References


import java.sql.*;
public class Test {
    public static void main(String[] args) throws Exception {
        Class.forName("org.sqlite.JDBC");
        Connection conn = DriverManager.getConnection("jdbc:sqlite:test.db");
        Statement stat = conn.createStatement();
        stat.executeUpdate("drop table if exists people;");
        stat.executeUpdate("create table people (name, occupation);");
        PreparedStatement prep = conn.prepareStatement("insert into people values (?, ?);");
        prep.setString(1, "Gandhi");
        prep.setString(2, "politics");
        prep.addBatch();
        conn.setAutoCommit(false);
        prep.executeBatch();
        conn.setAutoCommit(true);

        ResultSet rs = stat.executeQuery("select * from people;");
        while (rs.next()) {
            System.out.println("name = " + rs.getString("name"));
            System.out.println("job = " + rs.getString("occupation"));
        }
        rs.close();
        conn.close();
    }
}

This is the old way (Pre Java 6) of connecting.
SqliteJDBC Documentation

http://www.zentus.com/sqlitejdbc/
Drivers must be in your classpath
JDBC Drivers

Java supports four types of JDBC drivers

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

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

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

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

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

Sqlite
jdbc:sqlite:filename

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

In your code
Class.forName("com.mysql.jdbc.Driver");

Command line
java -Djdbc.drivers=org.postgresql.Driver
yourProgramName
Loading Driver - Java 6, JDBC 4

Auto discovery

String dbUrl = "jdbc:postgresql://bismarck.sdsu.edu/test";
String user = "whitney";
String password = "mysecret";
Connection bismarck = DriverManager.getConnection( dbUrl, user, password);
Statement getTables = bismarck.createStatement();
ResultSet tableList = getTables.executeQuery("SELECT * FROM names");
while (tableList.next() )
    System.out.println("Last Name: " + tableList.getString(1) + \t\t"First Name: " + tableList.getString( "first_name"));
bismarck.close();

Java 6 introduces auto discovery. We don't have to call Class.forName(). This requires Java 6 and JDBC 4 compliant drivers.
**DriverManager.getConnection**

Three forms:

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

Form 1

```java
static String ARS_URL = "jdbc:oracle:@PutDatabaseNameHere";

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

Form 2

```java
DriverManager.getConnection(
    "jdbc:oracle:whitney/secret@PutDatabaseNameHere");
```

Form 3

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

DriverManager getConnection (ARS_URL ,info );
```
java.sql verses javax.sql

java.sql
DriverManager

javax.sql
DataSource
Connection Pools
Distributed
Transactions
Normally uses JNDI
JNDI
Java Naming and Directory Interface

Need JNDI Service Provider

http://java.sun.com/docs/books/tutorial/jndi/overview/index.html
Queries

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
ResultSet

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);
```

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

ggetObject(String columnName) returns the object in the named column
<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
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 when 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) + 't' +
        "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
SQLiteJDBC Issues

Only one Connection can write to the database at a time

No write can finish while there is an open reader

So

Close resultsets as soon as you can
SQLiteJDBC Issues

Queries are cheap
SQLiteJDBC Issues

Use PreparedStatement

PreparedStatement prep = conn.prepareStatement("insert into mytable values (?)");

Converted into SQLite form
(true for most databases)

Second use is fast
SQLiteJDBC Issues

Transactions are Good

Slow

PreparedStatement prep = conn.prepareStatement( "insert into mytable values (?);" );
for (int i=0; i < 10000; i++) {
    prep.setInt(1, i);
    prep.executeUpdate();
}

Fast

PreparedStatement prep = conn.prepareStatement( "insert into mytable values (?);" );
conn.setAutoCommit(false);
for (int i=0; i < 10000; i++) {
    prep.setInt(1, i);
    prep.executeUpdate();
}
conn.commit();