



# Making Oracle and JDBC Work For You

Presented to:

**TOUG DBA/Developer Day 2004**



**October 25, 2004**

John Jay King  
King Training Resources  
john@kingtraining.com

Download this paper and code examples from:  
<http://www.kingtraining.com>



- Understand how to use JDBC (Java DataBase Connectivity) to access Oracle data
- Know the different types of Oracle JDBC connections supported
- Be able to query data from Oracle using JDBC
- Use an Oracle stored procedure via JDBC



- Connection
- Statement
- Result Set
- Thin Client
- OCI Client



- Oracle 10g, Oracle9i and Oracle8i provide a Java Virtual Machine (JVM) built in to the database
- Oracle provides JDBC drivers to allow Java to use the database
- Oracle provides an SQLJ translator allowing the use of embedded SQL in Java
- JDeveloper is an Integrated Development Environment (IDE) that may be used to create Java programs using JDBC or SQLJ
- Common Object Request Broker Architecture (CORBA) support
- Oracle Application Server (Oracle9i AS)
- Support for JavaBeans and EJB



- JDBC is a tool used to access SQL databases from Java
- JDBC 3.0 is supported by Oracle 10g, all of JDBC 2.0 and parts of JDBC 3.0 (the current release) are supported by Oracle9i JDBC or Oracle extensions
- JDBC allows programmers to:
  - Connect to a database
  - Query and Update a database
  - Perform PL/SQL and call stored Procedures/Functions
- JDBC programs are database vendor and platform independent (mostly) unless Oracle-specific features are used



- Oracle support three types of JDBC drivers:
  - Oracle Thin JDBC driver
    - Does not require Oracle client
    - Creates its own Oracle Net/Net8/SQL\*Net connection
    - Downloadable (about 900k)
    - Used for Applets
  - Oracle OCI JDBC driver
    - Requires Oracle Client
    - Uses existing Oracle Net/Net8/SQL\*Net connection
    - Used for Applications on client and middle-tier applications
  - Oracle Server JDBC driver (aka KPRB driver)
    - Server-side only
    - Used for server-side JDBC, Java stored procedures, and Enterprise Java Beans (EJBs)
    - Supports communication between PL/SQL and Java

# Which Driver for Me?



- Use the Thin JDBC driver for:
  - Applets
  - Most applications
  - TCP/IP connections only
- Use the OCI JDBC driver for:
  - Applications requiring the best performance
  - Connections not using TCP/IP
- Use the Server-side JDBC driver for:
  - Accessing other databases from within the database
- Use the Server-side Internal JDBC driver for:
  - Programs running in the server, accessing the server



- Thin JDBC drivers and OCI JDBC drivers work with:
  - Oracle 10g
  - Oracle 9.x
  - Oracle 8.1.x
  - Oracle 8.0.x (no support for objects)
  - Oracle 7.x (no support for objects or LOBs)
- Oracle Server JDBC Internal drivers became available with Oracle 8.1.5 but can access data in Oracle 8.1.4
- Oracle Server-side Thin driver became available in 8.1.7





- Oracle's JDBC drivers add several features to the standard Java JDBC
  - JDBC 1.22 compliant, supports most JDBC 2.0 and parts of JDBC 3.0
  - Supports object-relational data
  - Supports LOB data
  - Provides Oracle-specific performance features
  - Allows use of PL/SQL and Java stored procedures
  - Supports all Oracle character sets
- Once a connection has been established, JDBC works the same regardless of driver being used



- JDBC drivers for Oracle must be available for compilation and testing (.zip files gone in Oracle 10g):
  - classes111.zip           Java 1.1 (gone in Oracle 10g)
  - classes12.zip           Java 1.2, 1.3 (gone in Oracle 10g)
  - classes12.jar           Java 1.2, 1.3 (also classes12\_g.jar)
  - ojdbc14.jar           Java 1.4 (also ojdbc14\_g.jar)
- Additional of drivers support Oracle NLS features:
  - nls\_charset11.zip       Java 1.1.x NLS characters
  - nls\_charset12.zip       Java 1.2, 1.3, 1.4 NLS characters
  - orai18n.jar           Java 1.2, 1.3, 1.4 NLS characters
- CLASSPATH names ONLY ONE SET of zip/jar files
- Zip files may be found in:<ORACLE\_HOME>/jdbc/lib



1. Import java.sql.\* and other needed packages
2. Load/register the JDBC driver
3. Connect to the database using JDBC
4. Create a statement object
5. Execute SQL statements and process results
6. Close the result set  
(technically not required, but safest to avoid memory leak issues)
7. Close the statement
8. Close the connection  
(disconnect from database)



- This is one of the two parts of JDBC that is most impacted by switching database vendors
- Make sure that the JDBC driver has been made available to the JVM (in the classpath)
- Make sure the most current driver is used
- Connect using one of two mechanisms:

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

or

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



- Connections are the other part of JDBC that is vendor-dependent, Oracle's connection string varies depending upon the driver in use
- A DBA's (DataBase Administrator's) help may be required to properly format the connect string and identify the appropriate settings
- It is a bad idea to hard-code userid and password information in code as shown in these examples, be sure to prompt the current user for the necessary information
- Be sure to close the connection, this is code that is often placed in a finally block



- Thin client connection

- jdbc:oracle:thin:@\*\*hostcomputer\*\*:\*\*port\*\*:\*\*oraclesid\*\*

```
String url = "jdbc:oracle:thin:@tecra:1521:tec817";  
String uID = "scott";           // prompt user for this  
String uPswd = "tiger";        // prompt user for this  
Connection conn =  
    DriverManager.getConnection(url,uID,uPswd);
```

- OCI client connection

- jdbc:oracle:oci8:@\*\*oraclesid\*\*

```
String url = "jdbc:oracle:oci8:@tec817";  
String uID = "scott";           // prompt user for this  
String uPswd = "tiger";        // prompt user for this  
Connection conn =  
    DriverManager.getConnection(url,uId,uPswd);
```



- Creating and destroying database connections is one of the most expensive operations JDBC performs
- Performance may sometimes be improved dramatically by using “Connection Pools”
- Connection Pools represent a set of existing database connections that application programs share, eliminating the cost of create/destroy each time a program touches the database
- Creating and managing Connection Pools is often the province of the DBA staff, though, applications may create them manually



- With Connection Pools the Application Server opens a given number of Connections to the database (dependent upon system configuration) and Java programs share them via ConnectionPool
- Caching the Connection objects provides significant performance improvements
- Connection Pools take advantage of JDBC 2.0's JNDI (Java Naming and Directory Interface) technology using ConnectionPoolDataSource, DataSource, and PooledConnection interfaces





- As with normal Connections, Connection Pool implementations tend to be vendor-specific; many people find it useful to use a “Connection Factory” as part of framework to make the database vendor transparent to the Java programs
- The use of a “Connection Factory” encapsulates (hides) the complexity of the pool connection and looks to the JDBC developer like just another database connection
- A common mistake made by developers is failing to close() the pooled connection, closing the connection makes the pooled connection available for reuse (close() is polymorphic!)



```
try {
    // Retrieve the DataSource using logical JNDI name
    ctx = new InitialContext(env);
    ds = (DataSource) ctx.lookup("jdbc/mysample");
}
catch (NamingException eName) {
    System.out.println("Error in lookup jdbc/mysample");
    System.exit(0);
}
try {
    conn = ds.getConnection(); // NOT DriverManager
    // ** database access code goes here
}
catch (SQLException e) {
    // SQL exception code
}
```



- The Statement object contains the SQL statement to be executed
- For maximum portability use generic SQL
- For maximum performance/flexibility, it might be necessary to use Oracle-specific SQL and OracleStatement object
- Be sure to close the Statement object, this is code often found in a finally block



- Standard JDBC Statement:

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

- Oracle-specific Statements:

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

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



- Immediately after a query, the result set pointer points one record before the first row returned, use the next method to get the first record
- It is not normally necessary to manually close the Result Set, it will be closed automatically when the Statement it is associated with is closed or re-executed
- Due to memory-leak issues, it may be safest to close the Result Set



- To execute a query:

```
String mySql = "SELECT ENAME, JOB, DEPTNO, "  
              + " to_char(HIREDATE, 'MM/DD/YYYY') HIREDATE "  
              + " FROM EMP "  
              + " WHERE DEPTNO = '" + myParm + "'" "  
              + " ORDER BY ENAME";  
ResultSet rs = stmt.executeQuery(mySQL);
```

- To execute an update or stored procedure call:

```
String mySQL = "UPDATE EMP SET SAL = SAL * 1.06 "  
              + " WHERE ROWID = '" + saveRowId + "'";  
int      nbrRows = stmt.executeUpdate(mySql);
```



- When updating or deleting rows previously displayed, be sure to store and use the Oracle ROWID for the column
- ROWID is the fastest possible access to a given row
- Don't show the ROWID to the user...  
(pretty scary looking...)



- Data is processed a row at time in the ResultSet using the next method:

```
while (rs.next()) { /** process SQL row **/ }
```

- Retrieve values one column at a time by name, or, by position in the SQL statement:

```
String myEmpno = rs.getString("EMPNO");
```

```
String myEmpno = rs.getString(1);
```





- Some other methods available for result set processing include:
  - afterLast Moves cursor to the end of ResultSet
  - beforeFirst Moves cursor to the front of ResultSet
  - clearWarnings Clears all warnings on ResultSet
  - close Releases ResultSet object's database and JDBC resources immediately
  - deleteRow Deletes the current row
  - first Moves cursor to first row in ResultSet
  - getWarnings Returns the first warning reported
  - insertRow Inserts a row into this ResultSet
  - last Moves cursor to last row in ResultSet
  - next Moves the cursor down one row
  - previous Moves the cursor up one row



- Some ResultSet methods useful for testing:
  - rowDeleted
  - rowInserted
  - rowUpdated
  - wasNull
- Test wasNull for **each** column value that might be null

```
myEmpno = rs.getInt("EMPNO");  
if (rs.wasNull())  
    myEmpno = 0000;  
myEname = rs.getString("ENAME");  
if (rs.wasNull())  
    myEname = " ";
```



- Some standard methods used to process column values by name or by position:
  - getBigDecimal
  - getBlob
  - getBoolean
  - getByte
  - getClob
  - getDate
  - getDouble
  - getFloat
  - getInt
  - getLong
  - getObject
  - getShort
  - getStatement
  - getString



- Oracle extensions if using OracleResultSet instead of ResultSet:
  - getArray
  - getBfile
  - getBlob
  - getClob
  - getNumber
  - getOracleObject
  - getRaw
  - getRef
  - getRowid
  - getStruct



- Oracle 9i and 10g support scrollable cursors (JDBC 2.0 and later only)
- ResultSet objects may be specified as `SCROLL_SENSITIVE` when creating Statement, PreparedStatement, and CallableStatement objects
- Scrollable Cursors add new ResultSet methods:
  - `beforeFirst()`
  - `afterLast()`
  - `first()`
  - `last()`
  - `absolute(nn)`
  - `relative(nn)`



- It is a good idea to put the Statement close and Connection close in a finally block

```
finally {
    try {
        rs.close();           // Close result set
        stmt.close();        // Close statement
        conn.close();        // Close connection
    }
    catch (SQLException e) {
        // code for error closing things
    }
}
// More precise to have try-catch for
// each "close"
```



- Sometimes a program will be doing a large number of Insert, Update, or Delete statements
- Normally, JDBC performs an automatic commit after each SQL statement's execution
- This can be controlled using methods from the Connection object



- To turn off autocommit

```
conn.setAutoCommit(false);
```

- To add commands to the batch

```
stmt.addBatch(insertUpdateDeleteStmt);
```

- To execute a batch-style command

```
int [] updateCounts = stmt.executeBatch();
```

- To manually commit or rollback

```
conn.rollback(); // conn.commit();
```





- Using Stored Procedures is a simply a matter of knowing the procedure/function name, parameters, and return type (for functions)
- CallableStatement objects or OracleCallableStatement objects are used instead of the normal Statement objects
- Input parameters are set using setString, setArray, setAsciiStream, setBigDecimal, setBinaryStream, setBlob, setBoolean, setByte, setBytes, setCharacterStream, setClob, setDate, setDate, setDouble, setFloat, setInt, setLong, setNull, setObject, setRef, setShort, setString, setTime, setTimestamp, and setUnicodeStream
- Output parameters are defined using RegisterOutParameter
- Statement executeUpdate() is used
- Get methods are used to retrieve output values



- Note use of substitution/bind variables (OK with any SQL!)
- Variables are referenced via relative position in statement

```
// Following is Oracle-specific
CallableStatement stmt1
    = conn.prepareCall("begin addem(?,?,?); end;");
// Portable Stored Procedure call follows
// CallableStatement stmt1
//    = conn.prepareCall("{CALL addem(?,?,?)}");
stmt1.setString(1,"123");
stmt1.setString(2,"456");
stmt1.registerOutParameter(3,Types.VARCHAR);
stmt1.executeUpdate();
System.out.println("Value returned is "
    + stmt1.getString(3));
```



```
CallableStatement stmt2
```

```
    = conn.prepareStatement("begin ? := times_2(?); end;");  
stmt2.registerOutParameter(1,Types.NUMERIC);  
stmt2.setString(2,"1234.56");  
stmt2.executeUpdate();  
java.math.BigDecimal outVal = stmt2.getBigDecimal(1);  
System.out.println("Value returned is " + outVal);
```



- getBigDecimal
- getBoolean
- getByte
- getBytes
- getDate
- getDouble
- getFloat
- getInt
- getLong
- getObject
- getShort
- getString
- getTime
- getTimeStamp
- registerOutParameter
- wasNull



- `getArray`
- `getBfile`
- `getBlob`
- `getClob`
- `getCursor`
- `getCustomDatum`
- `getNumber`
- `getOracleObject`
- `getRaw`
- `getRef`
- `getRowid`
- `getStruct`



- JDBC's autocommit (the default) may be a poor performer when many SQL statements are involved
  - Controlling commit/rollback manually will probably improve performance
  - If many Insert/Update/Delete operations are involved, Batched SQL might improve performance more
  - If many Insert/Update/Delete operations are involved, a Stored Procedure might work best



- SQLJ is “Pro\*Java” in appearance and functionality
  - Relatively normal SQL statements are coded
  - A pre-processor converts the SQL to two files:
    - xxx.sqlj      SQLJ source code
    - xxx.ser      SQLJ profile
  - Tends to be a better performer for “static” SQL
- SQLJ is mentioned here because it is a viable alternative to JDBC in many cases
- SQLJ has been removed from Oracle 10g



- SQLJ clauses begin with a pound (#) sign and include standard looking SQL statements
- The statement below inserts a row into the EMP table passing three column values

```
#sql {INSERT INTO EMP (EMPNO, LASTNAME, SALARY)  
      VALUES( :empId, :lastName, :empPay) };
```

- Remember how complex JDBC calls with replaceable values were?
- SQLJ is far simpler to code
- Interestingly, most Java programmers seem to prefer the JDBC code





- Oracle online manuals (<http://tahiti.oracle.com>)
  - Oracle10g/9i/8i Java Developer's Guide
  - Oracle10g/9i/8i Java Stored Procedures Developer's Guide
  - Oracle10g/9i/8i Java Tools Reference
  - Oracle10g/9i/8i JDBC Developer's Guide and Reference
  - Oracle9i/8i SQLJ Developer's Guide and Reference
  - Oracle10g/9i/8i Supplied Java Packages Reference
- Javadoc for Oracle JDBC:  
<oraclehome>/jdbc/doc/javadoc.zip
- Lots of papers and examples:  
<http://technet.oracle.com>



- JDBC access allows full use of Oracle from Java programs
- Oracle-specific features improve performance at the cost of portability
- Review of the steps:
  1. Load/register the JDBC driver
  2. Connect to the database
  3. Create a statement object
  4. Execute SQL statements and process results
  5. Close the result set (if used), statement, and connection  
(probably from a finally block)



# **Training Days 2005**

**Mark your calendar for:**

**February 9-10, 2005!**



## To contact the author:

John King

King Training Resources

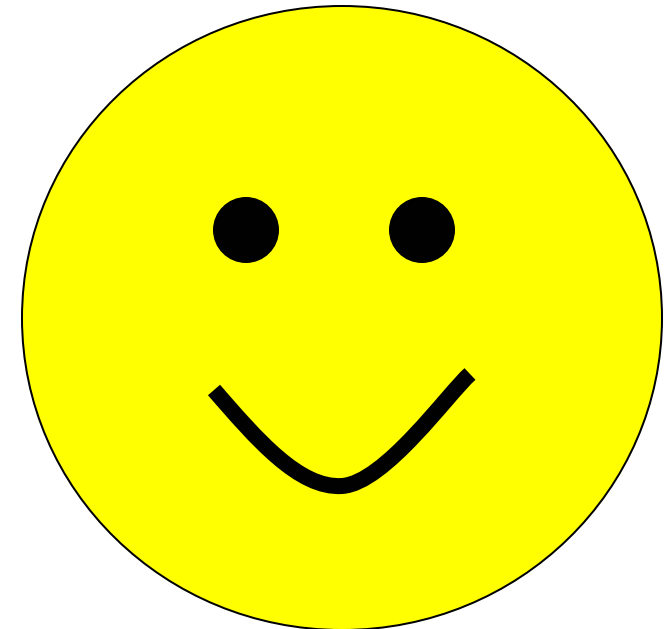
6341 South Williams Street

Littleton, CO 80121-2627 USA

1.800.252.0652 - 1.303.798.5727

Email: [john@kingtraining.com](mailto:john@kingtraining.com)

Paper & Sample Code: <http://www.kingtraining.com>



Thanks for your attention!