apropiado con la sentencia SQL que queremos enviar.
Para una sentencia **SELEC**T, el método a ejecutar es **executeQuer**y. Para sentencias que crean o modifican tablas, el método a utilizar es **executeUpdat**e.
Se toma un ejemplar de una conexión activa para crear un objeto **Statemen**t. En el siguiente ejemplo, utilizamos nuestro objeto **Connection: con** para crear el objeto **Statemen**t: **stm**t:

```
Statement stmt = con.createStatement();
```

En este momento **stmt** existe, pero no tiene ninguna sentencia SQL que pasarle al controlador de la base de datos. Necesitamos suministrarle el metodo que utilizaremos para ejecutar **stm**t. Por ejemplo, en el siguiente fragmento de código, suministramos **executeUpdate** con la sentencia SQL del ejemplo anterior:

```
stmt.executeUpdate("CREATE TABLE COFFEES " +"(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " + "SALES INTEGER, TOTAL INTEGER)");
```

Coma ya habíamos creado un String con la sentencia SQL y lo habíamos llamado **createTableCoffee**s, podríamos haber escrito el código de esta forma alternativa:

```
stmt.executeUpdate(createTableCoffees);
```

Ejecutar Sentencias
Utilizamos el método **executeUpdate** porque la sentencia SQL contenida en **createTableCoffees** es una sentencia DDL (data definition language). Las sentencias que crean, modifican o eliminan tablas son todas ejemplos de sentencias DDL y se ejecutan con el método **executeUpdate**. Cómo se podría esperar de su nombre, el método **executeUpdate** también se utiliza para ejecutar sentencias SQL que actualizan un tabla. En la práctica **executeUpdate** se utiliza más frecuentemente para actualizar tablas que para crearlas porque una tabla se crea sólo una vez, pero se puede actualizar muchas veces.
El método más utilizado para ejecutar sentencias SQL es **executeQuery**. Este método se utiliza para ejecutar sentencias **SELECT**, que comprenden la amplia mayoría de las sentencias SQL. Pronto veremos como utilizar este método.

Introducir Datos en una Tabla

Hemos visto como crear la tabla **COFFEES** especificando los nombres de columnas y los tipos de datos almacenados en esas columnas, pero esto sólo configura la estructura de la tabla. La tabla no contiene datos todavía. Introduciremos datos en nuestra tabla una fila cada vez, suministrando la información a almacenar en cada columna de la fila. Observa que los valores insertados en las columnas se listan en el mismo orden en que se declararon las columnas cuando se creó la tabla, que es el orden por defecto.

El siguiente código iserta una fila de datos con **Colombian** en la columna **COF_NAME**, **101** en **SUP_ID**, **7.99** en **PRICE**, **0** en **SALES**, y **0** en **TOTAL**. (Como acabamos de inaugurar "The Coffee Break", la cantidad vendida durante la semana y la cantidad total son cero para todos los cafés). Al igual que hicimos con el código que creaba la tabla **COFFEES**, crearemos un objeto **Statement** y lo ejecutaremos utilizando el método **executeUpdate**.
Como la sentencia SQL es damasiado larga como para entrar en una sóla línea, la hemos dividido en dos strings concatenándolas mediante un signo más (+) para que puedan compilarse. Presta especial atención a la necesidad de un espacio entre **COFFEES** y **VALUE**S. Este espacio debe estar dentro de las comillas y debe estar después de **COFFEES** y antes de **VALUES**; sin un espacio, la sentencia SQL sería leída erróneamente como **"INSERT INTO COFFEESVALUES . .** ." y el controlador de la base de datos buscaría la tabl**aCOFFEESVALUES**. Observa también que
utilizamos comilla simples alrededor del nombre del café porque está anidado dentro de las comillas dobles. Para la mayoría de controladores de bases de datos, la reglageneral es alternar comillas dobles y simples para indicar anidación.

```
Statement stmt = con.createStatement();
stmt.executeUpdate( "INSERT INTO COFFEES " + "VALUES ('Colombian', 101, 7.99, 0, 0)");
```

El siguiente código inserta una segunda línea dentro de la tabla **COFFEE**S. Observa que hemos reutilizado el objeto **Statemen**t: **stmt** en vez de tener que crear uno nuevo para cada ejecución.

```
stmt.executeUpdate("INSERT INTO COFFEES " + "VALUES ('French_Roast', 49, 8.99, 0, 0)");
```

Los valores de las siguientes filas se pueden insertar de esta forma:

```
stmt.executeUpdate("INSERT INTO COFFEES " + "VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " + "VALUES ('Colombian_Decaf', 101, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " + "VALUES ('French_Roast_Decaf', 49, 9.99, 0, 0)");
```

Obtener Datos desde una Tabla
Ahora que la tabl**aCOFFEES** tiene valores, podemos escribir una sentencia **SELECT** para acceder a dichos valores. El asterisco (*) en la siguiente sentencia SQL indica que la columna debería ser seleccionada. Como no hay claúsula **WHERE** que limite las columas a seleccionar, la siguiente sentencia SQL seleciona la tabla completa: SELECT * FROM COFFEES

El resultado, que es la tabla completa, se parecería a esto:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

El resultado anterior es lo que veríamos en nuestro terminal si introdujeramos la petición SQL directamente en el sistema de la base de datos. Cuando accedemos a una base de datos a través de una aplicación Java, como veremos pronto, necesitamos recuperar los resultados para poder utilizarlos. Veremos como hacer esto en la siguiente página.
Aquí tenemos otro ejemplo de una sentencia **SELEC**T, ésta obtiene una lista de cáfes y sus respectivos precios por libra: SELECT COF_NAME, PRICE FROM COFFEES

El resultado de esta consulta se parecería a esto:

COF_NAME	PRICE
Colombian	7.99
French_Roast	8.99
Espresso	9.99
Colombian_Decaf	8.99
French_Roast_Decaf	9.99

La sentencia **SELECT** genera los nombres y precios de todos los cáfes de la tabla. La siguiente sentencia SQL límita los cafés seleccionados a aquellos que cuesten menos de $9.00 por libra:

```
SELECT COF_NAME, PRICE
FROM COFFEES
WHERE PRICE < 9.00
```

El resultado se parecería es esto:

COF_NAME	PRICE
Colombian	7.99
French_Roast	8.99
Colombian Decaf	8.99

Recuperar Valores de una Hoja de Resultados

Ahora veremos como enviar la sentencia **SELECT** de la página anterior desde un programa escrito en Java y como obtener los resultados que hemos mostrado.

JDBC devuelve los resultados en un objeto **ResultSe**t, por eso necesitamos declarar un ejemplar de la clase **ResultSet** para contener los resultados. El siguiente código presenta el objeto **ResultSe**t: **rs** y le asigna el resultado de una consulta anterior:

```
ResultSet rs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
```

Utilizar el Método next

La variable **r**s, que es un ejemplar de **ResultSe**t, contiene las filas de cafés y sus precios mostrados en el juego de resultados de la página anterior. Para acceder a los nombres y los precios, iremos a la fila y recuperaremos los valores de acuerdo con sus tipos. El método **next** mueve algo llamado cursor a la siguiente fila y hace que esa fila (llamada fila actual) sea con la que podamos operar. Como el cursor inicialmente se posiciona justo encima de la primera fila de un objeto **ResultSe**t, primero debemos llamar al método **next** para mover el cursor a la primera fila y convertirla en la fila actual.

Sucesivas invocaciones del método **next** moverán el cursor de línea en línea de arriba a abajo. Observa que con el JDBC 2.0, cubierto en la siguiente sección, se puede mover el cursor hacia atrás, hacia posiciones específicas y a posiciones relativas a la fila actual además de mover el cursor hacia adelante.

Utilizar los métodos getXXX

Los métodos **getXXX** del tipo apropiado se utilizan para recuperar el valor de cada columna. Por ejemplo, la primera columna de cada fila de **rs** es **COF_NAM**E, que almacena un valor del tipo **VARCHAR** de SQL. El método para recuperar un valor **VARCHAR** es **getStrin**g. La segunda columna de cada fila almacena un valor del tipo **FLOAT** de SQL, y el método para recuperar valores de ese tipo es **getFloa**t. El siguiente código accede a los valores almacenados en la fila actual de **rs** e imprime una línea con el nombre seguido por tres espacios y el precio. Cada vez que se llama al método **nex**t, la siguiente fila se convierte en la actual, y el bucle continúa hasta que no haya más filas en **r**s:

```
String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
  String s = rs.getString("COF_NAME");
  Float n = rs.getFloat("PRICE");
  System.out.println(s + " " + n);
}
```

La salida se parecerá a esto:

Colombian	7.99
French_Roast	8.99
Espresso	9.99
Colombian_Decaf	8.99
French_Roast_Decaf	9.99

Veamos cómo funcionan los métodos **getXXX** examinando las dos sentencias **getXXX** de este código. Primero examinaremos **getStrin**g.

```
String s = rs.getString("COF_NAME");
```

El método **getString** es invocado sobre el objeto **ResultSe**t: **r**s, por eso **getString** recuperará (obtendrá) el valor almacenado en la columna **COF_NAME** de la fila actual de **r**s. El valor recuperado por **getString** se ha

convertido desde un **VARCHAR** de SQL a un **String** de Java y se ha asignado al objeto **String** s. Observa que utilizamos la variable **s** en la expresión **println** mostrada arriba, de esta forma: **println(s + " " + n)**
La situación es similar con el método **getFloat** excepto en que recupera el valor almacenado en la columna **PRIC**E, que es un **FLOAT** de SQL, y lo convierte a un **float** de Java antes de asignarlo a la variable n.

JDBC ofrece dos formas para identificar la columna de la que un método **getXXX** obtiene un valor. Una forma es dar el nombre de la columna, como se ha hecho arriba. La segunda forma es dar el índice de la columna (el número de columna), con un **1** significando la primera columna, un **2** para la segunda, etc. Si utilizáramos el número de columna en vez del nombre de columna el código anterior se podría parecer a esto:

```
String s = rs.getString(1);
float n = rs.getFloat(2);
```

La primera línea de código obtiene el valor de la primera columna de la fila actual de **rs** (columna **COF_NAM**E), convirtiéndolo a un objeto **String** de Java y asignándolo a s. La segunda línea de código obtiene el valor de la segunda columna de la fila actual de **r**s, lo convierte a un **float** de Java y lo asigna a n. Recuerda que el número de columna se refiere al número de columna en la hoja de resultados no en la tabla original.
En suma, JDBC permite utilizar tanto el nombre cómo el número de la columna como argumento a un método **getXX**X. Utilizar el número de columna es un poco más eficiente, y hay algunos casos
donde es necesario utilizarlo.
JDBC permite muchas lateralidades para utilizar los métodos **getXXX** para obtener diferentes tipos de datos SQL. Por ejemplo, el método **getInt** puede ser utilizado para recuperar cualquier tipo numérico de caracteres. Los datos recuperados serán convertidos a un **in**t; esto es, si el tipo SQL es **VARCHA**R, JDBC intentará convertirlo en un entero. Se recomienda utilizar el método **getInt** sólo para recuperar **INTEGER** de SQL, sin embargo, no puede utilizarse con los tipos **BINAR**Y, **VARBINAR**Y, **LONGVARBINAR**Y, **DAT**E, **TIM**E, o **TIMESTAMP** de SQL.

Métodos para Recuperar Tipos SQL muestra qué métodos pueden utilizarse legalmente para recuperar tipos SQL, y más importante, qué métodos están recomendados para recuperar los distintos tipos SQL. Observa que esta tabla utiliza el término "JDBC type" en lugar de "SQL type." Ambos términos se refieren a los tipos genéricos de SQL definidos en **java.sql.Type**s, y ambos son intercambiables.

Utilizar el método getString
Aunque el metodo **getString** está recomendado para recuperar tipos **CHAR** y **VARCHAR** de SQL, es posible recuperar cualquier tipo básico SQL con él. (Sin embargo, no se pueden recuperar los nuevos tipos de datoas del SQL3. Explicaremos el SQL3 más adelante).
Obtener un valor con **getString** puede ser muy útil, pero tiene sus limitaciones. Por ejemplo, si se está utilizando para recuperar un tipo numérico, **getString** lo convertirá en un **String** de Java, y el
valor tendrá que ser convertido de nuevo a número antes de poder operar con él.

Utilizar los métodos de ResultSet.getXXX para Recuperar tipos JDBC

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR
getByte	X	x	x	x	x	x	x	x	x	x	x
getShort	x	X	x	x	x	x	x	x	x	x	x
getInt	x	x	X	x	x	x	x	x	x	x	x
getLong	x	x	x	X	x	x	x	x	x	x	x
getFloat	x	x	x	x	X	x	x	x	x	x	x
getDouble	x	x	x	x	x	X	X	x	x	x	x
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x
getBoolean	x	x	x	x	x	x	x	x	x	X	x
getString	x	x	x	x	x	x	x	x	x	x	X
getBytes											
getDate											x
getTime											x
getTimestamp											x
getAsciiStream											x
getUnicodeStream											x

getBinaryStream											
getObject	x	x	x	x	x	x	x	x	x	x	x

	VARCHAR	LONG VARCHAR	BINARY	VARBINARY	LONG VARBINARY	DATE	TIME	TIMESTAMP
getByte	x	x						
getShort	x	x						
getInt	x	x						
getLong	x	x						
getFloat	x	x						
getDouble	x	x						
getBigDecimal	x	x						
getBoolean	x	x						
getString	X	x	x	x	x	x	x	x
getBytes			X	X	x			
getDate	x	x				X		x
getTime	x	x					X	x
getTimestamp	x	x				x	x	X
getAsciiStream	x	X	x	x	x			
getUnicodeStream	x	X	x	x	x			
getBinaryStream			x	x	X			
getObject	x	x	x	x	x	x	x	x

Una "x" indica que el método **getXXX** se puede utilizar legalmente para recuperar el tipo JDBC dado.
Una "X" indica que el método **getXXX** está recomendado para recuperar el tipo JDBC dado.

Actualizar Tablas

Supongamos que después de una primera semana exitosa, el propietario de "The Coffee Break" quiere actualizar la columna **SALES** de la tabla **COFFEES** introduciendo el número de libras vendidas de cada tipo de café. La sentencia SQL para actualizar una columna se podría parecer a esto:

```
String updateString = "UPDATE COFFEES " + "SET SALES = 75 " + "WHERE COF_NAME LIKE 'Colombian'";
```

Uitlizando el objeto **stm**t, este código JDBC ejecuta la sentencia SQL contenida en **updateStrin**g:

```
stmt.executeUpdate(updateString);
```

La tabl**aCOFFEES** ahora se parecerá a esto:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	75	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Observa que todavía no hemos actualizado la columna **TOTAL**, y por eso tiene valor 0.
Ahora seleccionaremos la fila que hemos actualizado, recuperando los valores de las columnas **COF_NAME** y **SALES**, e imprimiendo esos valores:

```
String query = "SELECT COF_NAME, SALES FROM COFFEES " +  "WHERE COF_NAME LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
  String s = rs.getString("COF_NAME");
  int n = rs.getInt("SALES");
  System.out.println(n + " pounds of " + s +" sold this week.")
  }
```

Esto imprimira lo siguiente:
75 pounds of Colombian sold this week.
Cómo la claúsula **WHERE** límita la selección a una sóla línea, sólo hay una línea en la **ResultSet: rs** y una línea en la salida. Por lo tanto, sería posible escribir el código sin un bucle **while**:

```
rs.next();
String s = rs.getString(1);
int n = rs.getInt(2);
System.out.println(n + " pounds of " + s + " sold this week.")
```

Aunque hay una sóla línea en la hoja de resultados, necesitamos utilizar el método **next** para acceder a ella. Un objeto **ResultSet** se crea con un cursor apuntando por encima de la primera fila. La primera llamada al método **next** posiciona el cursor en la primera fila (y en este caso, la única) de **r**s. En este código, sólo se llama una vez a **nex**t, si sucediera que existiera una línea, nunca se accedería a ella.

Ahora actualizaremos la columna **TOTAL** añadiendo la cantidad vendida durante la semana a la cantidad total existente, y luego imprimiremos el número de libras vendidas hasta la fecha:

```
String updateString = "UPDATE COFFEES " + "SET TOTAL = TOTAL + 75 " + "WHERE COF_NAME LIKE
'Colombian'";
stmt.executeUpdate(updateString);
String query = "SELECT COF_NAME, TOTAL FROM COFFEES " + "WHERE COF_NAME LIKE
'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
  String s = rs.getString(1);
  Int n = rs.getInt(2);
  System.out.println(n + " pounds of " + s + " sold to date.")
}
```

Observa que en este ejemplo, utilizamos el índice de columna en vez del nombre de columna, suministrando el índice **1** a **getString** (la primera columna de la hoja de resultados es **COF_NAM**E), y el índice **2** a **getInt** (la segunda columna de la hoja de resultados es **TOTA**L). Es importante distinguir entre un índice de columna en la tabla de la base de datos como opuesto al índice en la tabla de la hoja de resultados. Por ejemplo, **TOTAL** es la quinta columna en la tabla **COFFEES** pero es la segunda columna en la hoja de resultados generada por la petición del ejemplo anterior.

Utilizar Sentencias Prepared

Algunas veces es más conveniente o eficiente utilizar objetos **PreparedStatement** para enviar sentencias SQL a la base de datos. Este tipo especial de sentencias se deriva de una clase más general, Statement, que ya conocemos.

Cuándo utilizar un Objeto PreparedStatement

Si queremos ejecutar muchas veces un objeto **Statemen**t, reduciremos el tiempo de ejecución si utilizamos un objeto **PreparedStatemen**t, en su lugar.
La caracterísitca principal de un objeto **PreparedStatement** es que, al contrario que un objeto **Statemen**t, se le entrega una sentencia SQL cuando se crea. La ventaja de esto es que en la mayoría de los casos, esta sentencia SQL se enviará al controlador de la base de datos inmediatamente, donde será compilado. Como resultado, el objeto **PreparedStatement** no sólo contiene una sentencia SQL, sino una sentencia SQL que ha sido precompilada. Esto significa que cuando se ejecuta la **PreparedStatemen**t, el controlador de base de datos puede ejecutarla sin tener que compilarla primero.
Aunque los objetos **PreparedStatement** se pueden utilizar con sentencias SQL sin parámetros, probablemente nosotros utilizaremos más frecuentemente sentencias con parámetros. La ventaja de utilizar sentencias SQL que utilizan parámetros es que podemos utilizar la misma sentencia y suministrar distintos valores cada vez que la ejecutemos. Veremos un ejemplo de esto en las página siguientes.

Crear un Objeto PreparedStatement

Al igual que los objetos **Statemen**t, creamos un objeto **PreparedStatement** con un objeto **Connectio**n. Utilizando nuestra conexión **con** abierta en ejemplos anteriores, podríamos escribir lo siguiente para crear un objeto **PreparedStatement** que tome dos parámetros de entrada:

```
PreparedStatement updateSales = con.prepareStatement("UPDATE COFFEES SET SALES = ?
                                WHERE COF_NAME LIKE ?");
```

La variable **updateSales** contiene la sentencia SQL, **"UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?**", que también ha sido, en la mayoría de los casos, enviada al controlador de la base de datos, y ha sido precompilado.

Suministrar Valores para los Parámetros de un PreparedStatement

Necesitamos suministrar los valores que se utilizarán en los lugares donde están las marcas de interrogación, si hay alguno, antes de ejecutar un objeto **PreparedStatement**. Podemos hacer esto llamando a uno de los métodos **setXXX** definidos en la clase **PreparedStatement**. Si el valor que queremos sustituir por una marca de interrogación es un **int** de Java, podemos llamar al método **setInt**. Si el valor que queremos sustituir es un **String** de Java, podemos llamar al método **setString**, etc. En general, hay un método **setXXX** para cada tipo Java.

Utilizando el objeto **updateSales** del ejemplo anterior, la siguiente línea de código selecciona la primera marca de interrogación para un **int** de Java, con un valor de 75:

```
updateSales.setInt(1, 75);
```

Cómo podríamos asumir a partir de este ejemplo, el primer argumento de un método **setXXX** indica la marca de interrogación que queremos seleccionar, y el segundo argumento el valor que queremos ponerle. El siguiente ejemplo selecciona la segunda marca de interrogación con el string **"Colombian**":

```
updateSales.setString(2, "Colombian");
```

Después de que estos valores hayan sido asignados para sus dos parámetros, la sentencia SQL de **updateSales** será equivalente a la sentencia SQL que hay en string **updateString** que utilizando en el ejemplo anterior. Por lo tanto, los dos fragmentos de código siguientes consiguen la misma cosa:

Código 1:

```
String updateString = "UPDATE COFFEES SET SALES = 75 " + "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
```

Código 2:

```
PreparedStatement updateSales = con.prepareStatement( "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ? ");
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate():
```

Utilizamos el método **executeUpdate** para ejecutar ambas sentencias **stmt updateSales**.

Observa, sin embargo, que no se suministran argumentos a **executeUpdate** cuando se utiliza para ejecutar **updateSales**. Esto es cierto porque **updateSales** ya contiene la sentencia SQL a ejecutar.

Mirando esto ejemplos podríamos preguntarnos por qué utilizar un objeto **PreparedStatement** con parámetros en vez de una simple sentencia, ya que la sentencia simple implica menos pasos. Si actualizáramos la columna **SALES** sólo una o dos veces, no sería necesario utilizar una sentencia SQL con parámetros. Si por otro lado, tuvieramos que actualizarla frecuentemente, podría ser más fácil utilizar un objeto **PreparedStatement**, especialmente en situaciones cuando la utilizamos con un bucle while para seleccionar un parámetro a una sucesión de valores. Veremos este ejemplo más adelante en esta sección.

Una vez que a un parámetro se ha asignado un valor, el valor permanece hasta que lo resetee otro valor o se llame al método **clearParameters**. Utilizando el objeto **PreparedStatement**: **updateSales**, el siguiente fragmento de código reutiliza una sentencia prepared después de resetar el valor de uno de sus parámetros, dejando el otro igual:

```
updateSales.setInt(1, 100);
updateSales.setString(2, "French_Roast");
updateSales.executeUpdate();
// changes SALES column of French Roast row to 100
updateSales.setString(2, "Espresso");
updateSales.executeUpdate();
// changes SALES column of Espresso row to 100 (the first
// parameter stayed 100, and the second parameter was reset
// to "Espresso")
```

Utilizar una Bucle para asignar Valores

Normalmente se codifica más sencillo utilizando un bucle **for** o **while** para asignar valores de los parámetros de entrada.

El siguiente fragmento de código demuestra la utilización de un bucle **for** para asignar los parámetros en un objeto **PreparedStatement**: **updateSales**. El array **salesForWeek** contiene las

cantidades vendidas semanalmente. Estas cantidades corresponden con los nombres de los cafés listados en el array **coffee**s, por eso la primera cantidad de **salesForWeek** (175) se aplica al primer nombre de café de **coffees** ("**Colombia**n"), la segunda cantidad de **salesForWeek** (150) se aplica al segundo nombre de café en **coffees** ("**French_Roast**"), etc. Este fragmento de códigodemuestra la actualización de la columna **SALES** para todos los cafés de la tabla **COFFEES**

```
PreparedStatement updateSales;
String updateString = "update COFFEES " + "set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso", "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
  updateSales.setInt(1, salesForWeek[i]);
  updateSales.setString(2, coffees[i]);
  updateSales.executeUpdate();
}
```

Cuando el propietario quiera actualizar las ventas de la semana siguiente, puede utilizar el mismo código como una plantilla. Todo lo que tiene que haces es introducir las nuevas cantidades en el orden apropiado en el array **salesForWee**k. Los nombres de cafés del array **coffees** permanecen constantes, por eso no necesitan cambiarse. (En una aplicación real, los valores probablemente serían introducidos por el usuario en vez de desde un array inicializado).

Valores de retorno del método executeUpdate

Siempre que **executeQuery** devuelve un objeto **ResultSet** que contiene los resultados de una petición al controlador de la base datos, el valor devuelto por **executeUpdate** es un **int** que indica cuántas líneas de la tabla fueron actualizadas. Por ejemplo, el siguiente código muestra el valor de retorno de **executeUpdate** asignado a la variable n:

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
// n = 1 because one row had a change in it
```

La tabla **COFFEES** se ha actualziado poniendo el valor **50** en la columna **SALES** de la fila correspondiente a **Espresso**. La actualización afecta sólo a una línea de la tabla, por eso **n** es igual a 1.

Cuando el método **executeUpdate** es utilizado para ejecutar una sentecia DDL, como la creación de una tabla, devuelve el **int**: 0. Consecuentemente, en el siguiente fragmento de código, que ejecuta la sentencia DDL utilizada pra crear la tabla **COFFEE**S, **n** tendrá el valor 0:

```
int n = executeUpdate(createTableCoffees); // n = 0
```

Observa que cuando el valor devuelto por **executeUpdate** sea 0, puede significar dos cosas: (1) la sentencia ejecutada no ha actualizado ninguna fila, o (2) la sentencia ejecutada fue una sentencia DDL.

Utilizar Joins

Algunas veces necesitamos utilizar una o más tablas para obtener los datos que queremos. Por ejemplo, supongamos que el propietario del "The Coffee Break" quiere una lista de los cafés que le compra a Acme, Inc. Esto implica información de la tabla **COFFEES** y también de la que vamos a crear **SUPPLIER**S. Este es el caso en que se necesitan los "joins" (unión). Una unión es una operación de base de datos que relaciona dos o más tablas por medio de los valores que comparten.

En nuestro ejemplo, las tablas **COFFEES** y **SUPPLIERS** tienen la columna **SUP_I**D, que puede ser utilizada para unirlas.

Antes de ir más allá, necesitamos crear la tabla **SUPPLIERS** y rellenarla con valores.

El sigueinte código crea la tabla **SUPPLIER**S:

```
String createSUPPLIERS = "create table SUPPLIERS " + "(SUP_ID INTEGER, SUP_NAME
                                    VARCHAR(40), " + "STREET VARCHAR(40), CITY VARCHAR(20), "
                                  + "STATE CHAR(2), ZIP CHAR(5))";
stmt.executeUpdate(createSUPPLIERS);
```

El siguiente código inserta filas para tres suministradores dentro de **SUPPLIER**S:

```
stmt.executeUpdate("insert into SUPPLIERS values (101, " + "'Acme, Inc.', '99 Market Street', 'Groundsville',
" + "'CA', '95199'");
stmt.executeUpdate("Insert into SUPPLIERS values (49," + "'Superior Coffee', '1 Party Place', 'Mendocino',
'CA', " + "'95460'");
```

```
stmt.executeUpdate("Insert into SUPPLIERS values (150, " + "'The High Ground', '100 Coffee Lane',
'Meadows', 'CA', " + "'93966'");
```

El siguiente código selecciona la tabla y nos permite verla:

```
ResultSet rs = stmt.executeQuery("select * from SUPPLIERS");
```

El resultado sería algo similar a esto:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Ahora que tenemos las tablas **COFFEES** y **SUPPLIER**S, podremos proceder con el escenario en que el propietario quería una lista de los cafés comprados a un suministrador particular. Los nombres de los suminstradores están en la tabla **SUPPLIER**S, y los nombres de los cafés en la tabla **COFFEE**S.
Como ambas tablas tienen la columna **SUP_I**D, podemos utilizar esta columna en una unión. Lo siguiente que necesitamos es la forma de distinguir la columna **SUP_ID** a la que nos referimos. Esto se hace precediendo el nombre de la columna con el nombre de la tabla, **"COFFEES.SUP_I**D" para indicar que queremos referirnos a la columna **SUP_ID** de la tabla **COFFEE**S. En el siguiente código, donde **stmt** es un objeto **Statemen**t, seleccionamos los cafés comprados a Acme, Inc.:

```
String query = " SELECT COFFEES.COF_NAME " + "FROM COFFEES, SUPPLIERS " +
                  "WHERE SUPPLIERS.SUP_NAME LIKE 'Acme, Inc.'" +"and SUPPLIERS.SUP_ID
                    = COFFEES.SUP_ID";
ResultSet rs = stmt.executeQuery(query);
System.out.println("Coffees bought from Acme, Inc.: ");
while (rs.next()) {
  String coffeeName = getString("COF_NAME");
  System.out.println(" " + coffeeName);
}
```

Esto producirá la siguiente salida:

```
Coffees bought from Acme, Inc.:
Colombian
Colombian_Decaf
```

Utilizar Transacciones

Hay veces que no queremos que una sentencia tenga efecto a menos que otra también suceda. Por ejemplo, cuando el propietario del "The Coffee Break" actualiza la cantidad de café vendida semanalmente, también querrá actualizar la cantidad total vendida hasta la fecha. Sin embargo, el no querrá actualizar una sin actualizar la otra; de otro modo, los datos serían inconsistentes. La forma para asegurarnos que ocurren las dos acciones o que no ocurre ninguna es utilizar una transacción. **Una transacción es un conjunto de una o más sentencias que se ejecutan como una unidad, por eso o se ejecutan todas o no se ejecuta ninguna.**

Desactivar el modo Auto-entrega

Cuando se crea una conexión, está en modo auto-entrega. Esto significa que cada sentencia SQL individual es tratada como una transacción y será automáticamente entregada justo después de ser ejecutada. (Para ser más preciso, por defecto, una sentencia SQL será entregada cuando está completa, no cuando se ejecuta. Una sentencia está completa cuando todas sus hojas de resultados y cuentas de actualización han sido recuperadas. Sin embargo, en la mayoría de los casos, una sentencia está completa, y por lo tanto, entregada, justo después de ser ejecutada).
La forma de permitir que dos o más sentencia sean agrupadas en una transacción es desactivar el modo auto-entrega. Esto se demuestra en el siguiente código, donde **con** es una conexión activa:

```
con.setAutoCommit(false);
```

Entregar una Transacción

Una vez que se ha desactivado la auto-entrega, no se entregará ninguna sentencia SQL hasta que llamemos explícitamente al método **commit**. Todas las sentencias ejecutadas después de la anterior llamada al método **commit** serán incluidas en la transacción actual y serán entregadas juntas como una unidad. El siguiente código, en el que **con** es una conexión activa, ilustra una transacción:

```
con.setAutoCommit(false);
```

```
PreparedStatement updateSales = con.prepareStatement( "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement( "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

En este ejemplo, el modo auto-entrega se desactiva para la conexión **co**n, lo que significa que las dos sentencias prepared **updateSales** y **updateTotal** serán entregadas juntas cuando se llame al método **commi**t. Siempre que se llame al método **commit** (bien automáticamente, cuando está activado el modo auto-commit o explícitamente cuando está desactivado), todos los cambios resultantes de las sentencias de la transacción serán permanentes. En este caso, significa que las columnas **SALES** y **TOTAL** para el café Colombian han sido cambiadas a **50** (si **TOTAL** ha sido **0** anteriormente) y mantendrá este valor hasta que se cambie con otra sentencia de actualización.
La línea final del ejemplo anterior activa el modo auto-commit, lo que significa que cada sentencia será de nuevo entregada automáticamente cuando esté completa. Volvemos por lo tanto al estado por defecto, en el que no tenemos que llamar al método **commi**t. Es bueno desactivar el modo auto-commit sólo mientras queramos estar en modo transacción.
De esta forma, evitamos bloquear la base de datos durante varias sentencias, lo que incrementa los conflictos con otros usuarios.

Utilizar Transacciones para Preservar al Integridad de los Datos
Además de agrupar las sentencias para ejecutarlas como una unidad, las transacciones pueden ayudarnos a preservar la integridad de los datos de una tabla. Por ejemplo, supongamos que un empleado se ha propuesto introducir los nuevos precios de los cafés en la tabla **COFFEES** pero lo retrasa unos días. Mientras tanto, los precios han subido, y hoy el propietario está introduciendo los nuevos precios. Finalmente el empleado empieza a introducir los precios ahora desfasados al mismo tiempo que el propietario intenta actualizar la tabla. Después de insertar los precios desfasados, el empleado se da cuenta de que ya no son válidos y llama el método **rollback** de la **Connection** para deshacer sus efectos. (El método **rollback** aborta la transacción y restaura los valores que había antes de intentar la actualziación. Al mismo tiempo, el propietario está ejecutando una sentencia **SELECT** e imprime los nuevos precios. En esta situación, es posible que el propietario imprima los precios que más tarde serían devueltos a sus valores anteriores, haciendo que los precio impresos sean incorrectos.

Leer A A = 50	Leer A A = 70

Esta clase de situaciones puede evitarse utilizando Transacciones. Si un controlador de base de datos soporta transacciones, y casi todos lo hacen, proporcionará algún nivel de protección contra conflictos que pueden surgir cuando dos usuarios acceden a los datos a la misma vez.

Para evitar conflictos durante una transacción, un controlador de base de datos utiliza bloqueos, mecanismos para bloquear el acceso de otros a los datos que están siendo accedidos por una transacción. (Observa que en el modo auto-commit, donde cada sentencia es una transacción, el bloqueo sólo se mantiene durante una sentencia). Una vez activado, el bloqueo permanece hasta que la transacción sea entregada o anulada. Por ejemplo, un controlador de base de datos podría bloquear una fila de una tabla hasta que la actualización se haya entregado. El efecto de este bloqueo es evitar que usuario obtenga una lectura sucia, esto es, que lea un valor antes de que sea permanente. (Acceder a un valor actualizado que no haya sido entregado se considera una lectura sucia porque es posible que el valor sea devuelto a su valor anterior. Si leemos un valor que luego es devuelto a su valor antiguo, habremos leído un valor nulo).

La forma en que se configuran los bloqueos está determinado por lo que se llama nivel de aislamiento de transacción, que pude variar desde no soportar transacciones en absoluto a soportar todas las transacciones que fuerzan una reglas de acceso muy estrictas.

Un ejemplo de nivel de aislamiento de transacción es **TRANSACTION_READ_COMMITTE**D, que no permite que se acceda a un valor hasta que haya sido entregado. En otras palabras, si nivel de aislamiento de transacción se selecciona a **TRANSACTION_READ_COMMITTE**D, el controlador de la base de datos no permitirá que ocurran lecturas sucias. El interface **Connection** incluye cinco valores que representan los niveles de aislamiento de transacción que se pueden utilizar en JDBC.
Normalmente, no se necesita cambiar el nivel de aislamiento de transacción; podemos utilizar el valor por defecto de nuestro controlador. JDBC permite averiguar el nivel de aislamiento de transacción de nuestro controlador de la base de datos (utilizando el método **getTransactionIsolation** de **Connectio**n) y permite configurarlo a otro nivel (utilizando el método **setTransactionIsolation** de **Connectio**n). Sin embargo, ten en cuenta, que aunque JDBC permite seleccionar un nivel de aislamiento, hacer esto no tendrá ningún efecto a no ser que el driver del controlador de la base de datos lo soporte.

Cuándo llamar al método rollback
Como se mencionó anteriormente, llamar al método **rollback** aborta la transacción y devuelve cualquier valor que fuera modificado a sus valores anteriores. Si estamos intentando ejecutar una o más sentencias en una transacción y obtenemos una **SQLExceptio**n, deberíamos llamar al método **rollback** para abortar la transacción y empezarla de nuevo. Esta es la única forma para asegurarnos de cuál ha sido entregada y cuál no ha sido entregada. Capturar una **SQLException** nos dice que hay algo erróneo, pero no nos dice si fue o no fue entregada. Como no podemos contar con el hecho de que nada fue entregado, llamar al método **rollback** es la única forma de asegurarnos.

Procedimientos Almacenados
Un procedimiento almacenado es un grupo de sentencias SQL que forman una unidad lógica y que realizan una tarea particular. Los procedimientos almacenados se utilizan para encapsular un conjunto de operaciones o peticiones para ejecutar en un servidor de base de datos. Por ejemplo, las operaciones sobre una base de datos de empleados (salarios, despidos, promociones, bloqueos) podrían ser codificados como procedimientos almacenados ejecutados por el código de la aplicación. Los procedimientos almacenados pueden compilarse y ejecutarse con diferentes parámetros y resultados, y podrían tener cualquier combinación de parámtros de entrada/salida.
Los procedimientos almacenados están soportados por la mayoría de los controladores de bases de datos, pero existe una gran cantidad de variaciones en su síntaxis y capacidades. Por esta razón, sólo mostraremos un ejemplo sencillo de lo que podría ser un procedimiento almacenado y cómo llamarlos desde JDBC, pero este ejemplo no está diseñado para ejecutarse.

Conectar JDBC-ODBC

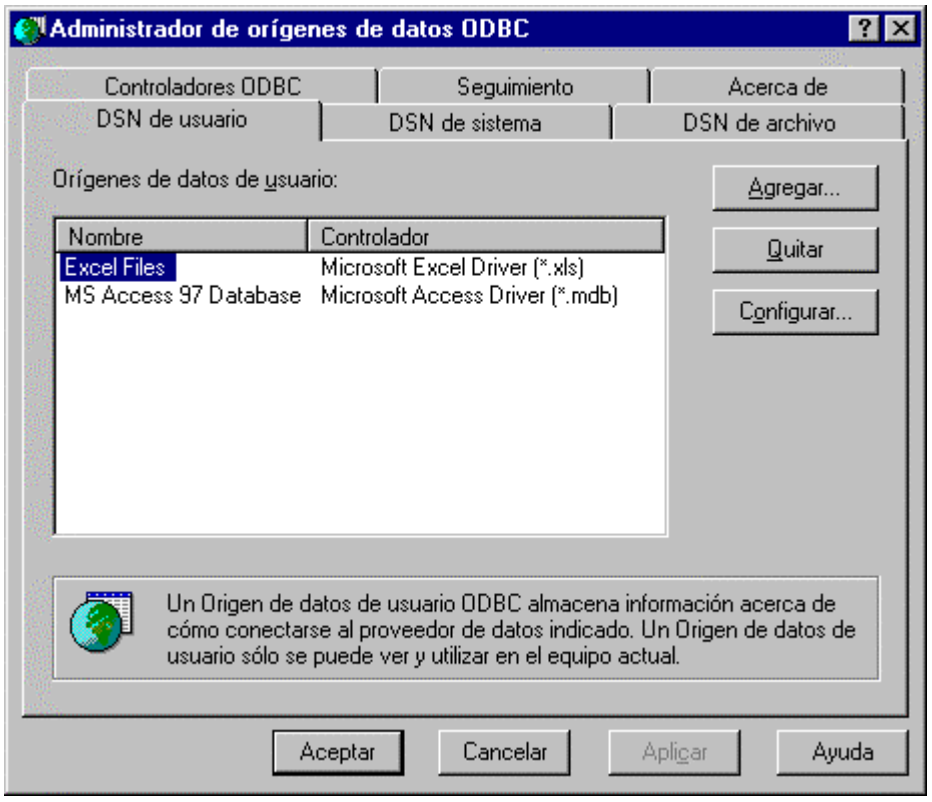
Crear un origen de datos ODBC

El administrador de origen de datos ODBC está situado en el panel de control con el nombre de *Fuentes de Datos 0DBC32 bits*, entonces abrir el **Panel de Control** y localizar el icono llamado *32bit ODBC*.

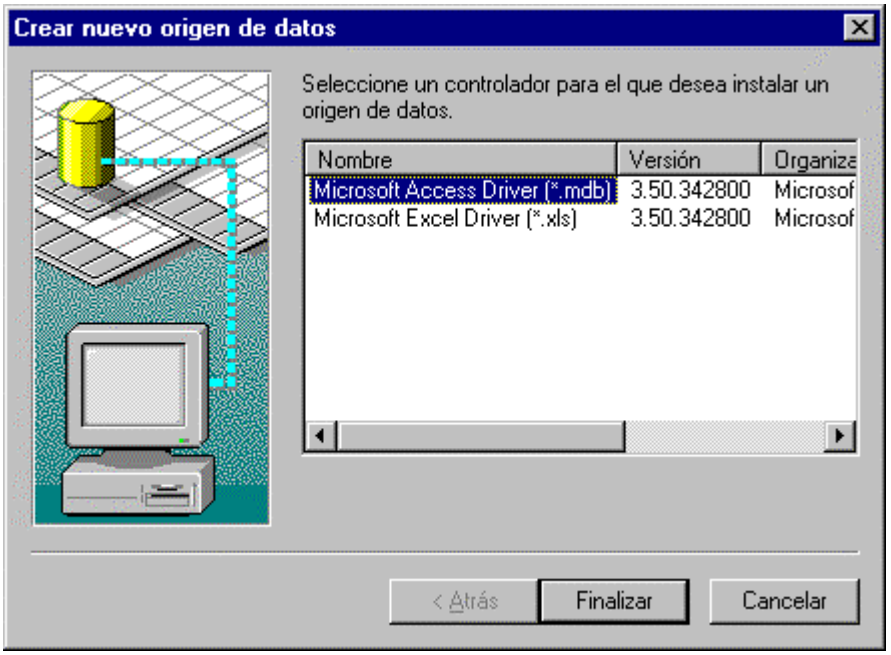
Una vez que pulsamos sobre el icono aparece una ficha llamada **Controladores ODBC** en la cual se pueden apreciar los distintos controladores que tenemos instalados.

Junto a la ficha de **Controladores ODBC** aparecen otras fichas que nos permiten establecer el origen de datos de usuario (DSN usuario), de sistema (DSN de sistema) y de archivo (DSN de archivo). En cada una de estas fichas podemos agregar, configurar y quitar orígenes de datos.
Lo primero que vamos hacer para crear el origen de datos es establecer el nivel de acceso, para ello existen tres niveles de acceso: de usuario, de sistema y de archivo. En este caso, utilizamos un nivel de acceso de usuario (DSN usuario), por lo tanto, seleccionamos la ficha **DSN de usuario**.

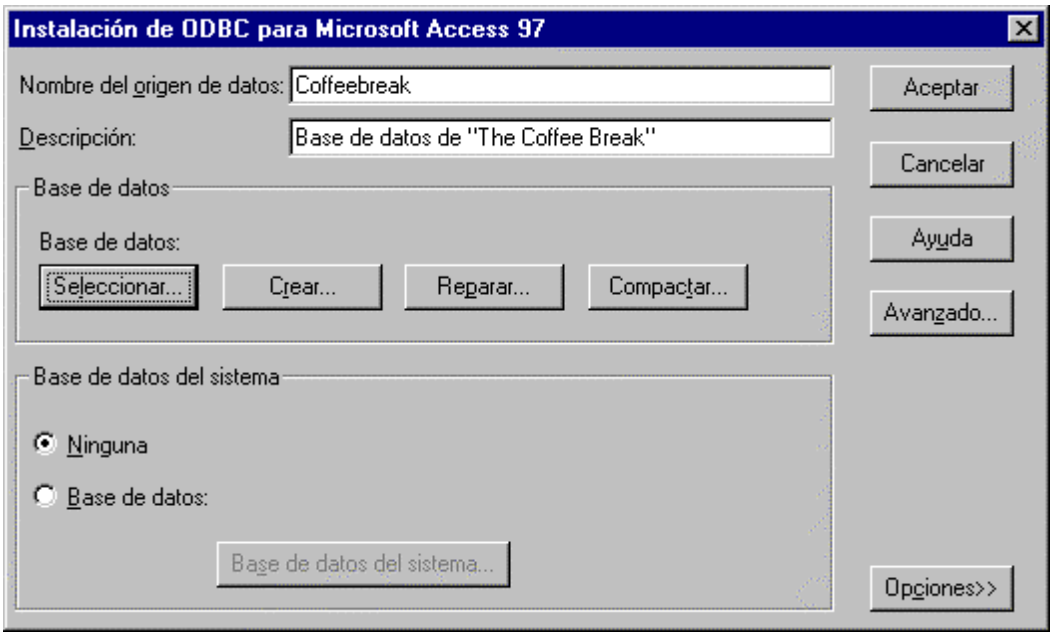
La ficha **DSN de usuario** posee tres botones, uno para agregar, otro para configurar y otro para eliminar el origen de datos. Entonces, pulsamos el botón de *Agregar* para crear el origen de datos “Coffeebreak”.



Una vez realizado esto, aparecen los controladores ODBC que se encuentran ya instalados, en este caso, escogeremos **Microsoft Access Driver** y después el botón *Finalizar*.



Una vez elegido el tipo de controlador, insertaremos el nombre del origen de datos (**Coffeebreak**) en la siguiente ventana:



Con el botón *Seleccionar*, elegimos la bases de datos **Coffeebreak.mdb**, que fue creada anteriormente en MS Access.

Finalmente, ya tenemos instalado el origen de datos de nuestra aplicación, y pulsamos el botón *Aceptar* para guardar los cambios.