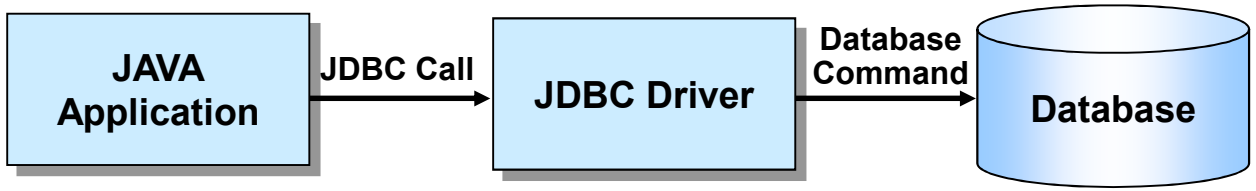
JDBC (Java Database Connectivity)

Database System Concepts, 6th Ed.

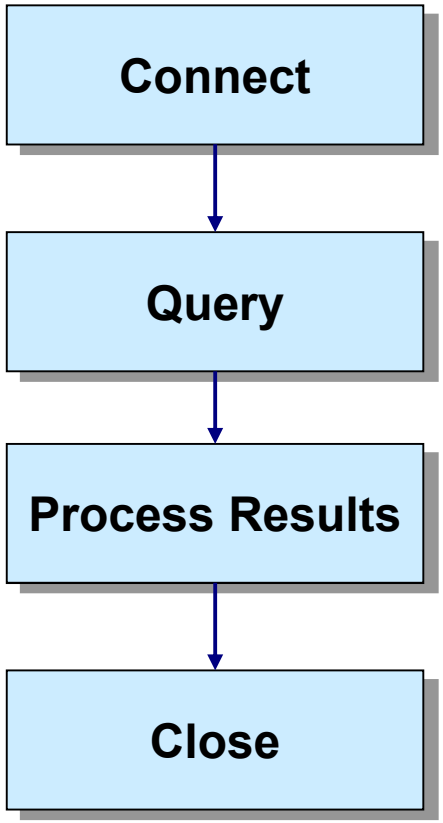
JDBC

- Java API for communicating with database systems supporting SQL
  - Supports a variety of features for querying and updating data, and for retrieving query results
  - Also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes

- What does JDBC do?
  - Establish a connection with a database
  - Send SQL statements
  - Process the results



JDBC Programming Steps



1) Register the driver
2) Create a connection to the database

1) Create a statement
2) Query the database

1) Get a result set
2) Assign results to Java variables

1) Close the result set
2) Close the statement
3) Close the connection

Skeleton Code

```
import java.sql.*;

Class.forName(DRIVERNAME);

Connection con = DriverManager.getConnection(
                    CONNECTIONURL, DBID, DBPASSWORD);

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM member);

While(rs.next())  {
      Int x = rs.getInt("a");
      String s = rs.getString("b");
      Float f = rs.getFloat("c");
}

rs.close();
stmt.close();
con.close();
```

Loading a JDBC driver

Connecting to a database

Executing SQL

Processing the result set

Closing the connections

Step 1 : Loading a JDBC Driver

- A JDBC driver is needed to connect to a database
- Loading a driver requires the class name of the driver
  - Tibero: com.tmax.tibero.jdbc.TbDriver
    - Add $TB_HOME/client/lib/jar/tibero5-jdbc.jar to Java classpath
    - Import tbJDBC package in Java file

      import com.tmax.tibero.jdbc.;*
      import com.tmax.tibero.jdbc.ext.;*
  - Oracle: oracle.jdbc.driver.OracleDriver
  - MySQL: com.mysql.jdbc.Driver
- Loading the driver class

  Class.forName("com.tmax.tibero.jdbc.TbDriver");
- It is possible to load several drivers
- The class *DriverManager* manages the loaded driver(s)

Step 2 : Connecting to a Database

- JDBC URL for a database
  - Identifies the database to be connected
  - Consists of three-part:

jdbc:<subprotocol>:<subname>

Protocol: JDBC is the only protocol in JDBC

Sub-protocol: identifies a database driver

Subname: indicates the **location** and **name** of the database to be accessed. **Syntax is driver specific**

- Creating a *Connection* object (in java.sql.*)

```
Connection conn =
   DriverManager.getConnection("jdbc:tibero:thin:@localhost:8629:tibero",
                               "tibero", "tmax");
```

(server IP):(port):(SID)

DB user id, password

- *DriverManager*
  - Allows you to connect to a database using the specified JDBC driver, database location, database name, username and password
  - Returns a *Connection* object which can then be used to communicate with the database

Step 3 : Executing SQL

- *Statement* object (in java.sql.*)
  - Sends SQL to the database to be executed
  - Can be obtained from a *Connection* object

    Statement statement = conn.createStatement();

- *Statement* has three methods to execute a SQL statement:
  - **executeQuery()** for QUERY statements
    - Returns a *ResultSet* which contains the query results

```
ResultSet rset = stmt.executeQuery
  ("select RENTAL_ID, STATUS from ACME_RENTALS");
```

  - **executeUpdate()** for INSERT, UPDATE, DELETE, or DDL statements
    - Returns an integer, the number of affected rows from the SQL

```
int rowcount = stmt.executeUpdate
  ("delete from ACME_RENTAL_ITEMS where rental_id =1011");
```

  - **execute()** for either type of statement

Step 4 : Processing the Results

- JDBC returns the results of a query in a *ResultSet* object (in java.sql.*)
  - *ResultSet* object contains all of the rows which satisfied the conditions in a SQL statement
- A *ResultSet* object maintains a cursor pointing to its current row of data
  - Use next() to step through the result set row by row
    - next() returns TRUE if there are still remaining records
  - getString(), getInt(), and getXXX() assign each value to a Java variable
- Example

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(
    "SELECT ID, name, score FROM table1");
While (rs.next()) {
  int id = rs.getInt("ID");
  String name = rs.getString("name");
  float score = rs.getFloat("score");
  System.out.println("ID=" + id + " " + name + " " + score);
}
```

NOTE
This code will not skip the first record

Table1

ID	name	score
1	James	90.5
2	Smith	45.7
3	Donald	80.2

Output

ID=1 James 90.5
ID=2 Smith 45.7
ID=3 Donald 80.2

Step 5 : Closing Database Connection

- It is a good idea to close the *Statement* and *Connection* objects when you have finished with them

- Close the *ResultSet* object

  rs.close();

- Close the *Statement* object

  stmt.close();

- Close the *Connection* object

  conn.close();

The PreparedStatement Object

- A *PreparedStatement* object holds precompiled SQL statements
- Use this object for statements you want to execute more than once
- A *PreparedStatement* can contain variables (?) that you supply each time you execute the statement

```
// Create the prepared statement
PreparedStatement pstmt = con.prepareStatement("
            UPDATE table1 SET status = ? WHERE id =?")

// Supply values for the variables
pstmt.setString (1, "out");
pstmt.setInt(2, id);

// Execute the statement
pstmt.executeUpdate();
```

Transactions Control in JDBC

- Transaction: more than one statement that must all succeed (or all fail) together
  - If one fails, the system must reverse all previous actions
  - E.g., updating several tables due to customer purchase
- COMMIT = complete transaction
- ROLLBACK = cancel all actions

- By default, each SQL statement is treated as a separate transaction that is committed automatically in JDBC
  - bad idea for transactions with multiple updates
- Can turn off automatic commit on a connection
  - conn.setAutoCommit(false);
- Transactions must then be committed or rolled back explicitly
  - conn.commit(); or
  - conn.rollback();
- conn.setAutoCommit(true) turns on automatic commit

Transactions Control Example

```
conn.setAutoCommit(false);
try {
    PreparedStatement pstmt = con.prepareStatement(
        "update BankAccount set amount = amount + ? where accountId = ?");
    pstmt.setInt(1,-100); pstmt.setInt(2, 13);
    pstmt.executeUpdate();
    pstmt.setInt(1, 100); pstmt.setInt(2, 72);
    pstmt.executeUpdate();
    conn.commit();
catch (SQLException e) {
    conn.rollback();
}
```

Other JDBC Features

- Handling large object types
  - getBlob() and getClob() that are similar to the getString() method, but return objects of type Blob and Clob, respectively
  - get data from these objects by getBytes()
  - associate an open stream with Java Blob or Clob object to update large objects
    - blob.setBlob(int parameterIndex, InputStream inputStream).

References

- Database System Concepts, Ch. 5.1.1 JDBC
- Oracle JDBC site
  - http://www.oracle.com/technetwork/java/javase/jdbc/index.html
- Java JDBC Tutorial
  - http://docs.oracle.com/javase/tutorial/jdbc/
- Java API for java.sql package
  - http://docs.oracle.com/javase/6/docs/api/java/sql/package-summary.html
- Tibero JDBC 개발자 안내서
  - http://technet.tmax.co.kr > 기술문서 > Tibero