BEA WebLogic Enterprise

JDBC Driver
Programming Reference

BEA WebLogic Enterprise 4.2
Document Edition 4.2
July 1999

Copyright

Restricted Rights Legend

Trademarks or Service Marks

JDBC Driver Programming Reference

Document Edition	Date	Software Version
4.2	July 1999	BEA WebLogic Enterprise 4.2

Contents

3. Using the jdbcKona/ MSSQLServer4 Driver

4. Extensions to the JDBC API

Preface

Purpose of This Document

This document provides reference information on the jdbcKona drivers, which are packaged and installed with the BEA WebLogic Enterprise (sometimes referred to as WLE) software.

Note: Effective February 1999, the BEA M3 product is renamed. The new name of the product is BEA WebLogic Enterprise (WLE).

Who Should Read This Document

This document is intended for WebLogic Enterprise server application developers who need to use a JDBC driver to access a database management system (DBMS).

How This Document Is Organized

The *JDBC Driver Programming Reference* is organized as follows:

- Chapter 1, "Using the jdbcKona Drivers," provides an overview of how to use the jdbcKona drivers with the WebLogic Enterprise system. This chapter also provides some vendor-specific details on the jdbcKona drivers, and also contains a sample implementation that lists and describes the procedure for using a jdbcKona driver with a WebLogic Enterprise Java server application.

- Chapter 2, “Using the jdbcKona/Oracle Driver,” provides specific details about how to use the jdbcKona/Oracle driver to connect a WebLogic Enterprise Java server application to an Oracle DBMS.
- Chapter 3, “Using the jdbcKona/ MSSQLServer4 Driver,” provides specific details about how to use the jdbcKona/MSSQLServer4 driver to connect a WebLogic Enterprise Java server application to Microsoft’s SQL Server.
- Chapter 4, “Extensions to the JDBC API,” documents the API to the jdbcKona extensions to the JDBC application programming interface (API).

How to Use This Document

This document, *JDBC Driver Programming Reference,* is designed primarily as an online, hypertext document. If you are reading this as a paper publication, note that to get full use from this document you should access it as an online document via the Online Documentation CD for the BEA WebLogic Enterprise 4.2 release.

The following sections explain how to view this document online, and how to print a copy of this document.

Opening the Document in a Web Browser

To access the online version of this document, open the following file:

```
\doc\wle\v42\index.htm
```

Note: The online documentation requires Netscape Communicator version 4.0 or later, or Microsoft Internet Explorer version 4.0 or later.

Printing from a Web Browser

You can print a copy of this document, one file at a time, from the Web browser. Before you print, make sure that the chapter or appendix you want is displayed and *selected* in your browser. To select a chapter or appendix, click anywhere inside the chapter or appendix you want to print.

The Online Documentation CD includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document. On the CD Home Page, click the PDF Files button and scroll to the entry for the document you want to print.

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`

Convention	Item
monospace italic text	Identifies variables in code. *Example*: `String expr`
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
`...`	Indicates one of the following in a command line: ♦ That an argument can be repeated several times in a command line ♦ That the statement omits additional optional arguments ♦ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Related Documentation

The following sections list the documentation provided with the BEA WebLogic Enterprise software, related BEA publications, and other publications related to the technology.

BEA WebLogic Enterprise Documentation

The BEA WebLogic Enterprise information set consists of the following documents:

Installation Guide

C++ Release Notes

Java Release Notes

Getting Started

Guide to the University Sample Applications

Guide to the Java Sample Applications

Creating Client Applications

Creating C++ Server Applications

Creating Java Server Applications

Administration Guide

Using Server-to-Server Communication

C++ Programming Reference

Java Programming Reference

Java API Reference

JDBC Driver Programming Reference (this document)

System Messages

Glossary

Technical Articles

Note: The Online Documentation CD also includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document.

BEA Publications

Selected BEA TUXEDO Release 6.5 for BEA WebLogic Enterprise version 4.2 documents are available on the Online Documentation CD.

To access these documents:

1. Click the Other Reference button from the main menu.
2. Click the TUXEDO Documents option.

Other Publications

For more information about CORBA, Java, and related technologies, refer to the following books and specifications:

Cobb, E. 1997. *The Impact of Object Technology on Commercial Transaction Processing*. VLDB Journal, Volume 6. 173-190.

Edwards, J. with DeVoe, D. 1997. *3-Tier Client/Server At Work*. Wiley Computer Publishing.

Edwards, J., Harkey, D., and Orfali, R. 1996. *The Essential Client/Server Survival Guide*. Wiley Computer Publishing.

Flanagan, David. May 1997. *Java in a Nutshell*, 2nd Edition. O'Reilly & Associates, Incorporated.

Flanagan, David. September 1997. *Java Examples in a Nutshell.* O'Reilly & Associates, Incorporated.

Fowler, M. with Scott, K. 1997. *UML Distilled, Applying the Standard Object Modeling Language*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series.

Jacobson, I. 1994. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley.

Mowbray, Thomas J. and Malveau, Raphael C. (Contributor). 1997. *CORBA Design Patterns*, Paper Back and CD-ROM Edition. John Wiley & Sons, Inc.

Orfali, R., Harkey, D., and Edwards, J. 1997. *Instant Corba*. Wiley Computer Publishing.

Orfali, R., Harkey, D. February 1998. *Client/Server Programming with Java and CORBA*, 2nd Edition. John Wiley & Sons, Inc.

Otte, R., Patrick, P., and Roy, M. 1996. *Understanding CORBA*. Prentice Hall PTR.

Rosen, M. and Curtis, D. 1998. *Integrating CORBA and COM Applications*. Wiley Computer Publishing.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Loresen, W. 1991. *Object-Oriented Modeling and Design*. Prentice Hall.

The Common Object Request Broker: Architecture and Specification. Revision 2.2, February 1998. Published by the Object Management Group (OMG).

CORBAservices: Common Object Services Specification. Revised Edition. Updated: November 1997. Published by the Object Management Group (OMG).

Contact Information

The following sections provide information about how to obtain support for the documentation and the software.

Documentation Support

If you have questions or comments on the documentation, you can contact the BEA Information Engineering Group by e-mail at **docsupport@beasys.com**. (For information about how to contact Customer Support, refer to the following section.)

Customer Support

If you have any questions about this version of the BEA WebLogic Enterprise product, or if you have problems installing and running the BEA WebLogic Enterprise software, contact BEA Customer Support through BEA WebSupport at `www.beasys.com`. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

1 Using the jdbcKona Drivers

This chapter covers general guidelines for using the jdbcKona drivers and some vendor-specific notes on each driver. Included at the end of this chapter is a summary of the steps you take, including sample code, to use a JDBC driver in a WebLogic Enterprise Java application.

Note: The jdbcKona drivers are based on JDBC 1.22

The jdbcKona drivers include both Type 2 and Type 4 drivers. The Type 2 drivers (for Oracle) employ client libraries supplied by the database vendors. The Type 4 drivers (for the Microsoft SQL Server) are 100% pure Java; they connect to the database server at the wire level without vendor-supplied client libraries.

Platforms Supported by the jdbcKona Drivers

The following table lists the platforms supported by the jdbcKona drivers.

JDBC Driver	Operating System and Version	JVM	DBMS	Client Libraries
jdbcKona/ Oracle	Windows NT 4.0 (SP4)	Java 2	Oracle 7.3.4 or higher	Oracle 7.3.4
	Solaris 2.4, 2.5, and 2.6	Java 2	Oracle 7.3.4 or higher	Oracle 7.3.4
jdbcKona/ MSSQLServer4	Not applicable	Java 2	Microsoft SQL Server 6.5 (SP3)	Not applicable

Adding the jdbcKona JAR File to Your CLASSPATH

Be sure to add the jdbcKona JAR file, which applies to both jdbcKona drivers, to your environment. You can do this by appending the following to your `CLASSPATH` system environment variable:

On Solaris Systems:

```
$TUXDIR/udataobj/java/jdbc/jdbcKona.jar
```

On NT Systems:

```
$TUXDIR\udataobj\java\jdbc\jdbcKona.jar
```

jdbcKona/Oracle Shared Libraries and Dynamic Link Libraries

The jdbcKona/Oracle (Type 2) driver calls native libraries that are supplied with the driver. The UNIX libraries (shared object files) are in the `$TUXDIR/lib` directory. The Windows `DLL` files are included in the WLE Java software kit in the `$TUXDIR\bin` directory.

The following table lists the names of the driver files included with the WLE Java system.

JDBC Driver	Windows NT/95	UNIX
jdbcKona/Oracle	`weblogicoci33.dll`	`libweblogicoci33.so`

For the jdbcKona/Oracle driver, you also need the vendor-supplied libraries for the database.

Requirements for Making a Connection to a Database Management System (DBMS)

You need the following components to connect to a DBMS using a jdbcKona driver:

- A database server (Oracle or Microsoft SQL Server)
- The jdbcKona driver for your database
- The Java 2 software

Support for JDBC Extended SQL

The Sun Microsystems, Inc. JDBC specification includes *SQL Extensions*, also called *SQL Escape Syntax*. All jdbcKona drivers support Extended SQL. Extended SQL provides access to common SQL extensions in a way that is portable between DBMSs.

For example, the function to extract the day name from a date is not defined by the SQL standards. For Oracle, the SQL is:

```
select to_char(date_column, 'DAY') from table_with_dates
```

Using Extended SQL, you can retrieve the day name for both DBMSs, as follows:

```
select {fn dayname(date_column)} from table_with_dates
```

The following is an example that demonstrates several features of Extended SQL:

```
String insert=
  "-- This SQL includes comments and JDBC extended SQL syntax.       \n" +
  "insert into date_table values( {fn now()},        -- current time \n" +
  "                               {d '1997-05-24'},  -- a date       \n" +
  "                               {t '10:30:29' },   -- a time       \n" +
  "                {ts '1997-05-24 10:30:29.123'},  -- a timestamp  \n" +
  "                '{string data with { or } will not be altered')  \n" +
  "-- Also note that you can safely include { and } in comments or   \n" +
  "-- string data.";
Statement stmt = conn.createStatement();
stmt.executeUpdate(query);
```

Extended SQL is delimited with curly braces ({}) to differentiate it from common SQL. Comments are preceded by two hyphens, and are ended by a newline character (\n). The entire Extended SQL sequence, including comments, SQL, and Extended SQL, is placed within double quotes and is passed to the `execute` method of a `Statement` object.

The following is Extended SQL used as part of a `CallableStatement` object:

```
CallableStatement cstmt =
  conn.prepareCall("{ ? = call func_squareInt(?)}");
```

The following example shows that you can nest extended SQL expressions:

```
select {fn dayname({fn now()})}
```

You can retrieve lists of supported Extended SQL functions from a `DatabaseMetaData` object. The following example shows how to list all the functions a JDBC driver supports:

```
DatabaseMetaData md = conn.getMetaData();
System.out.println("Numeric functions:     " + md.getNumericFunctions());
System.out.println("\nString functions:    " + md.getStringFunctions());
System.out.println("\nTime/date functions: " + md.getTimeDateFunctions());
System.out.println("\nSystem functions:    " + md.getSystemFunctions());
conn.close();
```

Refer to Chapter 11 of the JDBC 1.2 specification at the Sun Microsystems, Inc. Web site for a description of Extended SQL.

The JDBC API, with WebLogic Extensions

For the complete set of JDBC API documentation, see the following Web site:

`http://www.weblogic.com/docs/classdocs/packages.html#jdbc`

The following packages, classes, interfaces, and WebLogic extensions compose the JDBC API:

```
Package java.sql
Package java.math

Class java.lang.Object
   Interface java.sql.CallableStatement
    (extends java.sql.PreparedStatement)
   Interface java.sql.Connection
   Interface java.sql.DatabaseMetaData
   Class java.util.Date
      Class java.sql.Date
      Class java.sql.Time
      Class java.sql.Timestamp
   Class java.util.Dictionary
      Class java.util.Hashtable
       (implements java.lang.Cloneable)
          Class java.util.Properties
   Interface java.sql.Driver
   Class java.sql.DriverManager
   Class java.sql.DriverPropertyInfo
   Class java.lang.Math
```

```
Class java.lang.Number
   Class java.math.BigDecimal
   Class java.math.BigInteger
Interface java.sql.PreparedStatement
 (extends java.sql.Statement)
Interface java.sql.ResultSet
Interface java.sql.ResultSetMetaData
Interface java.sql.Statement
Class java.lang.Throwable
   Class java.lang.Exception
      Class java.sql.SQLException
         Class java.sql.SQLWarning
            Class java.sql.DataTruncation
Class java.sql.Types
Class weblogic.jdbc.oci.Connection
 (implements java.sql.Connection)
Class weblogic.jdbc.oci.Statement
 (implements java.sql.Statement)
    Class weblogic.jdbc.oci.PreparedStatement
       Class weblogic.jdbc.oci.CallableStatement
        (implements java.sql.CallableStatement)
```

The jdbcKona drivers provide extensions to JDBC for certain database-specific enhancements. The jdbcKona drivers have the following extended classes:

```
Class weblogic.jdbc.oci.CallableStatement
Class weblogic.jdbc.oci.Connection
Class weblogic.jdbc.oci.Statement
```

For more information about these extensions, see Chapter 4, "Extensions to the JDBC API."

Implementing a WLE Java Application Using the jdbcKona Drivers

This section describes the following steps involved in implementing a simple WLE Java application that uses a jdbcKona driver to connect to a DBMS:

- Importing Packages
- Setting Properties for Connecting to the DBMS

- Connecting to the DBMS
- Making a Simple SQL Query
- Inserting, Updating, and Deleting Records
- Creating and Using Stored Procedures and Functions
- Disconnecting and Closing Objects

Many of the steps described in this section include code snippets from a comprehensive code example that is provided at the end of this chapter.

For database-specific details on implementing WLE Java applications using the jdbcKona drivers, see Chapter 2, "Using the jdbcKona/Oracle Driver," and Chapter 3, "Using the jdbcKona/ MSSQLServer4 Driver."

Importing Packages

The classes that you import into your WLE Java server application that uses a jdbcKona driver should include:

```
import java.sql.*;
import java.util.Properties;
```

The jdbcKona drivers implement the `java.sql` interface. You write your WLE Java application using the `java.sql` classes; the `java.sql.DriverManager` maps the jdbcKona driver to the `java.sql` classes.

You do not import the jdbcKona driver class; instead, you load the driver inside the application. This allows you to select an appropriate driver at runtime. You can even decide after the program is compiled what DBMS to connect to.

Included in the WLE Java software is the latest version of the JDBC API class files. Make sure you do not have any earlier versions of the `java.sql` classes in your `CLASSPATH`.

You need to import the `java.util.Properties` class only if you use a `Properties` object to set parameters for connecting to the DBMS.

Setting Properties for Connecting to the DBMS

In the following example, a `java.util.Properties` object sets the parameters for connecting to the DBMS. There are other ways of passing these parameters to the DBMS that do not require a `Properties` object, as in the following snippet:

```
Properties props = new Properties();
props.put("user",          "scott");
props.put("password",      "tiger");
props.put("server",        "DEMO");
```

The value for the server property may be vendor-specific; in this example, it is the V2 alias of an Oracle database running over TCP. You may also add the server name to the URL (see the next section) instead of setting it with the `java.util.Properties` object.

Connecting to the DBMS

In general, to a connect to a DBMS, you need to perform the following steps:

1. Load the proper jdbcKona driver.

   The most efficient way to load the jdbcKona driver is to invoke the `Class.forName().newInstance` method with the name of the driver class. This loads and registers the jdbcKona driver, as in the following example for jdbcKona/Oracle:

   ```
   Class.forName("weblogic.jdbc.oci.Driver").newInstance();
   ```

2. Obtain a JDBC connection.

   You request a JDBC connection by invoking the `DriverManager.getConnection` method, which takes as its parameters the URL of the driver and other information about the connection, such as the location of the database and login information.

Note that both steps describe the jdbcKona driver, but in different formats. The full pathname for the driver is period-separated, while the URL is colon-separated. The following table lists the class paths and URLs for the jdbcKona drivers:

JDBC Driver	Driver Type	Class Pathname	Class URL
jdbcKona/Oracle	Type 2	`weblogic.jdbc.oci.Driver`	`jdbc:weblogic:oci`
jdbcKona/ MSSQLServer4	Type 4	`weblogic.jdbc. mssqlserver4.Driver`	`jdbc:weblogic:mssqlserver4`

Additional information required to form a database connection varies by DBMS vendor and by whether the jdbcKona driver is of Type 2 or Type 4. There are also a variety of methods for specifying this information in your program.

For full details about the jdbcKona drivers, refer to Chapter 2, "Using the jdbcKona/Oracle Driver," and Chapter 3, "Using the jdbcKona/ MSSQLServer4 Driver." For a complete code example, see "Implementing a WLE Java Application Using the jdbcKona Drivers" on page 1-6.

The connection to the DBMS is handled by the jdbcKona driver. You use both the class name of the driver (in dot-notation) and the URL of the driver (with colons as separators). Class names are case sensitive.

The `Class.forName().newInstance` method loads the driver and registers the driver with the `DriverManager` object.

Note: The Sun Microsystems, Inc. *JDBC API Reference* for the `java.sql.Driver` interface recommends simply invoking `Class.forName("`*`driver-class`*`")` to load the driver.

The connection is created with the `DriverManager.getConnection` method, which takes as arguments the URL of the driver and a `Properties` object, as in the following snippet. The URL is not case sensitive.

```
Class.forName("weblogic.jdbc.oci.Driver").newInstance();
Connection conn =
  DriverManager.getConnection("jdbc:weblogic:oracle",
                              props);
conn.setAutoCommit(false);
```

The default transaction mode for JDBC assumes autocommit to be true. Setting autocommit to false improves performance.

The `Connection` object is an important part of the application. The `Connection` class has constructors for many fundamental database objects that you will use throughout the application. In the examples that follow, you will see the Connection object `conn` used repeatedly.

Connecting to the database completes the initial portion of a WLE Java application, which will be very much the same for any application.

Invoke the `close` method on the Connection object as soon as you finish working with the object, usually at the end of a class.

Making a Simple SQL Query

The most fundamental task in database access is to retrieve data. With a jdbcKona driver, retrieving data is a three-step process:

1. Create a `Statement` object to send an SQL query to the DBMS.
2. Execute the `Statement`.
3. Retrieve the results into a `ResultSet` object.

In the following code snippet, we execute a simple query on the `Employee` table (alias `"emp"`) and display data from three of the columns. We also access and display metadata about the table from which the data was retrieved. Note that we close the Statement at the end.

```
Statement stmt = conn.createStatement();
stmt.execute("select * from emp");
ResultSet rs = stmt.getResultSet();

while (rs.next()) {
  System.out.println(rs.getString("empid") + " - " +
                     rs.getString("name")  + " - " +
                     rs.getString("dept"));
  }

ResultSetMetaData md = rs.getMetaData();

System.out.println("Number of columns: " + md.getColumnCount());
for (int i = 1; i <= md.getColumnCount(); i++) {
    System.out.println("Column Name: "     + md.getColumnName(i));
    System.out.println("Nullable: "        + md.isNullable(i));
    System.out.println("Precision: "       + md.getPrecision(i));
```

```
    System.out.println("Scale: "           + md.getScale(i));
    System.out.println("Size: "            + md.getColumnDisplaySize(i));
    System.out.println("Column Type: "     + md.getColumnType(i));
    System.out.println("Column Type Name: "+ md.getColumnTypeName(i));
    System.out.println("");
  }

stmt.close();
```

Inserting, Updating, and Deleting Records

The following snippet shows three common database tasks: inserting, updating, and deleting records from a database table. We use a JDBC `PreparedStatement` object for these operations; we create the `PreparedStatement` object, then execute the object and close it.

A `PreparedStatement` object (subclassed from JDBC `Statement`) allows you to execute the same SQL over and over again with different values. `PreparedStatement` objects use the JDBC `"?"` syntax.

```
String inssql = "insert into emp(empid, name, dept) values (?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(inssql);

for (int i = 0; i < 100; i++) {
  pstmt.setInt(1, i);
  pstmt.setString(2, "Person " + i);
  pstmt.setInt(3, i);
  pstmt.execute():
}
pstmt.close();
```

We also use a `PreparedStatement` object to update records. In the following code snippet, we add the value of the counter `"i"` to the current value of the `"dept"` field.

```
String updsql = "update emp set dept = dept + ? where empid = ?";
PreparedStatement pstmt2 = conn.prepareStatement(updsql);

for (int i = 0; i < 100; i++) {
  pstmt2.setInt(1, i);
  pstmt2.setInt(2, i);
  pstmt2.execute();
}
pstmt2.close();
```

Finally, we use a `PreparedStatement` object to delete the records that we added and then updated, as in the following snippet:

```
String delsql = "delete from emp where empid = ?";
PreparedStatement pstmt3 = conn.prepareStatement(delsql);

for (int i = 0; i < 100; i++) {
  pstmt3.setInt(1, i);
  pstmt3.execute();
}
pstmt3.close();
```

Creating and Using Stored Procedures and Functions

You can use a jdbcKona driver to create, use, and drop stored procedures and functions. First, we execute a series of `Statement` objects to drop a set of stored procedures and functions from the database, as in the following code snippet:

```
Statement stmt = conn.createStatement();
try {stmt.execute("drop procedure proc_squareInt");}
catch (SQLException e) {;}
try {stmt.execute("drop procedure func_squareInt");}
catch (SQLException e) {;}
try {stmt.execute("drop procedure proc_getresults");}
catch (SQLException e) {;}
stmt.close();
```

We use a JDBC `Statement` object to create a stored procedure or function, and then we use a JDBC `CallableStatement` object (subclassed from the `Statement` object) with the JDBC `"?"` syntax to set `IN` and `OUT` parameters. For information about doing this with the jdbcKona/Oracle driver, see Chapter 2, “Using the jdbcKona/Oracle Driver.”

The first two code snippets that follow use the jdbcKona/Oracle driver. Note that Oracle does not natively support binding to `"?"` values in an SQL statement. Instead, it uses `":1"`, `":2"`, and so forth. You can use either syntax in your SQL with the jdbcKona/Oracle driver.

Stored procedure input parameters are mapped to JDBC `IN` parameters, using the `CallableStatement.set`*xxx* methods, such as `setInt()`, and the `"?"` syntax of the JDBC `PreparedStatement` object. Stored procedure output parameters are mapped to JDBC `OUT` parameters, using the `CallableStatement.registerOutParameter`

methods and the "?" syntax of the JDBC `PreparedStatement` object. A parameter may be both `IN` and `OUT`, which requires both a `set`*xxx*() and a `registerOutParameter()` invocation to be made on the same parameter number.

In the following code snippet, we use a JDBC `Statement` object to create an Oracle stored procedure; then we execute the stored procedure with a `CallableStatement` object. We use the `registerOutParameter` method to set an output parameter for the squared value.

```
Statement stmt1 = conn.createStatement();
stmt1.execute("CREATE OR REPLACE PROCEDURE proc_squareInt " +
              "(field1 IN OUT INTEGER, field2 OUT INTEGER) IS " +
              "BEGIN field2 := field1 * field1; field1 := " +
              "field1 * field1; END proc_squareInt;");
stmt1.close();

// Native Oracle SQL is commented out here
// String sql = "BEGIN proc_squareInt(?, ?); END;";

// This is the correct syntax as specified by JDBC
String sql = "{call proc_squareInt(?, ?)}";
CallableStatement cstmt1 = conn.prepareCall(sql);

// Register out parameters
cstmt1.registerOutParameter(2, java.sql.Types.INTEGER);
for (int i = 0; i < 5; i++) {
  cstmt1.setInt(1, i);
  cstmt1.execute();
  System.out.println(i + " " + cstmt1.getInt(1) +
" " + cstmt1.getInt(2));
}
cstmt1.close();
```

In the following code snippet, we use similar code to create and execute a stored function that squares an integer.

```
Statement stmt2 = conn.createStatement();
stmt2.execute("CREATE OR REPLACE FUNCTION func_squareInt " +
              "(field1 IN INTEGER) RETURN INTEGER IS " +
              "BEGIN return field1 * field1; " +
              "END func_squareInt;");
stmt2.close();

// Native Oracle SQL is commented out here
// sql = "BEGIN ? := func_squareInt(?); END;";

// This is the correct syntax specified by JDBC
sql = "{ ? = call func_squareInt(?)}";
```

```
CallableStatement cstmt2 = conn.prepareCall(sql);

cstmt2.registerOutParameter(1, Types.INTEGER);
for (int i = 0; i < 5; i++) {
  cstmt2.setInt(2, i);
  cstmt2.execute();
  System.out.println(i + " " + cstmt2.getInt(1) +
                     " " + cstmt2.getInt(2));
}
cstmt2.close();
```

Disconnecting and Closing Objects

Close `Statement`, `ResultSet`, `Connection`, and other such objects with their `close` methods after you have finished using them. Closing these objects releases resources on the remote DBMS and within your application. When you use one object to construct another, close the objects in the reverse order in which they were created. For example:

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select * from empno");

     (process the ResultSet)

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

Always close the `java.sql.Connection` as well, usually as one of the last steps in your program. Every connection should be closed, even if a login fails. An Oracle connection will cause a system failure (such as a segment violation) when the finalizer thread attempts to close a connection that you have inadvertently left open. If you do not close connections to log out of the database, you may also exceed the maximum number of database logins. Once a connection is closed, all of the objects created in its context become unusable.

There are occasions on which you will want to invoke the `commit` method to commit changes you have made to the database before you close the connection.

When `autocommit` is set to true (the default JDBC transaction mode), each SQL statement is its own transaction. After we created the `Connection` object for these examples, however, we set `autocommit` to false; in this mode, the `Connection` object always has an implicit transaction associated with it, and any invocation to the

rollback or commit methods will end the current transaction and start a new one. Invoking commit() before close() ensures that all of the transactions are completed before closing the connection.

Just as you close Statement, PreparedStatement, and CallableStatement objects when you have finished working with them, always invoke the close method on the connection as final cleanup in your application, in a try {} block, and catch exceptions and deal with them appropriately. The final two lines of the example include an invocation to commit() and then close() to close the connection, as in the following snippet:

```
conn.commit();
conn.close();
```

Code Example

The following is a sample implementation to give you an overall idea of the structure for a WLE Java application that uses a jdbcKona driver to access a DBMS. The code example shown here includes retrieving data, displaying metadata, inserting, deleting, and updating data, and stored procedures and functions. Note the explicit invocations to close() for each JDBC-related object, and note also that we close the connection itself in a finally {} block, with the invocation to close() wrapped in a try {} block.

```
import java.sql.*;
import java.util.Properties;
import weblogic.common.*;

public class test {
  static int i;
  Statement stmt = null;

  public static void main(String[] argv) {
    try {
      Properties props = new Properties();
      props.put("user",              "scott");
      props.put("password",          "tiger");
      props.put("server",            "DEMO");

      Class.forName("weblogic.jdbc.oci.Driver").newInstance();
      Connection conn =
        DriverManager.getConnection("jdbc:weblogic:oracle",
                                    props);
```

```
}
catch (Exception e)
e.printStackTrace();
}

try {
  // This will improve performance in Oracle
  // You'll need an explicit commit() call later
  conn.setAutoCommit(false);

  stmt = conn.createStatement();
  stmt.execute("select * from emp");
  ResultSet rs = stmt.getResultSet();

  while (rs.next()) {
    System.out.println(rs.getString("empid") + " - " +
                       rs.getString("name")  + " - " +
                       rs.getString("dept"));
  }

  ResultSetMetaData md = rs.getMetaData();

  System.out.println("Number of Columns: " + md.getColumnCount());
  for (i = 1; i <= md.getColumnCount(); i++) {
    System.out.println("Column Name: "     + md.getColumnName(i));
    System.out.println("Nullable: "        + md.isNullable(i));
    System.out.println("Precision: "       + md.getPrecision(i));
    System.out.println("Scale: "           + md.getScale(i));
    System.out.println("Size: "            + md.getColumnDisplaySize(i));
    System.out.println("Column Type: "     + md.getColumnType(i));
    System.out.println("Column Type Name: "+ md.getColumnTypeName(i));
    System.out.println("");
  }
  rs.close();
  stmt.close();

  Statement stmtdrop = conn.createStatement();
  try {stmtdrop.execute("drop procedure proc_squareInt");}
  catch (SQLException e) {;}
  try {stmtdrop.execute("drop procedure func_squareInt"); }
  catch (SQLException e) {;}
  try {stmtdrop.execute("drop procedure proc_getresults"); }
  catch (SQLException e) {;}
  stmtdrop.close();

  // Create a stored procedure
  Statement stmt1 = conn.createStatement();
  stmt1.execute("CREATE OR REPLACE PROCEDURE proc_squareInt " +
                "(field1 IN OUT INTEGER, " +
```

```
              "field2 OUT INTEGER) IS " +
              "BEGIN field2 := field1 * field1; " +
              "field1 := field1 * field1; " +
              "END proc_squareInt;");
stmt1.close();


CallableStatement cstmt1 =
  conn.prepareCall("BEGIN proc_squareInt(?, ?); END;");
cstmt1.registerOutParameter(2, Types.INTEGER);
for (i = 0; i < 100; i++) {
  cstmt1.setInt(1, i);
  cstmt1.execute();
  System.out.println(i + " " + cstmt1.getInt(1) +
                     " " + cstmt1.getInt(2));
}
cstmt1.close();

// Create a stored function
Statement stmt2 = conn.createStatement();
stmt2.execute("CREATE OR REPLACE FUNCTION func_squareInt " +
              "(field1 IN INTEGER) RETURN INTEGER IS " +
              "BEGIN return field1 * field1; END func_squareInt;");
stmt2.close();

CallableStatement cstmt2 =
  conn.prepareCall("BEGIN ? := func_squareInt(?); END;");
cstmt2.registerOutParameter(1, Types.INTEGER);
for (i = 0; i < 100; i++) {
  cstmt2.setInt(2, i);
  cstmt2.execute();
  System.out.println(i + " " + cstmt2.getInt(1) +
                     " " + cstmt2.getInt(2));
}
cstmt2.close();

// Insert 100 records
System.out.println("Inserting 100 records...");
String inssql = "insert into emp(empid, name, dept) values (?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(inssql);

for (i = 0; i < 100; i++) {
  pstmt.setInt(1, i);
  pstmt.setString(2, "Person " + i);
  pstmt.setInt(3, i);
  pstmt.execute();
}
pstmt.close();
```

```
      // Update 100 records
      System.out.println("Updating 100 records...");
      String updsql = "update emp set dept = dept + ? where empid = ?";
      PreparedStatement pstmt2 = conn.prepareStatement(updsql);

      for (i = 0; i < 100; i++) {
        pstmt2.setInt(1, i);
        pstmt2.setInt(2, i);
        pstmt2.execute();
      }
      pstmt2.close();

      // Delete 100 records
      System.out.println("Deleting 100 records...");
      String delsql = "delete from emp where empid = ?";
      PreparedStatement pstmt3 = conn.prepareStatement(delsql);

      for (i = 0; i < 100; i++) {
        pstmt3.setInt(1, i);
        pstmt3.execute();
      }
      pstmt3.close();

      conn.commit();
      }
      catch (Exception e) {
        // Deal with failures appropriately
      }
      finally {
        try {conn.close();}
        catch (Exception e) {
          // Catch and deal with exception
        }
      }
  }
}
```

2 Using the jdbcKona/Oracle Driver

This chapter provides general guidelines for using the jdbcKona/Oracle Type 2 driver. For general notes about and an example of using the jdbcKona drivers, see Chapter 1, "Using the jdbcKona Drivers."

Data Type Mapping

Mapping of types between Oracle and the jdbcKona/Oracle driver are provided in the following table.

Oracle	jdbcKona/Oracle driver
Varchar	String
Number	Tinyint
Number	Smallint
Number	Integer
Number	Long
Number	Float
Number	Numeric
Number	Double

Oracle	jdbcKona/Oracle driver
Long	Longvarchar
RowID	String
Date	Timestamp
Raw	(var)Binary
Long raw	Longvarbinary
Char	(var)Char
Boolean*	Number OR Varchar
MLS label	String

Note that when the `PreparedStatement.setBoolean` method is invoked, this method converts a `VARCHAR` type to `"1"` or `"0"` (`string`), and it converts a `NUMBER` type to 1 or 0 (number).

Note that the `PreparedStatement.setBoolean` method converts a `VARCHAR` type to `"1"` or `"0"` (`string`), and it converts a `NUMBER` type to 1 or 0 (number).

Connecting the jdbcKona/Oracle Driver to an Oracle DBMS

In general, to make a DBMS connection, you perform the following steps:

1. Load the proper jdbcKona driver.

   The most efficient way to do this is to invoke the `Class.forName().newInstance()` method with the name of the driver class, which properly loads and registers the jdbcKona driver, as in the following example:

   ```
   Class.forName("weblogic.jdbc.oci.Driver").newInstance();
   ```

2. Request a JDBC connection by invoking the `DriverManager.getConnection` method, which takes as its parameters the URL of the driver and other information about the connection.

Note that both steps describe the jdbcKona driver, but in a different format. The full package name is period-separated, and the URL is colon-separated. The URL must include at least `weblogic:jdbc:oracle`, and may include other information, including server name and database name.

There are several variations on this basic pattern, which are described here for Oracle. For a full code example, see "Implementing a WLE Java Application Using the jdbcKona Drivers" on page 1-6.

Method 1

The simplest way to connect to an Oracle DBMS is by passing the URL of the driver that includes the name of the server, along with a username and a password, as arguments to the `DriverManager.getConnection` method, as in the following jdbcKona/Oracle example:

```
Class.forName("weblogic.jdbc.oci.Driver").newInstance();
Connection conn =
  DriverManager.getConnection("jdbc:weblogic:oracle:DEMO",
                              "scott",
                               "tiger");
```

In the preceding example, `DEMO` is the V2 alias of an Oracle database. Note that invoking the `Class.forName().newInstance()` method properly loads and registers the driver.

Method 2

You can also pass a `java.util.Properties` object with parameters for connection as an argument to the `DriverManager.getConnection` method. The following example shows how to connect to the `DEMO` database:

```
Properties props = new Properties();
props.put("user",         "scott");
props.put("password",     "tiger");
props.put("server",       "DEMO");
```

```
Class.forName("weblogic.jdbc.oci.Driver").newInstance();
Connection conn =
  DriverManager.getConnection("jdbc:weblogic:oracle",
                              props);
```

If you do not supply a server name (`DEMO` in the preceding example), the system looks for an environment variable (`ORACLE_SID` in the case of Oracle). You can also add the server name to the URL, using the following format:

```
"jdbc:weblogic:oracle:DEMO"
```

When you use the preceding format, you do not need to provide a `"server"` property.

Other Properties You Can Set for the jdbcKona/Oracle Driver

There are other properties that you can set for the jdbcKona/Oracle driver, which are covered later in this document. The jdbcKona/Oracle driver also allows setting a property -- `allowMixedCaseMetaData` -- to the boolean `true`. This property sets up the connection to use mixed case letters in invocation to `DatabaseMetaData` methods. Otherwise, Oracle defaults to uppercase letters for database metadata.

The following is an example of setting up the properties to include this feature:

```
Properties props = new Properties();
props.put("user",                   "scott");
props.put("password",               "tiger");
props.put("server",                 "DEMO");
props.put("allowMixedCaseMetaData", "true");

Connection conn =
    DriverManager.getConnection("jdbc:weblogic:oracle",
                                                        props);
```

If you do not set this property, the jdbcKona/Oracle driver defaults to the Oracle default, which uses uppercase letters for database metadata.

General Notes

Always invoke the `Connection.close` method to close the connection when you have finished working with it. Closing objects releases resources on the remote DBMS and within your application, as well as being good programming practice. Other jdbcKona objects on which you should invoke the `close` method after final use include:

- `Statement` (`PreparedStatement`, `CallableStatement`)
- `ResultSet`

Waiting for Oracle DBMS Resources

The jdbcKona/Oracle driver supports the Oracle `oopt()` C API, which allows a client to wait until resources become available. The Oracle C function sets options in cases where requested resources are not available; for example, whether to wait for locks.

You can set whether a client waits for DBMS resources, or receives an immediate exception. The following is an example:

```
java.util.Properties props = new java.util.Properties();
props.put("user",     "scott");
props.put("password", "tiger");
props.put("server",   "goldengate");

Class.forName("weblogic.jdbc.oci.Driver").newInstance();

// You must cast the Connection as a weblogic.jdbc.oci.Connection
// to take advantage of this extension
Connection conn =
  (weblogic.jdbc.oci.Connection)
     DriverManager.getConnection("jdbc:weblogic:oracle", props);

// After constructing the Connection object, immediately call
// the waitOnResources method
conn.waitOnResources(true);
```

Note that use of this method can cause several error return codes while waiting for internal resources that are locked for short durations.

To take advantage of this feature, you must first cast your `Connection` object as a `weblogic.jdbc.oci.Connection` object, and then invoke the `waitOnResources` method.

This functionality is described in section 4-97 of *The OCI Functions for C*, published by Oracle Corporation.

Autocommit

The default transaction mode for JDBC assumes `autocommit` to be true. You will improve the performance of your programs by setting `autocommit` to false after creating a `Connection` object with the following statement:

```
Connection.setAutoCommit(false);
```

Using Oracle Blobs

The jdbcKona/Oracle driver supports two new properties to support Oracle Blob chunking:

- `weblogic.oci.insertBlobChunkSize`

  This property affects the buffer size of input streams bound to a `PreparedStatement` object. Blob chunking requires an Oracle 7.3.x or higher Oracle Server; to use this property, you must be connected to an Oracle Server that supports this feature.

  Set this property to a positive integer to insert Blobs into an Oracle DBMS with the Blob chunking feature. By default, this property is set to 0 (zero), which means that BLOB chunking is turned off.

- `weblogic.oci.selectBlobChunkSize`

  This property sets the size of output streams associated with a JDBC `ResultSet` object. The mechanism for piecewise selects does not have the same use restrictions as that for Blob inserts, so this property is set to 65534 by default. It is not necessary to turn this property off.

Set this property to the size of the desired output stream, in bytes.

Support for Oracle Array Fetches

With WLE Java, the jdbcKona/Oracle driver supports Oracle array fetches. With this feature support, invoking the `ResultSet.next` method the first time gets an array of rows and stores it in memory, rather than retrieving a single row. Each subsequent invocation of the `next` method reads a row from the rows in memory until they are exhausted, and only then does the `next` method go back to the database.

You set a property (`java.util.Property`) to control the size of the array fetch. The property is `weblogic.oci.cacheRows`; it is set by default to 100. The following is an example of setting this property to 300, which means that invocations to the `next` method hit the database only once for each 300 rows retrieved by the client:

```
Properties props = new Properties();
props.put("user",                  "scott");
props.put("password",              "tiger");
props.put("server",                "DEMO");
props.put("weblogic.oci.cacheRows", "300");

Class.forName("weblogic.jdbc.oci.Driver").newInstance();
Connection conn =
  DriverManager.getConnection("jdbc:weblogic:oracle",
                              props);
```

You can improve client performance and lower the load on the database server by taking advantage of this JDBC extension. Caching rows in the client, however, requires client resources. Tune your application for the best balance between performance and client resources, depending upon your network configuration and your application.

If any columns in a `SELECT` statement are of type `LONG`, the cache size will be temporarily reset to 1 (one) for the `ResultSet` object associated with that select statement.

Using Stored Procedures

The following sections describe how to use stored procedures:

- Syntax for Stored Procedures in the jdbcKona/Oracle Driver
- Binding a Parameter to an Oracle Cursor
- Using CallableStatement

Syntax for Stored Procedures in the jdbcKona/Oracle Driver

The syntax for stored procedures in Oracle was altered in the jdbcKona/Oracle driver examples to match the JDBC specification. (All of the examples also show native Oracle SQL, commented out, just above the correct usage; the native Oracle syntax works as it did in the past.) You can read more about stored procedures for the jdbcKona drivers in Chapter 1, "Using the jdbcKona Drivers."

Note that Oracle does not natively support binding to `"?"` values in an SQL statement. Instead it uses ":1", ":2", and so forth. We allow you to use either in your SQL with the jdbcKona/Oracle driver.

Binding a Parameter to an Oracle Cursor

BEA Systems, Inc. has created an extension to JDBC (`weblogic.jdbc.oci.CallableStatement`) that allows you to bind a parameter for a stored procedure to an Oracle cursor. You can create a JDBC `ResultSet` object with the results of the stored procedure. This allows you to return multiple `ResultSet` objects in an organized way. The `ResultSet` objects are determined at run time in the stored procedure. An example procedure follows.

First, define the stored procedures, as follows:

```
create or replace package
curs_types as
```

```
type EmpCurType is REF CURSOR RETURN emp%ROWTYPE;
end curs_types;
/

create or replace procedure
single_cursor(curs1 IN OUT curs_types.EmpCurType,
ctype in number) AS BEGIN
   if ctype = 1 then
     OPEN curs1 FOR SELECT * FROM emp;
   elsif ctype = 2 then
     OPEN curs1 FOR SELECT * FROM emp where sal > 2000;
   elsif ctype = 3 then
     OPEN curs1 FOR SELECT * FROM emp where deptno = 20;
   end if;
END single_cursor;
/

create or replace procedure
multi_cursor(curs1 IN OUT curs_types.EmpCurType,
             curs2 IN OUT curs_types.EmpCurType,
             curs3 IN OUT curs_types.EmpCurType) AS
BEGIN
    OPEN curs1 FOR SELECT * FROM emp;
    OPEN curs2 FOR SELECT * FROM emp where sal > 2000;
    OPEN curs3 FOR SELECT * FROM emp where deptno = 20;
END multi_cursor;
/
```

In your Java code, construct `CallableStatement` objects with the stored procedures and register the output parameter as data type `java.sql.Types.OTHER`. When you retrieve the data into a `ResultSet` object, use the output parameter index as an argument for the `getResultSet` method. For example:

```
weblogic.jdbc.oci.CallableStatement cstmt =
  (weblogic.jdbc.oci.CallableStatement)conn.prepareCall(
                   "BEGIN OPEN ? " +
                   "FOR select * from emp; END;");
cstmt.registerOutParameter(1, java.sql.Types.OTHER);

cstmt.execute();
ResultSet rs = cstmt.getResultSet(1);
printResultSet(rs);
rs.close();
cstmt.close();

weblogic.jdbc.oci.CallableStatement cstmt2 =
  (weblogic.jdbc.oci.CallableStatement)conn.prepareCall(
                    "BEGIN single_cursor(?, ?); END;");
```

```
cstmt2.registerOutParameter(1, java.sql.Types.OTHER);

cstmt2.setInt(2, 1);
cstmt2.execute();
rs = cstmt2.getResultSet(1);
printResultSet(rs);

cstmt2.setInt(2, 2);
cstmt2.execute();
rs = cstmt2.getResultSet(1);
printResultSet(rs);

cstmt2.setInt(2, 3);
cstmt2.execute();
rs = cstmt2.getResultSet(1);
printResultSet(rs);

cstmt2.close();

weblogic.jdbc.oci.CallableStatement cstmt3 =
  (weblogic.jdbc.oci.CallableStatement)conn.prepareCall(
                    "BEGIN multi_cursor(?, ?, ?); END;");
cstmt3.registerOutParameter(1, java.sql.Types.OTHER);
cstmt3.registerOutParameter(2, java.sql.Types.OTHER);
cstmt3.registerOutParameter(3, java.sql.Types.OTHER);

cstmt3.execute();

ResultSet rs1 = cstmt3.getResultSet(1);
ResultSet rs2 = cstmt3.getResultSet(2);
ResultSet rs3 = cstmt3.getResultSet(3);
```

Note that the default size of an Oracle-stored procedure string is 256K.

Using CallableStatement

The default length of a string bound to an `OUTPUT` parameter of a `CallableStatement` object is 128 characters. If the value you assign to the bound parameter exceeds that length, you get the following error:

```
ORA-6502: value or numeric error
```

You can adjust the length of the value of the bound parameter by passing an explicit length with the scale argument to the `CallableStatement.registerOutputParameter` method. The following is a code example that binds a `VARCHAR` that will never be larger than 256 characters:

```
CallableStatement cstmt =
  conn.prepareCall("BEGIN testproc(?); END;");

cstmt.registerOutputParameter(1, Types.VARCHAR, 256);
cstmt.execute();
System.out.println(cstmt.getString());
cstmt.close();
```

DatabaseMetaData Methods

DatabaseMetaData is implemented in its entirety in the jdbcKona/Oracle driver. There are some variations that are specific to Oracle, which are as follows:

- As a general rule, the String catalog argument is ignored in all DatabaseMetaData methods.
- In the `DatabaseMetaData.getProcedureColumns` method:
  - The String `catalog` argument is ignored.
  - The String `schemaPattern` argument accepts only exact matches (no pattern matching).
  - The String `procedureNamePattern` argument accepts only exact matches (no pattern matching).
  - The String `columnNamePattern` argument is ignored.

jdbcKona/Oracle and the Oracle NUMBER Column

Oracle provides a column type called `NUMBER`, which can be optionally specified with a precision and a scale, in the forms `NUMBER(P)` and `NUMBER(P,S)`. Even in the simple, unqualified `NUMBER` form, this column can hold all number types from small integer values to very large floating point numbers, with high precision.

The jdbcKona/Oracle driver reliably converts the values in a column to the Java type requested when a WLE Java application asks for a value from such a column. Of course, if a value of `123.456` is asked for with `getInt()`, the value will be rounded.

The method `getObject`, however, poses a little more complexity. The jdbcKona/Oracle driver guarantees to return a Java object that will represent any value in a `NUMBER` column with no loss in precision. This means that a value of `1` can be returned in an `Integer`, but a value like `123434567890.123456789` can only be returned in a `BigDecimal`.

There is no metadata from Oracle to report the maximum precision of the values in the column, so the jdbcKona/Oracle driver must decide what sort of object to return based on each value. This means that one `ResultSet` object may return multiple Java types from the `getObject` method for a given `NUMBER` column. A table full of integer values may all be returned as `Integer` from the `getObject` method, whereas a table of floating point measurements may be returned primarily as `Double`, with some `Integer` if any value happens to be something like `123.00`. Oracle does not provide any information to distinguish between a `NUMBER` value of `1` and a `NUMBER` of `1.0000000000`.

There is more reliable behavior with qualified `NUMBER` columns; that is, those defined with a specific precision. Oracle's metadata provides these parameters to the driver so the jdbcKona/Oracle driver always returns a Java object appropriate for the given precision and scale, regardless of the values shown in the following table. The following table shows the Java objects returned for each qualified `NUMBER` column.

Column Definition	Returned by `getObject()`
`NUMBER(P <= 9)`	`Integer`
`NUMBER(P <= 18)`	`Long`
`NUMBER(P >= 19)`	`BigDecimal`
`NUMBER(P <=16, S > 0)`	`Double`
`NUMBER(P >= 17, S > 0)`	`BigDecimal`

3 Using the jdbcKona/ MSSQLServer4 Driver

The jdbcKona/MSSQLServer4 is a Type 4, pure-Java, two-tier driver. It requires no client-side libraries because it connects to the database via a proprietary vendor protocol at the wire-format level. Unlike Type 2 JDBC drivers, Type 4 drivers make no native calls, so they can be used in Java applets.

A Type 4 JDBC driver is similar to a Type 2 driver in many other ways. Type 2 and Type 4 drivers are two-tier drivers: each client requires an in-memory copy of the driver to support its connection to the database.

The API reference for JDBC, of which this driver is a fully compliant implementation, is available online in several formats at the Sun Microsystems, Inc. Web site.

Connecting to an SQL Server with the jdbcKona/MSSQLServer4 Driver

To connect to an SQL Server database in a WLE Java server application, perform the following steps:

1. Load the jdbcKona/MSSQLServer4 JDBC driver.
2. Request a JDBC connection.

An efficient way to load the JDBC driver is to invoke the `Class.forName().newInstance()` method, specifying the name of the driver class, as in the following example:

```
Class.forName("weblogic.jdbc.mssqlserver4.Driver").newInstance();
```

After loading the JDBC driver, request a JDBC connection by invoking the `DriverManager.getConnection` method. You invoke this method with a connection URL, which, again, specifies the JDBC driver and other connection information.

There are several ways to specify connection information in the `DriverManager.getConnection` method. The following sections describe three methods.

Method 1

The simplest method is to use a connection URL that includes the database name, host name and port number of the database server, and two additional arguments to specify the database user name and password, as in the following example:

```
Class.forName("weblogic.jdbc.mssqlserver4.Driver").newInstance();
Connection conn =
  DriverManager.getConnection(
              "jdbc:weblogic:mssqlserver4:database@host:port",
              "sa",               // database user name
               "");                // password for database user
```

In this example, `host` is the name or IP number of the computer running SQL Server, and `port` is the port number the SQL Server is listening on.

Method 2

You can set connection information in a `Properties` object and pass this information to the `DriverManager.getConnection` method. The following example specifies the `server`, `user`, and `password` in a `Properties` object:

```
Properties props = new Properties();
props.put("server",     "pubs@myhost:1433");
props.put("user",       "sa");
```

```
props.put("password",   "");

Class.forName("weblogic.jdbc.mssqlserver4.Driver").newInstance();
Connection conn =
   DriverManager.getConnection("jdbc:weblogic:mssqlserver4",
                               props);
```

Method 3

You can add connection options to the end of the connection URL, instead of creating a `Properties` object. Separate the URL from the connection options with a question mark (?), and separate options with ampersands (&), as in the following example:

```
Class.forName("weblogic.jdbc.mssqlserver4.Driver).newInstance();
DriverManager.getConnection(
  "jdbc:weblogic:mssqlserver4:database@myhost:myport?user=
   sa&password=");
```

You can use the `Driver.getPropertyInfo` method to find out more about URL options at run time.

Setting Properties for Microsoft SQL Server 7

The jdbcKona/MSSQLServer4 driver recognizes SQL Server 7 automatically. You must set the `sql7` property in the connection URL or in a `Properties` object to `true` to connect to SQL Server 7. For example, the connection URL for an SQL Server 7 connection would be similar to the following:

```
"jdbc:weblogic:mssqlserver4:pubs@myhost:myport?sql7=true"
```

Using the jdbcKona/MSSQLServer4 Driver in Java Development Environments

The jdbcKona/MSSQLServer4 driver has been used successfully in the Java SDK 1.2 for Sun and Windows NT development environment.

JDBC Extensions and Limitations

This section describes the following JDBC extensions and limitations:

- Support for JDBC Extended SQL
- cursorName Method Not Supported
- java.sql.TimeStamp Limitations
- Querying Metadata
- Changing autoCommit Mode
- Statement.executeWriteText() Methods Not Supported
- Sharing a Connection Object in Multithreaded Applications
- EXECUTE Keyword with Stored Procedures

Support for JDBC Extended SQL

The Sun Microsystems, Inc. JDBC specification includes a feature called SQL Extensions, or SQL Escape Syntax. The jdbcKona/MSSQLServer4 driver supports Extended SQL. For information about this feature, see Chapter 1, "Using the jdbcKona Drivers."

cursorName Method Not Supported

The `cursorName` method is not supported, because its definition does not apply to the Microsoft SQL Server.

java.sql.TimeStamp Limitations

The `java.sql.TimeStamp` class in the Java 2 software is limited to dates after 1970. Earlier dates raise an exception. However, if you retrieve dates using the `getString` method, the jdbcKona/MSSQLServer4 driver uses its own date class to overcome the limitation.

Querying Metadata

You can only query metadata for the current database. The metadata methods call the corresponding SQL Server stored procedures, which operate only on the current database. For example, if the current database is master, only the metadata relative to master is available on the connection.

Changing autoCommit Mode

Invoke the `Connection.setAutoCommit` method with a `true` or `false` argument to enable or disable chained transaction mode. When `autoCommit` is `true`, the jdbcKona/MSSQLServer4 driver begins a transaction whenever the previous transaction is committed or rolled back. You must explicitly end your transactions with a `commit` or a `rollback`. If there is an uncommitted transaction when you invoke the `setAutoCommit` method, the driver rolls back the transaction before changing the mode. Be sure to commit any changes before you invoke this method.

Statement.executeWriteText() Methods Not Supported

The jdbcKona Type 2 drivers support an extension that allows you to write text and image data into a row as part of an `SQL INSERT` or `UPDATE` statement without using a text pointer. This extension, `Statement.executeWriteText()`, requires the DB-Library native libraries, and thus is not supported by the jdbcKona/MSSQLServer4 JDBC driver.

To read and write text and image data with streams, you can use the `prepareStatement.setAsciiStream()`, `prepareStatement.setBinaryStream()`, `ResultSet.getAsciiStream()`, and `ResultSet.getBinaryStream()` JDBC methods.

Sharing a Connection Object in Multithreaded Applications

The jdbcKona/MSSQLServer4 driver allows you to write multithreaded applications in which multiple threads can share a single `Connection` object. Each thread can have an active `Statement` object. However, if you invoke the `Statement.cancel` method on one thread, SQL Server may cancel a `Statement` on a different thread. The `Statement` object that is cancelled depends on timing issues in the SQL Server. To avoid this unexpected behavior, we recommend that you get a separate `Connection` object for each thread.

EXECUTE Keyword with Stored Procedures

A Transact-SQL feature allows you to omit the `EXECUTE` keyword on a stored procedure when the stored procedure is the first command in the batch. However, when a stored procedure has parameters, the jdbcKona/MSSQLServer4 driver adds variable declarations (specific to the JDBC implementation) before the procedure call. Because of this, it is good practice to use the `EXECUTE` keyword for stored procedures. Note that the JDBC extended SQL stored procedure syntax, which does not include the `EXECUTE` keyword, is not affected by this issue.

4 Extensions to the JDBC API

This chapter describes the following jdbcKona extensions to the JDBC API:

- `Class weblogic.jdbc.oci.CallableStatement`
- `Class weblogic.jdbc.oci.Connection`
- `Class weblogic.jdbc.oci.Statement`

For complete details on the JDBC API, refer to the following Web site:

`http://www.weblogic.com/docs/classdocs/packages.html#jdbc`

Class CallableStatement

Class `weblogic.jdbc.oci.CallableStatement` contains jdbcKona extensions to JDBC to support the use of cursors as parameters in `CallableStatement` objects.

The `CallableStatement` class:

- Extends the `PreparedStatement` class
- Implements the `CallableStatement` interface
- Has the following inheritance hierarchy:

```
java.lang.Object
   |
   +----weblogic.jdbc.oci.Statement
           |
           +----weblogic.jdbc.oci.PreparedStatement
                   |
                    +----weblogic.jdbc.oci.CallableStatement
```

- Has the `getResultSet` method

weblogic.jdbc.oci.CallableStatement.getResultSet

Synopsis Returns a `ResultSet` object from a stored procedure where the specified parameter has been bound to an Oracle cursor. Register the output parameter with the `registerOutputParameter` method, using `java.sql.Types.OTHER` as the data type.

Java Mapping

```
public ResultSet getResultSet(int parameterIndex) throws
                                                    SQLException
```

Parameters

`parameterIndex`
This parameter is an index into the set of parameters for the stored procedure.

Throws

`SQLException`
This exception is thrown if the operation cannot be completed.

Class Connection

This section describes only the jdbcKona extension to JDBC that accesses the Oracle OCI C Function `oopt()`. Other information about this class is in the description for class `java.sql.Connection`. A `Connection` object is usually constructed as a `java.sql.Connection` class. To use this extension to JDBC, you must explicitly cast your `Connection` object as a `weblogic.jdbc.oci.Connection` class.

The public Connection class:

- Extends the `Object` class
- Implements the `Connection` interface
- Has the following inheritance hierarchy:

```
java.lang.Object
   |
   +----weblogic.jdbc.oci.Connection
```

- Has the `waitOnResources` method

weblogic.jdbc.oci.Connection.waitOnResources

Synopsis

Use this method to access the Oracle `oopt()` function for C (see section 4-97 of The OCI Functions for C). The Oracle C function sets options in cases where requested resources are not available; for example, whether to wait for locks.

When the argument to this method is true, this jdbcKona extension to JDBC sets this option so that your program will receive an error return code whenever a resource is requested but is unavailable. Use of this method can cause several error return codes while waiting for internal resources that are locked for short durations.

Java Mapping

```
public void waitOnResources(boolean val)
```

Parameters

`val`

This parameter is set to true if the connection should wait on resources.

Class weblogic.jdbc.oci.Statement

This class contains jdbcKona extensions to JDBC to support parsing of SQL statements and adjusting of the fetch size. Only those methods are documented here.

The `weblogic.jdbc.oci.Statement` class:

- Extends the Object base class
- Has the following inheritance hierarchy:

```
java.lang.Object
   |
   +----weblogic.jdbc.oci.Statement
```

- Has the following methods:
  - `fetchsize`
  - `parse`

weblogic.jdbc.oci.Statement.fetchsize

Synopsis

Allows tuning of the size of prefetch array used for Oracle row results. Oracle provides the means to do data prefetch in batches, which decreases network traffic and latency for row requests.

The default batch size is 100. Memory for 100 rows is allocated in the native stack for every query. For queries that need fewer rows, this size can be adjusted appropriately. This saves on the swappable image size of the application and will benefit performance if only as many rows as needed are fetched.

Java Mapping

```
public void fetchSize(int size)
```

Parameters

`size`

This parameter specifies the number of rows to be prefetched.

weblogic.jdbc.oci.Statement.parse

Synopsis

Allows tuning of the size of prefetch array used for Oracle row results. Oracle provides the means to do data prefetch in batches, which decreases network traffic and latency for row requests.

The default batch size is 100. Memory for 100 rows is allocated in the native stack for every query. For queries that need fewer rows, this size can be adjusted appropriately. This saves on the swappable image size of the application and will benefit performance if only as many rows as needed are fetched.

Java Mapping

```
public int parse(String sql) throws SQLException
```

Parameters

`sql`

This parameter is the SQL statement to be verified.

Throws

SQLException

This exception is thrown if the operation cannot be completed.

Index

R

S

W