# Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL and Java</td>
<td>3</td>
</tr>
<tr>
<td>JDBC</td>
<td>5</td>
</tr>
<tr>
<td>JDBC Architecture</td>
<td>9</td>
</tr>
<tr>
<td>Using JDBC</td>
<td>14</td>
</tr>
<tr>
<td>JDBC Drivers</td>
<td>17</td>
</tr>
<tr>
<td>Queries</td>
<td>23</td>
</tr>
<tr>
<td>SQL Data Types and Java</td>
<td>27</td>
</tr>
<tr>
<td>Threads &amp; Connections</td>
<td>31</td>
</tr>
<tr>
<td>Transactions</td>
<td>32</td>
</tr>
<tr>
<td>PreparedStatement</td>
<td>35</td>
</tr>
<tr>
<td>Batch Updates</td>
<td>37</td>
</tr>
<tr>
<td>CallableStatement</td>
<td>38</td>
</tr>
</tbody>
</table>

*Copyright ©, All rights reserved. 2005 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent ([http://www.opencontent.org/opl.shtml](http://www.opencontent.org/opl.shtml)) license defines the copyright on this document.*
References


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


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)

ODBC 3.0

• Unicode
• Aligns ODBC closer to ISO SQL-92 CLI
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 (In JDK 1.4)
• JDBC 4.0

JDBC 1.x

Basic SQL functionality
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 Package

Now java.sql

Once was optional Java package javax.sql

• 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

java.sql & javax.sql in JDK 1.4

Most advanced features are in javax.sql

- 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

Note Database must support a feature for JDBC can use it
JDBC Architecture

Java Program -> JDBC Driver Manager
    |                  JDCB-ODBC Driver -> ODBC Database
    |                  Oracle Driver -> Oracle Database
    |                  MM MySQL Driver -> MySQL Database
    |                  etc.

JDBC driver provides connections to database via drivers
JDBC Drivers, JDBC Versions & Java API

java.sql.* is mainly interfaces for JDBC Drivers

The driver for the database determines the actual functionality
Sample JDBC Use

```java
import java.sql.*;

public class SampleConnection
{
    public static void main (String args[])
        throws Exception
    {
        String dbUrl = "jdbc:mysql://rugby.sdsu.edu:8777/test";
        String user = "whitney";
        String password = "mylittleSecret";
        System.out.println("Load Driver!");

        Class.forName("com.mysql.jdbc.Driver");
        Connection rugby;
        rugby = DriverManager.getConnection(dbUrl, user, password);
        Statement getTables = rugby.createStatement();
        ResultSet tableList =
            getTables.executeQuery("SELECT * FROM name");
        while (tableList.next() )
            System.out.println("Last Name: " + tableList.getString(1) + \t + "First Name: " + tableList.getString("first_name"));
        rugby.close();
    }
}
```
DownLoading the MySQL JDBC Driver


Contains the jar file mysql-connector-java-3.0.10-stable-bin.jar

Needs to be in your classpath

Implements JDBC 3.0, but some of the advance features are not functional

See the documentation at:

Downloading PostgreSQL JDBC Driver

PostgreSQL JDBC Driver web site

Download site

Course accounts use PostgreSQL 7.4

So use: pg74.215.jdbc3.jar

Driver 8.0-311 JDBC 3 also seems to work
Using JDBC

Step 1. Load the driver(s)

Step 2. Connect to the database

Step 3. Issue queries and receive results
Loading a Driver

The most commonly used way

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

To load the Oracle driver

```java
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
Loading a Driver

The Recommended Way

Use the command line to specify the driver

java –Djdbc.drivers=org.postgresql.Driver yourProgramName

Makes it easier to change database vendors without recompiling the code

Long command lines need script to run
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:qegra;UID=kgh;PWD=fooey
PostgreSQL Subprotocol

jdbc:postgresql:database
jdbc:postgresql://host/database
jdbc:postgresql://host:port/database

MySQL Subprotocol

jdbc:mysql://[host][,failoverhost...][:port]/[database]
[?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]...

hostname defaults to 127.0.0.1
port defaults to 3306

Legal values for properties can be found at:
http://www.mysql.com/documentation/connector-j/
index.html#id2800782
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
**DataSource**

The current recommended way to get a connection is using a DataSource

In theory DataSource supports connection pooling

DataSources are optional in JDBC 2.0

MySql DataSource is a wrapper for DriverManager
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
```java
public static void main (String args[]) throws Exception {

    String dbUrl = "jdbc:mysql://rugby.sdsu.edu:8777/test";
    String user = "whitney";
    String password = "SCECmysql";
    System.out.println("Load Driver!");

    Class.forName("com.mysql.jdbc.Driver");
    Connection rugby = DriverManager.getConnection(dbUrl, user, password);
    Statement getTables = rugby.createStatement();
    int rowsModified =
        getTables.executeUpdate("UPDATE name
                           SET first_name = 'foo' WHERE last_name='Whitney'");
    System.out.println("Number of rows modified: " + rowsModified);

    rugby.close();
}
```
**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
**getObject**

A replacement for the getXXX methods

Rather than

```java
ResultSet tableList = getTables.executeQuery("SELECT * FROM name");
String firstName = tableList.getString(1);
```

Can use

```java
ResultSet tableList =
    getTables.executeQuery("SELECT * FROM name");
String firstName = (String) tableList.getObject(1);
```

g getObject( int k) returns the object in the k’th column of the current row

g getObject( String columnName) returns the object in the named column
### 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>
Some Result Set Issues

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

What happens if we try to access data before we call next?

In both cases an java.sql.SQLException is thrown
Mixing ResultSets

Can't have two active result sets on same statement

```java
Statement namesTable = toFargo.createStatement();
Connection rugby;
rugby = DriverManager.getConnection( dbUrl, user, password);
Statement getTables = rugby.createStatement();
ResultSet count =
    getTables.executeQuery("SELECT COUNT(*) FROM name");
ResultSet tableList =
    getTables.executeQuery("SELECT * FROM name");

while (tableList.next() )
    System.out.println("Last Name: "+ tableList.getObject(1) + 	"First Name: " + tableList.getObject("first_name"));

// Raises java.sql.SQLException
count.getObject(1);

rugby.close();
```

this can happen two threads have access to the same statement
Two Statements on one Connection work

Connection rugby;
rugby = DriverManager.getConnection( dbUrl, user, password);
Statement getTables = rugby.createStatement();
Statement tableSize = rugby.createStatement();

ResultSet count =
    getTables.executeQuery("SELECT COUNT(*) FROM name");
ResultSet tableList =
    tableSize.executeQuery("SELECT * FROM name");

while (tableList.next() )
    System.out.println("Last Name: "+ tableList.getObject(1) + \\
                     \"First Name: "+ tableList.getObject("first_name"));

count.next();
System.out.println("Count: " + count.getObject(1) );
count.close();
tableList.close();
rugby.close();
Threads & Connections

Some JDBC drivers are not thread safe

If two threads access the same connection results may get mixed up

PostgreSQL & MySql drivers are thread safe

When two threads make a request on the same connection

• The second thread blocks until the first thread get it its results

Can use more than one connection but

• Each connection requires a process on the database
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.*;

public static void main (String args[]) throws Exception
{
    String dbUrl = "jdbc:mysql://rugby.sdsu.edu:8777/test";
    String user = "whitney";
    String password = "SCECmysql";
    System.out.println("Load Driver!");
    Class.forName("com.mysql.jdbc.Driver");

    Connection rugby;
    rugby = DriverManager.getConnection( dbUrl, user, password);
    rugby.setAutoCommit(false);
    Statement update  = rugby.createStatement();
    update.executeUpdate("INSERT INTO name
            (first_name, last_name) VALUES ( 'Donald', 'Duck') ");
    update.executeUpdate("INSERT INTO name
            (first_name, last_name) VALUES ( 'Micky', 'Mouse') ");
    rugby.rollback();
    update.close();
    Statement getTables  = rugby.createStatement();
    ResultSet tableList =
        getTables.executeQuery("SELECT * FROM name");
    while (tableList.next() )
        System.out.println("Last Name: " + tableList.getObject(1) + "\t" +
            "First Name: " + tableList.getObject( "first_name"));
    rugby.close();
}
MySql & Transactions

Note that MySql does not support transactions unless it is a recent version and is configured to do so.

There is a test to see if the database supports transactions

However, it did return the incorrect answer when I tested the code

```java
Connection rugby;
rugby = DriverManager.getConnection( dbUrl, user, password);
rugby.setAutoCommit(false);
boolean hasTransactions =
   ((com.mysql.jdbc.Connection)rugby).supportsTransactions();
System.out.println( "Supports transactions? " + hasTransactions);
```
**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.

```java
String flightOut = "SELECT * FROM AirlineSchedule WHERE from = ?";
```
PreparedStatement Example

```java
import java.sql.*;
import java.io.*;

Connection rugby;
rugby = DriverManager.getConnection( dbUrl, user, password);

String findPerson =
    "SELECT * FROM name WHERE last_name = ?";
PreparedStatement find = rugby.prepareStatement( findPerson );
find.setObject( 1, "whitney" );

ResultSet person =  find.executeQuery();
while (person.next() )
    System.out.println("Last Name: "+ person.getObject(1) + "\t" +
                       "First Name: "+ person.getObject("first_name"));

find.clearParameters();
find.setObject( 1, "olson" );

person = find.executeQuery();
while (person.next() )
    System.out.println("Last Name: " + person.getObject(1) + "\t" +
                       "First Name: " + person.getObject("first_name"));

rugby.close();
```
**Batch Updates**

Allows one to send a batch of updates in one request

```java
Connection rugby;
rugby = DriverManager.getConnection(dbUrl, user, password);

Statement batch = rugby.createStatement();
batch.addBatch("INSERT INTO name (first_name, last_name) VALUES ('Lion', 'King')");
batch.addBatch("INSERT INTO name (first_name, last_name) VALUES ('Little', 'Nemo')");

int[] updates = batch.executeBatch();

for (int k = 0; k < updates.length; k++)
   System.out.println("Update "+ k + " rows affected "+ updates[k]);

rugby.close();
```
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 give you more information than you thought possible