Notas técnicas de JAVA - Tip en detalle Nro. 2

(Lo nuevo, lo escondido, o simplemente lo de siempre pero bien explicado)

Una introducción a JDBC (Java Database Connectivity) (Acceso a bases de datos desde JAVA)

Tema: JDBC. ODBC. DBMS, Fuente de datos, API, SQL.

Descripción: Una introducción conceptual y práctica a la tecnología JDBC.

Nivel: Intermedio

Fecha pub: Enero 2004

---

*"Notas Técnicas de JAVA" se envía con frecuencia variable y absolutamente **sin cargo** como un servicio a nuestros clientes. Contiene notas/recursos/artículos técnicos desarrollados en forma totalmente objetiva e independiente. Teknoda es una organización de servicios de tecnología informática y **NO comercializa hardware, software ni otros productos**. Si desea suscribir otra dirección de e-mail para que comience a recibir los tips envíe un mensaje desde esa dirección a develop@teknoda.com, indicando su nombre, empresa a la que pertenece, cargo y país.*

Próximos Tips:

- *Concepto de "frameworks" y STRUTS.*
- *Cómo conformar un entorno de Programación JAVA Parte II: Application Servers*

Lista de Tips publicados hasta la fecha:

1. Cómo conformar un entorno de programación JAVA (serie de varios tips). Parte I: Selección e instalación de un IDE gratuito.

Tabla de contenido

El objetivo de este tip es cubrir los aspectos más relevantes en el uso de JDBC. Los siguientes puntos serán cubiertos en el presente tip:

I. Introducción / Resumen Ejecutivo

Qué es JDBC

JDBC (Java DataBase Connectivity), es una interfaz de programación que permite a los programas JAVA ejecutar sentencias SQL, para gestionar sobre cualquier base de datos "SQL-compliant". Dado que la mayoría de las bases de datos relacionales hoy soportan SQL, **JDBC se ha convertido en el estándar de la industria para que una aplicación JAVA pueda trabajar sobre sus datos**, creando archivos o realizando cualquier tipo de operación sobre ellos.

La tecnología JDBC parte de una especificación, que se materializa como **un conjunto de clases, métodos e interfaces**, que actúan en distintos niveles del entorno JAVA. El nivel **más bajo** de las componentes JDBC es **variable y configurable,** dado que es dependiente de el/los DBMS (Oracle, SQL Server, etc.) a los que se desea acceder. Existe un driver JDBC para Sybase, otro para Oracle, otro para DB2/400, etc.

Justamente, **la característica más importante de la API JDBC es que aísla por completo el código de la aplicación de las características particulares de cada fuente de datos.** En otras palabras, utilizando la API JDBC, el mismo código puede ejecutar sobre una base de datos Sybase, o sobre cualquier DBMS para la cual exista un driver JDBC disponible y configurado.

La natural portabilidad de Java, sumada a la API JDBC permite explotar realmente el tan ansiado "Write Once and Run Anywhere".

Aunque JDBC está principalmente orientado al envío de sentencias SQL, desde la versión 2.0 permite interactuar con otras clases de fuentes de datos, proporcionando también un soporte que posibilita la lectura y grabación sobre datos tabulados.

ODBC-JDBC Bridge

JDBC es similar a la especificación ODBC; de hecho se deriva de él; pero está diseñado específicamente para programas JAVA. Fue desarrollado por Sun. La API JDBC está incluida en la JAVA 2 Platform Standard Edition (J2SE). Implementa operaciones JDBC, traduciéndolas en operaciones ODBC. ODBC las recibe como las operaciones de un programa aplicativo normal. Este tipo de drivers es normalmente usado cuando no existen drivers JDBC nativos.

La implementación de JDBC

Por definición, una plataforma JAVA-enabled debe soportar un conjunto amplio y previamente especificado de librerías de API's. Existen dos librerías relacionadas con JDBC: **java.sql** y **javax.sql.**

- ***java.sql***: esta es la librería necesaria para la implementación de JDBC, y es la que ofrece las funcionalidades básicas de la herramienta. Es la encargada de proporcionar las API´s necesarias para acceder y procesar datos almacenados en una base de datos. Incluye drivers que pueden ser instalados dinámicamente y que permitirán el acceso a distintos DBMS. Dentro del paquete *java.sql* existe gran cantidad de clases, métodos e interfaces que permitirán entre otras posibilidades:
  - iniciar una conexión con la base de datos
  - enviar sentencias SQL a la base de datos
  - recuperar y actualizar los datos recibidos como resultado de una consulta
  - "mappear" tipos de datos de SQL a clases e interfaces en el lenguaje de programación JAVA
  - acceder a infomación propia de las tablas (cabeceras de columnas, tipos de campos, etc), utilizar interfaces que permiten la invocación de SQL stored procedures, manejo de excepciones relacionadas, entre otras posibilidades.

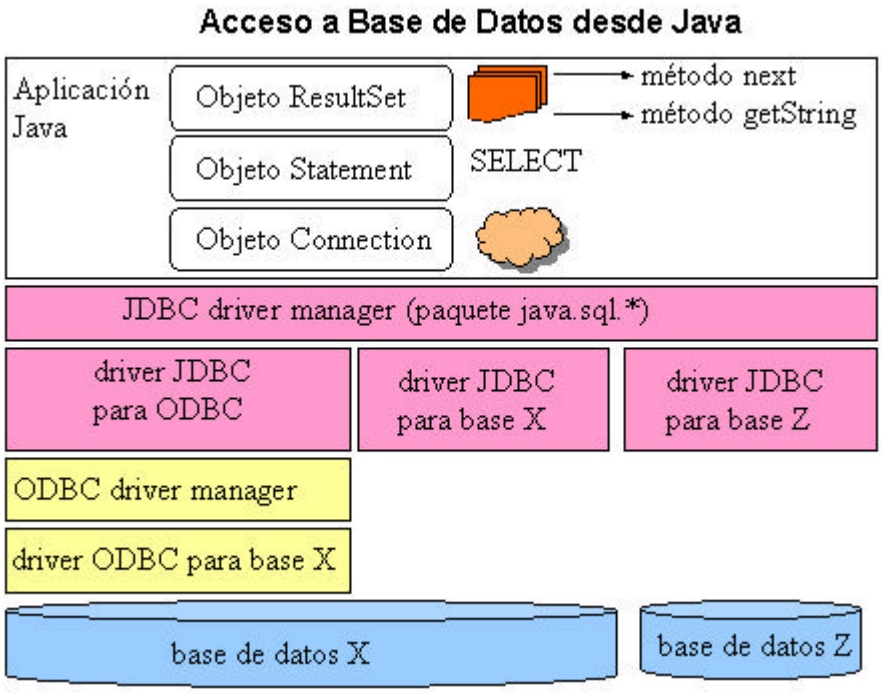
  Este paquete, de alguna manera, puede ser visto como una versión "portable" de ODBC.

- ***Javax.sql***: esta librería ofrece funcionalidades adicionales. Proporciona las API´s necesarias para acceder a fuentes de datos del lado del servidor y su procesamiento desde Java. **Este paquete funciona como un suplemento del paquete java.sql**, e incluye clases, interfaces y métodos que implementan formas más eficientes de llevar a cabo ciertas tareas, por ejemplo la conexión con la base de datos. Posee también **connection pooling**, que permite que una conexión sea utilizada y reutilizada, reduciendo el número de conexiones que es necesario crear y mejorando la performance. Permite también el manejo de **transacciones distribuidas**, que posibilita involucrar en una sola transacción, información proveniente desde distintos orígenes de datos.

Ambos paquetes integran el Java 2 Platform Standard Edition (J2SE) versión 1.4. El paquete *javax.sql* forma parte además del Java 2 Platform Enterprise Edition (J2EE) versión 1.3.1. En ambos casos, se incluye la versión 3.0 de JDBC, última de la familia.

Arquitectura de JDBC

La siguiente imagen muestra la aquitectura de JDBC, y cuales son los componentes que intervienen en la ejecución exitosa de una sentencia SQL. En la parte inferior del esquema está ubicada la **base de datos** que se desea acceder. Sobre ella, **el driver de base de datos** correspondiente. En el caso de un ODBC-JDBC bridge, como se mencionára anteriormente, funcionará como un traductor de sentencias SQL en sentencias comprensibles por el DBMS en uso. El **paquete java.sql**, implementación básica de JDBC. Dentro del paquete java.sql existen tres clases que juegan un papel importante en el uso de JDBC: **Connection, Statement y ResultSet**. **Connection**, el primero de ellos se utilizará en el establecimiento de la conexión con el origen de datos. **Statement** es el que contendrá la sentencia SQL, y a través de la aplicación de diferentes métodos se ejecutarán las sentencias. **ResultSet** juega el importante papel de almacenar el resultado de la ejecución de sentencias SELECT. Luego, "recorriendo" este tipo de objeto, se podrán obtener cada uno de los registros obtenidos en la ejecución.



II. Instalación de JDBC y del driver apropiado

Antes de comenzar a trabajar, es importante verificar que todo aquel producto necesario para trabajar, haya sido instalado. Los siguiente son los pasos que deben llevarse a cabo para poder escribir un programa que utilice JDBC:

Instalación de JDBC

Para instalar Java y JDBC, seguir las instrucciones para descargar la última versión de JDK. Cuando la descarga esté completa, también se tendrá instalado JDBC. Para descargar la última versión disponible, ingresar en http://java.sun.com/products/JDK/CurrentRelease .

En el Tip número 1 de esta serie "Cómo conformar un entorno de programación Java", se encuentra el detalle de cómo descargar la última versión disponible de JSDK.

Clasificación de drivers

Los drivers para tecnología JDBC se encuentran actualmente clasificados en cuatro categorías:

1. **JDBC-ODBC Bridge**: proporciona acceso JDBC API vía uno o más drivers ODBC. Este tipo de drivers es recomendado solamente para desarrollo de prototipos, bases poco masivas o cuando se decide dejar para una etapa posterior del proyecto la elección del driver. Implementa operaciones JDBC, traduciéndolas en operaciones ODBC. ODBC las recibe como las operaciones de un programa aplicativo normal. El "bridge" implementa JDBC para cualquier base de datos para la cual existe un driver ODBC disponible. Para información particular sobre el "bridge" provisto por Sun, ingresar en http://java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/bridge.doc.html#996747 .
2. **Native-API partly Java technology-enabled driver**: convierte llamadas JDBC en llamadas a las API del cliente para bases de datos Oracle, Sybase, Informix, DB2 o otros DBMS. Como el driver JDBC-ODBC bridge, este tipo de driver requiere que algún código binario sea cargado en cada puesto cliente.
3. **Net-protocol fully Java technology-enabled driver:** traduce llamadas a API de JDBC en un protocolo de red independiente del DBMS, el cual es luego traducido a un protocolo DBMS por un servidor. Este middleware de servidor de red permite a sus clientes basados en tecnología Java conectarse a muchas bases de datos diferentes.

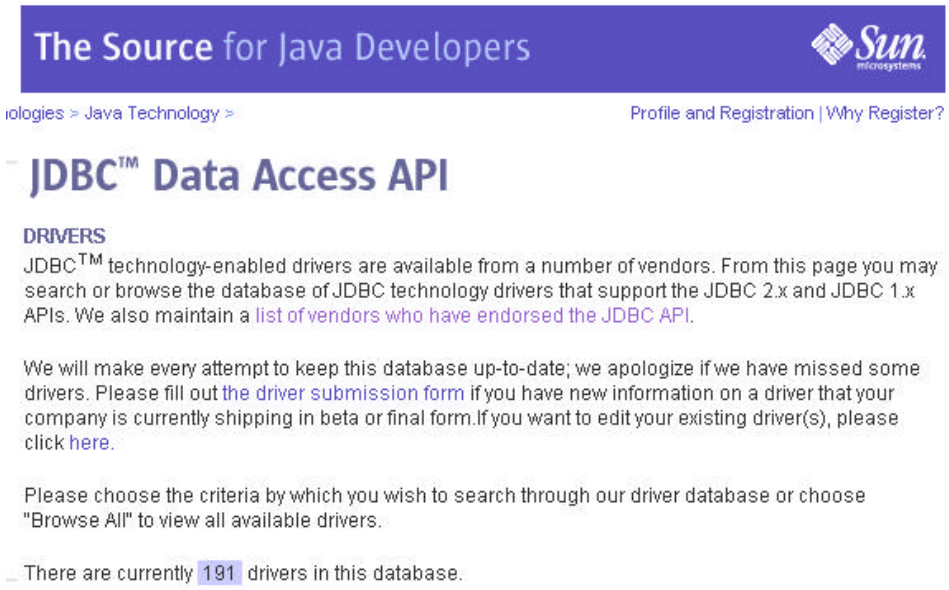
El protocolo específico utilizado es dependiente del vendedor. En general, esta opción puede ser la alternativa JDBC API más flexible.

4. **Native-protocol fully Java technology-enabled driver**: convierte llamadas a tecnología JDBC en invocaciones al protocolo de red usado por el DBMS instalado. Esto permite que una llamada directa desde el puesto cliente al servidor DBMS, y representa una solución práctica para el acceso Intranet. Puesto que muchos de estos protocolos son propietarios, los vendedores de bases de datos serán los proveedores primarios para este tipo de drivers. Varios proveedores de bases de datos tienen este tipo de drivers en desarrollo aún.

Cómo buscar el driver apropiado

Existe gran cantidad de proveedores de drivers en el mercado. El siguiente link permite buscar el driver necesario según su tipo, el DBMS instalado, la versión de JDBC que se desea usar, entre otros datos:

http://servlet.java.sun.com/products/jdbc/drivers



En la parte inferior de esta misma página se pueden observar los distintos datos solicitados para realizar búsquedas de drivers.

Una vez seleccionado el driver a utilizar, se deberán seguir las instrucciones de instalación provistas por el vendedor.

III. Estableciendo una conexión utilizando JDBC

Cuando se necesita acceder a una base de datos utilizando JDBC, existen ciertos"pasos" que deben respetarse: la carga del driver instalado y el establecimiento de la conexión propiamente dicho:

Carga del driver JDBC correspondiente

La carga del driver que se necesita utilizar es muy simple y se resuelve en sólo una línea de código. La documentación del driver incluye, entre otros datos, el nombre de la clase que se debe utilizar. Las siguientes sentencias Java son ejemplos de cargas de drivers para distintos DBMS:

```
Class.forName("com.ibm.as400.access.AS400JDBCDriver")
```

```
Class.forName("oracle.jdbc.driver.OracleDriver")
```

El nombre de la clase a cargar a través de la sentencia Java anterior, debe ser proporcionada por el proveedor del driver.

Establecimiento de una conexión

Luego de realizada la carga del driver, se está en condiciones de establecer una conexión con el DBMS. La siguiente línea de código ilustra el fromato de la sentencia Java a utilizar:

```
Connection con = DriverManager.getConnection(url,"user","password")
```

Donde:

`url`:	especifica el protocolo de JDBC, un subprotocolo y el dominio o dirección IP del sistema AS/400 al que se accederá. Tanto el protocolo como el subprotocolo deberán ser proporcionados por la firma proveedora del driver.
`user`:	usuario habilitado para ingresar al sistema.
`password`:	contraseña del usuario.

El objeto **`DriverManager`** se utiliza para, a través de sus métodos disponibles, gestionar los servicios básicos de JDBC, por ejemplo, obtener una conexión.

El método **`getConnection`**, aplicado sobre el objeto DriverManager, intenta establecer una conexión a la base de datos especificada en la URL que recibe como parámetro, utilizando el usuario y contraseña también proporcionados. Como resultado, devuelve un objeto de tipo **`Connection`** y de nombre **`con`** (en este ejemplo).

A partir de aquí, se dispone de una conexión abierta la cual puede utilizarse para enviar sentencias SQL al DBMS.

Ejemplo de conexión:

```
Connection con =
DriverManager.getConnection("jdbc:as400://169.145.220.37","java","java")
```

Parametrización de nombres de driver y atributos de conexión

En los ejemplos de carga de drivers mostrados previamente se puede observar que el nombre del driver a utilizar se encuentra "hardcodeado" dentro del código del programa Java. Si se necesitara acceder a otro DBMS, sería también necesario cambiar la sentencia anterior y, por lo tanto, recompilar el programa. Para evitar esto, **el nombre del driver a cargar puede almacenarse en un "properties file" (archivos de tipo txt). De esta manera, cambiar el driver en el archivo externo, no afecta al programa Java encargado de realizar la carga del driver.** Esta práctica es muy común, y no es sólo utilizada para parametrizar el nombre del driver, sino también para muchas otras propiedades de un proyecto. Por ejemplo: nombre del usuario, texto a imprimir en facturas, nombre de la empresa, etc. Actualmente, para almacenar las propiedades de los proyectos son utilizadas hojas XML o archivos de tipo JNDI (Java Naming and Directory Interface).

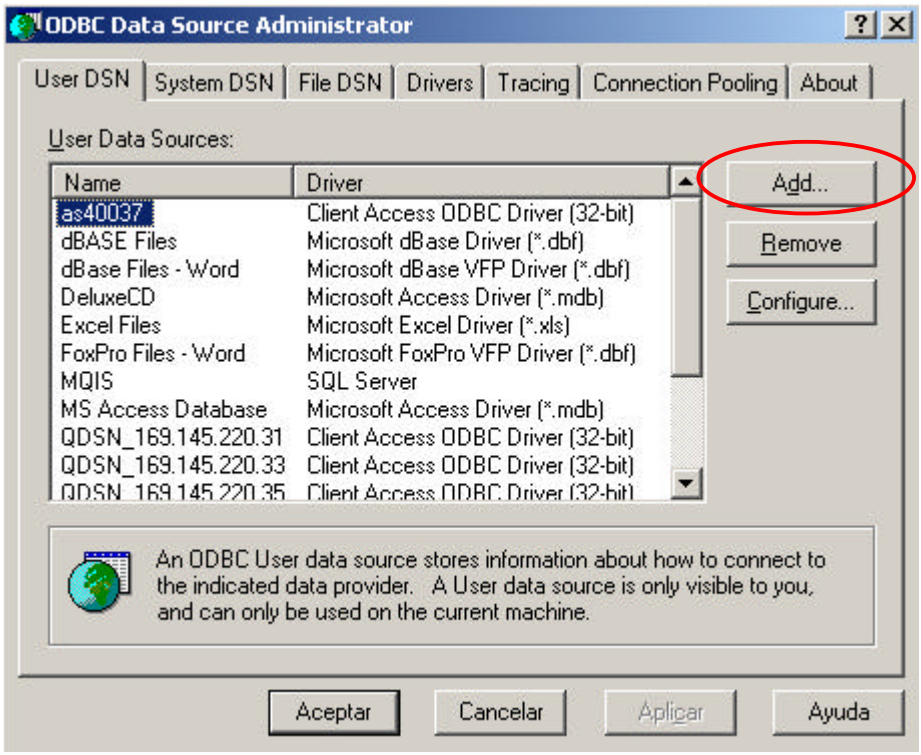
IV. Ejemplo completo de JDBC

El siguiente ejemplo cubrirá la creación, carga y visualización del contenido de una tabla operando sobre Microsoft Access como DBMS. La base de datos Access deberá ser creada con anterioridad, sin necesidad de crear tablas, ya que esa tarea se realizará desde Java. Para ello, es necesario utilizar un driver tipo 1 JDBC-ODBC Bridge, que traducirá los accesos SQL solicitados desde Java a ODBC, que serán finalmente enviadas al gestor de BD. El origen de datos ODBC también deberá ser previamente creado.

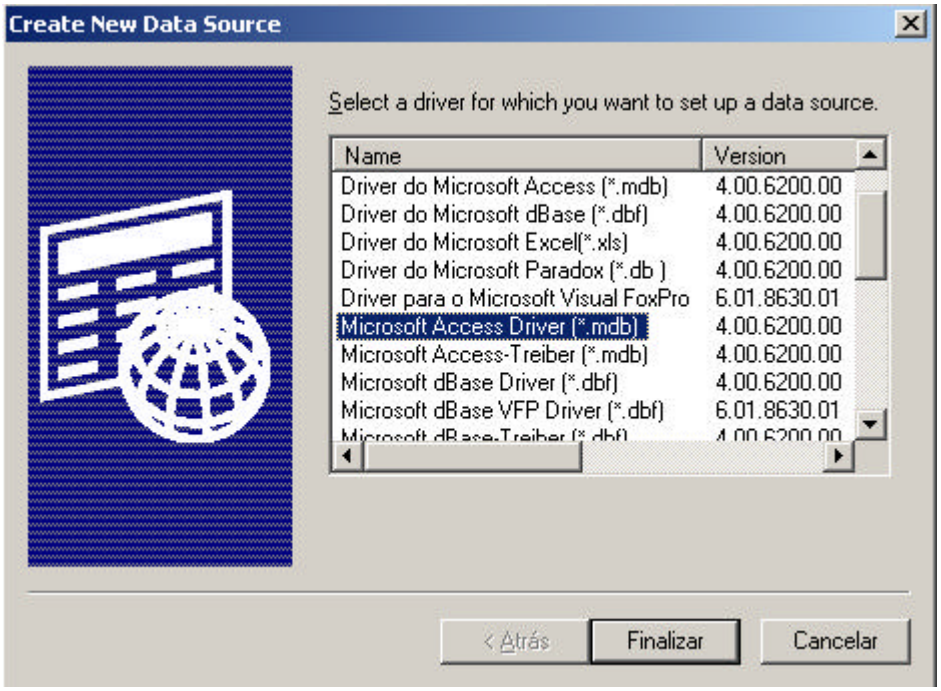
Creación del origen de datos ODBC

Para crear un rigen de datos ODBC que "apunte" a nuestra base de datos MS Access a utilizar, se deben seguir los siguientes pasos, desde el puesto donde se trabajará:

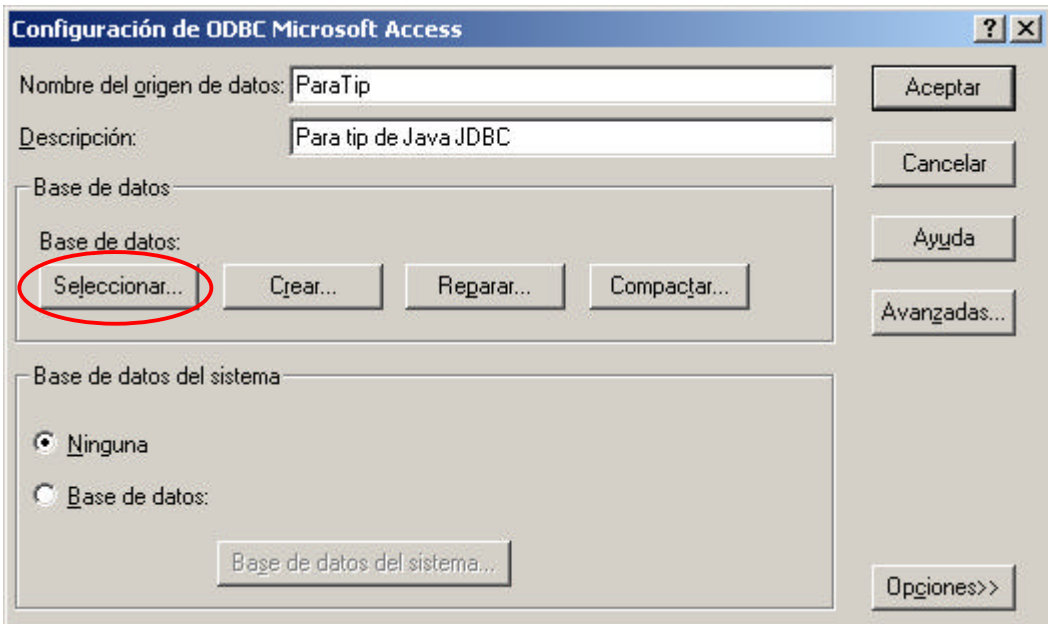
1. Presionar **Inicio → Configuración → Panel de control**.
2. Seleccionar **Herramientas administrativas**.
3. Seleccionar **Orígenes de datos ODBC**.
4. En la ventana **ODBC Data Source Administrator**, presionar el botón **Add**:



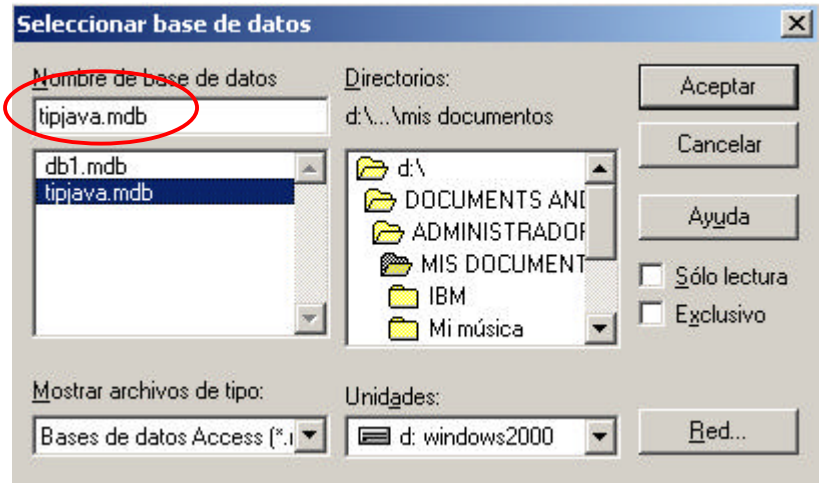
5. Se visualiza la ventana **Create New Data Source**, donde se selecciona el gestor de base de datos para el cual se necesita crear el origen de datos:



6. Presionar el botón **Finalizar**. Se visualiza la ventana **Configuración de ODBC Microsoft Access**. Completar un nombre para el origen de datos y una descripción. Presionar el botón **Seleccionar** para la sección **Base de datos**:



7. En la ventana **Seleccionar base de datos**, especificar el path donde la base de datos a utilizar está ubicada.



8. Presionar **Aceptar** en las siguientes ventanas hasta salir.

Una vez realizada esta tarea, procederemos con la creación de una tabla. Tanto para la creación de la tabla, como también para la carga y visualización de los datos, se crearán clases de Java que utilizarán JDBC. Los fuentes serán escritos utilizando Eclipse, uno de los IDE's disponibles para Java (para más información, consultar el Tip número 1: Cómo conformar un entorno de programación JAVA. Parte I: Selección e instalación de un IDE gratuito).

Creación de una tabla

El siguiente fuente Java muestra, de manera sencilla, los pasos necesarios para crear una archivo utilizando JDBC:

```
      // Imports
(1)   import java.sql.*;


      // Declaración de clase

(2)   public class CrearTabla
      {

          // Declaración de método main de la clase

(3)       public static void main(String[] args)
          {
(4)         Connection con  = null;
            Statement  stmt = null;

            try {
```

```
            // Carga del driver y establecimiento de la conexión

(5)         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con =DriverManager.getConnection("jdbc:odbc:ParaTip", "", "");

(6)         stmt = con.createStatement();


            // Ejecución de la sentencia SQL

(7)         stmt.executeUpdate("CREATE TABLE      CLIENTES " +
                               "(TIPO            VARCHAR (20 )    NOT NULL, " +
                               " NRODOC          INT              NOT NULL, " +
                               " APELLIDO_NOMBRE VARCHAR (50 )    NOT NULL, " +
                               " DOMICILIO       VARCHAR (50 )    NOT NULL, " +
                               " LIMITE_CREDITO  INT              NOT NULL, " +
                               " PRIMARY KEY     (NRODOC))");

(8)        System.out.println(" Tabla CLIENTES creada");


            // Manejo de errores


(9)         } catch ( ClassNotFoundException cnfe) {
              System.out.println("Driver de Base de Datos no encontrado");
              cnfe.getMessage();
              cnfe.printStackTrace();
            } catch ( SQLException sqle) {
              System.out.println("Problema con instrucción SQL");
              sqle.getMessage();
              sqle.printStackTrace();
            }
          }

      }
```

Las siguientes observaciones se corresponden con los números especificados al principio de algunas líneas del fuente anterior:

(1) La sentencia **import** incorpora, durante la ejecución de esta clase, el paquete java.sql. Recordar (fue nombrado anteriormente) que este paquete es el que contiene las clases que se utilizan bajo JDBC.

(2) Declaración de la clase CrearTabla.

(3) Declaración del método **main**. La existencia de este método indica que esta clase será ejecutada como un application.

(4) Definición de dos variables miembro dentro del método main: **con**, de tipo Connection, y **stmt**, de tipo Statement. La variable **con** se utilizará para el establecimiento de la conexión con la base de datos sobre la cual se creará la tabla. En el caso de la variable **stmt**, su objetivo es el de almacenar la sentencia SQL que se ejecutará, aplicando un determinado método sobre ella.

Los bloques **try – catch** permiten ejecutar sentencias Java y definir tratamientos de excepciones si se producen situaciones de error dentro de las sentencias incluidas. El bloque **catch** especifica los errores que se capturarán y que acciones se llevarán a cabo.

(5) Carga del driver JDBC correspondiente y establecimiento de la conexión con el origen de datos creado en esta misma sección.

(6) Se aplica el método **createStatement** al objeto **con** de tipo **Connection**. La aplicación del método devuelve un objeto de tipo **Statement**, aquí de nombre **stmt**.

(7) El método **executeUpdate** aplicado sobre **stmt**, recibe y ejecuta la sentencia SQL completa que creará la tabla CLIENTES.

(8) Si la creación es exitosa, en consola se leerá el mensaje ”Tabla CLIENTES creada”.

(9) Los bloques catch capturarán, de producirse, dos tipos de excepciones: por driver de base de datos no encontrado, y por problemas en la instrucción SQL cargada. En ambos casos, exhiben en consola mensajes que alertan de estas situaciones.

Recordar: en caso de no usar un IDE para el desarrollo de este ejemplo, el fuente podrá ser ingrsado en un archivo de tipo txt. Desde el prompt del DOS, con **javac** y el nombre del fuente, se procederá a la compilación. Utilizando **java** y el nombre de la nueva clase, se ejecutará el programa java especificado.

Carga de la tabla creada

El siguiente fuente muestra los pasos a seguir para cargar la tabla anteriormente creada, con registros.

```
   import java.sql.*;

   // Declaración de la clase CrearTabla.

   public class CargarTabla
   {

      // Declaración de variables miembro de la clase

(1)   static String[] SQLData =
        {
            "('DNI', 1588888, 'Juan',   'Avda Alberdi 5555', 3800)",
            "('DNI', 1528948, 'José',   'Avda Castro 99768', 5200)",
            "('DNI', 1689088, 'Pablo',  'Costa 555',         1500)",
            "('DNI', 6799672, 'Maria',  'J. Alvarez 2341',   7800)",
            "('DNI', 4327800, 'Carlos', 'J. Mitre 2671',     5000)",
        };

      // Declaración del método main

      public static void main(String[] args)
      {
         Connection con  = null;
         Statement stmt  = null;
         int iRowCount = 0;

         try {

            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con = DriverManager.getConnection("jdbc:odbc:ParaTip", "", "");
            stmt = con.createStatement();
```

```
(2)         for (int i = 0; i < SQLData.length; i++)
                    {
                        iRowCount +=
                        stmt.executeUpdate(
                          "INSERT INTO CLIENTES VALUES " +
                           SQLData[i] );
                    }

(3)         System.out.println(" Tabla CLIENTES cargada con registros");


          } catch ( ClassNotFoundException cnfe) {
            System.out.println("Driver de Base de Datos no encontrado");
            cnfe.getMessage();
            cnfe.printStackTrace();
          } catch ( SQLException sqle) {
            System.out.println("Problema con instrucción SQL");
            sqle.getMessage();
            sqle.printStackTrace();
          }
        }

    }
```

Comentarios:

(1) Declaración del vector de tipo String de nombre **SQLData**. Contiene los datos que se agregarán a la tabla CLIENTES.

(2) Se implementa un ciclo desde 0, con incrementos de 1, hasta la longitud del vector) especificada por el valor devuelto de aplicar el método **length** sobre el objeto **SQLData**. En cada iteración, se incorpora un registro tomado del vector de String **SQLData**.

(3) Si no se presentan errores en las ejecuciones de los INSERTs, se visualizará en consola el mensaje "Tabla CLIENTES cargada".

Es importante aclarar, que existen variadas y potentes formas de incorporar registros en tablas, sin necesidad de "hardcodearlos" dentro del código (JSP- Java Server Pages, Servlets). La forma seleccionada en este tip, responde a la elección de una forma sencilla de incorporación de los registros, que permitan al programador comenzar a familiarizarse con el lenguaje.

Visualización de la tabla

La visualización de los datos se realizará por consola del producto IDE Eclipse.

```
import java.sql.*;

// Creación de la clase VerTabla.

public class VerTabla
{

    public static void main(String[] args)
    {
        Connection con  = null;
        Statement stmt  = null;
        ResultSet rs;
        String sTipo, sApyn, sDomicilio;
        int iNrodoc, iLimite;

      try {

            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            con =DriverManager.getConnection("jdbc:odbc:ParaTip", "", "");
            stmt = con.createStatement();

(1)         rs = stmt.executeQuery("SELECT * FROM CLIENTES ORDER BY NRODOC");

(2)         while(rs.next())
                  {
                        sTipo      = rs.getString("TIPO");
                        iNrodoc    = rs.getInt("NRODOC");
                        sApyn      = rs.getString("APELLIDO_NOMBRE");
                        sDomicilio = rs.getString("DOMICILIO");
                        iLimite    = rs.getInt("LIMITE_CREDITO");

                        System.out.println( sTipo      + " " +
                                            iNrodoc    + " " +
                                            sApyn      + " " +
                                            sDomicilio + " " +
                                            iLimite );
                        }

            System.out.println(" Tabla CLIENTES visualizada");
```

```
        } catch ( ClassNotFoundException cnfe) {
          System.out.println("Driver de Base de Datos no encontrado");
          cnfe.getMessage();
          cnfe.printStackTrace();
        } catch ( SQLException sqle) {
          System.out.println("Problema con instrucción SQL");
          sqle.getMessage();
          sqle.printStackTrace();
        }
      }

}
```

Comentarios:

(1) Ejecución de la sentencia SQL SELECT. Las sentencias SELECT se ejecutan aplicando el método **executeQuery** sobre el objeto de tipo **Statement**, a diferencia de INSERTs, DELETEs y UPDATEs que usan el método executeUpdate. El resultado de la aplicación de este método es un objeto de tipo **ResultSet**, de nombre **rs**, que fue creado anteriormente.

(2) El ciclo while, aplicando el método **next** sobre el **ResultSet** obtenido, recorre toda la selección hecha y va "imprimiendo" en la consola los datos contenidos en la tabla CLIENTES.

V. JDBC como base para la construcción de otras herramientas

Existen varias tipos de APIs alternativas desarrolladas sobre las APIs tradicionales de JDBC. Algunas de estas APIs alternativas son:

- **SQL embebido para Java**: un consorcio incluyendo Oracle, IBM, Sun, entre otras empresas, definieron **SQLJ**, una especificación que contempla esta forma de utilización de SQL. JDBC requiere que las sentencias SQL se envien básicamente como strings "no interpretados" a métodos de Java. Un preprocesador SQL embebido proporciona chequeos en tiempo de compilación y permite al programador "intercalar" sentencias SQL con sentencias propias del lenguaje Java. **El preprocesador de SQLJ efectivamente traduce este mix Java-SQL en lenguaje de programación Java con invocaciones a JDBC.**

- Un "mapeo" directo de tablas de bases de datos relacionales a clases de Java. Existen productos que realizan un mapeo donde cada tabla se convierte en una clase y cada fila en una instancia de esa clase y cada valor de columna en un atributo de aquella instancia. De esta manera, los programadores pueden operar directamente sobre objetos Java, y las sentencias SQL requeridas para leer y actualizar datos son generadas de manera transparente al programador.

VI. Conceptos adicionales para tener en cuenta

Algunas definiciones importantes

API: (Application Programming Interface)

SQL: (Structured Query Language) es un lenguaje standarizado utilizado para crear, manipular, examinar y gestionar bases de datos relacionales. Considerando que SQL es un lenguaje en sí mismo, una única sentencia puede ser muy expresiva. Dispone de elementos suficientes como para disparar acciones que involucren, entre otras posibilidades, sorts o mergings de los datos seleccionados.

ODBC: (Open DataBase Connectivity) es una interface basada en C que proporciona un accesos a engines de bases de datos basados en SQL, proporcionado una interface consistente para comunicarse no sólo con los datos, sino también para acceder a la "database metadata" (información sobre las tablas que componen la base de datos, sus campos, y características que los definen). Existen diferentes drivers o bridges para cada DBMS provistos por los fabricantes. La combinación de ODBC y SQL hace posible conectarse a la base de datos y manipular la información de una manera standard. Aunque ODBC comenzó como un standard en PC, se ha convertido prácticamente en un standard de la industria.

VII. Dónde Obtener Información Adicional

www.java.sun.com

www.java.sun.com/docs/books/tutorial/jdbc/

Copyright 2003 Teknoda S.A. Diciembre 2003. JAVA es marca registrada de Sun. SAP, R/3 y ABAP son marcas registradas de SAP AG. AS/400 es marca registrada de IBM. Todas las marcas mencionadas son marcas registradas de las empresas proveedoras.

La información contenida en este artículo ha sido recolectada en la tarea cotidiana por nuestros especialistas a partir de fuentes consideradas confiables. No obstante, por la posibilidad de error humano, mecánico, cambios de versión u otro, Teknoda no garantiza la exactitud o completud de la información aquí volcada.

Dudas o consultas ***develop@teknoda.com***