**CS 580 Client-Server Programming**  
Fall Semester, 2000  
Doc 21 JDBC  

**Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL and Java</td>
<td>2</td>
</tr>
<tr>
<td>JDBC</td>
<td>4</td>
</tr>
<tr>
<td>JDBC Architecture</td>
<td>7</td>
</tr>
<tr>
<td>Using JDBC</td>
<td>9</td>
</tr>
<tr>
<td>JDBC Drivers</td>
<td>10</td>
</tr>
<tr>
<td>Queries</td>
<td>14</td>
</tr>
<tr>
<td>Transactions</td>
<td>21</td>
</tr>
<tr>
<td>PreparedStatement</td>
<td>23</td>
</tr>
<tr>
<td>CallableStatement</td>
<td>25</td>
</tr>
</tbody>
</table>

**References**

http://www.sdsu.edu/doc/java-1.1.1/doc/guide/jdbc/index.html,  
Sun’s on-line JDBC Tutorial/Documentation

Client/Server Programming with Java and CORBA, Orfali and Harkey, John Wiley and Sons, Inc. 1997


JDBC Documentation at:  

**Copyright ©, All rights reserved.**

2000 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA.  
OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.
SQL and Java
Some Jargon

SQL Access Group (SAG) - multivendor "Standards" group

SQL Call Level Interface (CLI)
SAG standard for remote connects to a database

CLI uses drivers to the database

Program uses a driver manager to talk to the driver

The driver is database specific

In 1994 X/Open adopted SQL CLI to produce X/Open CLI

In 1996 X/Open CLI was adapted by ISO to become ISO 9075-3 Call level Interface
Microsoft's Open Database Connectivity (ODBC)

Extension of the SAG CLI

ODBC 2.0 (32 bit) has three conformance levels

- Core
  23 API calls for basic SQL stuff

- Level 1
  19 API calls for large objects (BLOBs) and driver-specific

- Level 2
  19 API calls for scrolling (cursors)
JDBC
Java Database Connectivity

Sun states
  JDBC is a trademark and
  Not an abbreviation for Java Database Connectivity

JDBC is a portable SQL CLI written in Java.

Versions of JDBC
- JDBC 1.x
- JDBC 2.x
- JDBC 3.0

JDBC 1.x

Basic SQL functionality

MM MySQL stable driver only supports 1.x
JDBC 2.1 Core

Standard part of JDK 1.2

JDBC drivers must implement JDBC 2.x before you can use it

MySQL driver for JDBC 2.x is in pre-beta release

Additional Features
• Scrollable result sets
• Updateable result sets
  Can change the result of a query locally & in database
• Batch updates
• BLOB, CLOB support

JDBC 2.0 Optional Package (javax.sql)

Optional Java package

Downloadable from

• Java Naming and Directory Interface (JNDI) support
• Connection pooling
• Distributed transactions
• JavaBean RowSets
  Access any tabular data (files, spreadsheets)
  Make old drivers scrollable & updateable
  Wraps JDBC driver for use in GUI
JDBC 3.0

In final draft form

- Set, release, or rollback a transaction to designated savepoints
- Reuse of prepared statements by connection pools
- Connection pool configuration
- Retrieval of parameter metadata
- Retrieval of auto-generated keys
- Ability to have multiple open ResultSet objects
- Passing parameters to CallableStatement objects by name
- Holdable cursor support
- BOOLEAN data type
- Making internal updates to the data in Blob and Clob objects
- Retrieving and updating the object referenced by a Ref object
- Updating of columns containing BLOB, CLOB, ARRAY and REF types
- DATALINK/URL data type
- Transform groups and type mapping
- DatabaseMetadata APIs
JDBC Architecture

Java Program -> JDBC Driver Manager -> Oracle Driver -> Oracle Database

JDBC-ODBC Driver -> ODBC Database

MM MySQL Driver -> MySQL Database

e tc.

JDBC driver provides connections to database via drivers
import java.sql.*;

public class SampleMySQL
{
    public static void main(String[] args) throws Exception
    {
        Class.forName("org.gjt.mm.mysql.Driver").newInstance();
        String database = "jdbc:mysql://fargo.sdsu.edu:5555/WHITNEYR";
        String user = "WHITNEYR";
        String password = "Top_Secret:)";
        Connection toFargo =
            DriverManager.getConnection(database, user, password);

        Statement namesTable = toFargo.createStatement();

        ResultSet namesFound =
            namesTable.executeQuery("SELECT * FROM name");
        while (namesFound.next() )
        {
            System.out.print( "first: " + namesFound.getString( 1));
            System.out.println( "\tlast: " + namesFound.getString( 2));
        }
        toFargo.close();
    }
}
Using JDBC

Step 1. Load the driver(s)

Step 2. Connect to the database

Step 3. Issue queries and receive results

Loading a Driver

A well-written JDBC driver is loaded using Class.forName

To load the Oracle driver

import java.sql.*;

class JdbcTest
{
    public static void main (String args []) throws 
        ClassNotFoundException
    {
        Class.forName ("oracle.jdbc.OracleDriver");
    }
}

This requires that oracle package be in your path

A properly written driver will register itself with the DriverManager class
JDBC Drivers

Java supports four types of JDBC drivers

1. JDBC-ODBC bridge plus ODBC driver
   Java code access ODBC native binary drivers
   ODBC driver accesses databases
   ODBC drivers must be installed on each client

2. Native-API partly-Java driver
   Java code accesses database specific native binary drivers

3. JDBC-Net pure Java driver
   Java code accesses database via DBMS-independent net protocol

4. Native-protocol pure Java driver
   Java code accesses database via DBMS-specific net protocol
**JDBC URL Structure**

jdbc:<subprotocol>:<subname>

<subprotocol>
   Name of the driver or database connectivity mechanism

<subname>
   Depends on the <subprotocol>, can vary with vendor
   If connection goes over Internet subname is to contain net URL
   jdbc:mysql://fargo.sdsu.edu:5555/WHITNEYR

**ODBC Subprotocol**

jdbc:odbc:<data-source-name>[:<attribute-name>=<attribute-value>]*

Examples

jdbc:odbc:qeor7
jdbc:odbc:wombat
jdbc:odbc:wombat;CacheSize=20;ExtensionCase=LOWER
jdbc:odbc:qeora;UID=kgh;PWD=fooey
DriverManager.getConnection - Using JDBC URL

Three forms:

getConnection(URL, Properties)
getConnection(URL, userName, Password)
getConnection(URLWithUsernamePassword)

Form 1
static String ARS_URL = "jdbc:oracle:@PutDatabaseNameHere";

DriverManager.getConnection(ARS_URL, "whitney","secret");

Form 2
DriverManager.getConnection("jdbc:oracle:whitney/secret@PutDatabaseNameHere");

Form 3
java.util.Properties info = new java.util.Properties();
info.addProperty ("user", "whitney");
info.addProperty ("password","secret");

DriverManager getConnection (ARS_URL ,info );
java.sql.DriverManager

Driver related methods
deregisterDriver(Driver)
getDriver(String)
getDrivers()
registerDriver(Driver)

Connecting to a database
getConnection(String, Properties)
getConnection(String, String, String)
getConnection(String)

getLoginTimeout()
setLoginTimeout(int)

Logging/tracing/Debugging
getLogStream()
setLogStream(PrintStream)
println(String)
   Print a message to the current JDBC log stream
Queries

Connection toFargo =
    DriverManager.getConnection(database, user, password);

Statement namesTable = toFargo.createStatement();

ResultSet namesFound =
    namesTable.executeQuery("SELECT * FROM name");

executeUpdate
    Use for INSERT, UPDATE, DELETE or SQL that return nothing

executeQuery
    Use for SQL (SELECT) that return a result set

execute
    Use for SQL that return multiple result sets
    Uncommon
    Stored procedures can return
ResultSet - Result of a Query

JDBC returns a ResultSet as a result of a query

A ResultSet contains all the rows and columns that satisfy the SQL statement

A cursor is maintained to the current row of the data

The cursor is valid until the ResultSet object or its Statement object is closed

next() method advances the cursor to the next row

You can access columns of the current row by index or name

ResultSet has getXXX methods that:

- have either a column name or column index as argument
- return the data in that column converted to type XXX
Some Result Set Issues

What happens when we call next() too many times?

What happens before we call next

Example

Name Table

<table>
<thead>
<tr>
<th>first</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>roger</td>
<td>whitney</td>
</tr>
<tr>
<td>pete</td>
<td>stanley</td>
</tr>
<tr>
<td>rat</td>
<td>cat</td>
</tr>
</tbody>
</table>

Sample Table

<table>
<thead>
<tr>
<th>col</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>
Two Queries

public class SampleMySQL {
    public static void main(String[] args) throws Exception {
        Class.forName("org.gjt.mm.mysql.Driver").newInstance();
        String database = "jdbc:mysql://fargo.sdsu.edu:5555/foo";
        Connection toFargo =
            DriverManager.getConnection(database, "foo", "bar");
        Statement namesTable = toFargo.createStatement();
        ResultSet namesFound =
            namesTable.executeQuery("SELECT * FROM name");
        for (int k = 0;k< 3;k++) {
            System.out.println( "first: " + namesFound.getString( 1));
            namesFound.next();
        }
        for (int k = 0;k< 3;k++) {
            sample.next();
            System.out.println( "col: " + sample.getString( 1));
        }
        toFargo.close();
    }
}

Result

first: roger
first: roger
first: pete
col: a
col: b
col: b
Mixing ResultSets

Can't have two active result sets on same statement

Statement namesTable = toFargo.createStatement();

ResultSet namesFound =
    namesTable.executeQuery("SELECT * FROM name");
ResultSet sample =
    namesTable.executeQuery("SELECT * FROM sample");
for (int k = 0;k< 3;k++) {
    namesFound.next();
    sample.next();
    System.out.println( "first: " + namesFound.getString( 1));
    System.out.println( "col: " + sample.getString( 1));
}

Result

first: roger
col: a
first: roger
col: b
first: roger
col: b
Use Two Statements

Connection toFargo =
   DriverManager.getConnection(database, user, password);

Statement namesTable = toFargo.createStatement();
Statement exampleTable = toFargo.createStatement();

ResultSet namesFound =
   namesTable.executeQuery("SELECT * FROM name");
ResultSet sample =
   exampleTable.executeQuery("SELECT * FROM sample");
for (int k = 0;k< 3;k++) {
   namesFound.next();
   sample.next();
   System.out.println("first: " + namesFound.getString( 1));
   System.out.println("col: " + sample.getString( 1));
}

Result
first: roger
col: a
first: pete
col: b
first: rat
col: b
SQL Data Types and Java

<table>
<thead>
<tr>
<th>SQL type</th>
<th>Java type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>String</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>String</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>String</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>BIT</td>
<td>boolean</td>
</tr>
<tr>
<td>TINYINT</td>
<td>byte</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>long</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>FLOAT</td>
<td>double</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>BINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>byte[]</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
</tr>
</tbody>
</table>
Transactions

A transaction consists of one or more statements that have been executed and completed

A transaction ends when a commit or rollback is sent

Connections are opened in auto commit mode:

  when a statement is completed, it is committed

Transactions and Concurrency

What happens to data that is changed in a transaction, but not yet committed?

Can other programs access the old or new values?

Use setTransactionIsolation(int) in Connection class to set access levels

Access levels are given as static fields of Connection class

 TRANSACTION_NONE
 TRANSACTION_READ_COMMITTED
 TRANSACTION_READ_UNCOMMITTED
 TRANSACTION_REPEATABLE_READ
 TRANSACTION_SERIALIZABLE
import java.sql.*;
import java.io.*;

class JdbcTest {
    static String ARS_URL = "jdbc:oracle:@PutDatabaseNameHere";

    public static void main (String args []) throws SQLException, ClassNotFoundException, IOException {
        Class.forName ("oracle.jdbc.OracleDriver");
        Connection ARS;
        ARS = DriverManager.getConnection(ARS_URL, "whitney", "secret");
        ARS.setAutoCommit(false);

        String floodProblem = DELETE FROM AirlineSchedule WHERE from = 'FAR';

        String newflight = INSERT INTO AirlineSchedule VALUES ( 'DE', 'SAN', '8:00', '12:00', '909', 'A');

        Statement schedule = ARS.createStatement ();
        schedule.executeUpdate (floodProblem);
        schedule.executeUpdate (newflight);
        ARS.commit();
        ARS.close();
    }
}
**PreparedStatement**

PreparedStatement objects contain SQL statements that have been sent to the database to be prepared for execution.

The SQL statements contain variables (IN parameters) which are given values before statement is executed.

Only makes sense to use if database and driver keeps statements open after they have been committed.

IN parameters are indicated by a ?

Values are set by position.

String flightOut = "SELECT * FROM AirlineSchedule WHERE from = ?";
import java.sql.*;
import java.io.*;

class JdbcTest {
    static String ARS_URL = "jdbc:oracle:@PutDatabaseNameHere";

    public static void main (String args []) throws SQLException, ClassNotFoundException, IOException {
        Class.forName ("oracle.jdbc.OracleDriver");
        Connection ARS;
        ARS = DriverManager.getConnection(ARS_URL, "whitney", "secret");

        String flightOut =    "SELECT * FROM AirlineSchedule
                  WHERE from = ?";

        PreparedStatement schedule;
        schedule = ARS.prepareStatement (flightOut);

        schedule.setObject( 1, "SAN" );
        ResultSet fromSanDiego = schedule.executeQuery ();

        schedule. clearParameters();
        schedule.setObject( 1, "LAX" );
        ResultSet fromLA = schedule.executeQuery ();
    }
}
CallableStatement

Some databases have stored procedures (a procedure of SQL statements)

CallableStatement allows a Java program to invoke a stored procedure in a database

DatabaseMetaData

The class DatabaseMetaData allows you to obtain information about the database

The 113 methods in DatabaseMetaData gives you more information than you thought possible