Mimer JDBC Driver Guide

September 2007

Mimer JDBC, Driver Guide

Produced and published by Mimer Information Technology AB, Uppsala, Sweden.
P.O. Box 1713,
SE-751 47 Uppsala, Sweden.
Tel +46(0)18-780 92 00.
Fax +46(0)18-780 92 40.

Mimer SQL Web Sites:
http://developer.mimer.com
http://www.mimer.com

Contents

Chapter 1

Introduction

Mimer JDBC Drivers provide access to Mimer SQL databases from Java applications and applets. The drivers are type 4 drivers, which means that they are written entirely in Java. As they are written in Java, they can be downloaded in applets.

Mimer JDBC Drivers can also be used on all platforms that support Java Virtual Machine (JVM) and so provide a very high degree of portability.

About this Guide

The guide is intended for Java application developers working with Mimer SQL. It covers all available Mimer JDBC drivers.

The guide describes the usage of SQL in Java applications, and provides, together with the *Mimer SQL Reference Manual*, the complete reference material for Mimer SQL.

To read more about JDBC and JVM, visit http://java.sun.com/products/jdbc/index.html.

The JDBC API specification implemented by this driver (packages `java.sql` and `javax.sql`) is found at
http://java.sun.com/j2se/1.4.2/docs/api/java/sql/package-summary.html
and http://java.sun.com/j2se/1.4.2/docs/api/javax/sql/package-summary.html.

Definitions, Terms and Trademarks

API	Application Programming Interface
CDC	Connected Device Configuration
CLDC	Connected Limited Device Configuration
EJB	Enterprise Java Beans
J2EE	Java 2 platform Enterprise Edition
J2ME	Java 2 platform Micro Edition
J2SE	Java 2 platform Standard Edition
JCP	Java Community Process
JDBC	The Java database API
JDK	Java Development Kit
JNDI	Java Naming and Directory Interface

JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KVM	K Virtual Machine, a compact portable JVM, intended for small, resource constrained devices
MIDP	Mobile Interface Device Profile
OCC	Optimistic Concurrency Control
PSM	Persistent Stored Modules, the term used by ISO/ANSI for stored procedures
SQL	Structured Query Language
URL	Uniform Resource Locator

Available Drivers

At the moment, the following Mimer JDBC drivers are available:

- `mimjdbc3.jar` supports the JDBC 3 specification and is J2EE compliant. This is currently the main JDBC driver, which should be used in most situations. This driver requires a Java 1.4 runtime or later.
- `mimjdbc1.jar` and `mimjdbc2.jar` supports the JDBC 1.2 and JDBC 2 specifications respectively. These drivers support the same server features as `mimjdbc3.jar`, but are supposed to be used on platforms and in environments where modern Java runtime environments are not supported. mimjdbc2.jar is J2EE compliant.
- `minjdbc3.jar` supports the JDBC Optional Package for CDC/Foundation Profile specification and is J2ME compliant. This driver supports a subset of the JDBC 3 specification and is aimed at devices with limited memory resources. See further down in this document for specifics on what is supported and not.
- `midjdbc2.jar` implements a subset of the functionality provided by the minjdbc3 driver. It is J2ME compliant and is supposed to be used in environments supporting the CLDC/Mobile Information Device Profile specification. See further down in this document for specifics on what is supported and not.

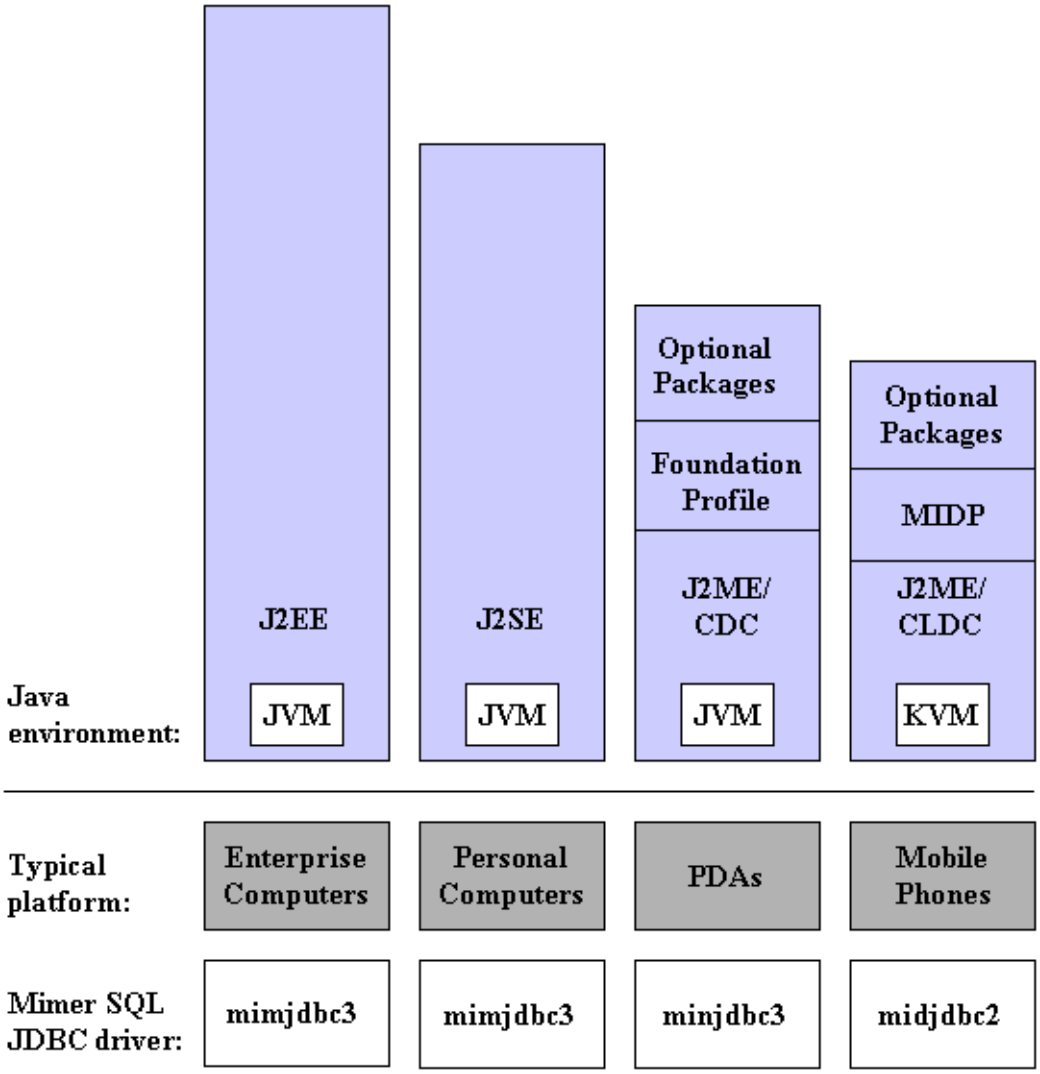
Requirements

Driver	Mimer SQL Product	Java Runtime
`mimjdbc3.jar`	Mimer SQL 8.2 or later	JRE 1.4 or later
`mimjdbc2.jar`	Mimer SQL 8.2 or later	JRE 1.2 or later
`mimjdbc1.jar`	Mimer SQL 8.2 or later	JRE 1.1.8 or later

Driver	Mimer SQL Product	Java Runtime
`minjdbc3.jar`	Mimer SQL 8.2 or later	J2ME CDC/Foundation Profile
`midjdbc2.jar`	Mimer SQL 8.2 or later	J2ME CLDC 1.0.4/MID Profile 2.0 or CLDC 1.0/MID Profile 1.0 with extra support for the socket protocol within the Generic Connection Framework

Environment

Mimer JDBC has a complete range of functionality and support for the smallest devices to the high-end systems and application servers. In the picture below the various Java environments are described and coupled with computer environments and Mimer JDBC drivers:



In addition, the `mimjdbc1` and `mimjdbc2` drivers are available for older environments (see *Available Drivers* on page *2*).

Differences Between the Drivers

In most cases the `mimjdbc3` driver should be used. The `mimjdbc1` and `mimjdbc2` drivers should be used when only older Java runtime environments are supported. The `minjdbc3` and `midjdbc2` drivers are mainly for smaller and resource constrained environments.

The following table lists differences between the drivers:

Description	mimjdbc1	mimjdbc2	mimjdbc3	minjdbc3	midjdbc2
Array Fetches	Yes	Yes	Yes	Yes	Yes
Batch Operations	No	Yes	Yes	Yes	Yes
CallableStatements	Yes	Yes	Yes	Yes	Yes
DatabaseMetaData Support	Yes	Yes	Yes	Yes	Yes
DataSource	No	Yes	Yes	Yes	Yes
Distributed Transactions (XA)	No	Yes	Yes	No	No
Driver identification support	Yes	Yes	Yes	No	No
Driver-class Support	Yes	Yes	Yes	No	No
Holdable cursors	No	No	Yes	Yes	No
`java.sql.Blob`, `java.sql.Clob`	No	Yes	Yes	Yes	Yes
JavaBean configuration	No	Yes	Yes	Yes	No
May be stored in connection pools	No	Yes	Yes	No	No
May be stored in JNDI	No	Yes	Yes	No	No
National character Support	Yes	Yes	Yes	Yes	Yes
Native SQL expansions	Yes	Yes	Yes	Yes	Yes
Parameter metadata	No	No	Yes	Yes	No
Scrollable ResultSets	No	Yes	Yes	Yes	Yes
Set FetchSize	No	Yes	Yes	Yes	Yes

Description	mimjdbc1	mimjdbc2	mimjdbc3	minjdbc3	midjdbc2
SQL data types: BLOB, CLOB, and NCLOB	Yes	Yes	Yes	Yes	Yes
Updatable result sets	No	No	No	No	No

Note: A requirement for functionality support regarding to the table above is that the database server version used is supporting the functionality.

The default fetch size for the mimjdbc drivers is about 64 kB. In the minjdbc and midjdbc drivers the buffer size is 10 kB. This buffer size can be manipulated using the `.setFetchSize` method.

About the JDBC Driver for J2ME/CDC (minjdbc3)

The Mimer JDBC driver for J2ME/CDC uses less memory resources than the regular drivers. The code size itself has been reduced by removing features not included in the JDBC for J2ME/CDC specification. Furthermore the J2ME-driver allocates smaller buffers for network packets and internal data.

The Mimer JDBC driver for J2ME/CDC is primarily intended to be used on a device with limited memory resources. It can connect to any type of Mimer SQL server.

About the JDBC Driver for Midlets (midjdbc3)

The Mimer MIDP driver should be used in environments supporting the CLDC/MID Profile specification. It is a strict subset of the regular JDBC for CDC configurations API. Also applications developed with Mimer MIDP will run with little change in a regular JDBC environment and vice versa. Programming skills obtained using regular JDBC programming may be applied to programming within the MIDP environment as well.

The following features are omitted in the Mimer JDBC for MIDP:

- All methods requiring floating point data types (such as `ResultSet.getDouble`). See below on floats, doubles and BigDecimal's for guidelines on handling these cases.
- All methods requiring a `java.sql.Date`, `java.sql.Time` and `java.sql.Timestamp` data type. When working with SQL date, time and timestamp data, please consider using `ResultSet.getString`, `PreparedStatement.setString` and other string getter and setter methods instead.
- The MIDP driver has no `finalize()` methods. This means that it is even more important for applications to explicitly close database objects when done with them.
- Since the `java.math` library is omitted from MIDP, no `BigDecimal`'s may be used. Please consider using the appropriate `java.lang.String` getter or setter method.
- There is no `java.sql` interface. Programs interact directly against the `com.mimer.jdbc` classes instead. See below for a list of corresponding classes. Basically, instead of importing java.sql.*, you should import com.mimer.jdbc.* instead and carry on as usual.

Importing the JDBC Classes

Because the K Virtual Machine per default does not allow loading system class (java.*) applications accessing Mimer SQL databases should import the com.mimer.jdbc.* classes instead. This makes the application less portable than a regular JDBC application, but these are the constraints set by the environment.

The following Mimer classes correspond to the named interface classes in the regular JDBC API.

JDBC class	Mimer class
java.sql.BatchUpdateException	com.mimer.jdbc.BatchUpdateException
java.sql.Blob	com.mimer.jdbc.Blob
java.sql.CallableStatement	com.mimer.jdbc.CallableStatement
java.sql.Clob	com.mimer.jdbc.Clob
java.sql.Connection	com.mimer.jdbc.Connection
java.sql.DatabaseMetaData	com.mimer.jdbc.DatabaseMetaData
java.sql.DataSource	com.mimer.jdbc.MimerDataSource
java.sql.DataTruncation	com.mimer.jdbc.DataTruncation
java.sql.PreparedStatement	com.mimer.jdbc.PreparedStatement
java.sql.ResultSet	com.mimer.jdbc.ResultSet
java.sql.ResultSetMetaData	com.mimer.jdbc.ResultSetMetaData
java.sql.SQLException	com.mimer.jdbc.SQLException
java.sql.SQLWarning	com.mimer.jdbc.SQLWarning
java.sql.Statement	com.mimer.jdbc.Statement
java.sql.Types	com.mimer.jdbc.Types

See also the javadoc for the Mimer MIDP driver. It is available on the Mimer SQL Developer site: http://developer.mimer.com/documentation/latest_javadoc_midp/index.html.

FLOAT and DOUBLE PRECISION

Databases with FLOAT, DOUBLE PRECISION, REAL and INTEGER(n) columns which are required to expose these numbers in a MIDlet can use one the following tricks to go around the float and BigDecimal limitation in the K Virtual Machine.

1. It is always possible to get the SQL FLOAT or DOUBLE PRECISION value into a java.lang.String. This will work particularly well if the data is only to be displayed.

2 To handle fractions so that they may be used to some calculations, make sure at the SQL level to always return integers and reserving the last digits of the integer for decimals. If the database has a `FLOAT` column and we are interested in 4 decimals, do the following SQL:

```
select cast(OUR_FLOAT*10000 as bigint) from OUR_TABLE
```

At the application level, place the result in an `int` or a `long`, but assume that the last 4 digits are decimals when displaying values to the user. In addition, decimals entered by the user must be handled similarly by the application before being inserted to the database. The following SQL statement may be used to make the `float` appear correctly in a table.

```
insert into OUR_TAB(OUR_FLOAT) values (cast(? as float)/10000);
```

DATE, TIME and TIMESTAMP

`DATE`, `TIME` and `TIMESTAMP` system classes (`java.util.Date`) are not included in the MIDP specification. Therefore the Mimer client cannot return `DATE`, `TIME` or `TIMESTAMP` objects. Please use string getter and setter methods instead.

Logging

To keep the driver size small and to optimize performance the Mimer JDBC drivers do not perform any logging. For logging, we provide a separate driver, Mimer JDBC Trace driver.

Mimer JDBC Trace driver is a full JDBC Driver that covers all of JDBC by calling the matching routines of the logged JDBC Driver.

It produces a log of every JDBC call an application makes, and also measures the elapsed time for each call. The log can be written to a file, or can be displayed directly in a window.

For more information, see http://developer.mimer.com/howto/howto_28.htm.

Chapter 2

Using the Mimer JDBC Driver

This chapter explains how to load the Mimer JDBC driver and how to connect to a Mimer SQL database. It also contains JDBC application examples and discusses driver characteristics.

Loading a Driver

To use the Mimer JDBC driver, it must be loaded into the Java environment. The Java environment locates a driver by a search along the class path, defined in the `CLASSPATH` environment variable.

The `CLASSPATH` environment variable informs the Java environment where to find Java class files, such as the Mimer JDBC drivers.

The Mimer JDBC driver jar file, including the directory specification, should be added to the Java class path, as can be seen in the following examples:

UNIX:
```
# echo $CLASSPATH
CLASSPATH=.:/usr/lib/mimjdbc3.jar
```

Win:
```
% set CLASSPATH=.;D:\MIMJDBC3.JAR
```

Besides defining the `CLASSPATH` environment variable explicitly, it can also be defined for a specific session when executing the application. For example:

```
java -classpath /usr/lib/mimjdbc3.jar JdbcApplication
```

Connecting the Traditional Way

The connection provides the link between the application and the Mimer SQL database server. To make a connection using the `DriverManager` class requires two operations, i.e. loading the driver and making the connection.

The class name of the Mimer JDBC Driver is:

```
com.mimer.jdbc.Driver
```

The class name of the Mimer JDBC Trace Driver is:

```
com.mimer.jtrace.Driver
```

The jar file referenced in the `CLASSPATH` determines which driver is loaded.

A driver can be explicitly loaded using the standard `Class.forName` method:

```
import java.io.*;
import java.sql.*;

…

try {
        Class.forName("com.mimer.jdbc.Driver");

    } catch (java.lang.ClassNotFoundException cnf) {
        System.err.println("JDBC driver not found");
        return;
    }
```

Alternatively, `DriverManager`, when it initializes, looks for a `jdbc.drivers` property in the system properties. The `jdbc.drivers` property is a colon-separated list of drivers.

The `DriverManager` attempts to load each of the named drivers in this list of drivers. The `jdbc.drivers` property can be set like any other Java property, by using the `-D` option:

```
java -Djdbc.drivers=com.mimer.jdbc.Driver class
```

The property can also be set from within the Java application or applet:

```
Properties prp = System.getProperties();
prp.put("jdbc.drivers",
           "com.mimer.jdbc.Driver:com.mimer.jtrace.Driver");
System.setProperties(prp);
```

Note: Neither of the mechanisms used to load the driver specify that the application will actually use the driver. The driver is merely loaded and registered with the DriverManager.

Connecting With URL

To make the actual database connection, a URL string is passed to the `DriverManager.getConnection` method in the JDBC management layer.

The URL defines the data source to connect to. The JDBC management layer locates a registered driver that can connect to the database represented by the URL.

URL Syntax

The Mimer JDBC drivers support the following URL syntax:

```
jdbc:mimer:[protocol:][URL-field-list][property-list]
```

`URL-field-list` options can be combined with `property-list` options.

Protocol

If a protocol is specified, the driver will load the mimcomm JNI library and use native routines to connect to the database. If the protocol is not specified (or is an empty string), no JNI library will be loaded and a TCP/IP connection will be made using standard Java network packages in you Java runtime.

Supported protocols include:

protocol	Explanation
`local`	Use shared memory communication to a server that runs on your local machine. This protocol is often much faster than TCP/IP-based communication.
`tcp`	Connects to the server using TCP/IP, but through the mimcomm JNI library.
`rapi`	Use the RAPI protocol to connect to mobile devices (Windows only).
`decnet`	Use Decnet to connect to a remote server (VMS only).

URL-field-list

All fields in the `URL-field-list` are optional.

The database server host computer, with or without a user specification, is introduced by `//` and the database name is introduced by `/`, like:

```
[//[user[:password]@]serverName[:portNumber]] [/databaseName]
```

A Connection object is returned from the `getConnection` method, for example:

```
String url = "jdbc:mimer://MIMER_ADM:admin@localhost/ExampleDB";
Connection con = DriverManager.getConnection(url);
```

Alternatively, the `getConnection` method allows the user name and password to be passed as parameters:

```
url = "jdbc:mimer://localhost/ExampleDB";
con = DriverManager.getConnection(url, "MIMER_ADM", "admin");
```

Property-list

The `property-list` for the Mimer JDBC Driver is optional. The list is introduced by a leading question mark `?` and where there are several properties defined they are separated by ampersands `&`, like:

```
?property=value[&property=value[&property=value]]
```

The following properties are supported:

Property	Explanation
`databaseName`	Name of database server to access
`user`	User name used to log in to the database
`password`	Password used for the login
`serverName`	Computer on which the database server is running, the default is `localhost`
`portNumber`	Port number to use on the database server host, the default is `1360`
`protocol`	The protocol to use when connecting. If set, load the mimcomm JNI library. If empty, use standard Java TCP/IP support.

The following example demonstrates a connection using the driver properties:

```
url = "jdbc:mimer:?databaseName=ExampleDB"
                + "&user=MIMER_ADM"
                + "&password=admin"
                + "&serverName=srv2.mimer.com";
con = DriverManager.getConnection(url);
```

Alternatively a `java.util.Properties` object can be used:

```
Properties dbProp = new Properties();

dbProp.put("databaseName", "ExampleDB");
dbProp.put("user", "MIMER_ADM");
dbProp.put("password", "admin");
con = DriverManager.getConnection("jdbc:mimer:", dbProp);
```

Elements from the `URL-field-list` and the `property-list` can be combined:

```
url = "jdbc:mimer:/ExampleDB"
                + "?user=MIMER_ADM"
                + "&password=admin";
```

The `DriverPropertyInfo` class is available for programmers who need to interact with a driver to discover the properties that are required to make a connection. This enables a generic GUI tool to prompt the user for the Mimer SQL connection properties:

```
Driver drv;
DriverPropertyInfo [] drvInfo;

drv = DriverManager.getDriver("jdbc:mimer:");
drvInfo = drv.getPropertyInfo("jdbc:mimer:", null);
for (int i = 0; i < drvInfo.length; i++) {
    System.out.println(drvInfo[i].name + ": " + drvInfo[i].value);
}
```

After connecting to the database, all sorts of information about the driver and database is available through the use of the `getMetadata` method:

```
DatabaseMetaData dbmd;

dbmd = con.getMetaData();

System.out.println("Driver      " + dbmd.getDriverName());
System.out.println("    Version " + dbmd.getDriverVersion());
System.out.println("Database    " + dbmd.getDatabaseProductName());
System.out.println("    Version " + dbmd.getDatabaseProductVersion ());
con.close();
```

The `close` method tells JDBC to disconnect from the Mimer SQL database server. JDBC resources are also released.

It is usual for connections to be explicitly closed when no longer required. The normal Java garbage collection has no way of freeing external resources, such as the Mimer SQL database server.

Connecting the J2EE Way

Along with J2EE came a new way for JDBC drivers to connect to database servers. Instead of requesting connections through the `java.sql.DriverManager` class, applications should connect using the `javax.sql.DataSource`, `com.mimer.jdbc.MimerConnectionPoolDataSource` or `com.mimer.jdbc.MimerXADataSource` interfaces.

Deploying Mimer JDBC in JNDI

The Mimer DataSource class is `com.mimer.jdbc.MimerDataSource`. When applications are deployed within the J2EE environment, a properly initiated MimerDataSource object should be stored in JNDI for the application server to retrieve at runtime. Application servers may use the JavaBean interface to obtain configuration parameters for MimerDataSource objects.

These are the DataSource attributes recognized by the Mimer JDBC drivers:

DataSource Attributes	Description
`serverName`	The computer on which the database server is running, the default is `localhost`
`portNumber`	The port number to use on the server host, the default is `1360`
`description`	A textual description
`databaseName`	The name of the database on the server (required)
`user`	User name
`password`	Password
`protocol`	The protocol to use when connecting via the mimcomm JNI library
`service`	The service to connect to. This field plays the same role as the portNumber field, but any string can be used for protocols that don't use integer-valued port numbers (such as Decnet or named pipes). If a service value is specified, any portNumber value is ignored.

See sample programs further down for programming examples.

Deploying Mimer JDBC in a Connection Pool

Mimer JDBC may be deployed in J2EE compliant connection pools.

When deploying Mimer JDBC in a connection pool, the class `com.mimer.jdbc.MimerConnectionPoolDataSource` should be used. This class features the same attributes as described above for `.MimerDataSource`.

Deploying Mimer JDBC in Distributed Transaction Environments

Mimer JDBC may be used in J2EE compliant distributed transaction environments.

When deploying Mimer JDBC to be used in distributed transactions, the class `com.mimer.jdbc.MimerXADataSource` should be used. Whenever connections are created using this factory class, Mimer SQL may cooperate in transactions with any other XA compliant database server.

Read more about Mimer SQL and distributed transactions in *Mimer SQL Programmer's Manual*.

Mimer JDBC/CDC Optional Package

The Mimer SQL product contains a Mimer JDBC driver suitable for CDC/FP environments. This driver follows a specification laid out by the Java Community Process (JCP) 169. The detailed specification, *JDBC for CDC/FP Optional Package specification,* can be found at the Sun web site (http://java.sun.com).

This driver is targeted at more powerful handheld environments, such as PDA's, handheld computers and high-end mobile telephones.

The Mimer JDBC/CDC driver is located in the Mimer installation directory under the name `minjdbc3.jar`. The steps required to use the driver in a project varies somewhat depending on the target platform.

Sony Ericsson CDC Platform

The Sony Ericsson CDC Platform is targeted at their high-end Symbian telephones, for example M600, P990 and W950. This package is available as a download from their developer site.

In order to use the Mimer JDBC driver in a project with the Sony Ericsson CDC Platform, two things must be done:

1. Make sure the Mimer JDBC driver is installed in the private directory of the application in question. If the UID of the application is FF000000, this directory would be `\private\FF000000` on any drive. To make the driver available to your application in the Symbian emulator, you need to copy the `minjdbc3.jar` file to the application private folder. For example:

   ```
   copy "c:\program files\mimer sql 9.3\minjdbc3.jar"
   c:\symbian\uiq3sdk\epoc32\winscw\c\private\FF000000\minjdbc3.jar
   ```

   To make the driver available for deployment on the actual telephone, a line similar to the following need to be included in the application package specification file (`.PKG`):

```
"c:\program files\mimer sql 9.3\minjdbc3.jar" "!:\private\FF000000\minjdb3.jar"
```

2. Specify the Mimer JDBC driver in the classpath. This is done in an invocation specification file, with the extension `.j9`. For example, an application named `HelloMimerCDC` with the UID FF000000 may use this invocation file:

   ```
   -cp c:\private\FF000000\HelloMimerCDC.jar;c:\private\FF000000\minjdbc3.jar
   HelloMimerCDC
   ```

A complete programming example is available for download at the Mimer SQL Developer site.

Error Handling

Error handling is taken care of by using the classes `SQLException` and `SQLWarning`.

The Class SQLException

The `SQLException` class provides information relating to database errors. Details include a textual description of the error, an SQLState string, and an error code. There may be a number of `SQLException` objects for a failure.

```
try {

…

} catch(SQLException sqe) {
    SQLException stk;

    stk = sqe;   // Save initial exception for stack trace

    System.err.println("\n*** SQLException:\n");
    while (sqe != null) {
        System.err.println("Message:     " + sqe.getMessage());
        System.err.println("SQLState:    " + sqe.getSQLState());
        System.err.println("NativeError: " + sqe.getErrorCode());
        System.err.println();

        sqe = sqe.getNextException();
    }

    stk.printStackTrace(System.err);
}
```

The Class SQLWarning

The `SQLWarning` class provides information relating to database warnings. The difference between warnings and exceptions is that warnings, unlike exceptions, are not thrown.

The `getWarnings` method of the appropriate object (`Connection`, `Statement` or `ResultSet`) is used to determine whether warnings exist.

Warning information can be retrieved using the same mechanisms as in the `SQLException` example above but with the method `getNextWarning` retrieving the next warning in the chain:

```
con = DriverManager.getConnection(url);
checkSQLWarning(con.getWarnings());

…

private static boolean checkSQLWarning( SQLWarning sqw )
throws SQLException {
    boolean rc = false;

    if (sqw != null) {
        rc = true;

        System.err.println("\n*** SQLWarning:\n");
        while (sqw != null) {
            System.err.println("Message:     " + sqw.getMessage());
            System.err.println("SQLState:    " + sqw.getSQLState());
            System.err.println("NativeError: " + sqw.getErrorCode());
            System.err.println();

            sqw = sqw.getNextWarning();
        }
    }

    return rc;
}
```

Viewing Driver Characteristics

By using the `java com.mimer.jdbc.Driver` command, you can view characteristics of a specific driver and the current environment:

```
java com.mimer.jdbc.Driver options
```

The options available are:

Option	Description
`-version`	Display driver version
`-sysprop`	Display all system properties
`-errors`	List all JDBC error codes
`-ping url`	Test the database at the specified url
`-mimcomm`	Load the mimcomm JNI library and show its version number. Displays informational messages to help fix any problems.

The following is an example that uses the `-version` option:

```
java com.mimer.jdbc.Driver -version
Mimer JDBC driver version 3.15
```

Used without any arguments, the command will display usage information.

Note: This functionality is only supported for the `mimjdbc1`, `mimjdbc2`, and `mimjdbc3` drivers. Other drivers must use the `getMetadata` method.

The mimcomm JNI library

The Mimer JDBC driver can be used in a 100% native Java environment. In this case, the connection to a Mimer database server is done by the TCP/IP support included in the Java platform.

However, it is also possible to load an external library called mimcomm that includes support for all the communication protocols available on the particular platform. Please note that the mimcomm library may not be available on platforms that don't have a recent version of Mimer SQL installed.

The name of the mimcomm library varies between platforms. It is called mimcomm.dll on Windows, libmimcomm.so on Unix and MIMCOMM.EXE on VMS.

When you install a Mimer SQL distribution, the mimcomm library will normally be installed in a place where the Java environment can find it. You can test this by using the `-mimcomm` switch as a command line argument to the JDBC driver:

```
unix $ java -cp mimjdbc3.jar com.mimer.jdbc.Driver -mimcomm
System.getProperty("java.library.path"):
/usr/lib/SunJava2-1.4.2/jre/lib/i386/client:/usr/lib/SunJava2
-1.4.2/jre/lib/i386:/usr/lib/SunJava2-1.4.2/jre/../lib/i386:/
mimer/v925/dist/lib

System.loadLibrary("mimcomm"):

mimcomm library Version: V925B
JNI parameter method:    JNI_COPY
```

When the JDBC driver loads the mimcomm library, it looks for the library in the path specified by the java.library.path system property. If the JDBC driver cannot find the library in the path listed, you should either move the mimcomm library to a directory listed in the path or consult your Java manual for instructions on how to change the `java.library.path` system property.

Java Program Examples

Below are a collection of small basic Java programs for different environments, showing a database connection and a simple database operation with some error handling.

JDBC Application Example

The example Java program below creates a result set containing all rows of the data dictionary view `INFORMATION_SCHEMA.TABLES`, then each row is fetched and displayed on standard output.

In this example, the user name and password are given separately using the `DriverManager.getConnection` method, i.e. not given in the URL specification.

The below example will work with the mimjdbc drivers.

```
import java.sql.*;

class Example
{
    public static void main(String[] args)
    {
        try {
            Class.forName("com.mimer.jdbc.Driver");
            String url = "jdbc:mimer://my_node.mimer.se/customers";
            Connection con = DriverManager.getConnection(url,
                                                         "SYSADM","SYSPW");
            Statement stmt = con.createStatement();
            String sql = "select TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE
                          from INFORMATION_SCHEMA.TABLES";
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                String schema = rs.getString(1);
                String name = rs.getString(2);
                String type = rs.getString(3);
                System.out.println(schema + "  " + name + "  " + type);
            }
            rs.close();
            stmt.close();
            con.close();
        } catch (SQLException e) {
            System.out.println("SQLException!");
            while (e != null) {
                System.out.println("SQLState  : " + e.getSQLState());
                System.out.println("Message   : " + e.getMessage());
                System.out.println("ErrorCode : " + e.getErrorCode());
                e = e.getNextException();
                System.out.println("");
            }
        } catch (Exception e) {
            System.out.println("Other Exception");
            e.printStackTrace();
        }
    }
}
```

Another way to provide connection properties is to supply a `java.util.Properties` object to the `DriverManager.getConnection` method.

JDBC Application Example for J2EE

This example Java program deploys a `com.mimer.jdbc.MimerDataSource` in a file system JNDI repository. Note that the file system JNDI repository have to be downloaded from Sun. It is available for download at http://java.sun.com/products/jndi/serviceproviders.html. At this site, several other service providers may be downloaded as well.

Examples provided in this section will only work with the `mimjdbc2` and `mimjdbc3` drivers.

```
import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import javax.naming.directory.*;
import java.util.Hashtable;

public class RegisterJNDI
{
    public static void main(String argv[])
    {
        try {
            com.mimer.jdbc.MimerDataSource ds =
                new com.mimer.jdbc.MimerDataSource();

            ds.setDescription("Our Mimer data source");
            ds.setServerName("my_node.mimer.se");
            ds.setDatabaseName("customers");
            ds.setPortNumber("1360");
            ds.setUser("SYSADM");
            ds.setPassword("SYSPW");

            // Set up environment for creating initial context
            Hashtable env = new Hashtable();

            env.put(Context.INITIAL_CONTEXT_FACTORY,
                    "com.sun.jndi.fscontext.RefFSContextFactory");
            env.put(Context.PROVIDER_URL, "file:.");
            Context ctx = new InitialContext(env);

            // Register the data source to JNDI naming service
            ctx.bind("jdbc/customers", ds);

        } catch (Exception e) {
            System.out.println(e);
            return;
        }
    }
}
```

Once the data source is deployed, applications may connect using the deployed `DataSource` object. For instance like the below code snippet:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL, "file:.");
Context ctx = new InitialContext(env);
DataSource ds = (DataSource)ctx.lookup("jdbc/customers");
return ds.getConnection();
```

Using the Driver from Applets

The example Java applet below creates a result set containing all rows of the data dictionary view `INFORMATION_SCHEMA.TABLES`, then each row is fetched and displayed on standard output.

In this example, the user name and password are given separately using the `DriverManager.getConnection` method, i.e. not given in the URL specification.

The example will work with the mimjdbc drivers.

```
import java.sql.*;
import java.applet.*;
import java.awt.*;

public class ExampleApplet extends java.applet.Applet {
    public void init() {
        resize(1200, 600);
    }

    public void paint(Graphics g) {
        int row = 1;
        g.drawString("Listing tables:", 20, 10 * row++);
        try {
            Class.forName("com.mimer.jdbc.Driver");
            String url = "jdbc:mimer://my_node.mimer.se/customers";
            Connection con = DriverManager.getConnection(url, "SYSADM",
                                                         "SYSPW");
            Statement stmt = con.createStatement();
            String sql = "select TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE
                          from INFORMATION_SCHEMA.TABLES";
            ResultSet rs = stmt.executeQuery(sql);
            while (rs.next()) {
                String schema = rs.getString(1);
                String name = rs.getString(2);
                String type = rs.getString(3);
                g.drawString(schema + "  " + name + "  " + type, 50,
                             10 * row++);
            }
            rs.close();
            stmt.close();
            con.close();
        } catch (SQLException e) {
            g.drawString("SQLException!", 20, 10 * row++);
            while (e != null) {
                g.drawString("SQLState  : " + e.getSQLState(), 20,
                             10 * row++);
                g.drawString("Message   : " + e.getMessage(), 20,
                             10 * row++);
                g.drawString("ErrorCode : " + e.getErrorCode(), 20,
                             10 * row++);
                e = e.getNextException();
                g.drawString("", 20, 10*row++);
            }
        } catch (Exception e) {
            g.drawString("Other Exception!", 20, 10 * row++);
            g.drawString(e.toString(), 20, 10 * row++);
        }
    }
}
```

Executing the Java Applet Example

To use a Mimer JDBC Driver in a Java applet, copy the driver jar file to the directory containing the applet's Java classes.

This directory must be accessible to the Web server. The driver jar file name should be given as the applet tag's `ARCHIVE` parameter in the HTML file. For example:

```
<html>
 <head>
  <title> The Example Applet
 </head>
 <body>
 Example Applet:
  <applet archive="mimjdbc2.jar"
          code="ExampleApplet.class"
          width=800
          height=600>
  </applet>
 </body>
</html>
```

You execute the applet by accessing the HTML file from a browser, for example:

```
http://my_node/ExampleApplet.html
```

Note: There is a security restriction for Java applets, which states that a network connection can only be opened to the host from which the applet itself was downloaded. This means that both the Web server distributing the applet code and the database server must reside on the same host computer.

Mimer JDBC Midlet Example

This example midlet connects to a Mimer SQL database `db` on the host `my_node.mimer.se` using TCP/IP port `4711`. Instructions on compiling and executing the example is found in the article 'Java programming for mobile phones with Mimer SQL', found at http://developer.mimer.com/howto/howto_43.htm. The example uses the environment described in the article.

Example program:

```
import com.mimer.jdbc.*;
import java.lang.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

// A MIDlet which browse the corporate telephone directory
public class BrowsePhoneNumbers extends MIDlet
    implements CommandListener,Runnable {
    private Command exitCommand,browseCommand,backCommand;
    private Display display;
    private Form mainForm;
    private List resultList;
    private boolean firstTime;
    private MimerDataSource mds;
    private TextField namField,grpField,phnField;
    private Thread dbthread;

    public BrowsePhoneNumbers() {
    display = Display.getDisplay(this);
    exitCommand = new Command("Exit", Command.EXIT, 1);
    browseCommand = new Command("Browse", Command.ITEM, 1);
    backCommand = new Command("Back", Command.BACK, 1);
    mainForm = new Form("Corporate phone dictionary");
    mds = new MimerDataSource();
    mds.setServerName("my_node.mimer.se");
    mds.setDatabaseName("db");
    mds.setPortNumber(4711);
    namField = new TextField("Name", "", 25, TextField.ANY);
    grpField = new TextField("Group", "", 15, TextField.ANY);
    phnField = new TextField("Phone", "", 15, TextField.ANY);
    resultList = new List("Found numbers", Choice.IMPLICIT);
    resultList.addCommand(backCommand);
    resultList.setCommandListener(this);
    firstTime = true;
    }

    // Start the MIDlet by creating the TextBox and
    // associating the exit command and listener.
    public void startApp() {
    if (firstTime) {
        mainForm.append(namField);
        mainForm.append(grpField);
        mainForm.append(phnField);

        firstTime = false;
    }

    mainForm.addCommand(exitCommand);
    mainForm.addCommand(browseCommand);
    mainForm.setCommandListener(this);
    display.setCurrent(mainForm);
    }
```

```
    // Pause is a no-op because there are no background
    // activities or record stores to be closed.
    public void pauseApp() { }

    // Destroy must cleanup everything not handled
    // by the garbage collector.
    // In this case there is nothing to cleanup.
    public void destroyApp(boolean unconditional) { }

    public void commandAction(Command c, Displayable s) {
    if (c == exitCommand) {
        destroyApp(false);
        notifyDestroyed();
    }
    if (c == browseCommand) {
        //    resultList.deleteAll();  This may be uncommented on MIDP 2.0
        display.setCurrent(resultList);
        Thread thread = new Thread(this);
        thread.start();
    }
    if (c == backCommand) {
        display.setCurrent(mainForm);
    }
}

    public void run()
    {
    Connection con;
    try {
        con = mds.getConnection("SYSADM","SYSADM");
        try {
            PreparedStatement ps =
                con.prepareStatement("call PHNQRY(?,?,?)");
            ps.setString(1,namField.getString());
            ps.setString(2,grpField.getString());
            ps.setString(3,phnField.getString());
            ps.execute();
            ResultSet rs = ps.getResultSet();
            while (rs.next()) {
                resultList.append(rs.getString(1)+", "+rs.getString(2)+",
                    "+rs.getString(3),null);
            }
        } finally {
        con.close();
        }
    } catch (SQLException se) {
        resultList.append(se.getMessage()+
                          " SQLCODE "+se.getErrorCode(),null);
    }
    }
}
```

Chapter 3

Programming With JDBC

This chapter describes some programming aspects when using the Mimer JDBC Driver.

We recommend you to read 'JDBCBench, a Java Database Case Study' available on our developer Web site, http://developer.mimer.com/features/feature_16.htm.

Examples in this Chapter

The examples are based on the sample schema that is provided as part of the Mimer SQL distribution. They assume that the example database environment has been created.

Transaction Processing

Mimer SQL uses a method for transaction management called Optimistic Concurrency Control. OCC does not involve any locking of rows as such, and therefore cannot cause a deadlock.

JDBC Transactions

JDBC transactions are controlled through the `Connection` object. There are two modes for managing transactions within JDBC:

- auto-commit
- manual-commit.

The `setAutoCommit` method is used to switch between the two modes.

Auto-commit Mode

Auto-commit mode is the default transaction mode for JDBC. When a connection is made, it is in auto-commit mode until `setAutoCommit` is used to disable auto-commit.

In auto-commit mode each individual statement is automatically committed when it completes successfully, no explicit transaction management is necessary. However, the return code must still be checked, as it is possible for the implicit transaction to fail.

Manual-commit Mode

When auto-commit is disabled, i.e. manual-commit is set, all executed statements are included in the same transaction until it is explicitly completed.

When an application turns auto-commit off, the next statement against the database starts a transaction. The transaction continues either the `commit` or the `rollback` method is called. The next command sent to the database after that starts a new transaction.

Calling the `commit` method ends the transaction. At that stage, Mimer SQL checks whether the transaction is valid and raises an exception if a conflict is identified.

If a conflict is encountered, the application determines how to continue, for example whether to automatically retry the transaction or inform the user of the failure. The application is notified about the conflict by an exception that must be caught and evaluated.

A request to rollback a transaction causes Mimer SQL to discard any changes made since the start of the transaction and to end the transaction.

Use the `commit` or `rollback` methods, rather than using the `SQL COMMIT` or `ROLLBACK` statements to complete transactions, for example:

```
Statement stmt;
int transactionAttempts;

final int MAX_ATTEMPTS = 5; // Maximum transaction attempts

// Open a connection
url = "jdbc:mimer:/ExampleDB";
con = DriverManager.getConnection(url, "MIMER_ADM", "admin");

con.setAutoCommit(false); // Explicit transaction handling

stmt = con.createStatement();

// Loop until transaction successful (or max attempts exceeded)
for (transactionAttempts = 1; ; transactionAttempts++) {
    // Perform an operation under transaction control
    stmt.executeUpdate("UPDATE mimer_store.currencies"
                     + "   SET exchange_rate = exchange_rate * 1.05"
                     + "   WHERE code = 'USD'");

    try {
        con.commit(); // Commit transaction

        System.out.println("Transaction successful");
        break;

    } catch(SQLException sqe) {
        // Check commit error - allow serialization failure
        if (sqe.getSQLState().equals("40001")) {
            // Check number of times the transaction has been attempted
            if (transactionAttempts >= MAX_ATTEMPTS) {
                // Raise exception with application defined SQL state
                throw new SQLException("Transaction failure", "UET01");
            }
        }
        else {
            // Raise all other exceptions to outer handler
            throw sqe;
        }
    } finally {
        con.close();
    }
}
```

Setting the Transaction Isolation Level

The `setTransactionIsolation` method sets the transaction isolation level. The default isolation level for Mimer SQL is `TRANSACTION_REPEATABLE_READ`.

Note: With Enterprise Java Beans, the EJB environment provides the transaction management and therefore explicit transaction management is not required.

Executing an SQL Statement

The `Connection` object supports three types of Statement objects that can be used to execute an SQL statement or stored procedure:

- a Statement object is used to send SQL statements to the database
- the PreparedStatement interface inherits from Statement
- the CallableStatement object inherits both Statement and PreparedStatement methods.

Using a Statement Object

The Connection method `createStatement` is used to create a Statement object that can be used to execute SQL statements without parameters.

The `executeUpdate` method of the Statement object is used to execute an SQL DELETE, INSERT, or UPDATE statement, i.e. a statement that does not return a result set, it returns an `int` indicating the number of rows affected by the statement, for example:

```
int rowCount;

stmt = con.createStatement();

rowCount = stmt.executeUpdate(
                   "UPDATE mimer_store.currencies"
                 + "   SET exchange_rate = exchange_rate * 1.05"
                 + "   WHERE code = 'USD'");

System.out.println(rowCount + " rows have been updated");
```

Using a PreparedStatement Object

Where an SQL statement is being repeatedly executed, a `PreparedStatement` object is more efficient than repeated use of the `executeUpdate` method against a Statement object.

In this case the values for the parameters in the SQL statement (indicated by `?`) are supplied with the `setXXX` method, where `XXX` is the appropriate type for the parameter.

For example:

```
PreparedStatement pstmt;
int rowCount;

pstmt = con.prepareStatement(
                    "UPDATE mimer_store.currencies"
                  + "   SET exchange_rate = exchange_rate * ?"
                  + "   WHERE code = ?");

pstmt.setFloat(1, 1.05f);
pstmt.setString(2, "USD");
rowCount = pstmt.executeUpdate();

pstmt.setFloat(1, 1.08f);
pstmt.setString(2, "GBP");
rowCount = pstmt.executeUpdate();
```

Using a CallableStatement Object

Similarly, when using stored procedures, a `CallableStatement` object allows parameter values to be supplied, for example:

```
CallableStatement cstmt;

cstmt = con.prepareCall("CALL mimer_store.order_item( ?, ?, ? )");

cstmt.setInt(1, 700001);
cstmt.setInt(2, 60158);
cstmt.setInt(3, 2);
cstmt.executeUpdate();
```

The `setNull` method allows a JDBC null value to be specified as an `IN` parameter. Alternatively, use a Java null value with a `setXXX` method.

For example:

```
pstmt.setString(4, null);
```

A more complicated example illustrates how to handle an output parameter:

```
CallableStatement cstmt;

cstmt = con.prepareCall("CALL mimer_store.age_of_adult( ?, ? )");

cstmt.setString(1, "US");
cstmt.registerOutParameter(2, Types.CHAR);

cstmt.executeUpdate();
System.out.println(cstmt.getString(2) + " years");
```

Batch Update Operations

JDBC provides support for batch update operations. The `BatchUpdateException` class provides information about errors that occur during a batch update using the Statement method `executeBatch`.

The class inherits all the method from the class `SQLException` and also the method `getUpdateCounts` which returns an array of update counts for those commands in the batch that were executed successfully before the error was encountered.

For example:

```
try {

…

} catch(BatchUpdateException bue) {
    System.err.println("\n*** BatchUpdateException:\n");

    int [] affectedCount = bue.getUpdateCounts();
    for (int i = 0; i < affectedCount.length; i++) {
        System.err.print(affectedCount[i] + "  ");
    }
    System.err.println();

    System.err.println("Message:     " + bue.getMessage());
    System.err.println("SQLState:    " + bue.getSQLState());
    System.err.println("NativeError: " + bue.getErrorCode());
    System.err.println();

    SQLException sqe = bue.getNextException();
    while (sqe != null) {
        System.err.println("Message:     " + sqe.getMessage());
        System.err.println("SQLState:    " + sqe.getSQLState());
        System.err.println("NativeError: " + sqe.getErrorCode());
        System.err.println();

        sqe = sqe.getNextException();
    }
}
```

Note: The `BatchUpdateException` object points to a chain of `SQLException` objects.

Enhancing Performance

The batch update functionality allows the statement objects to support the submission of a number of update commands as a single batch.

The ability to batch a number of commands together can have significant performance benefits. The methods `addBatch`, `clearBatch` and `executeBatch` are used in processing batch updates.

The `PreparedStatement` example above could be simply rewritten to batch the commands.

For example:

```
PreparedStatement pstmt;
int [] affectedCount;

pstmt = con.prepareStatement(
                   "UPDATE mimer_store.currencies"
                 + "   SET exchange_rate = exchange_rate * ?"
                 + "   WHERE code = ?");

pstmt.setFloat(1, 1.05f);
pstmt.setString(2, "USD");
pstmt.addBatch();

pstmt.setFloat(1, 1.08f);
pstmt.setString(2, "GBP");
pstmt.addBatch();

affectedCount = pstmt.executeBatch();
```

The Mimer SQL database server executes each command in the order it was added to the batch and returns an update count for each completed command.

If an error is encountered while a command in the batch is being processed then a `BatchUpdateException` is thrown (see *Error Handling* on page *14*) and the unprocessed commands in the batch are ignored.

In general it may be advisable to treat all the commands in the batch as a single transaction, allowing the application to have control over whether those commands that succeeded are committed or not.

Set the Connection object's auto-commit mode to off to group the statements together in a single transaction. The application can then commit or rollback the transaction as required.

Calling the method `clearBatch` clears a Statement object's list of commands.

Using the `Close` method to close any of the Statement objects releases the database and JDBC resources immediately. It is recommended that Statement objects be explicitly closed as soon as they are no longer required.

Result Set Processing

There are a number of ways of returning a result set. Perhaps the simplest is as the result of executing an SQL statement using the `executeQuery` method, for example:

```
Statement stmt;
ResultSet rs;

stmt = con.createStatement();

rs = stmt.executeQuery("SELECT *"
                     + "   FROM mimer_store.currencies");

while (rs.next()) {
    System.out.println(rs.getString("CURRENCY"));
```

A ResultSet can be thought of as an array of rows. The 'current row' is the row being examined and manipulated at any given time, and the location in the ResultSet is the 'current row position'.

Information about the columns in a result set can be retrieved from the metadata, for example:

```
Statement stmt;
ResultSet rs;
ResultSetMetaData rsmd;

stmt = con.createStatement();

rs = stmt.executeQuery("SELECT *"
                     + "   FROM mimer_store.currencies");

rsmd = rs.getMetaData();
for (int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(" Type: " + rsmd.getColumnTypeName(i));
    System.out.println(" Size: " + rsmd.getColumnDisplaySize(i));
}
```

Scrolling in Result Sets

The previous examples used forward-only cursors (`TYPE_FORWARD_ONLY`), which means that they only support fetching rows serially from the start to the end of the cursor, this is the default cursor type.

In modern, screen-based applications, the user expects to be able to scroll backwards and forwards through the data. While it is possible to cache small result sets in memory on the client, this is not feasible when dealing with large result sets. Support for scrollable cursors provide the answer.

Scrollable cursors allow you to move forward and back as well as to a particular row within the ResultSet. With scrollable cursors it is possible to iterate through the result set many times.

The Mimer drivers' scrollable cursors are of type `TYPE_SCROLL_INSENSITIVE`, which means that the result set is scrollable but also that the result set does not show changes that have been made to the underlying database by other users, i.e. the view of the database is consistent. To allow changes to be reflected may cause logically inconsistent results.

Positioning the Cursor

There are a number of methods provided to position the cursor:

- `absolute`
- `afterLast`
- `beforeFirst`
- `first`
- `last`
- `next`
- `previous`
- `relative`

There are also methods to determine the current position of the cursor:

- `isAfterLast`
- `isBeforeFirst`
- `isFirst`
- `isLast`

The `getRow` method returns the current cursor position, starting from 1. This provides a simple means of finding the number of rows in a result set.

For example:

```
Statement stmt;
ResultSet rs;

stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                           ResultSet.CONCUR_READ_ONLY);

rs = stmt.executeQuery("SELECT code, currency"
                     + "   FROM mimer_store.currencies"
                     + "   WHERE code LIKE 'A%'");

System.out.println("\nOriginal sort order");
while (rs.next()) {
    System.out.println(rs.getString(1) + "  " + rs.getString(2));
}
```

```
System.out.println("\nReverse order");
while (rs.previous()) {
    System.out.println(rs.getString(1) + "  " + rs.getString(2));
}

rs.last();
System.out.println("\nThere are " + rs.getRow() + " rows");
```

The Mimer JDBC Driver will automatically perform a pre-fetch whenever a result set is created. This means that a number of rows are transferred to the client in a single communication across the network. If only a small number of rows are actually required use `setMaxRows` to limit the number of rows returned in the result set.

Result Set Capabilities

A instance of the `ResultSet` class is created when a query is executed. The capabilities of the result set depend on the arguments used with the `createStatement` (or `prepareStatement` or `prepareCall`) method.

The first argument defines the type of the `ResultSet`, whether it is scrollable or non-scrollable, and the second argument defines the concurrency option, i.e. the update capabilities.

A `ResultSet` should only be made updatable if the functionality is going to be used, otherwise the option `CONCUR_READ_ONLY` should be used. If used, both the type and the concurrency option must be specified.

The following example creates a scrollable result set cursor that is also updatable:

```
stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                           ResultSet.CONCUR_UPDATABLE);
```

Even if the options used specify that the result set will be scrollable and updatable, it is possible that the actual SQL query will return a `ResultSet` that is non-scrollable or non-updatable.

Holdable cursors

The `mimjdbc3.jar` driver supports the JDBC 3 specification. As such it provides an opportunity for application developers to create holdable cursors. The difference between a holdable cursor and a regular cursor is that regular cursors are closed at the end of the transaction. The holdable cursor can (theoretically) stay opened for an unlimited period of time. However, leaving a cursor open for a long period of time may have serious performance implications for the same reason long lasting transactions may impair server performance.

Updating Data

Applications can update data by executing the `UPDATE`, `DELETE`, and `INSERT` statements. An alternative method is to position the cursor on a particular row and then use `DELETE CURRENT`, or `UPDATE CURRENT` statements.

The following example illustrates how this can be done:

```
Statement select;
PreparedStatement update;
ResultSet rs;
```

```
select = con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                             ResultSet.CONCUR_UPDATABLE);

select.setCursorName("CRN"); /* Name the cursor */

rs = select.executeQuery("SELECT currency"
                       + "   FROM mimer_store.currencies"
                       + "   WHERE code = 'ALL'"
                       + "   FOR UPDATE OF currency");

update = con.prepareStatement("UPDATE mimer_store.currencies"
                            + "   SET currency = ?"
                            + "   WHERE CURRENT OF crn");

while (rs.next()) {
    if (rs.getString("CURRENCY").startsWith("Leke")) {
        update.setString(1, "Albanian Leke");
    }
    else {
         update.setString(1, "Leke");
    }
    update.executeUpdate();
}
```

Programming Considerations

Below is a summary of issues to be considered when programming with Mimer JDBC.

Interval Data

Both the JDBC specification and the Java language lack support for `INTERVAL` data types.

You can use the `getString` and `setString` methods for values accessed by a driver from database columns containing `INTERVAL` data.

Closing Objects

Although Java has automatic garbage collection, it is essential that you close JDBC objects, such as `ResultSets`, `Statements` and `Connections`, when done with them.

Closing objects gives your application better control over resources.

If you don't close objects, resources are kept allocated in the database server until garbage collection is triggered, this can exhaust server resources.

Increasing Performance

- **Use Stored Procedures**

  One of the main issues when trying to increase performance is to reduce network traffic. For example, you can increase performance by using the database server to return more accurate data instead of returning a large amount of unqualified data which your application must handle. You can achieve this by using more sophisticated SQL statements or by using stored procedures (PSM).

- **Use More Than One Connection**

  Mimer JDBC drivers are thread-safe and use one lock per connection. So, to achieve higher concurrency, an application should use several connections.

- **Prefetching Data**

  The drivers are implemented to perform automatic `prefetch`, i.e. whenever a `resultSet` is created, a buffer is filled with successive rows. This is an optimization for throughput, allowing more data to be fetched to the client in the single communication made.

  The flip side of the coin is that response time, i.e. the time it takes to receive the first record, may be increased (see *Use setMaxRows* below.)

- **Use setMaxRows**

  If you know that only a small number of records are to be fetched, then you can use the `setMaxRows` method to optimize the response time, i.e. to avoid an array fetch.

- **Use PreparedStatements**

  Another way of increasing performance is to avoid recompiling SQL statements unnecessarily. Whenever you use `Statement.executeXXX` methods, statements are compiled. Instead, use parameterized precompiled statements, i.e. `PreparedStatement`, whenever possible.

- **Use Batched Statements**

  Using the Mimer JDBC Driver version 2 or later, you can reduce the number of network requests by using batched statements.

  If, for example, you replace 20 calls to `Statement.execute()` with 20 calls to `Statement.addBatch()` followed by a `Statement.executeBatch()` call, 20 server network requests are replaced by a single network request.

  If response time is an issue, this simple change may give a twenty-fold performance improvement!

  Note that batched statements for `PreparedStatement` and `CallableStatement` differ from the implementation for the `Statement` class. When using `PreparedStatement` or `CallableStatement`, only a single SQL statement can be used in a batch. The `addBatch()` call (without argument) adds a set of parameters to the batch. When you use batches, the same SQL statement is called repeatedly with different parameters in a single network request.

  In versions 2 and later, you can use the `setFetchSize` method to control the amount of data fetched.

Chapter A

Change History

The following sections document changes in the drivers.

New Functions

This section describes the main new functions of each Mimer JDBC version.

New Functions in 3.18, 2.18 and 1.18

The JDBC version 18 drivers may now connect to Mimer SQL Micro servers. Note however, that many features you normally expect in a Mimer SQL Engine are not available in the Mimer SQL Micro Edition server.

An application may detect the Mimer SQL product type by calling `DatabaseMetaData.getDatabaseProductName()`. This will return "Mimer SQL Micro", "Mimer SQL Mobile", or "Mimer SQL Engine" - depending on the server type.

New Functions in 3.17, 2.17 and 1.17

Support for the `BOOLEAN` SQL data type that was introduced in Mimer SQL 9.3 servers.

New Functions in 3.16, 2.16 and 1.16

- The driver can load and use the mimcomm JNI library which allows the JDBC driver to use all communication methods supported by Mimer on the platform.
- The classes `MimerDataSource`, `MimerConnectionPoolDataSource` and `MimerXADataSource` have two additional properties: protocol and service. These are needed when using the mimcomm JNI library. The new properties are explained further in *Deploying Mimer JDBC in JNDI* on page *13*.

New Functions in 3.15

- The first release of a JDBC 3 compliant driver.
- Holdable cursors.

New Functions in 2.9

Server data type `NATIONAL CHARACTER LARGE OBJECT` (`NCLOB`) is now supported.

New Functions in 2.8

The method PreparedStatement.setBytes is now supported on LONGVARBINARY and PreparedStatement.setString on LONGVARCHAR. In the case of Mimer, LONGVARBINARY is the same as a BLOB, and LONGVARCHAR is the same as a CLOB.

New Functions in 2.7

The object returned when calling .getBinaryStream, .getAsciiStream and .getCharacterStream on BLOB and CLOB objects now implements the .mark(), .reset() and .skip() methods.

New Functions in 2.5

Support for large objects; BINARY LARGE OBJECT (BLOB) and CHARACTER LARGE OBJECT (CLOB). BLOB's store any sequence of bytes, CLOB's store Latin-1 character data.

New Functions in 2.4

Support for server NCHAR and NCHAR VARYING data types. They are used to store Unicode data. By using these data types, any Java String object can now be stored in the database. This is not the case when using CHARACTER or CHARACTER VARYING data types since these can only store Latin-1 characters.

New Functions in 2.3

- Support for javax.sql.DataSource.
- Support for connection pooling using javax.sql.ConnectionPool DataSource
- Support for distributed transactions (XA).

New Functions in 2.0

- Scrollable cursors are now fully supported.
- All date, time and timestamp methods now support the java.util.Calendar class for handling time zones. Mimer SQL 8.2 servers do not currently support time zones and this feature enables you to use time zones.
- Batches of statements are supported.
- Batches of prepared statements are supported. Batches of prepared statements are really useful for increasing performance when executing several INSERT, UPDATE or DELETE statements.
- Batches of callable statements are supported.
- There are now setter and getter methods for CharacterStreams.
- Several new DatabaseMetaData methods.
- Support for the Mimer SQL statements ENTER and LEAVE.

New Functions in 1.9

Server data type NATIONAL CHARACTER LARGE OBJECT (NCLOB) is now supported.

New Functions in 1.7

- When working with a Mimer SQL version 9 server, the JDBC 1 driver now supports the new version 9 data types (`NCHAR`, `NCHAR VARYING`, `BINARY LARGE OBJECTS`, and `CHARACTER LARGE OBJECT`).
- The SQL statements `ENTER` and `LEAVE` are now supported.

New Functions in 1.2

Support for query timeout and cancel (connection timeout is not supported).

Changed Functions

This section describes the main changed functions of each Mimer JDBC version.

Changed Functions in 3.16, 2.16 and 1.16

- **Changed type mapping for FLOAT(n)**

  Mimer SQL supports the datatype FLOAT(n) which can store a floating point number with n digits of mantissa and an exponent ranging from -999 to 999.

  This datatype was previously mapped to the Java type double (which only supports exponents ranging from -308 to 308). This was problematic since some routines (in particular getObject()) would fail for very large (or small) values in the database.

  The FLOAT(n) data type is now mapped to java.math.BigDecimal. While not a perfect match, this datatype can accurately represent all values that can be stored in FLOAT(n) columns in the database.

  Note that it is still possible to use the methods getDouble() and getFloat() on FLOAT(n) data, but those methods will fail when the data is out of range for a Java double (or float).

  To store Java double and float values, consider using the Mimer datatype DOUBLE PRECISION for Java double and the Mimer datatype REAL for Java float.

  Note that the Mimer datatype FLOAT (without a precision) is synonymous to DOUBLE PRECISION and is a bad match for the Java float type which is single precision.

- **Changed string representation for floating point data**

  The JDBC driver supports the getString() method on all Mimer floating point columns. Previously this method padded the returned value with zeroes to its declared precision (a FLOAT(15) could return "1.00000000000000"). This version will not add those zeroes (getString() on the same value will return "1").

Changed Functions in 2.15 and 1.15

- **Login failure now returns SQLSTATE 08004.**

  Previously login failure threw an `SQLException` with the SQLState `28000`. According to the SQL-2003 standard, this is incorrect, and has been corrected to return `08004`. The 08-class of SQLStates relates to error conditions during the connect phase.

- **Several error messages have been clarified**

  Error texts returned when a cast from a character column to something else now more clearly state the failed cast. Note that this particular improvement applies to client side casts only. For instance, this includes casts where an SQL parameter type is `INTEGER` and its value is set using the `PreparedStatement.setString` method.

  The driver now displays an accurate error text when a connection attempt fails because the application hasn't specified the database name, or it has specified an empty database name.

  When the application refers to a column name that does not exist in the result set, the error text now includes existing columns names. To keep error texts reasonably short, if this error occurs on a result set with many columns, only a selection of column names. This situation is indicated with three consecutive periods in the error text.

  Errors returned from the Mimer TCP server (listening on port 1360 on behalf of Mimer SQL servers) now include a descriptive text, previously only the error code was displayed to the caller.

Changed Functions in 2.14 and 1.14

JDBC clients now present more detailed information to the servers about who it is, which version it is and in which environment it is executing in. Future servers will provide tools to monitor this information.

Changes in 2.14 and 1.14

- **Changing autocommit mode always commits open transactions**

  Earlier on, the Mimer JDBC driver mimiced the ODBC behavior when autocommit mode is changed. The ODBC spec says that open transactions should be automatically committed when the autocommit mode goes from off to on. The JDBC specification requires drivers to commit open transactions on all changes in autocommit state. From version 14 onwards, the Mimer JDBC driver implements this behavior.

  An observant reader might question why this has any significance at all? After all, when autocommit mode is on, we expect all statements to be committed automatically anyway? The difference lies in how open result sets are treated. As you may know, result sets are by default closed when transactions are committed. In practice, running in autocommit mode means that transactions are committed _as_soon_as_possible_. For instance, a statement returning a result set will typically be committed when the application explicitly closes the result set, or if the result set is forward-only when the entire set has been read. Changing the autocommit mode during the life of the result set will now always trigger a commit which will close the result set.

Changes in 2.9

The driver now returns the correct object type when doing `CallableStatement.getObject`. According to the JDBC specification, `getObject` should return a Java object whose type corresponds to what type the output parameter was registered to with the `CallableStatement.registerOutParameter` method call. Earlier drivers always returned the default Java object type.

Changes in 2.7

- All `.getUnicodeStream` on `NCHAR` columns no longer throw `IndexOutOfBoundsException`.
- All `.getCharacterStream` returned incorrect results for `NCHAR` and `NCHAR VARYING` columns. This problem is corrected.
- All `.getAsciiStream`, `.getBinaryStream`, and `.getCharacterStream` on `CHAR`, `CHAR VARYING`, `NCHAR`, `NCHAR VARYING`, `BINARY` and `BINARY VARYING` columns have been reworked to reduce memory footprint, and also to provide more efficient `.mark()`, `.reset()`, and `.skip()` implementations.

Changes in 2.2

- Column names and labels are now regarded as equal. From an SQL standard point of view, the column name should be hidden when a correlation name is specified.
- Both `ResultSetMetaData.getColumnName` and `ResultSetMetaData.getColumnLabel` return the correlation name when one is specified.
- Default network buffers have been reduced in size to increase server scalability.

Changes in 2.1

A Mimer SQL beta licence key is no longer required on the server.

Changes in 1.3

`Statement.executeQuery` no longer accepts non-query statements and `Statement.executeUpdate` no longer accepts statements other than updates, inserts or deletes.

Changes in 1.2

The name of the Mimer driver class is changed to `com.mimer.jdbc.Driver` (earlier `com.mimer.jdbc1.Driver`).

Corrected Problems

This section describes the main corrected functions of each Mimer JDBC version.

Correction in 3.20, 2.20 and 1.20

If the connection with the server was lost (or if the server is shut down), earlier JDBC drivers could produce a null pointer exception in some circumstances. The new JDBC driver will produce an appropriate SQLException.

Also, the method `Connection.isClosed()` will now return true on any connection that has received an SQLException indicating that the connection with the server was lost.

Correction in 3.19, 2.19 and 1.19

Version numbers for servers older than 9.3 was not returned properly by the Mimer JDBC *n*.18 drivers. This problem was seen in `DatabaseMetaData.getProductVersion`, `DatabasMetaData.getDatabaseMajorVersion` and `DatabaseMetaData.getDatabaseMinorVersion`.

Corrections in 3.18, 2.18 and 1.18

- **PreparedStatement.setString threw no exception on BOOLEAN data types**

  Version 17 JDBC drivers did never throw an exception if the application called `PreparedStatement.setString` with an illegal string. That is, a string that is not '`true`' or '`false`'. This is corrected in version 18.

- **LITERAL_SUFFIX and LITERAL_PREFIX for DATE, TIME, TIMESTAMP and INTERVAL data types**

  Result sets returned by `DatabaseMetaData.getTypeInfo` did not contain any data in the columns `LITERAL_SUFFIX` and `LITERAL_PREFIX` for data types DATE, TIME, TIMESTAMP and all INTERVAL data types. From version 18, these columns have a relevant value.

  This correction applies for all server versions. Using an older JDBC driver against a v9.3.5 Mimer SQL server or later will also return correct values for these columns.

- **Changed behavior for protocol type tcp**

  Specifying the protocol TCP in the Mimer JDBC URL, would instruct the driver to connect using the native TCP/IP-stack. From version 18, specifying the TCP protocol makes the driver connect using the Java TCP/IP-stack.

  The behavior when the protocol is unspecified has not been changed, that is, the Java TCP/IP-stack is used.

Correction in 3.16, 2.16 and 1.16

- **DatabaseMetaData.getColumns returns too many columns**

  Older versions of the Mimer JDBC driver returned, when calling `DatabaseMetaData.getColumns`, duplicate rows for BINARY LARGE OBJECT, CHARACTER LARGE OBJECT and NATIONAL CHARACTER LARGE OBJECT columns. For example, querying a table with one CHARACTER column and one BINARY LARGE OBJECT column returned a result set of three rows. One row for the CHARACTER column, and two for the BINARY LARGE OBJECT column. This is now corrected.

- **DATE/TIME comparison problems**

  Previous drivers did not recognize TIMESTAMP's representing a value prior to the timestamp 1000-01-01 00:00:00, DATE values prior to the date 1000-01-01 and TIME values prior to 10:00:00 correctly. Although the driver would retrieve and display those values correctly, comparison operations may fail for identical values, leading to potentially duplicate primary keys, or that query conditions may fail for no obvious reason. These problems are now corrected.

Corrections in 2.14

- **PreparedStatement batches whose size exactly matched the network buffer size failed**

  Batches of PreparedStatements failed if data in the entire batch exactly matched the amount of space available in the network buffer.

  This could mean, for instance, that a batch of 19 rows would fail, while batches of 18 and 20 rows would succeed.

- **Improved default connection values for MIDP drivers**

  The MIDP driver can only connect using the javax.sql.DataSource compliant `com.mimer.jdbc.MimerDataSource` class. From version 14 onwards, MIDP drivers default all values (host name, database name, port number) to those required to connect to the local database 'db'. This is the default name of the local database on Symbian devices.

  Ident name and password must always be completed by the application.

Corrections in 2.14 and 1.14

- **MIDP:Reduced network buffers**

  MIDP drivers has from the start only utilized network buffers of about 10 kb. Unfortunately, at times the driver would allocate network buffers larger than necessary. This is no longer the case. The programmer might influence the size of the network buffer by using the `ResultSet.setFetchSize` function. If this function is not called, the driver will use a fetch size corresponding to a size of about 10 kb.

- **Non-public constructor in the Driver-class made applications fail loading the Mimer JDBC driver**

  Applications that don't rely on `DriverManager.getConnection`, or the `javax.sql.DataSource` class, to create a Connection to Mimer, but instead are creating a connection by using the Driver class couldn't load the `com.mimer.jdbc.Driver` class of the 1.13 and 2.13 Mimer JDBC drivers. More specifically, the following didn't work:

  ```
  Class dc = Class.forName("com.mimer.jdbc.Driver");
  Driver d = (Driver)dc.getInstance();
  ```

  Example of products using this (or similar) techniques, and thus avoiding the `DriverManager` object, are Sun Java Studio Creator and the Squirrel SQL database viewer.

- **Reading a BLOB stream might hang the connection**

  Reading Binary Large Objects through an `InputStream` (obtained through `ResultSet.getBinaryStream`) would place the network connection in an inconsistent state, in practice the session would hang, if the size of the object is larger than the size of the default network packet size, and the application tries to read the entire object in one call (`InputStream.read(b,off,len)` where len is larger than the size of the object). This is no longer the case.

- **Clarified error texts when streams are closed**

  The error texts saying that streams have been closed now explain why the stream was closed. This could be of several reasons, the server connection went dead, the transaction was committed or rolled back, the statement in which the result set containing the stream was closed, and so forth. This is now explained in the error message.

Corrections in 2.13 and 1.13

- **Fetching data might throw an SQLException with vendor code 1**

  Fetching data (`ResultSet.next`) could erroneously throw SQLExceptions with vendor code 1. This was wrong and is now corrected.

- **Large BLOB problem**

  Whenever BLOB's was read in several passes from the server, and the application specified a length longer than the actual BLOB, the driver hanged. Many platforms deliver TCP/IP packets in chunks of about 64 kb so this problem would occur when reading BLOB's larger than that. On smaller platforms, such as the Sony Ericsson P800 mobile telephone, the problem is more evident since the MIDP environment on that platform delivers TCP/IP packets in sizes of 512 bytes. This problem is now corrected.

Corrections in 2.12 and 1.12

- The midjdbc2 driver now fully supports SQL `DATE`, `TIME`, and `TIMESTAMP` data types.

- By mistake the Beta release (1.10/2.10) lacked support of the SQL constructs for manipulating session and transaction characteristics, such as `SET TRANSACTION READ ONLY`. These SQL statements are now supported, see the *Mimer SQL Reference Manual* for more information.
- Scrollable cursors now take the value set by `Statement.setFetchDirection` into account when selecting fetch strategy on scrollable result sets. This does not apply to 1.12.
- A problem which caused a premature end of table when a scrollable cursor has seen the end of the result set, is fetching backwards (using `ResultSet.previous`) and the fetch size has been set to a value less than the size of the result set. This problem did not apply to the JDBC 1 driver.
- When earlier versions did a `ResultSet.afterLast` or `ResultSet.last` after setting `Statement.maxRows` to a value that actually limits the result set size, the cursor was positioned on the wrong row, beyond the end of the result set. This is now corrected.

Corrections in 2.11 and 1.11

- Previously an `InputStream.skip(n)` on a stream derived from a `BLOB` column, or a `CharacterStream.skip(n)` on a stream derived from a `CLOB` or `NCLOB` column may leave the network state out of sync. This was seen with the error -22046 'An internal error occurred in ReadFromServer'. This problem is now corrected.

  Note: For the JDBC 1 driver, this problem applies to streams derived from the `getUnicodeStream` method call on `CLOB` and `NCLOB` columns.
- A problem with large SQL statements has been fixed. SQL statements larger than about 20 000 characters were unable to compile because of an `ArrayIndexOutOfBoundsException`.

Corrections in 2.10 and 1.10

- Previous versions of the driver did not return the correct data type on `CallableStatement.getObject` calls. The specification states that the object type returned should match whatever type was specified when the output parameter was registered through the `CallableStatement.registerOutParameter` call. Previously, the object type returned matched the data type on the server.
- When calling `ResultSet.findColumn`, `ResultSet.getString(String)` and similar column name related methods, the Mimer driver previously did a case sensitive search. This was incorrect. The search should be case insensitive, which it now is.
- 2.8 and 1.8 versions of the Mimer JDBC driver introduced a problem setting `CHARACTER` and `CHARACTER VARYING` columns via a `CharacterStream` object. The end result lost characters without throwing errors. This is now corrected.
- A JDBC driver using a database server with many indexes could have performance problems with the `DatabaseMetaData.getSchemas` call. This is now corrected for 9.2-servers and later. Unfortunately, since the problem is server related, older servers cannot easily be corrected.

- JDBC drivers connecting to Mimer SQL 8.2 servers unexpectedly threw `SQLException` exceptions when using `DatabaseMetaData.getCatalogs` or `DatabaseMetaData.getUDTs`. This is now corrected. Note that neither of these queries should return any rows with Mimer SQL 8.2 servers.
- `java.sql.Blob` and `java.sql.Clob` objects returned from calls to `ResultSet.getBlob` and `ResultSet.getClob` now stay alive throughout the entire transaction. Once the transaction in which the object is created is ended, all calls to the objects will throw a 'transaction has ended' exception. Previously, these objects could not be used once the resultset was closed.

Corrections in 2.9

- Scrollable result sets returned an error when calling `setFetchDirection`. This is no longer the case.
- `ResultSet.getString` did not return correct characters for `å`, `ä`, `ö` and similar Latin-1 but non-ASCII characters when other default character encoding than ISO 8859-1 was used. This included for instance Macintosh computers. This is now corrected.

Corrections in 2.7

- Earlier versions incorrectly returned `SQLSTATE 22001` for numeric value out of range. The correct `22003` is now returned.
- Procedure calls with large output parameters (typically `CHAR(100)`, `VARCHAR(100)` or larger) could end with the following exception message:

  ```
  An internal error occurred in MimConnection.readFromServer (packlen=148,
  bufLen=100, maxReceive=0).
  ```

  This problem is now corrected.
- Batches of statements were not cleared when being executed. This forced the programmer to call `Statement.clearBatch()` before building another batch. From now on, batches are automatically cleared after being executed.

Corrections in 2.6

Server resources was not released even when the application was properly closing `Statement`, `PreparedStatement` and `CallableStatement` objects. This could sometimes cause the following error when attempting to drop a table:

```
Error code: -16002, msg: Table locked by another cursor, state: S1000
```

This problem is now corrected.

Corrections in 2.2

- Correction of `DatabaseMetaData.supportsTransactionIsolationLevel(0)` which erroneously returned true.
- `ResultSet.getConcurrency()` and `ResultSet.getType()` returned wrong values for scrollable cursors.

- `DatabaseMetaData.getSystemFunctions()` returned the nonexisting `USERNAME` function.
- A `ResultSet.fetchSize` with a large number no longer throws an `ArrayIndexException`.
- `ResultSets` created from a `PreparedStatement` or `CallableStatement` no longer fails on the second `.next` call.

Corrections in 1.9

- `ResultSet.getString` did not return correct characters for `å`, `ä`, `ö` and similar Latin-1 but non-ASCII characters when other default character encoding than ISO 8859-1 was used. This included for instance Macintosh computers. This is now corrected.
- The driver now returns the correct object type when doing `CallableStatement.getObject`. According to the JDBC specification, `getObject` should return a Java object whose type corresponds to what type the output parameter was registered to with the `CallableStatement.registerOutParameter` method call. Earlier drivers always returned the default Java object type.

Corrections in 1.7

Earlier versions incorrectly returned `SQLSTATE 22001` for numeric value out of range. The correct 22003 is now returned.

Known Restrictions

- Mimer SQL Mobile does not support `.mark()` on `.getBinaryStream`, `.getAsciiStream` and `.getCharacterStream`. Use `Bufferedxxx` manually.
- The following optional features described in the JDBC 2 specification are not yet supported:
  - Connection timeout
  - Updatable `ResultSets`
  - The `java.sql.Blob.position` and `java.sql.Clob.position` methods
  - The `java.sql.Array`, `java.sql.Ref`, and `java.sql.Struct` objects among with the getter and setter methods for the objects.

Known Problems

This section describes the known problems with Mimer JDBC.

Update Counts on Errors in Batched Statements

Whenever an error occurs in a batched Statement, the driver is unable to return the correct information about the number of executed rows. The correct behavior is to return an integer array within a thrown `BatchUpdateException` object whose length corresponds to the number of batch statements. The Mimer driver is now returning an integer array with one entry per statement, with all entries set to 0.

Index