
MiniSoft® JDBC® Driver

MiniSoft, Inc.
1024 First street
Snohomish, WA 98290
U.S.A.

1-800-682-0200
360-568-6602
Fax: 360-568-2923

MiniSoft Marketing AG
Papiermuhleweg 1
Postfach 107
Ch-6048 Horw
Switzerland

Phone: +41-41-340 23 20
Fax: +41-41-340 38 66
CompuServe: 100046,450
minisoftag@centralnet.ch

Internet access:

sales@minisoft.com
support@minisoft.com
<http://www.minisoft.com>
<ftp://ftp.minisoft.com>

Disclaimer

The information contained in this document is subject to change without notice.

MiniSoft, Inc. makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. MiniSoft, Inc. or its agents shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishings, performance, or use of this material.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another programming language without the prior written consent of MiniSoft, Inc.

©1999 by MiniSoft, Inc., Printed in U.S.A.
1st version of this manual printed

All product names and services identified in this document are trademarks or registered trademarks of their respective companies and are used throughout this document in editorial fashion only and are not intended to convey an endorsement or other affiliation with MiniSoft, Inc.

JDBC is a registered trademark of Sun Microsystems.

License Agreement

In return for payment of a onetime fee for this software product, the Customer receives from MiniSoft, Inc. a license to use the product subject to the following terms and conditions:

- The product may be used on one computer system at a time: i.e., its use is not limited to a particular machine or user but to one machine at a time.
- The software may be copied for archive purposes, program error verification, or to replace defective media. All copies must bear copyright notices contained in the original copy.
- The software may not be installed on a network server for access by more than one personal computer without written permission from MiniSoft, Inc.

Purchase of this license does not transfer any right, title, or interest in the software product to the Customer except as specifically set forth in the License Agreement, and Customer is on notice that the software product is protected under the copyright laws.

90-Day Limited Warranty

MiniSoft, Inc. warrants that this product will execute its programming instructions when properly installed on a properly configured personal computer for which it is intended. MiniSoft, Inc. does not warrant that the operation of the software will be uninterrupted or error free. In the event that this software product fails to execute its programming instructions, Customer's exclusive remedy shall be to return the product to MiniSoft, Inc. to obtain replacement. Should MiniSoft, Inc. be unable to replace the product within a reasonable amount of time, Customer shall be entitled to a refund of the purchase price upon the return of the product and all copies. MiniSoft, Inc. warrants the medium upon which this product is recorded to be free from defects in materials and workmanship under normal use for a period of 90 days from the date of purchase. During the warranty period MiniSoft, Inc. will replace media which prove to be defective. Customer's exclusive remedy for any media which proves to be defective shall be to return the media to MiniSoft, Inc. for replacement.

ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS IS LIMITED TO THE 90-DAY DURATION OF THIS WRITTEN WARRANTY. Some states or provinces do not allow limitations on how long an implied warranty lasts, so the above limitation or exclusion may not apply to you. This warranty gives you specific rights, and you may also have other rights which vary from state to state or province to province.

LIMITATION OF WARRANTY: MiniSoft, Inc. makes no other warranty expressed or implied with respect to this product. MiniSoft, Inc. specifically disclaims the implied warranty of merchantability and fitness for a particular purpose.

EXCLUSIVE REMEDIES: The remedies herein are Customer's sole and exclusive remedies. In no event shall MiniSoft, Inc. be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory.

Table of Contents

Disclaimer iii

License Agreement iv

Chapter 1

Introduction 1-1

What is JDBC? 1-1

Why use JDBC? 1-2

Chapter 2

Installation 2-1

JDBC from Web Site 2-1

JDBC diskettes 2-2

Installing MiniSoft JDBC 2-2

 Installing the MiniSoft JDBC/ODBC/32 host server 2-2

 Installing the MiniSoft JDBC driver 2-3

 Installing the MiniSoft ODBC/32 PC Administrator 2-4

 Running the middle-tier server 2-4

 Samples 2-4

 Datasources 2-5

Chapter 3

Database Access with JDBC 3-1

JDBC Classes and Interfaces	3-1
The Big Four	3-2
Registering the driver	3-2
Connecting to a database	3-2
Connecting using connection properties	3-3
Creating a Statement Object	3-4
Exiting JDBC	3-5
Prepared Statements and Parameters	3-7
Metadata	3-8
Adding, Updating and Deleting Data	3-10

Chapter 4

JDBC API Reference 4-1

DriverManager Methods	4-1
Connection Methods	4-3
Statement Methods	4-5
PreparedStatement Methods	4-7
ResultSet Methods	4-8

ResultSetMetaData Methods 4-11

DatabaseMetaData Methods 4-12

Index..... 1

Chapter 1

Introduction

What is JDBC?

Java Database Connectivity Driver (JDBC) is a standard or open application-programming interface (API) for accessing a database from JAVA programs. The MiniSoft JDBC driver implements this API to give JAVA programs access to data in TurboImage databases, KSAM files, MPE files, Self-describing files, and PowerHouse subfiles.

JDBC is based on and closely aligned with the Open Group standard Structured Query Language (SQL) Call-Level Interface. It allows programs to use SQL requests that will access databases without having to know the proprietary interfaces to the databases. The JDBC driver handles the SQL request and converts it into a request the individual database system understands.

Why use JDBC?

MiniSoft's JDBC driver lets you read and write to HP3000 files as if they were like any other database. JDBC supports advanced features such as access to multiple databases, KSAM files, MPE files and self-describing files. It also supports HP3000 features like third party indexing and B-trees. This makes MiniSoft's JDBC driver the perfect tool to give JAVA programs access to valuable enterprise data.

Chapter 2

Installation

You can receive this product on a set of three diskettes, or you can download it from our Web site.

JDBC from Web Site

Navigate to www.minisoft.com. Under Product Descriptions select JDBC under Client Server and Web Development Tools, then click Download MiniSoft's JDBC free demo and follow the instructions.

You will then receive three files:

- ◆ ODBC215s.exe, contains the server software to go to the HP3000.
- ◆ JDBC215c.exe, contains the JDBC driver and the mid-tier server.
- ◆ ODBC215a.exe, contains the administration software, which you need to install only on one PC: SCHEMA EDITOR and CATALOG EDITOR.

JDBC diskettes

The MiniSoft JDBC driver is shipped on three diskettes:

- ♦ Diskette 1 contains the files for a client installation and documentation on JDBC.
- ♦ Diskette 2 contains the files for an administrator installation. Install these files only if you need access to the catalog and schema editor.
- ♦ Diskette 3 contains the files for a server installation. No emulator is needed for this installation. Follow the instructions in the next section to install the files on the three diskettes.

Installing MiniSoft JDBC

The MiniSoft JDBC driver runs in a 3-tier model. The HP3000 contains a server program, which it shares with MiniSoft's ODBC/32 driver. A mid-tier server name 3KMidTierListener.exe runs on a Windows NT, Window 98, or Windows 95 machine. It connects JDBC clients to the HP3000 server. The JDBC driver classes for the client are in MSJDBC.jar. They need to be installed where the JAVA class loader can find them.

Installing the MiniSoft JDBC/ODBC/32 host server

This needs to be done only once on one PC—*not* on each individual JAVA development machine. (Remember that no terminal emulator is needed for this installation.)

1. Insert diskette 3.
2. Click Windows' Start menu on the Taskbar, then click run.

3. Type: A:\HPSETUP.EXE (A being the letter of your floppy drive).
4. Follow the on-screen instructions.

NOTE: The Server Installation creates a sample database and populates the customer table. The database that is created is called MSCARD.MM.MINISOFT.

Configuring for Read / Write access via JDBC calls

To have the ability to update JDBC you must modify MSJOB. By default the listener job, MSJOB, is installed with the following line that allows read-only access via JDBC calls:

```
SETVAR MSSERVER000004 "30006 0 ODBC SRVR.MM.MINISOFT S"
```

To enable write / update access, modify the command line to include the "w" (write), option:

```
SETVAR MSSERVER000004 "30006 0 ODBC SRVR.MM.MINISOFT S W"
```

Installing the MiniSoft JDBC driver

1. Place the JDBC diskette labeled #1 into drive A:\ or B:\.
2. Run JSETUP.EXE.

This will extract the driver *MSJDBC.jar* and the middle-tier server *3KMidTierListener.exe* into the C:\MiniSoft\JDBC directory. To use the driver from a Java application, the directory MSJDBC.jar resides in must be listed in the CLASSPATH environment variable. To use the driver from an applet, MSJDBC.jar should be installed on the Web Server, and the directory it resides in should be referenced with the ARCHIVE parameter in the APPLET tag in the appropriate HTML document.

Installing the MiniSoft ODBC/32 PC Administrator

Only install the Administrator files if one would need to access the Schema or Catalog Editor. An administrator installment is not necessary to run JDBC.

1. Place the JDBC diskette labeled #2 into A:\.
2. Click Windows' Start menu on the Taskbar, then select Run.
3. Type: A:\ADMIN.EXE (A being the letter of your floppy drive).
4. Follow the on screen instructions.

Running the middle-tier server

In order for the mid-tier server, *3KMidTierListener.exe*, to accept requests from clients, you must run it. Either set up a shortcut or run it from the Start|Run dialog. By default the mid-tier is listening on port 30504 for client requests. This can be changed with the */P:<port>* parameter. Only one copy of *3KMidTierListener.exe* should be running on the mid-tier machine, unless the port numbers are different.

Samples

There are three samples provided. The first two are a Java application (*ViewDB.java*) and a Java applet (*ViewDBApplet.java*). The third is a sample HTML file (*ViewDBApplet.htm*) supplied to launch *ViewDBApplet*.

Here is a sample of where files would be installed on a Web Server to run the applet:

ViewDBApplet.htm → <web root>

ViewDBApplet.class—> <web root>\ViewDBApplet\ViewDBApplet.class
MSJDBC.jar—> <web root>

Datasources

The JDBC driver can use ODBC/32 datasources on the mid-tier machine. ODBC/32 must be on the mid-tier machine to create and edit the datasources. The samples show how the datasource is referenced from the Java application or applet.

Chapter 3

Database Access with JDBC

JDBC is a standard and open application programming interface for accessing databases from the JAVA programming language. It allows JAVA programs to use SQL requests that will access databases without having to know the proprietary interfaces to the database. A JDBC driver handles the SQL requests submitted through the JDBC interface and converts them into requests the database the driver supports will understand.

JDBC Classes and Interfaces

The JDBC classes and interfaces are declared in *java.sql*. The classes are the common components of JDBC. The main class is the ***DriverManager***. Use the ***DriverManager*** class to make a connection to a database. Other classes in *java.sql* are used for date handling, exception handling and common constants. The interfaces in *java.sql* are the blueprints used by JDBC driver developers to create a JDBC driver. Each JDBC driver must contain classes that implement the interfaces in *java.sql*. A typical JDBC driver is a set of these classes, plus some support classes, contained in an archive.

The Big Four

Basic data access through JDBC can be accomplished using only four JDBC classes. They are:

DriverManager
Connection
Statement
ResultSet

You can access these classes through the interfaces since each of the driver's classes are implementations of the interfaces described in *java.sql*. This means that you only need to specify which driver to use during the connection request. After that the code is driver independent.

Registering the driver

The first step in using a JDBC driver is to register it with the *DriverManager*. A common way to do this is shown below:

```
Class.forName ( "com.minisoft.jdbc.MSJDBCdriver" );
```

The *forName* method of the *Class* class returns a *Class* object associated with the class with the given string name. This is not needed but the code will force the class named by the string to be loaded. The class name of the JDBC driver is "com.minisoft.jdbc.MSJDBCdriver". When the JDBC driver is loaded it will register itself with the *DriverManager*.

Connecting to a database

Now that the driver is registered with the *DriverManager*, you can request a connection to a database. The *getConnection* method of the

DriverManager class will do this. This is illustrated below:

```
Connection con=DriverManager.getConnection ("jdbc:MSJDBC//  
127.0.0.1:30504/MSDB");
```

The string parameter is a url which describes the connection. The first part “jdbc” is the main protocol. The second part “MSJDBC” is a sub-protocol that identifies MiniSoft’s JDBC driver. The *DriverManager* will use the sub-protocol to query each registered driver to see if the driver handles this sub-protocol. The third and subsequent parts identify how to connect to the database.

Note: This is driver specific. In this case, using MiniSoft’s JDBC driver, the third part is specifying an IP address and port number for the mid-tier server, while the fourth part is specifying a datasource. The datasource contains the HP3000 specific information to connect to the database(s).

Connecting using connection properties

MiniSoft’s JDBC driver also supports another form of *getConnection*, in which connection properties can be passed. This form of *getConnection* is shown in figure 1.

```
String url = "jdbc:MSJDBC://  
127.0.0.1:30504/";  
Properties p = new Properties ();  
p.put ( "Server", "data.minisoft.com" );  
p.put ( "Server Port", "31100" );  
p.put ( "User", "MGR" );  
p.put ( "User Password", "HEREYAGO" );  
p.put ( "Account", "MINISOFT" );  
p.put ( "Group", "MM" );  
p.put ( "Database0", "MSDB,DBPSWD,1,5"  
);  
Connection con =  
DriverManager.getConnection ( url, p );
```

Figure 1

For a list of all properties supported see chapter 4 on *JDBC API Reference*. By using this form of *getConnection* you do not need a datasource set up on the mid-tier machine. You can specify everything that is needed to connect to the database as a property.

The *getConnection* method returns an object from the selected driver that conforms to the *Connection* interface. From now on, objects created by the driver that conform to an interface in *java.sql*, will be referred to by their interface name, not the internal name of the class in the driver. This is because you never need to know what the internal name is, you always access the object through its interface. The *Connection* object can now be used to create a *Statement* object.

Creating a Statement Object

To create a Statement Object use the example shown below:

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

The *createStatement* method of a *Connection* object returns a *Statement* object. The *Statement* object can now be used to execute a SQL query with the *executeQuery* method as shown below:

```
ResultSet rs = stmt.executeQuery ( "SELECT * FROM CUSTOMERS" );
```

This will execute the SQL statement passed as a parameter and create a *ResultSet* object. The *ResultSet* object can now be used to retrieve the results of the SQL statement as shown below:

```
while ( rs.next () ) {  
s = rs.getString ( "CUSTOMER_NUMBER" );  
System.out.println ( s );  
s = rs.getString ( "CUSTOMER_NAME" );  
System.out.println ( s );  
s = rs.getString ( "ADDRESS1" );  
System.out.println ( s );  
}
```

```
s = rs.getString ( "ADDRESS2" );  
System.out.println ( s );  
s = rs.getString ( "CITY" );  
System.out.println ( s );  
s = rs.getString ( "STATE" );  
System.out.println ( s );  
s = rs.getString ( "COUNTRY" );  
System.out.println ( s );  
s = rs.getString ( "ZIP" );  
System.out.println ( s );  
s = rs.getString ( "DATE" );  
System.out.println ( s );  
}
```

The *Next* method obtains the next record from the result set. The *getString* method is used to return data from a column. The parameter can be either a column name as a string or a column number (1 based). The *getString* method is normally used to return alphanumeric columns, although some drivers will return any data type as a string. If you need to retrieve the data in its normal data type, there are *get...* methods for all the various data types.

Exiting JDBC

It is good practice to exit a program gracefully, although the JDBC driver will take care of things if you do not. To exit the JDBC program use the code shown below:

```
rs.close ( );  
stmt.close ( );  
con.close ( );
```

When the *close* method of a **Connection** object is called, it will request the *close* method of any **Statement** objects it created. Similarly, when the *close* method of a **Statement** object is called it will request the *close* method of any **ResultSet** objects it created. Given this scenario, you could just call *con.close*.

```
import java.sql.*;
public class FirstDBAccess
{
    public static void main (String [] args)
    {
        try {
            Class.forName ( "com.minisoft.jdbc.MSJDBCDriver" );
            String url = "jdbc:MSJDBC://127.0.0.1:30504/MSDB";
            Connection con = DriverManager.getConnection ( url );
            try {
                Statement stmt = con.createStatement ( );
                String query = "SELECT * FROM CUSTOMERS";
                ResultSet rs = stmt.executeQuery ( query );
                while ( rs.next ( ) ) {
                    s = rs.getString ( "CUSTOMER_NUMBER" );
                    System.out.println ( s );
                }
                stmt.close ( );
            }
            catch ( SQLException e2 )
            {
                System.out.println ( e2 );
            }
            con.close ( );
        }
        catch ( SQLException e0 )
        {
            System.out.println ( e0 );
        }
        catch ( ClassNotFoundException e1 )
        {
            System.out.println ( e1 );
        }
    }
}
```

Figure 2: A complete application that retrieves data from a TurboImage database.

A complete example that puts together all the above code pieces with added exception handling is shown in figure 2.

The exception handling catches any exceptions thrown by the JDBC methods. Many of the methods in JDBC can throw a *SQLException*. Printing out the exception will typically show the error message.

Prepared Statements and Parameters

In many situations you will want to execute essentially the same statement a number of times. The only difference between each execution of the statement might be some selection criteria or update values.

For example, if you needed to retrieve customer information by customer number based upon a customer number entered by the user. When the user requested customer number '000001', you could build a string that contained "SELECT * FROM CUSTOMERS WHERE CUSTOMER_NUMBER = '000001'" and execute it. Then when the user requested customer number '000002', you would build a string that contained "SELECT * FROM CUSTOMERS WHERE CUSTOMER_NUMBER = '000001'" and execute it. This method is very inefficient because the driver will have to compile essentially the same statement many times.

The preferred way to accomplish this task is to use a statement that contains parameters and compile it. Once prepared, you can supply values for the parameters and execute the statement as many times as needed without compiling it again. Instead of using the *createStatement* method of a **Connection** object, you can use the *prepareStatement* method. Its one parameter is a string that contains the SQL statement to prepare. The *prepareStatement* method returns a **PreparedStatement** object, which is a subclass of the **Statement** class. The **PreparedStatement** class has *set...* methods to set the values of parameters. Figure 3 illustrates preparing a statement, setting its parameters and executing it many times.

SQL statements use the question mark ('?') to mark the position of the parameters. The parameters are numbered, starting at 1, in the order they appear in the statement. The *setString* method's first parameter is the parameter number, its second is the value for the parameter. All parameter values can be cleared with the *clearParameters* method.

```
void DoneOneTime ()
{
  ...
  Connection con = DriverManager.getConnection ( url );
  String query = "SELECT * FROM CUSTOMERS WHERE
    CUSTOMER_NUMBER = ?";
  PreparedStatement stmt = con.prepareStatement ( query );
  ...
}
void DoneManyTimes ( String CustomerNumber )
{
  stmt.setString ( 1, CustomerNumber );
  ResultSet rs = stmt.executeQuery ();
  ...
}
```

Figure 3: A prepared statement with parameters.

Metadata

Metadata is data which describes data. There are two types of metadata objects that a JDBC driver can provide. One is based upon the *DatabaseMetaData* interface and the other is based upon the *ResultSetMetaData* interface.

The *DatabaseMetaData* class mainly contains methods to gather information concerning a database. The most common information is the names of the tables in the database and the layout of those tables. Other less frequently used information is access privileges and the relationships between tables. A *DatabaseMetaData* object is created with the *getMetaData* method of a *Connection* object. Figure 4 illustrates using a *DatabaseMetaData* object to retrieve the names of all the tables in a database.

The *getTables* method returns a *ResultSet* object that is used to retrieve the information about the tables. Each row in the result set has 5 columns as follows:

1. **TABLE_CAT** String => table catalog (may be null).
2. **TABLE_SCHEM** String => table schema (may be null).
3. **TABLE_NAME** String => table name.
4. **TABLE_TYPE** String => table type. Typical types are “TABLE”, “VIEW”, “SYSTEM TABLE”, “GLOBAL TEMPORARY”, “LOCAL TEMPORARY”, “ALIAS” and “SYNONYM”.
5. **REMARKS** String => explanatory comment on the table.

Depending to the driver, the **TABLE_CAT** and/or the **TABLE_SCHEM** columns may be null indicating that these are not attributes of the database the driver supports. Figure 4 illustrates loading table names into a list box.

```

...
Connection con = DriverManager.getConnection ( url );
Choice tableNames = new Choice ( );
DatabaseMetaData md = con.getMetaData ( );
String [] types = { "TABLE" };
ResultSet mrs = md.getTables ( null, "", "", types );
while ( mrs.next ( ) ) {
    tableNames.addItem ( mrs.getString ( 3 ) );
}
}
...

```

Figure 4: Loading table names into a list box.

The *ResultSetMetaData* class contains methods to gather information about a result set. This information contains the number of columns in each row of the result set and the layout of each column. This information is very useful to programs that need to dynamically create the layout for displaying data from a database. A *ResultSetMetaData* object is created with the *getMetaData* method of a *ResultSet* object. Figure 5 shows how to gather and use the data type of a result set column.

```
String s;
int i;
...
Connection con = DriverManager.getConnection ( url );
Statement stmt = con.createStatement ();
ResultSet rs = stmt.executeQuery ( "SELECT * FROM CUSTOMERS"
);
ResultSetMetaData md = rs.getMetaData ();
while ( rs.next () ) {
    int col;
    for ( col = 1; col <= md.getColumnCount; ++col ) {
        switch ( md.getColumnType ( col ) ) {
            case Types.CHAR :
                s = rs.getString ( col );
                ...
                break;
            case Types.INTEGER:
                i = rs.getInt ( col );
                ...
                break;
            ...
        }
    }
}
```

Figure 5: Using ResultSetMetaData to get a column's data type.

Adding, Updating and Deleting Data

JDBC can be used to add, update and/or delete records in a table. The *executeUpdate* method of a *Statement* (or *PreparedStatement*) object is used to execute SQL INSERT, UPDATE and DELETE statements. The return of the *executeUpdate* method indicates the number of records affected by the SQL statement. The setting of auto-commit for the *Connection* object determines if each statement is committed automatically, or if an explicit commit or rollback must be done. If auto-commit is on, then each statement executed with *executeUpdate* method will be committed immediately. If auto-commit is off, a commit or rollback will only be done when a *commit* or *rollback* method of the *Connection* object is called.

By default, new connections start with auto-commit on. Different drivers

handle locking, transaction isolation and concurrency differently. The driver's documentation will need to be consulted to determine how the driver behaves, and how compatible it will be with other applications that are accessing the database. Figure 6 demonstrates adding a customer record to the customers table, while querying and updating the next value for customer numbers.

```
Connection con = DriverManager.getConnection ( url );
con.setAutoCommit ( false );
...
Statement stmt1 = con.createStatement ( );
ResultSet rs = stmt1.executeQuery ( "SELECT NEXT_NUMBER FROM NEXT_NUMBERS
WHERE CATAGORY = 'CU'" );
rs.next ( );
int nextCust = getInt ( 1 );
PreparedStatement stmt2 = con.prepareStatement ( "INSERT INTO CUSTOMERS
(CUSTOMER_NUMBER, ...) VALUES (?,...)" );
stmt2.setInt ( 1, nextCust );
...
stmt2.executeUpdate ( );
PreparedStatement stmt3 = con.prepareStatement ( "UPDATE NEXT_NUMBERS SET
NEXT_NUMBER = ? WHERE CATAGORY = 'CU'" );
stmt3.setInt ( 1, ++nextCust );
stmt3.executeUpdate ( );
con.commit ( );
...
```

Figure 6: INSERT and UPDATE in a single transaction.

Chapter 4

JDBC API Reference

The following is a reference to the most commonly used methods of the JDBC API. A reference of the complete JDBC API can be found on the JavaSoft web site at www.javasoft.com.

DriverManager Methods

getConnection

Requests a connection to a database. If a connection is made, a *Connection* object is returned.

Example:

```
public static synchronized Connection getConnection(String url)
throws SQLException
```

```
public static synchronized Connection getConnection(String url,
Properties info) throws SQLException
```

Parameters:

- ♦ url - A string which describes the connection in the form “jdbc:MSJDBC://<mid-tier server host name>[:<mid-tier server port>][/<datasource>].
- ♦ The main protocol. <mid-tier server host name> and < mid-tier server port> identify the machine the mid-tier server is running on and the TCP port it is listening on.
- ♦ <datasource> specifies datasource on the mid-tier server machine that contains the HP3000 specific information to connect to the database(s).
- ♦ info – A Properties collection that contains the HP3000 specific information to connect to the database(s). Information in the info parameter will override information contained in a datasource. The available properties are:

2DriverTable - <name of a translation table file for translating data coming to the client>

2HostTable - <name of a translation table file for translating data going from the client>

Account - <HP3000 logon account>

Account Password - <HP3000 logon account password>

Database<n> - <database name to access on the HP3000>,<database password>,<load auto master flag>,<database open mode>

DecimalPoint - <decimal point character>

Group - <HP3000 logon group>

Group Password - <HP3000 logon group password>

Jobname - <jobname>

Langauge - <NLS language>

Schema<n> - <name of the schema file>,<lockword>

Server - <HP3000 IP address or hostname>

Server Port - <server TCP port>

User - <HP3000 user logon>

User Password=<HP3000 user password>

Note: All property names are case sensitive.

For the properties Database<n>, and Schema<n>, <n> indicates a number, starting with 0. The first database defined would be with property “Database0”, the second would be with “Database1”, and so on.

The values for <load auto master flag> is 1 to load automatic masters and 0 not to load automatic masters.

The values for <database open mode> are the same as on the DBOPEN intrinsic, i.e. 1 – 8.

Connection Methods

createStatement

Creates a *Statement* object. This is typically used when the statement has no parameters and is only going to be executed once.

Example:

```
public abstract Statement createStatement() throws SQLException
```

PrepareStatement

Creates a *PreparedStatement* object that represents a compiled SQL statement. This is typically used when the statement contains parameters, or when the statement will be executed more than once.

Example:

```
public abstract PreparedStatement prepareStatement(String sql)
throws SQLException
```

Parameters:

- ◆ sql - A string which contains the SQL statement to compile.

setAutoCommit

Sets the state of the connection's auto-commit mode. If a connection's auto-commit mode is true, then each statement that modifies the database will be committed upon completion. If a connection's auto-commit mode is false, then data will only be committed to the database when the commit method is called.

Example:

```
public abstract void setAutoCommit(boolean autoCommit) throws  
SQLException
```

Parameters:

- ◆ autoCommit - The state to set the auto-commit mode.

commit

Commits any changes made by SQL statements since the transaction started to the database. Transactions are automatically started when the first SQL statement that modifies the database is executed.

Example:

```
public abstract void commit() throws SQLException
```

rollback

Cancels and changes made by SQL statements since the transaction started.

Example:

```
public abstract void rollback() throws SQLException
```

close

Closes the connection and all open databases, statements and result sets.

Example:

```
public abstract void close() throws SQLException
```

getMetaData

Creates a *DatabaseMetaData* object. The *DatabaseMetaData* object is used to provide information about the connection and the open database(s).

Example:

```
public abstract DatabaseMetaData getMetaData() throws SQLException
```

Statement Methods

executeQuery

Compiles and executes a SQL statement and returns a *ResultSet* object.

Example:

```
public abstract ResultSet executeQuery(String sql) throws
```

SQLException

Parameters:

- ◆ sql - A string containing the statement to be executed.

executeUpdate

Executes a DELETE, INSERT or UPDATE SQL statement. The return is the number of records effected by the SQL statement.

Example:

```
public abstract int executeUpdate(String sql) throws SQLException
```

Parameters:

- ◆ sql - A string containing the statement to be executed.

close

Closes the *Statement* and its current *ResultSet* if one exists.

Example:

```
public abstract void close() throws SQLException
```

PreparedStatement Methods

executeQuery

Executes a previously compiled SQL statement and returns a *ResultSet* object.

Example:

```
public abstract ResultSet executeQuery() throws SQLException
```

executeUpdate

Executes a previously compiled DELETE, INSERT or UPDATE SQL statement.

Example:

```
public abstract int executeUpdate() throws SQLException
```

Set...

```
public abstract void setShort(int parameterIndex, short x) throws  
SQLException  
public abstract void setInt(int parameterIndex, int x) throws  
SQLException  
public abstract void setLong(int parameterIndex, long x) throws  
SQLException  
public abstract void setFloat(int parameterIndex, float x) throws  
SQLException  
public abstract void setDouble(int parameterIndex, double x) throws  
SQLException  
public abstract void setBigDecimal(int parameterIndex, BigDecimal x)  
throws SQLException
```

```
public abstract void setString(int parameterIndex, String x) throws  
SQLException  
public abstract void setDate(int parameterIndex, Date x) throws  
SQLException
```

Set the value of a parameter in a SQL statement.

Parameters:

- ◆ `parameterIndex` - The number of the parameter in the SQL statement, starting at 1.
- ◆ `x` - The value for the parameter.

clear Parameters

Clears all SQL statement parameters.

Example:

```
public abstract void clearParameters() throws SQLException
```

ResultSet Methods

next

Fetches the next available row of a result set.

Example:

```
public abstract boolean next() throws SQLException
```

close

Closes a *ResultSet*.

Example:

```
public abstract void close() throws SQLException
```

Get...

```
public abstract String getString(int columnIndex) throws
SQLException
public abstract short getShort(int columnIndex) throws SQLException
public abstract int getInt(int columnIndex) throws SQLException
public abstract long getLong(int columnIndex) throws SQLException
public abstract float getFloat(int columnIndex) throws SQLException
public abstract double getDouble(int columnIndex) throws
SQLException
public abstract BigDecimal getBigDecimal(int columnIndex, int scale)
throws SQLException
public abstract Date getDate(int columnIndex) throws SQLException
public abstract String getString(String columnName) throws
SQLException
public abstract short getShort(String columnName) throws
SQLException
public abstract int getInt(String columnName) throws SQLException
public abstract long getLong(String columnName) throws
SQLException
public abstract float getFloat(String columnName) throws
SQLException
public abstract double getDouble(String columnName) throws
SQLException
public abstract BigDecimal getBigDecimal(String columnName, int
scale) throws SQLException
public abstract Date getDate(String columnName) throws
SQLException
```

Returns the value of a column from the current row within the result set.

Parameters:

- ◆ columnIndex - The number of the columns together starting at 1.
- ◆ columnName - The name of the column to get.

getMetaData

Returns a *ResultSetMetaData* object that has information about this *ResultSet*.

Example:

```
public abstract ResultSetMetaData getMetaData() throws  
SQLException
```

findColumn

Obtains a column number that corresponds to a column name.

Example:

```
public abstract int findColumn(String columnName) throws  
SQLException
```

Parameters:

- ◆ columnName - The name of the column.

ResultSetMetaData Methods

getColumnCount

Finds the number of columns in the *ResultSet*.

Example:

```
public abstract int getColumnCount() throws SQLException
```

getColumnDisplaySize

Obtains the number of characters necessary to display the data from a column.

Example:

```
public abstract int getColumnDisplaySize(int column) throws  
SQLException
```

Parameters:

- ◆ column - Column number.

getColumnName

Obtains the name of a column in the *ResultSet*.

Example:

```
public abstract String getColumnName(int column) throws  
SQLException
```

Parameters:

- ◆ column - Column number.

getColumnType

Obtains the data type of a column in the *ResultSet*. Data type constants are defined in `java.sql.Types`.

Example:

```
public abstract int getColumnType(int column) throws SQLException
```

Parameters:

- ◆ columns - Column number.

DatabaseMetaData Methods

getTables

Obtains a *ResultSet* object containing information about the tables within the database.

Example:

```
public abstract ResultSet getTables(String catalog, String schema,  
String tableName, String types[]) throws SQLException
```

Parameters:

- ◆ catalog - Ignored. Catalog is not supported by MiniSoft's JDBC

driver.

- ◆ `schema` - The name of schema to search for tables. A schema name is either a database name or a schema editor file name. If empty all schemas are searched.
- ◆ `tableName` - The name of a table to gather information about. If empty information about all the tables is returned.
- ◆ `types` - An array of strings designating the types of tables to be included. If empty all types are included. The only type supported is "TABLE".

The columns in the *ResultSet* are:

1. **TABLE_CAT** String => table catalog (may be null).
2. **TABLE_SCHEM** String => table schema (may be null).
3. **TABLE_NAME** String => table name.
4. **TABLE_TYPE** String => table. Typical types are "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS" and "SYNONYM".
5. **REMARKS** String => explanatory comment on the table.

getColumns

Obtains a *ResultSet* object containing information about the columns in the table.

Example:

```
public abstract ResultSet getColumns(String catalog, String schema,  
String tableName, String columnName) throws SQLException
```

Parameters:

- ◆ catalog - ignored. Catalog is not supported by MiniSoft's JDBC driver.
- ◆ schema - The name of a schema. A schema name is either a database name or a schema editor file name. If empty all schemas are searched.
- ◆ tableName - The name of a table to gather the column information for.
- ◆ columnName - The name of a column. If empty includes all columns.

The columns in the *ResultSet* are:

1. **TABLE_CAT** String => table catalog (may be null)
2. **TABLE_SCHEM** String => table schema (may be null)
3. **TABLE_NAME** String => table name
4. **COLUMN_NAME** String => column name
5. **DATA_TYPE** short => SQL type from java.sql.Types
6. **TYPE_NAME** String => Data source dependent type name
7. **COLUMN_SIZE** int => column size. For char or date types this is the maximum number of characters, for numeric or decimal types this is precision.
8. **BUFFER_LENGTH** is not used
9. **DECIMAL_DIGITS** int => the number of fractional digits
10. **NUM_PREC_RADIX** int => Radix (typically either 10 or 2)

11. **NULLABLE** int => is NULL allowed?
 - ◆ columnNoNulls - might not allow NULL values
 - ◆ columnNullable - allows NULL values
 - ◆ columnNullableUnknown - nullability unknown
12. **REMARKS** String => comment describing column (may be null)
13. **COLUMN_DEF** String => default value (may be null)
14. **SQL_DATA_TYPE** int => unused
15. **SQL_DATETIME_SUB** int => unused
16. **CHAR_OCTET_LENGTH** int => for char types the maximum number of bytes in the column
17. **ORDINAL_POSITION** int => index of column in table (starting at 1)
18. **IS_NULLABLE** String => “NO” means column definitely does not allow NULL values; “YES” means the column might allow NULL values. An empty string means no one knows.

getIndexInfo

Get a *ResultSet* object containing information about the indexes or keys for the table.

Example:

```
public abstract ResultSet getIndexInfo(String catalog, String schema,  
String table, boolean unique, boolean approximate) throws  
SQLException
```

Parameters:

- ◆ catalog - Ignored. Catalog is not supported by MiniSoft's JDBC driver.
- ◆ schema - The name of schema. A schema name is either a database name or a schema editor file name. If empty all schemas are searched.
- ◆ table - The name of a table to gather the column information for.
- ◆ approximate - Ignored. All statistics are exact.

The columns in the *ResultSet* are:

1. **TABLE_CAT** String => table catalog (may be null)
2. **TABLE_SCHEM** String => table schema (may be null)
3. **TABLE_NAME** String => table name
4. **NON_UNIQUE** boolean => Can index values be non-unique? False when TYPE is tableIndexStatistic.
5. **INDEX_QUALIFIER** String => index catalog (may be null); null when TYPE is tableIndexStatistic
6. **INDEX_NAME** String => index name; null when TYPE is tableIndexStatistic
7. **TYPE** short => index type:
 - ◆ tableIndexStatistic - identifies table statistics that are returned in conjugation with a table's index descriptions.
 - ◆ tableIndexClustered - is a clustered index.
 - ◆ tableIndexHashed - is a hashed index.
 - ◆ tableIndexOther - is some other style of index.
8. **ORDINAL_POSITION** short => column sequence number within

index; zero when TYPE is tableIndexStatistic

9. **COLUMN_NAME** String => column name; null when TYPE is tableIndexStatistic
10. **ASC_OR_DESC** String => column sort sequence, “A” => ascending, “D” => descending, may be null if sort sequence is not supported; null when TYPE is tableIndexStatistic
11. **CARDINALITY** int => When TYPE is tableIndexStatistic, then this is the number of rows in the table; otherwise, it is the number of unique values in the index.
12. **PAGES** int => When TYPE is tableIndexStatistic then this is the number of pages used for the table, otherwise it is the number of pages used for the current index.
13. **FILTER_CONDITION** String => Filter condition, if any (may be null).

Index

A

Adding data 3-10

C

Classes of JDBC 3-1, 3-2

Connecting to a database 3-2

Connecting using connection properties 3-3

Connection Class 3-2

Connection Methods 4-3

close 4-5

commit 4-4

createStatement 4-3

getMetaData 4-5

PreparedStatement 4-3

rollback 4-4

setAutoCommit 4-4

Creating a ResultSet object 3-4

Creating a Statement Object 3-4

D

Database Access with JDBC 3-1

DatabaseMetaData Methods 4-12

getColumnns 4-13

getIndexInfo 4-15

getTables 4-12

Deleting data 3-10

DriverManager class 3-2

DriverManager Methods

getConnection 4-1

E

Exiting JDBC 3-5

G

getConnection 4-1

I

Installation of JDBC 2-1

Configuring Read/Write access 2-3

Downloading JDBC from Web Site 2-1

Installing JDBC driver 2-3

Installing JDBC/ODBC/32 host server 2-2

Installing ODBC/32 PC Administrator 2-4

JDBC diskettes 2-2

Running the middle-tier server 2-4

Interfaces of JDBC 3-1

J

JDBC API Reference 4-1

JDBC Description 1-1

M

Metadata 3-8

P

Prepared Parameters 3-7

Prepared Statements 3-7

PreparedStatement Methods 4-7

clear Parameters 4-8

executeQuery 4-7

executeUpdate 4-7

R

Registering the driver 3-2

ResultSet Class 3-2

ResultSet Methods 4-8

close 4-9

findColumn 4-10

getMetaData 4-10

next 4-8

ResultSetMetaData Methods 4-11

getColumnCount 4-11

getColumnDisplaySize 4-11

getColumnName 4-11

getColumnType 4-12

S

Statement Class 3-2

Statement Methods 4-5

close 4-6

executeQuery 4-5

executeUpdate 4-6

U

Updating data 3-10