Standard Interfaces and Oracle Class Names

Key JDBC Classes:

ORACLE INTERFACE	SUN INTERFACE
OracleDriver	implements Driver
OracleConnection	implements Connection
OracleStatement	implements Statement
OraclePreparedStatement	implements PreparedStatement -- extends OracleStatement
OracleCallableStatement	implements CallableStatement -- extends OraclePreparedStatement
OracleResultSet	implements ResultSet
OracleResultSetMetaData	implements ResultSetMetaData
OracleDatabaseMetaData	implements DatabaseMetaData

Oracle interfaces and classes above are in package
oracle.jdbc.driver

Sun interfaces are in package
java.sql

Import Commands

For standard JDBC:
```
import java.sql.*;
```

For BigDecimal and BigInteger classes:
```
import java.math.*;
```

For Oracle implementations and extensions to JDBC:
```
import oracle.jdbc.*;
import oracle.sql.*;
```

Register Drivers

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

Open a Connection

For Thin driver (e.g., user pat, password tiger):
```
Connection conn=
  DriverManager.getConnection
  ("jdbc:oracle:thin:@<mach-name>:
  <port-no>:<sid>", "pat", "tiger");
```

For OCI driver (e.g., default database):
```
Connection conn=
  DriverManager.getConnection
  ("jdbc:oracle:oci8:@", "pat", "tiger");
```

For server-side internal driver:
```
Connection conn =
  new oracle.jdbc.
  OracleDriver().defaultConnection( );
        OR
Connection conn =
  DriverManager.getConnection("jdbc:oracle:kprb:");
```

Using a properties object, first specify the properties object (e.g., user pat, password tiger):
```
java.util.Properties info =
  new java.util.Properties( );
info.put("user", "pat");
info.put("password", "tiger");
```

Then open the connection:
```
Connection conn =
  DriverManager.getConnection
  ("jdbc:oracle:oci8:@", info);
```

Close a Connection

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

Create a Statement

To create a generic statement:
```
Statement stmt = conn.createStatement( );
```

To create a prepared statement:
```
PreparedStatement pstmt =
  conn.prepareStatement("insert into EMP
  (EMPNO, ENAME) values(?, ?)");
```

Bind the parameter and execute the query:
```
Prepared Statement pstmt =
  conn.prepareStatement("select ENAME from
  EMP where EMPNO = ?");
pstmt.setInt(1,123);
ResultSet rset = pstmt.executeQuery( );
```

Create callable statements (for stored procedure and function):
```
CallableStatement cs1 =
  conn.prepareCall( " {call proc(?,?) } " ) ;
CallableStatement cs2 =
  conn.prepareCall( " { ? = call func(?,?) } " ) ;
```

Register OUT parameters (e.g., for function call in PL/SQL block):
```
CallableStatement cstmt =
  conn.prepareCall
  ("begin ? : = funcout(?); end;");
cstmt.registerOutParameter
  (1, Types.CHAR);
cstmt.registerOutParameter
  (2, Types.CHAR);
```

Where funcout is:
```
create or replace function funcout(y out char)
  return char is
begin
  y : = 'tested';
  return 'returned';
end;
```

Close a Statement

```
stmt.close( );
pstmt.close( );
cstmt.close( );
```

Execute a Query and Process a Result Set

To execute the query (returns a result set):
```
ResultSet rset = stmt.executeQuery
  ("select ENAME from EMP");
```

Process the result set (e.g., character data in first column):
```
while(rset.next( ))
  System.out.println(rset.getString(1));
```

Close a Result Set

```
rset.close( );
```

Processing SQL Exceptions

```
try {
  while(rset.next( ))
    System.out.println(rset.getString(5));
} catch(SQLException e) {
 e.printStackTrace( ); }
```

Insert - Update - Delete

Insert new employees into EMP table:
```
PreparedStatement pstmt =
  conn.prepareStatement("insert into EMP
  (EMPNO, ENAME) values(?, ?)");
pstmt.setInt(1, 1500);
pstmt.setString(2, "PAT");
pstmt.executeUpdate( );
pstmt.setInt(1, 507);
pstmt.setString(2, "LESLIE");
pstmt.executeUpdate( );
```

Update an employee:
```
PreparedStatement psmt = conn.prepareStatement
  (update EMP set ENAME = ? where EMPNO = ?);
psmt.setString(1, "SHANNON");
psmt.setInt(2, 507);
psmt.executeUpdate( );
```

Delete an employee:
```
PreparedStatement pstmt = conn.prepareStatement
  (delete from EMP where EMPNO = ?);
pstmt.setInt(1, 507);
pstmt.executeUpdate( );
```

Stored Procedure and Function Calls

```
CallableStatement cs = conn.prepareCall
  ("begin ? : = foo(?); end;") ;
cs.registerOutParameter(1, Types.CHAR);
cs.setString(2, "aa");
cs.executeUpdate( );
```

Commit or Rollback

Default is auto-commit ON.
To commit manually, set auto-commit OFF:
```
conn.setAutoCommit(false);
```

Once auto-commit mode is disabled, then manually commit or roll back changes:
```
conn.commit( );
      OR
conn.rollback( );
```

Note: For server-side internal driver, default is auto-commit off, and setAutoCommit( ) does not work.

Datatype Mappings

SQL DATATYPES	STANDARD JAVA TYPES
CHAR	java.lang.String
VARCHAR2	java.lang.String
LONG	java.lang.String
NUMBER	java.math. BigDecimal
NUMBER	boolean
NUMBER	byte
NUMBER	short
NUMBER	int
NUMBER	long
NUMBER	float
NUMBER	double
RAW	byte [ ]
LONGRAW	byte [ ]
DATE	java.sql.Date
DATE	java.sql.Time
DATE	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
user-defined object	java.sql.Struct
user-def. reference	java.sql.Ref
user-def. collection	java.sql.Array

Oracle type extensions:

BFILE (maps to oracle.sql.BFILE)
ROWID (maps to oracle.sql.ROWID)
REF CURSOR types (map to java.sql.ResultSet)
oracle.sql.* mapping classes are also available for all of the above datatypes for faster, more precise processing.

Note: Typecodes are specified in oracle.jdbc.OracleTypes
For standard types, definitions duplicate those in java.sql.Types

Streams

Long columns in JDBC are streamed.
Set a stream column:

```
pstmt.setAsciiStream(1, <input-stream>,
  <input-stream-length>);
```

If string data is in character format, use setCharacterStream( ):

```
pstmt.setCharacterStream(1, <input-stream>,
  <input-stream-length>);
```

For long raw columns, use setBinaryStream( ):

```
psmt.setBinaryStream(1, <input-stream>,
  <input-stream-length>);
```

Retrieve a stream column:

```
ResultSet rset =
  stmt.executeQuery
  ("select * from streamexample");
InputStream ascii_data =
  rset.getAsciiStream(1);
int c;
while((c = ascii_data.read(byte[ ] b)) ! = -1)
  System.out.println(b);
```

LOBs

Read a piece of a LOB (inputting result set column numbers in setXXX( ) calls):

```
BLOB blob =
  ((OracleResultSet)rset).getBLOB(1);
byte[ ] bytes =
  blob.getBytes(<begin_index>, <length>);
CLOB clob =
  ((OracleResultSet)rset).getCLOB(2);
String str =
  clob.getSubString(<begin index>, <length>);
BFILE bfile =
  ((OracleResultSet)rset).getBFILE(3);
byte[ ] bytes =
  bfile.getBytes(<begin_index>, <length>);
```

Read the LOB content as a stream:

```
BLOB blob =
  ((OracleResultSet)rset).getBLOB(1);
InputStream input_stream =
  blob.getBinaryStream( );
input_stream.read(...);
CLOB clob=
  ((OracleResultSet)rset).getCLOB(2);
InputStream input_stream =
  clob.getAsciiStream( );
input_stream.read(...);
BFILE bfile =
  ((OracleResultSet)rset).getBFILE(3);
InputStream input_stream =
  bfile.getBinaryStream( );
input_stream.read(...);
```

Write specified amount of data into a LOB:

```
BLOB blob=
  ((OracleResultSet)rset).getBLOB(1);
byte[ ] data = ... ;
int amount_written =
  blob.putBytes(<begin_index>, data);
CLOB clob=
  ((OracleResultSet)rset).getCLOB(2);
String data = ... ;
int amount_written =
  clob.putString(<begin_index>, data);
```

Note: begin_index starts with 1, not 0.

Replace the LOB content from a stream:

```
CLOB clob =
  ((OracleResultSet)rset).getCLOB(2);
Writer char_stream =
  clob.getCharacterOutputStream( );
char_stream.write(...);
BLOB blob =
  ((OracleResultSet)rset).getBLOB(1);
OutputStream output_stream =
  blob.getBinaryOutputStream( );
output_stream.write(...);
CLOB clob =
  ((OracleResultSet)rset).getCLOB(2);
OutputStream output_stream =
  clob.getAsciiOutputStream( );
output_stream.write(...);
```

Get a LOB length.

```
long length = blob.length( );
long length = clob.length( );
long length = bfile.length( );
```

Performance Enhancements

Oracle update batching --
Set connection batch size (acts as default for statements):

```
((OracleConnection)conn).
  setDefaultExecuteBatch(15);
```

Set statement batch size:

```
((OraclePreparedStatement)ps).
  setExecuteBatch(20);
```

Explicitly send the row to the server in batch mode:

```
int = ((OracleStatement)stmt).sendBatch( );
```

(Oracle JDBC also supports standard update batching.)

Oracle row prefetching --
Set connection prefetch size (default 10) (acts as default for statements):

```
((OracleConnection)conn).
  setDefaultRowPrefetch(15);
```

Set statement prefetch size:

```
((OracleStatement)stmt).setRowPrefetch(20);
```