Oracle8™ JDBC Drivers

Oracle's JDBC drivers, Release 8.0.4.0.0, implement the standard JDBC (Java Database Connectivity) interface as defined by JavaSoft. These drivers comply with JDBC version 1.22. In addition to the standard JDBC API, Oracle drivers have extensions to properties, types, and performance.

This document describes the installation and use of the drivers, as well as Oracle's extensions.

 JDBC is based on the X/Open SQL Call Level Interface, and complies with the SQL92 Entry Level standard.

A description of JDBC can be found at http://www.javasoft.com.

Contents

- Copyright Information.

- Introduction. JDBC versions and how they differ.

- Getting Started. How to obtain and install JDBC.

- Using Oracle's JDBC Drivers.

- Features of All Oracle JDBC Drivers.

- Features of Specific Oracle JDBC Drivers. Thin versus OCI drivers.

- Extensions to JDBC. Features added to the drivers by Oracle.

- Features Not Implemented.

- Applets. How to make applets with the JDBC Thin driver.

- Streams Tutorial. Short code samples and explanations.

- Common Problems and Frequently Asked Questions.

Copyright Information

Introduction

Oracle supports JDBC 1.22, not JDBC 1.01.

Oracle provides two categories of JDBC drivers:

- **JDBC Thin** for Java applets and applications
- **JDBC OCI** for Java applications

JDBC Thin

Oracle's JDBC Thin driver is a Type 4 driver that uses Java sockets to connect directly to Oracle. It provides its own implementation of a TCP/IP version of Oracle's Net8. Because it is written entirely in Java, this driver is platform-independent.

The Thin driver does not require Oracle software on the client side. It connects to any Oracle database of version 8.0.4 and higher. The driver requires a TCP/IP listener on the server side.

JDBC OCI

Oracle's JDBC OCI drivers are Type 2 JDBC drivers. They provide an implementation of the JDBC interfaces that uses the OCI (Oracle Call Interface) to interact with an Oracle database. This driver can access Oracle8.0.4 and higher servers.

Because they use native methods, they are platform-specific. The supported platforms are:

- Solaris: version 2.5 and above.
- Windows: 95 and NT 3.51 and above.

The JDBC OCI driver requires an Oracle 8.0.4 client installation including Net8 and all other dependent files.

JDK Versions

Because Java has undergone significant changes over its brief life, you must use a driver version that matches your Java Development Kit.

- JDK 1.0.2
- JDK 1.1.1 and higher

3

The Java classes for JDK 1.0.2 contain the JDBC 1.22 classes from Javasoft. The Java classes for JDK 1.1.1 do not contain the JDBC classes, because those are a standard part of JDK 1.1.1.

Configuration

You can use the JDBC Thin driver in Java applets that can be downloaded into a web browser, such as Netscape 3.0 or 4.0.

The Thin driver is entirely self contained, requiring no Oracle-specific software or files on the client side. It does, however, need to open a Java socket. It cannot run successfully in a browser that does not allow that operation.

The Oracle JDBC OCI driver is not appropriate for Java applets, because it uses a C library that is platform specific and is not downloadable into a Web browser.

It is appropriate for Java applications and Java code running in the Oracle Web Application Server 3.0 and higher.

Changes From the Beta Release

The JDBC Thin driver supports databases that use multibyte character sets.

Getting Started

Getting started with JDBC has a few basic steps:

1. Identify and obtain the correct distribution file for your platform.

2. Install the files.

3. Set environment variables.

4. Test the installation.

Distribution Files

Oracle provides three distribution files. The correct choice depends only on your platform. Each distribution file contains all versions that run on that platform.

The choices are:

- Oracle8.0.4 CD

- Windows zip file (requires Windows 95 or Windows NT 3.51 or higher)

- Solaris tar file (requires Solaris 2.5 or higher)

- Other (tar or zip)

Installation

If you have used a previous version of Oracle JDBC drivers, deinstall them before proceeding.

Installing From CD

- Run Oracle Installer.

- Select JDBC driver from the list of products and install.

For Downloaded Windows95 or Windows NT, with the Oracle Installer

- Unzip the distribution in c:\temp.

- Point the installer to c:\temp.

- Use Oracle Installer to install JDBC OCI for **Windows95** or **Windows NT**. Select it in the Products Available window, and click the Install button. The installer places all files within a hierarchy whose top directory appears following the words "Products Installed on" on the installer screen above the Products Installed window. The remainder of this document refers to that top

directory as [ORACLE HOME]. The JDBC files reside in a directory structure beginning at [ORACLE HOME]\JDBC.

- Add [ORACLE HOME]\JDBC\LIB\CLASSES111.zip to your CLASSPATH.

- Add [ORACLE HOME]\JDBC\LIB\CLASSES102.zip to your CLASSPATH instead, if you are using JDK 1.0.2.

For Downloaded Windows95 or Windows NT, without the Oracle Installer

- Un-zip the distribution in C:\JDBC.

- Add C:\JDBC\LIB\CLASSES111.zip to your CLASSPATH. Or,

- Add C:\JDBC\LIB\CLASSES102.zip to your CLASSPATH instead, if you are using JDK 1.0.2.

- Add C:\JDBC\LIB to your PATH.

The **Windows** version contains the dynamically linked library file OCI80JDBC.DLL for JDBC OCI8. The directory containing it must be in your PATH. If you used the Oracle Installer it moved the DLLs to the [ORACLE_HOME]\BIN directory, which is already in your **PATH**.

For Downloaded Solaris

- Create a directory /local/jdbc.

- Un-tar the distribution in /local/jdbc.

- Add /local/jdbc/lib/classes111.zip to CLASSPATH (For JDK 1.1.1). Or,

- Add /local/jdbc/lib/classes102.zip to CLASSPATH instead, if you are using JDK 1.0.2.

- Add /local/jdbc/lib to LD_LIBRARY_PATH.

The **Solaris** version contains the shared object library liboci80jdbc.so for JDBC OCI8. The directory containing it must be in your **LD_LIBRARY_PATH**.

Other Platforms
See your platform-specific documentation.

Files Installed

Aside from differences in upper and lower case and the direction in which the slashes point, all three installations produce the same contents of the **jdbc** directory:

readme.txt doc/ samples/ lib/

Read the readme.txt file that contains a concise presentation of up-to-the-minute facts that may not be in this document.

The **doc** directory contains documentation.

The samples directory contains sample programs. These include examples of how to use SQL92 and Oracle SQL syntax, PL/SQL blocks, streams, and the Oracle JDBC type and performance extensions.

The **lib** directory contains the Java classes in zip files: classes111.zip and classes102.zip, the first for JDK 1.1.1 and the second for JDK 1.0.2. Place *only* one of these zip files into your **CLASSPATH**. Do not unzip them.

The **other** distribution contains only the JDBC Thin driver, so there are no additional files in the **lib** directory.

Environment Variables

For all Oracle JDBC drivers you must set your **CLASSPATH** to include the zip file containing the Java classes that implement the driver. One way to do this is to place [ORACLE HOME]/jdbc/classes111.zip or [ORACLE HOME]/jdbc/classes102.zip into your **CLASSPATH**.

For Oracle JDBC OCI drivers you must also set your **PATH** (Windows) or **LD_LIBRARY_PATH** (Solaris) to include the directory containing the appropriate DLL or so library file.

Testing the Installation

The samples directory contains a subdirectory of sample programs for each Oracle JDBC driver. Two programs are common to all of these directories: JdbcCheckup.java and Employee.java. The first is designed to test the installation, and the second performs an elementary database operation.

The following two sections of this document contain a summary of the principles of using an Oracle JDBC driver to connect to an Oracle database and a sample program.

Using Oracle's JDBC Drivers

This section describes what you need to do in your Java programs to use the Oracle JDBC drivers.

There are subtle differences in using the JDBC OCI, the JDBC Thin for JDK 1.1.1, and the JDBC Thin for JDK 1.0.2 drivers. Please read the information corresponding to the JDBC driver that you want to use.

Using JDBC OCI

Your program needs to do the following three steps before using the JDBC API to access the database:

1- Import the JDBC classes

2- Register the JDBC OCI driver

3- Open a connection to the database

Import the JDBC Classes

Import the JDBC classes by adding the following import statements at the beginning of your program. The first import statement brings in the JDBC classes, the second adds the BigDecimal classes.

```
import java.sql.*;
import java.math.*;
```

Register the JDBC OCI Driver

Register the JDBC driver with the following call. This needs to be done only once in your Java application.

```
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
```

Open a Connection to the Database

Open a connection to the database with the JDBC getConnection method. This method needs a "connect string" that identifies the JDBC driver to use and the database to connect to. You also need to pass the user logon and password.

For the JDBC OCI driver, the database can be specified by a TNSNAMES entry; this is one of the database names you use from SQL*Plus. The available TNSNAMES

entries are listed in the file [ORACLE_HOME]/network/admin/tnsnames.ora on the client computer you are connecting from.

For example, if you want to connect to the database "mydatabase" as user "scott" with password "tiger":

```
Connection conn =
   DriverManager.getConnection ("jdbc:oracle:oci8:@mydatabase",
      "scott", "tiger");
```

Note that the ":" and "@" characters are both necessary.

For the JDBC OCI driver you can also specify the database with a Net8 name-value pair. This is less readable than a TNSNAMES entry but does not depend on the accuracy of the TNSNAMES.ORA file. This also works with the other JDBC drivers.

For example, if you want to connect to the database on host "myhost" that has a TCP/IP listener up on port 1521, and the SID (system identifier) is "orcl", use a statement such as:

```
Connection conn =
DriverManager.getConnection("jdbc:oracle:oci8:@(description=(address=(host=
myhost)(protocol=tcp)(port=1521))(connect_data=(sid=orcl)))",
"scott", "tiger");
```

NOTE: All parentheses and equal signs are necessary.

Sample Program for JDBC OCI8

The following program lists the contents of the ENAME column of the EMP table. It loads an Oracle JDBC driver, connects to the database *mydatabase*, submits a query, receives a result set, and outputs the employee names.

```
import java.sql.*;

class JdbcTest {
  public static void main (String args []) throws SQLException {
 // Load Oracle driver
 DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());

 // Connect to the local database
 Connection conn =
 DriverManager.getConnection ("jdbc:oracle:oci8:@mydatabase", "scott", "tiger");

 // Query the employee names
 Statement stmt = conn.createStatement ();
 ResultSet rset = stmt.executeQuery ("select ename from emp");
```

```
 // Print the name out
 while (rset.next ())
      System.out.println (rset.getString (1));
  }
}
```

Using JDBC Thin Driver With JDK 1.1.1

The JDBC Thin driver does not support TNSNAMES entries for the database name. See Step 3 for how to specify the database.

Your program must execute the following three steps before using the JDBC API to access the database:

1- Import the JDBC classes

2- Register the JDBC Thin driver

3- Open a connection to the database

Import the JDBC Classes

Import the JDBC classes by adding the following import statements at the beginning of your program. The first import statement brings in the JDBC classes, the second adds the BigDecimal classes.

```
import java.sql.*;
import java.math.*;
```

Register the JDBC Thin Driver

Register the JDBC driver with the following call. This needs to be done only once in your Java application.

```
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver ());
```

Open a Connection to the Database

Open a connection to the database with the JDBC *getConnection* method. This method needs a connect string that identifies the JDBC driver you are using and the database you are connecting to. You also need to pass the user logon and password.

Since the JDBC Thin driver can be used in applets that *do not* have an Oracle installation you *cannot* use a TNSNAMES entry to identify the database you want to connect to. You have to list explicitly the host name, TCP/IP port and Oracle SID

of the database you want to connect to. Please see your database administrator if you are not sure of the correct values.

For example, if you want to connect to the database on host "myhost" , that has a TCP/IP listener on port 1521 for the database SID (system identifier) "orcl", logon as user "scott", with password "tiger", write:

```
Connection conn =
    DriverManager.getConnection
        ("jdbc:oracle:thin:@myhost:1521:orcl", "scott", "tiger");
```

You can also specify the database with a Net8 name-value pair. This is less readable than the first version, but also works with the other JDBC drivers.

```
Connection conn =
    DriverManager.getConnection
("jdbc:oracle:thin:@(description=(address=(host=myhost)(protocol=tcp)
        (port=1521))(connect_data=(sid=orcl)))", "scott", "tiger");
```

Sample for JDBC Thin and JDK 1.1.1

The following program lists the contents of the ENAME column of the EMP table. It loads an Oracle JDBC driver, connects to the database, submits a query, receives a result set, and outputs the employee names.

```java
import java.sql.*;

class JdbcTest {
  public static void main (String args []) throws SQLException {
    // Load Oracle driver
    DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());

    // Connect to the local database
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:thin:@myhost:1521:orcl",
                                   "scott", "tiger");

    // Query the employee names
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("select ename from emp");

      // Print the name out
    while (rset.next ())
      System.out.println (rset.getString (1));
  }
```

```
        }
```

Using JDBC Thin with JDK 1.0.2

With JDK 1.0.2 you have to import the JDBC classes from the package jdbc.sql instead of java.sql. The driver class is oracle.jdbc.dnlddriver.OracleDriver.

Your program must execute the following three steps before using the JDBC API to access the database:

 1- Import the JDBC classes

 2- Register the JDBC Thin driver

 3- Open a connection to the database

Import the JDBC Classes

Import the JDBC classes by adding the following import statements at the beginning of your program. The first import line brings in the JDBC classes, the second line adds the BigDecimal classes.

```
import jdbc.sql.*;
import jdbc.math.*;
```

Register the Thin Driver

Register the JDBC driver with the following call. This needs to be done only once in your Java application. Note that the driver is called "dnlddriver".

```
DriverManager.registerDriver (new oracle.jdbc.dnlddriver.OracleDriver();
```

Open a Connection to the Database

Open a connection to the database with the JDBC *getConnection* method. This method needs a connect string that identifies the JDBC driver you want to use, the database you are connecting to, the user logon and password.

 Since the JDBC Thin driver can be used in applets that *do not* have an Oracle installation you cannot use a TNSNAMES entry to identify the database you want to connect to. You have to list explicitly the host name, TCP/IP port, and Oracle SID of the database you are connecting to. Please see your database administrator if you are not sure of the correct values.

For example, if you want to connect to the database on host "myhost", that has a TCP/IP listener on port 1521, an SID (system identifier) "orcl", and logon is as user "scott", with password "tiger":

```
Connection conn = DriverManager.getConnection
("jdbc:oracle:dnldthin:@myhost:1521:orcl","scott", "tiger");
```

Note that the driver is called "dnldthin". You can also specify the database with a Net8 name-value pair. This is less readable than the first version but also works with the other JDBC drivers.

```
Connection conn =
DriverManager.getConnection
("jdbc:oracle:dnldthin:@(description=(address=(host=myhost)
(protocol=tcp)(port=1521))(connect_data=(sid=orcl)))", "scott", "tiger");
```

Sample Program for JDBC Thin and JDK 1.0.2

The following program lists the contents of the ENAME column of the EMP table. It loads an Oracle JDBC driver, connects to the database, submits a query, receives a result set, and outputs the employee names.

```
import jdbc.sql.*;

class JdbcTest {
  public static void main (String args []) throws SQLException {
    // Load Oracle driver
    DriverManager.registerDriver (new oracle.jdbc.dnlddriver.OracleDriver());

    // Connect to the local database
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:dnldthin:@myhost:1521:orcl",
                                   "scott", "tiger");

    // Query the employee names
    Statement stmt = conn.createStatement ();
    ResultSet rset = stmt.executeQuery ("select ename from emp");

      // Print the name out
    while (rset.next ())
      System.out.println (rset.getString (1));
  }
}
```

An Oracle extension to the JDBC drivers is a form of the *getConnection* method that uses a `Properties` object. See Connection Properties.

Features of All Oracle JDBC Drivers

Oracle JDBC drivers support JDBC 1.22, the version supplied with the Java Development Kit (JDK) version 1.1.1 and available as an add-on to JDK 1.0.2.

Datatypes

The Oracle JDBC driver supports the SQL datatypes required by JDBC 1.22. In addition, Oracle JDBC drivers support the Oracle-specific datatypes ROWID and REFCURSOR. Discussion of how to use these Oracle-specific datatypes appears in Type Extensions.

The tables show how codes in the `java.sql.Types` class and `oracle.jdbc.driver.OracleTypes` class map into Oracle datatypes.

Table 1–1   Mapping JDBC Type Codes to Oracle Datatypes

JDBC Type Code	Oracle Datatype
Types.CHAR	CHAR
Types.VARCHAR	VARCHAR2
Types.LONGVARCHAR	LONG
Types.VARBINARY	RAW
Types.LONGVARBINARY	LONG RAW
All numeric types	NUMBER
All date types	DATE

Table 1–2   Mapping Oracle Type Codes to Oracle Datatypes

Oracle Type Code	Oracle Datatype
OracleTypes.ROWID	ROWID
OracleTypes.REFCURSOR	REFCURSOR
OracleTypes.BLOB	BLOB
Oracle.Types.CLOB	CLOB
Oracle.Types.BFILE	BFILE
Oracle.Types.CFILE	CFILE

LOB datatypes

Oracle8 provides datatypes for LOBs (large objects and external files). The datatypes are BLOB (unstructured binary data), CLOB (single-byte character data), BFILE (external file of binary data), CFILE (external file of single-byte character data).

Note: Only the JDBC OCI8 driver supports LOBs.

The JDBC extensions for LOB access to be used off the ResultSet or CallableStatement (the resultSet and callableStatement objects will have to be cast to OracleResultSet and OracleCallableStatement respectively in order to use these API) :

```
 OracleBlob getBlobValue (int index)

OracleClob getClobValue (int index)

OracleBfile getBfileValue (int index)

OracleCfile getCfileValue (int index)
```

These API return the LOB descriptor or file descriptor. The JDBC API off the PreparedStatement to bind the descriptor values are (the prepared statement object will have to be cast to an OraclePreparedStatement in order to use these API):

```
void setBlob (int parameterIndex, OracleBlob lob)

void setClob (int parameterIndex, OracleClob lob)

void setBfile (int parameterIndex, OracleBfile file)

void setCfile (int parameterIndex, OracleCfile file)
```

Please refer to the sample program, LobExample.java for usage examples.

Multibyte Character Sets

The JDBC Thin driver can access databases that use any Oracle character set. This is achieved by converting the characters to Unicode 1.2. Java itself uses Unicode 2.0, so there is a mismatch, largely affecting Korean characters.

Streaming

Oracle JDBC drivers support streaming of data in either direction between server and client. They support all stream conversions: binary, ASCII, and Unicode.

NOTE: Receiving LONG or LONG RAW columns in a streaming fashion (the default case) requires you to pay special attention to the order in which you receive data from the database.

A separate section explains the details of streaming. See Streams Tutorial.

Stored Procedures

Oracle JDBC drivers support execution of PL/SQL stored procedures and anonymous blocks. They support both SQL92 escape syntax and Oracle escape syntax. The following PL/SQL calls are all available from any Oracle JDBC driver:

```
// SQL92 Syntax
    CallableStatement cs1 = conn.prepareCall
                            ( "{call proc (?,?)}" ) ;
    CallableStatement cs2 = conn.prepareCall
                            ( "{? = call func (?,?)}" ) ;
// Oracle Syntax
    CallableStatement cs3 = conn.prepareCall
                            ( "begin proc (:1, :2); end;" ) ;
    CallableStatement cs4 = conn.prepareCall
                            ( "begin :1 := func(:2,:3); end;" ) ;
```

Database Metadata

Oracle JDBC drivers support all database metadata entry points. They do so by issuing queries against Oracle metadata tables. The distribution includes the source code of the OracleDatabaseMetadata class, which you can use to design your own metadata calls.

SQL92 Syntax

Oracle JDBC drivers support SQL92 escapes, except for outer joins. See Oracle SQL documentation for instructions on specifying outer joins.

Extensions to JDBC 1.22

Oracle JDBC drivers provide a variety of extensions to JDBC 1.22. These are summarized and discussed in Features of All Oracle JDBC Drivers.

Oracle Types

Oracle JDBC drivers support **ROWID** as a Java string and **REFCURSOR** as a Java ResultSet.

Row Prefetching

Oracle JDBC drivers allow you to set a number (default is 10) of rows to prefetch into the client during queries, thereby reducing round trips to the server. You can set the amount of prefetching for either the connection or the statement.

Execution Batching

Oracle JDBC drivers allow you to accumulate inserts and updates at the client and send them to the server in batches, thereby reducing round trips to the server. You can set the batch size (default is 1) for a statement.

Define Query Columns

Oracle JDBC drivers allow you to inform the driver of the types of the columns in an upcoming query, thereby saving a round trip to the database.

Database Metadata Remarks

Oracle JDBC drivers execute the DatabaseMetaData calls getTables and getColumns with reporting of the **TABLE_REMARKS** column turned off by default, thereby avoiding a time-consuming outer join. You can turn reporting of the **TABLE_REMARKS** column back on if you wish.

Limitations

There are a few requirements of JDBC 1.22 that Oracle JDBC drivers do not support:

CursorName

Oracle JDBC drivers do not support the getCursorName and setCursorName calls, because there is no convenient way to map them to Oracle constructs. Oracle recommends using **ROWID** instead.

Catalog Arguments to DatabaseMetaData Calls

There is no Oracle equivalent of the JDBC catalog arguments to DatabaseMetaData calls. Oracle JDBC drivers ignore catalog arguments.

SQL92 Outer Join Escapes

Oracle JDBC drivers do not support SQL92 outer join escapes. Use Oracle syntax with "(+)" instead.

PL/SQL BOOLEAN and RECORD Types

Oracle JDBC drivers do not support calling arguments or return values of the PL/SQL **BOOLEAN** or **RECORD** types. For more information, see Features Not Implemented

IEEE 754 Floating Point Compliance

Oracle's arithmetic on its **NUMBER** type is not compliant with the IEEE 754 standard for floating point arithmetic. Therefore there can be small disagreements between the results of computations performed by Oracle and the same computations performed by Java.

Oracle stores numbers in a format compatible with decimal arithmetic and guarantees 38 decimal digits of precision. It represents zero, minus infinity, and plus infinity exactly. For each positive number it represents, it represents a negative number of the same absolute value.

It represents every positive number between 10^{-30} and $(1 - 10^{-38}) * 10^{126}$ to full 38-digit precision.

Features of Specific Oracle JDBC Drivers

While all Oracle JDBC drivers are similar, some features apply only to JDBC OCI drivers and some apply only to the JDBC Thin driver.

JDBC OCI Features

The JDBC OCI drivers are Type 2 drivers that use Java native methods to call the C entry points of the OCI library. The use of native methods makes JDBC OCI drivers platform specific. They provide support for Solaris, Windows, and other platforms. The Windows version works both with Windows 95 and with Windows NT, versions 3.51 and 4.0.

JDBC OCI drivers, because they are platform specific, are not suitable for use in applets intended to be downloaded into browsers running on unknown platforms. They are, however, excellent choices for Java applications or Java middle tiers like the Oracle Web Application Server 3.0 Java Cartridge.

The JDBC OCI drivers require installation of Net8, version 8.0 or above, on the client side. Since they interface to Oracle databases through OCI, the JDBC OCI drivers support all installed Net8 adapters—IPC, named pipes, TCP/IP, DECnet, and others. They also support all features of the Advanced Networking Option, including encrypted Net8.

JDBC OCI drivers convert **CHAR** data represented in multibyte character sets into Java strings represented in Unicode. They do so on the client side using the conversion routines that OCI provides.

JDBC Thin Features

The JDBC Thin driver is a 100% Java Type 4 driver. It connects directly to Oracle via Java sockets without the need for a JDBC-specific middle tier. The JDBC Thin driver can only connect to a database if a TNS Listener is up and listening on TCP/IP sockets.

The JDBC Thin driver is only as platform-specific as Java is. It works on any system that provides a correct implementation of Java.

The JDBC Thin driver can be downloaded into any browser as part of a Java application. It is suitable for applets on an intranet, but firewall issues limit its use in applets for general distribution via the World Wide Web. Discussion of applets, firewalls, and browser security issues occur in section Applets.

The `samples` subdirectory of the driver distribution contains an applet that uses the JDBC Thin driver.

Extensions to JDBC

The extensions to JDBC fall into these categories:

- Connection Properties
- Type Extensions
- Performance Extensions

Connection Properties

Another form of the *getConnection* method uses the *Properties* class. For example:

```
java.util.Properties info = new java.util.Properties();
info.addProperty ("user", "scott");
info.addProperty ("password","tiger");
getConnection ("jdbc:oracle:oci8:",info);
```

Oracle JDBC drivers support other properties as well. The following is a complete list:

Table 1–3   Properties Recognized by Oracle JDBC Drivers

Name	Short Name	Type	Description	Equivalent to
user	N/A	String	The user name for logging into the database	N/A
password	N/A	String	The password for logging into the database	N/A
database	server	String	The connect string for the database	N/A
defaultRowPrefetch	prefetch	Integer	The default row prefetch	setDefaultRowPrefetch
remarksReporting	remarks	Boolean	True if getTables and getColumns should report TABLE_REMARKS	setRemarksReporting

If you wish to use JDBC Thin in an applet in a browser that supports only JDK 1.0.2 (Netscape Navigator 3.0, for example), change the first statements in your program to:

```
import jdbc.sql.*;
import jdbc.math.*;
 DriverManager.registerDriver (new oracle.jdbc.dnlddriver.OracleDriver());
```

Use the JDBC Thin 1.0.2 URL this way:

```
Connection conn = DriverManager.getConnection
("jdbc:oracle:dnlthin:@database","user","password");
```

Type Extensions

The JDBC drivers support the Oracle ROWID and REFCURSOR types.

Oracle ROWID Type

We do not support the getCursorName and setCursorName JDBC entry points. Instead we provide access to **ROWID**s, which provide similar functionality.

If you add the **ROWID** pseudocolumn to a query you can retrieve it in JDBC with the ResultSet getString entry point. You can also bind a ROWID to a preparedStatement parameter with the setString entry point.

This allows in-place updates, as in the following example:

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

// Query the employee names with "FOR UPDATE" to lock the rows.
// Select the ROWID to identify the rows to be updated.

ResultSet rset =
stmt.executeQuery ("select ENAME, ROWID from EMP for update");

// Prepare a statement to update the ENAME column at a given ROWID

PreparedStatement pstmt =
conn.prepareStatement ("update EMP set ENAME = ? where ROWID = ?");

// Loop through the results of the query
while (rset.next ())
{
    String ename = rset.getString (1);
    String rowid = rset.getString (2);  // Get the ROWID as a String
    pstmt.setString (1, ename.toLowerCase ());
    pstmt.setString (2, rowid); // Pass ROWID to the update statement
    pstmt.executeUpdate ();      // Do the update
}
```

In the *ResultSetMetaData* class, columns containing **ROWID**s are reported with the type oracle.jdbc.driver.OracleTypes.ROWID, whose value is -**8**.

Oracle REFCURSOR Type

The Oracle JDBC driver supports bind variables of type REFCURSOR. A REFCURSOR is represented by a JDBC `ResultSet`. Use the *getCursor* method of the *CallableStatement* to convert a REFCURSOR value returned by a PL/SQL block into a `ResultSet`. JDBC lets you call a stored procedure that executes a query and returns a results set. Cast the corresponding `CallableStatement` to `oracle.jdbc.driver.OracleCallableStatement` to use the getCursor method.

Importing classes from the *oracle.jdbc.driver* package makes programs more readable. Here is a simple example. The `samples` subdirectory of the distribution has additional examples.

```
import oracle.jdbc.driver.*;
...
  CallableStatement cstmt;
  ResultSet cursor;

  // Use a PL/SQL block to open the cursor
  cstmt = conn.prepareCall
            ("begin open ? for select ename from emp; end;");

  cstmt.registerOutParameter (1, OracleTypes.CURSOR);
  cstmt.execute ();
  cursor = ((OracleCallableStatement)cstmt).getCursor (1);

  // Use the cursor like a normal ResultSet
  while (cursor.next ())
    {System.out.println (cursor.getString (1));}
```

Performance Extensions

Oracle JDBC drivers supports extensions that improve performance by reducing round trips to the database.

Prefetching Rows uses client-side buffers to replace expensive round trips by inexpensive local pointer manipulation for most rows returned by a query.

Batching Updates does for data headed toward the database what prefetching does for data coming from it.

Specifying Column Types gets around an inefficiency in the usual JDBC protocol for performing and returning the results of queries.

Suppressing DatabaseMetaData TABLE_REMARKS Columns avoids an expensive outer join operation.

Prefetching Rows

Standard JDBC receives the result sets of queries one row at a time. Each row costs a round trip to the database. This feature associates with each statement object an integer called its *row prefetch setting*. JDBC fetches that number of rows at a time from result sets associated with the statement.

Use OracleStatement.setRowPrefetch to set a statement object's row prefetch setting or OracleConnection.setDefaultRowPrefetch to establish an initial value for all statement objects created for a given connection object. If you use the form of *getConnection* that takes a Properties object as an argument, you can set the connection's default row prefetch value that way. See the table Properties Recognized by Oracle JDBC Drivers.

If you do not set a default row prefetch value for a connection, DefaultRowPrefetch, its default row prefetch value is 10.

A statement object receives the default row prefetch setting from the associated connection at the time of the statement's creation. Subsequent changes to the connection's default row prefetch setting have no effect on the statement's row prefetch setting. Use setRowPrefetch to change the statement's row prefetch setting.

If a column of a result set is of type **long data** or **long raw data** (that is, the streaming types), JDBC changes the statement's row prefetch setting to one, even if JDBC never actually reads a value of either of those types.

The methods OracleConnection.getDefaultRowPrefetch and OracleStatement.getRowPrefetch return current values of these two settings.

Notes:

1. To use the setDefaultRowPrefetch and getDefaultRowPrefetch methods, cast the connection object returned by the getConnection method to type `oracle.jdbc.driver.OracleConnection`.

2. To use the setRowPrefetch and getRowPrefetch methods, cast the statement object returned by the connection's createStatement method to type `oracle.jdbc.driver.OracleStatement`.

Example

The following example illustrates the use of this feature. It assumes you have imported the classes `oracle.jdbc.driver.*`

```
Connection conn =
 DriverManager.getConnection ("jdbc:oracle:oci8:","scott","tiger");

//Set the default row prefetch setting for this connection
((OracleConnection)conn).setDefaultRowPrefetch (7);

/* The following statement gets the default row prefetch value for
   the connection, that is, 7.
*/
Statement stmt = conn.createStatement ();

/* Subsequent statements look the same, regardless of the row
   prefetch value. Only execution time changes.
*/
ResultSet rset = stmt.executeQuery ("select ename from emp");
System.out.println ( rset.next () );

while( rset.next () )
   System.out.println ( rset.getString (1) );

//Override the default row prefetch setting for this statement
( (OracleStatement)stmt ).setRowPrefetch (2);

ResultSet rset = stmt.executeQuery ("select ename from emp");
System.out.println ( rset.next () );

while( rset.next () )
   System.out.println ( rset.getString (1) );

stmt.close ();
```

Batching Updates

Standard JDBC makes a round trip to the database to execute a prepared statement whenever the statement's executeUpdate method is executed. This feature associates with each prepared statement object an integer called its *batch size*. JDBC accumulates that many execution requests for the statement before passing the requests to the database for execution.

Use OraclePreparedStatement.setExecuteBatch to set a prepared statement object's batch size.

Whenever the executeUpdate method of a prepared statement object is invoked, JDBC queues an execution request. When the number of queued requests reaches

the batch size, JDBC sends them to the database for execution. Calling the method OraclePreparedStatement.sendBatch also causes JDBC to send queued execution requests for the given prepared statement to the database for execution.

Regardless of the batch size, if any of a prepared statement's bind variables is (or becomes) a streaming type, JDBC sets the statement's batch size to one and sends any queued requests to the database for execution.

JDBC automatically executes the statement's sendBatch method whenever the connection receives a commit request or either the statement or the connection receives a close request.

The method OracleStatement.getExecuteBatch returns the current values of this setting.

Notes:

1. To use the sendBatch, setExecuteBatch and getExecuteBatch methods, cast the statement object returned by the connection's createStatement method to type `oracle.jdbc.driver.OraclePreparedStatement`.

2. Queued requests are invisible to the database. They are not available to queries. Use a batch size of one, or call the sendBatch method whenever you need immediate updates.

Example

The following example illustrates the use of this feature. It assumes you have imported the classes `oracle.jdbc.driver.*`

```
Connection conn =
  DriverManager.getConnection ("jdbc:oracle:oci8:","scott","tiger");

PreparedStatement ps =
  conn.prepareStatement ("insert into dept values (?, ?, ?)");

//Change batch size for this statement to 3
((OraclePreparedStatement)ps).setExecuteBatch (3);

ps.setInt (1, 23);
ps.setString (2, "Sales");
ps.setString (3, "USA");
ps.executeUpdate (); //JDBC queues this for later execution

ps.setInt (1, 24);
ps.setString (2, "Blue Sky");
```

```
ps.setString (3, "Montana");
ps.executeUpdate (); //JDBC queues this for later execution

ps.setInt (1, 25);
ps.setString (2, "Applications");
ps.setString (3, "India");
ps.executeUpdate (); //The queue size equals the batch value of 3
                //JDBC sends the requests to the database

ps.setInt (1, 26);
ps.setString (2, "HR");
ps.setString (3, "Mongolia");
ps.executeUpdate (); //JDBC queues this for later execution

((OraclePreparedStatement)ps).sendBatch ();
                            //JDBC sends the queued request
ps.close();
```

Specifying Column Types

When standard JDBC performs a query, it first uses a round trip to the database to determine the types of the columns of the result set. Then, when JDBC receives data from the query, it converts the data, if necessary, to the requested return type.

When you specify column types for a query, JDBC makes one fewer round trip to the database. The server, which is optimized to do so, performs any necessary type conversions.

To use this feature you must specify a data type for each column of the expected result set. If the number of columns for which you specify types does not match the number of columns in the result set, the process fails.

Use the following procedure to specify column types for a query:

1. Use the method OracleStatement.clearDefines (if necessary) to clear any previous column definitions for this statement object.

2. Determine, for each column of the expected result set

   ■ The integer column index (position).

   ■ The integer code for the type of the expected return data.
   (This can differ from the column type.)

3. For each column of the expected result set, invoke the method OracleStatement.defineColumnType, passing it

- Column index.
- Type code.
- (Optionally) maximum field size.

4. Use the statement's executeQuery method to perform the query.

Example

The following example illustrates the use of this feature. It assumes you have imported the classes `oracle.jdbc.driver.*`

```
Connection conn =
DriverManager.getConnection ("jdbc:oracle:oci8:","scott","tiger");

Statement stmt = conn.createStatement ();

/*Ask for the column as a string:
   Avoid a round trip to get the column type.
   Convert from number to string on the server.
*/
((OracleStatement)stmt).defineColumnType (1, Types.VARCHAR);

ResultSet rset = stmt.executeQuery ("select empno from emp");

while (rset.next () )
    System.out.println (rset.getString (1));

stmt.close ();
```

Notice the cast of **stmt** to type **OracleStatement** in the invocation of the defineColumnType method. The connection's createStatement method returns an object of type **Statement**. The **Statement** type does not have the defineColumnType and clearDefines methods.

The define extensions use JDBC types to specify the desired types. The allowed define types for columns depends on the internal Oracle type of the column.

All columns can be defined to their "natural" JDBC types, in most cases, to **Types.CHAR** or **Types.VARCHAR**.

Table 1–4, "Valid Define Types" is a list of valid define arguments to use in DefineColumnType:

Table 1–4   Valid Define Types

Oracle Type	Valid JDBC Define Type
NUMBER, VARNUM	BIGINT, TINYINT, SMALLINT, INTEGER, FLOAT, REAL, DOUBLE, NUMERIC, DECIMAL, CHAR, VARCHAR
CHAR, VARCHAR	CHAR, VARCHAR
LONG	LONGVARCHAR
LONGRAW	LONGVARBINARY
RAW	VARBINARY, BINARY
DATE	DATE, TIME, TIMESTAMP, CHAR, VARCHAR

Suppressing DatabaseMetaData TABLE_REMARKS Columns

The *DatabaseMetaData* calls *getTables* and *getColumns* are slow if they must report *TABLE_REMARKS* columns, because this necessitates an expensive outer join. By default the JDBC driver does not report *TABLE_REMARKS* columns.

The *OracleConnection* class provides two entry points for controlling the reporting of *TABLE_REMARKS* columns:

- OracleConnection.setRemarksReporting (boolean)

- OracleConnectiongetRemarksReporting ()

You can turn on *TABLE_REMARKS* reporting by passing a *true* argument to the *Connection.setRemarksReporting* method. You turn it back off by passing a *false* argument. First, cast your *Connection* object to the class *oracle.jdbc.driver.OracleConnection*.

The following code turns *TABLE_REMARKS* reporting on:

```
((oracle.jdbc.driver.OracleConnection)conn).setRemarksReporting (true);
```

Features Not Implemented

- `setCursorName` and `getCursorName` are not implemented. Use the ROWID type (see Features of All Oracle JDBC Drivers) rather than the `setCursorName` and `getCursorName` calls.

- We do not support ODBC escapes for outer joins. Use the usual Oracle SQL syntax for outer joins instead of the ODBC escapes syntax.

- We do not support arguments of type BOOLEAN to PL/SQL stored procedures. This is a restriction of the OCI.

  Workaround: define a second PL/SQL stored procedure that accepts the BOOLEAN argument as a CHAR or NUMBER and passes it as a BOOLEAN to the first stored procedure.

Applets

Browser Security Considerations

The communication between an applet that uses the JDBC Thin driver and the Oracle database happens on top of Java TCP/IP sockets. The connection can only be made if the web browser where the applet is executing allows a sockets connection to be made.

In a JDK 1.0.2 based Web Browser, such as Netscape 3.0, an applet can only open sockets to the host from which it was downloaded. For Oracle8 this means that the applet can only connect to a database running on the same host as the web server.

In a JDK 1.1.1 based web browser, such as Netscape 4.0, an applet can request socket connection privileges and, if the user grants them, the applet can connect to a database running on a different host from the web server host.

In Netscape 4.0 this involves signing your applet, then opening your connection as follows. Please refer to your browser documentation for the many details you have to take care of.

```
netscape.security.PrivilegeManager.enablePrivilege
            ("UniversalConnect");
connection = DriverManager.getConnection
            ("jdbc:oracle:thin:scott/tiger@dlsun511:1721:orcl");
```

Firewall Considerations

The JDBC Thin driver cannot connect to a database from behind a firewall. The firewall prevents the browser from opening a TCP/IP socket to the database.

This problem can be solved by using a Net8 compliant firewall and using connect strings in the applet that are compliant with the firewall configuration. This solution really only works for an intranet, because the connect string is dependent on the firewall behind which the *client* browser is running.

Writing the Applet Code

Write a JDBC applet like any other Java applet. You must import the JDBC interfaces to be able to access the JDBC entry points.

If you're targeting a JDK 1.1.1 browser, import the JDBC interfaces from the *java.sql* package. You load the Oracle JDBC Thin driver as usual. We recommend that your applet have a Connection local variable to contain the JDBC connection to the

database. (Note: you might prefer to connect to the database just when needed and keep the connection closed at other times).

```
import java.sql.*;
public class JdbcApplet extends java.applet.Applet
{
  Connection conn;   // Holds the connection to the database
  public void init()
  {
    // Load the driver
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
    // Connects to the database
    conn = DriverManager.getConnection
 ("jdbc:oracle:thin:scott/tiger@www-aurora.us.oracle.com:1521:orcl");
    ...
  }
}
```

In this example the connect string contains the username and password, but you can also pass them as arguments to getConnection after obtaining them from the user. See the standard JDBC documentation for more information.

For a JDK 1.0.2 browser, import the JDBC interfaces from the *jdbc.sql* package, load the driver from the *oracle.jdbc.dnlddriver.OracleDriver* class and use the dnldthin sub-protocol in your connect string:

```
import jdbc.sql.*;
public class JdbcApplet extends java.applet.Applet
{
  Connection conn;   // Holds the connection to the database
  public void init ()
  {
    // Load the driver
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
    // Connects to the database
    conn = DriverManager.getConnection
("jdbc:oracle:dnldthin:scott/tiger@www-aurora.us.oracle.com:1521:orcl");
    ...
  }
}
```

Packaging the Applet Code

The HTML page that runs the applet must have an APPLET tag with an initial width and height. For example, if **JdbcApplet.htm** contains the lines:

```
<applet code="JdbcApplet" archive="JdbcApplet.zip" width=500 height=200>
```

```
</applet>
```

If you use that form, the classes for the applet and the classes for the JDBC Thin driver must be in the same directory as the HTML page.

You can use the **CODEBASE** or **ARCHIVE** tags in the applet tag to place the applet and JDBC driver classes in a different directory on the server, or in a zip or jar file. Oracle recommends the use of a **zip** file. This saves many extra round-trips to the server. **CODEBASE** gives the directory name that your class files are in. It is a directory below the directory where the current page is. **ARCHIVE** gives the name of the **zip** or **jar** file.

Version 3.0 browsers do not support ARCHIVE.

What JDBC Files to Put in a zip or jar for an Applet

For a browser running the JDK 1.1.1, put the JDK 1.1.1 driver classes in the zip or jar for your applet. This is done by copying classes111.zip to a file, such as **myclasses.zip**, and then adding the application classes to **myclass.zip**. If you're not using the DatabaseMetaData entry points you can omit the oracle/jdbc/driver/OracleDatabaseMetaData.class file, which is large.

For a browser running the JDK 1.0.2, put the JDK 1.0.2 driver classes *and* the jdbc interface files from the *jdbc.sql* package (in the classes/jdbc/sql directory of the JDBC distribution) in the zip file. (JDK 1.0.2 browsers support zip files but not jar files). Because the classes of the JDBC Thin driver are delivered in a zip file, you have to extract the driver files from that zip before putting them in the zip or jar for your applet.

Streams Tutorial

Streams and LONG or LONG RAW Columns

When a query selects one or more **LONG** or **LONG RAW** columns the JDBC driver transfers these columns to the client in streaming mode: after a call to executeQuery or next, the data of the **LONG** column is waiting to be read on the connection to the database. To access the data you can get the column as a Java *InputStream* and use the read method of the *InputStream* object.

You can also get the data as a **String** or byte array, in which case the driver will do the streaming for you.

Example

The next Java example dumps the contents of a **LONG RAW** column to a file on the local file system. To create the table:

```
-- SQL code:
create table streamexample (NAME varchar2 (256), GIFDATA long raw);
insert into streamexample values ('LESLIE', '00010203040506070809')
```

Java code to dump the LESLIE **LONG RAW** column's data into a file called `leslie.dat`:

```
// Do a query to get the images named 'LESLIE'
ResultSet rset = stmt.executeQuery
      ("select GIFDATA from streamexample where NAME='LESLIE'");

// Get the first row
if (rset.next ())
{
  // Get the Gif data as a Stream from Oracle to the client
  InputStream gif_data = rset.getBinaryStream (1);
  // Open a file to store the gif data
  FileOutputStream file = new FileOutputStream ("leslie.gif");

  // Loop, reading from the gif stream and writing to the file
  int c;
  while ((c = gif_data.read ()) != -1)
    file.write (c);
  // Close the file
  file.close ();
}
```

In the example the contents of the **GIFDATA** column are transferred incrementally between the database and the client: The *InputStream* object returned by the call to getBinaryStream reads the data directly from the database connection.

Instead of getting the column with getBinaryStream you can get it with getBytes. In that case the driver fetches all data in one call into a byte array.

The previous example can be rewritten as:

```
// Do a query to get the images named 'LESLIE'
ResultSet rset = stmt.executeQuery
      ("select GIFDATA from streamexample where NAME='LESLIE'");

// Get the first row
if (rset.next ())
{
  // Get the Gif data as a Stream from Oracle to the client
  byte [] bytes = rset.getBytes (1);
  // Open a file to store the gif data
  FileOutputStream file = new FileOutputStream ("leslie.gif");

  // Write all the bytes in a single call
  file.write (bytes);
  // Close the file
  file.close ();
}
```

Because a **LONG RAW** column can contain up to 2 Gigabytes of data, the second example will likely use much more memory than the first example. You should use streams if you do not know the maximum size of the data in your **LONG** or **LONG RAW** columns.

Avoiding Streaming

The JDBC driver automatically streams any **LONG** and **LONG RAW** column. This is because streaming has to be decided when the query is executed: at this point the driver does not know yet if you will fetch **LONG** and **LONG RAW** columns as streams or not. So the driver assumes that you will stream. The opposite assumption uses much more memory if one of your **LONG** column is extremely large.

You can use the Define extension (see Prefetching Rows) to prevent the driver from streaming long columns. If you tell the driver that a **LONG** or **LONG RAW** column is actually of type **VARCHAR** or **VARBINARY** then the driver will not stream the data.

In the following example the data is not streaming:

```
oracle.jdbc.driver.OracleStatement ostmt =
  (oracle.jdbc.driver.OracleStatement)stmt;
ostmt.defineColumnType (1, Types.VARBINARY);
// Do a query to get the images named 'LESLIE'
ResultSet rset = stmt.executeQuery
        ("select GIFDATA from streamexample where NAME='LESLIE'");
// The data is not streaming here
rset.next ();
byte [] bytes = rset.getBytes (1);
```

Streaming and Multiple Columns

If your query selects multiple columns and one of them is streaming, the contents of the columns coming after the stream are normally not available until the stream has been read. This is because the database sends each row as a set of bytes representing the columns in the SELECT order: the data after a streaming column can only be read after the stream has been read.

For example, consider the following query:

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
```

The incoming data for each row has the following shape:

```
<a date><the characters of the long column><a number>
```

When you call rset.next() the JDBC driver stops reading the row data just before the first character of the **LONG** column. The data for the **NUMBER** has not yet been read. The Java Stream you get with rset.getAsciiStream reads the characters of the **LONG** column directly out of the database connection. The driver reads the data for the **NUMBER** from the third column only after it reads the last byte of the data from the stream.

If you do not want to read the data for the streaming column you can just call the close method of the stream object. This discards the stream data and reads the data for all the non-streaming columns that follow the stream.

If you try to access the data for the **NUMBER** column *before* reading the data from the streaming column the JDBC driver discards the streaming data automatically. You cannot access that data any more. If you try to get a stream for the **LONG** column the driver raises a "Stream Closed" error.

For example:

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
```

```
int n = rset.getInt (3);  // This discards the streaming data
InputStream is = rset.getAsciiStream (2);
                      // Raises an error: stream closed.
```

If you get the stream *before* getting the **NUMBER** column the stream still gets closed automatically:

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
InputStream is = rset.getAsciiStream (2); // Get the stream
int n = rset.getInt (3);
                  // Discards streaming data and closes the stream
int c = is.read (); // c is -1: no more characters to read
```

Closing a Stream

You can discard the data from a stream at any time by calling the stream's close method. Data from a stream is automatically discarded if you do any JDBC operation that talks to the database, other than reading the current stream. For example:

- Fetching the next row.

- Executing a different statement.

- Closing the result set or the connection.

Streaming non-LONG or LONG RAW Data

If you get a CHAR, VARCHAR or RAW column with one of the *ResultSet*getStream methods you get a Java *InputStream* but no real streaming occurs. The data is fully fetched during the call to executeQuery or next and the getXXXStream entry points returns a stream that reads data from an in-memory buffer.

Streaming and Row Prefetching

In the presence of a streaming column row prefetching is set back to 1.

Streaming and the Define Extension

You can use the define extension to define a **CHAR, VARCHAR** or **RAW** column as a stream by passing the type codes **LONGVARCHAR** or **LONGVARBINARY**. The program behaves as if that column was actually of the type **LONG** or **LONG RAW**. Note that there is not much point to this, because these columns are usually short.

Stream Types and Column Types

JDBC provides 3 types of streams:

- `BinaryStream`: returns the **RAW** bytes of the data.

- `AsciiStream`: returns ASCII bytes. Actually returns ISO-Latin-1 bytes.

- `UnicodeStream`: returns Unicode bytes with the UCS-2 encoding in big-endian format. You first get the high byte (character / 256) and then the low byte (character % 256)

You can get **LONG** and **LONG RAW** data with any of the 3 stream types. The driver does conversions for you depending on the character set of your database and on the type of driver.

LONG RAW Data Conversions

RAW data is returned as-is by a **BinaryStream** but converted to a hexadecimal representation by an ASCII or Unicode stream. The ASCII streams return the ASCII bytes of the hexadecimal representation and the Unicode streams returns the Unicode bytes.

If your **LONG RAW** column contains the bytes 20 21 22 you receive the following bytes:

LONG RAW	BinaryStream	ASCIIStream	UnicodeStream
20 21 22	20 21 22	49 52 49 53 49 54	0 49 0 52 0 49 0 53 0 49 0 54
		which is also	which is also
		'1' '4' '1' '5' '1' '6'	0 '1' 0 '4' 0 '1' 0 '5' 0 '1' 0 '6'

LONG Data Conversions

When you get **LONG** data as an ASCII stream you get a stream of ISO-Latin-1 characters.

When you get **LONG** data as a Unicode stream you get a stream of Unicode characters in the UCS2 encoding. The bytes are returned in big-endian ordering.

When getting **LONG** data as a Binary stream you get a stream of bytes representing the characters encoded in Unicode UTF8 format.

Common Problems and Frequently Asked Questions

This section lists questions you may have when using the Oracle JDBC drivers. Installation problems are listed first, then applet questions, and finally general questions.

Installation

DriverManager.getConnection Gives the Error: "No suitable driver"

Make sure that the driver is registered and that you use a connection URL consistent with your JDBC driver. See Using Oracle's JDBC Drivers for the correct values.

"Unimplemented Method Interface"

You are using a a JDK 1.0.2 driver with JDK 1.1.1. Use classes102.zip for JDK 1.0.2 and classes111.zip for JDK 1.1.1.

"UnsatisfiedLinkError with OCI driver"

When using Win NT or Win95, the Java Virtual Machine complains that it cannot load **OCI804JDBC.DLL**, when one of the DLLs called by **OCI804JDBC.DLL** cannot be loaded.

The JDBC OCI drivers use shared libraries that contain the C code portions of the driver. The library is `OCI804JDBC.DLL` for the Oracle8 client program.

The shared library is normally installed in `[ORACLE_HOME]\BIN` when you install the JDBC driver from the distribution. Make sure that directory is in your **PATH**. See Installation for more details.

The shared library also depends on other libraries. If any of those DLLs are missing, you will end up with an error saying `OCI804JDBC.DLL` is missing.

You can find the list of dependent DLLs by going to the Windows Explorer program, right-clicking on the DLL, and choosing Quick View. The Quick View screen shows, among other things, the Import Table which lists the dependent DLLs.

You can reinstall missing required support files from the Oracle8.0.4 installation CD.

"ORA-12705: invalid or unknown NLS parameter value specified"

Try explicitly setting NLS_LANG. If NLS_LANG is not set or is correctly set, then you may have a client other than Oracle8.0.4. Install Oracle8.0.4 on the client.

"ORA-1019: unable to allocate memory"

You are using the OCI8 driver in an Oracle7 client installation. Use the OCI7 driver.

"Invalid driver designator"

You are using an older version of Net8. The version of Oracle on the client may be older than Oracle8.0.4. Install Oracle8.0.4 on the client.

The JDBC Drivers do not Work with Oracle Webserver 2.1 on Windows/Windows NT

You need the patch release 2.1.0.3.2 of the Oracle Webserver on Windows/NT to be able to use the JDBC drivers in the Java cartridge.

Error While Trying to Retrieve Text for Error ORA-12705.

There is no Oracle installation on the client. Install Oracle8.0.4 on the client.

Applets

FileNotFound Exception

"I am using the Thin JDBC driver. When I run my applet using Appletviewer on the local machine where the classes111.zip file is present in the CLASSPATH, my applet runs correctly. However, when I run it from a remote machine, I get a FileNotFoundException:

```
oracle.jdbc.driver.OracleDriver not found
```

The best solution is to create your own zip file, which must be un-compressed, that contains all the JDBC classes plus the classes of your application. Then in your applet you set your ARCHIVE value to point to that zip file.

How do I Use JDBC OCI in an Applet?

You can't use JDBC OCI in an applet because it uses native methods. You must use the JDBC Thin driver for applets.

Getting Security Exceptions from Netscape 3.0 when Connecting to Oracle

With Netscape 3.0 an applet using the JDBC Thin driver can only connect to an Oracle database on the same host as the web server it was downloaded from. You can solve this problem by upgrading to Netscape 4.0. See Applets for more information.

General Questions

How Do I Distinguish the Arguments to Overloaded Stored Procedures?

The ResultSet returned by the *getProceduresColumns* calls contain an additional **VARCHAR** column named OVERLOAD to distinguish overloaded procedures. Arguments belonging to the same overloaded procedure all have the same value in the OVERLOAD column.

For example if you have the following package declaration:

```
create or replace package pack is
  procedure proc (x date, y number);
  procedure proc (z number);
end p;
```
The ResultSet returned by getProceduresColumns has the following contents:

Table 1–5   Using the OVERLOAD column to distinguish between overloaded procedures

PROCEDURE_NAME	COLUMN_NAME	PACKAGE_NAME	SEQUENCE	OVERLOAD
PROC	X	PACK	1	1
PROC	Y	PACK	2	1
PROC	Z	PACK	1	2

It shows that X and Y are the first and second parameters of the first procedure (OVERLOAD is 1) and that Z is the first parameter of the second procedure (OVERLOAD is 2).

How Do I Call Stored Procedures?

See Stored Procedures and the PL/SQL samples in the samples directory.

How Can I Stream Data to and from the Database?

See Streams Tutorial and the stream samples in the samples directory.

"Stream has already been closed"

If you fetch LONG or LONG RAW data in the wrong order you can get the SQL Exception "Stream has already been closed". See Streams Tutorial for more information.

How can I Debug with Symantec Visual Cafe?

It is not possible to debug JDBC OCI programs with Symantec Visual Cafe. You can debug programs that use the JDBC Thin driver.

"ORA-01000: maximum open cursors exceeded"

The number of cursors one client can open at a time on a connection is limited (50 is the default value). Close the cursors explicitly by using method *stmt.close()*.

The JDBC Thin Driver Gives Me "invalid character" Errors for Unicode Literals

The JDBC Thin driver requires double quotes around literals that contain Unicode characters.

For example:

```
ResultSet rset = stmt.executeQuery ("select * from
\"\u6d82\u6d85\u6886\u5384\"");
```

Slow INSERT or UPDATE

By default the driver commits all INSERTs and UPDATEs as soon as you execute the statement. This is known as `autoCommit` mode in JDBC. You can get better performance by turning `autoCommit` off and using explicit COMMIT statements.

Use the setAutoCommit entry point of the *Connection* class to turn off `autoCommit`:

```
connection.setAutoCommit(false);
```

See Batching Updates for information about the Oracle extensions for batching calls to INSERT and UPDATE. Batching these commands can achieve even more speed than turning off `autoCommit`.

How do I Set the Database Wait and Rollback Options

The `waitOption` and `autoRollback` parameters control rollback options for non-fatal errors when executing statements that affect multiple rows. This is only relevant if you are batching calls to INSERT and UPDATE as described in Batching Updates.

You can set the Wait and Rollback options on a per-statement basis with the Statement setAutoRollback and setWaitOption methods.

First cast the statement to the class *oracle.jdbc.driver.OracleStatement*. We recommend you import classes from the package *oracle.jdbc.driver* to make your code more readable.

The *OracleStatement* class provides the following entry points:

- `public void setAutoRollback (int autoRollback);`

  Set the Rollback option. See table below for valid values.

- `public int getAutoRollback();`

  Return the current Rollback option for the statement.

- `public void setWaitOption(int waitOption);`

  Set the Wait option. See table below for valid values.

- `public int getWaitOption();`

  Return the current Wait option for the statement.

For example:

```
import oracle.jdbc.driver.*;
 ...
  OracleStatement s = (OracleStatement)conn.createStatement ();
  s.setWaitOption (4);
  s.setAutoRollback (2);
```

Table 1–6   Valid values for the Wait and Rollback options

Parameter	Value	Effect
waitOption	0	The program waits until the requested resource is available. This is the default setting.
waitOption	4	The driver returns an error code if a requested resource is not available.
autoRollback	0	Any error (even non-fatal) causes the current transaction to be rolled back.
autoRollback	2	A non-fatal row-level error causes only the failing row to be rolled back.