Database Connectivity: JDBC

Database Connectivity Basics

Application-level database connectivity:

- Host language (Java, C/C++, Perl, PHP, etc)
- Target DBMS (Oracle, MS SQL server, IBM DB2, MySQL, etc)
- Client — Server environment
  - Client: application program
  - Server: DBMS

General structure:

- Load database driver/database support functionality
- form an SQL statement
- connect to the DBMS
- pass SQL statement to the DBMS
- receive result
- close connection

JDBC

JDBC originally, an abbreviation for Java Database Connectivity is the database connectivity package for Java.

JDBC driver

On CSL, JDBC drivers are located in the

/app/oracle/product/10.2.0/client1/jdbc/lib
To successfully compile Java programs using JDBC, this directory needs to be added to your CLASSPATH environment variable.

**in bash** the command is:
```
export CLASSPATH=/app/oracle/product/10.2.0/client1/jdbc/lib/*.jar
```

**in tcsh/csh** the command is:
```
setenv CLASSPATH /app/oracle/product/10.2.0/client1/jdbc/lib/*.jar
```

Alternatively, run your Java programs using the following command
```
$ java -cp $ORACLE_HOME/jdbc/lib/ojdbc14.jar:. <Program>
```

**JDBC Package**

The JDBC package name is `java.sql`. When writing Java applications, which include JDBC connectivity, add to your import section

```
import java.sql.*
```

**Loading the database driver**

First task in any JDBC application is loading the JDBC driver for the right DBMS. This is achieved using `Class.forName(String Name)` method. The argument passed to the `Class.forName` method is the name of the JDBC driver for a specific DBMS server.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Driver name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>oracle.jdbc.OracleDriver</td>
</tr>
<tr>
<td>MySQL</td>
<td>com.mysql.jdbc.Driver</td>
</tr>
<tr>
<td>Microsoft SQL Server</td>
<td>com.microsoft.jdbc.sqlserver.SQLServerDriver</td>
</tr>
<tr>
<td>IBM DB2</td>
<td>COM.ibm.db2.jdbc.app.DB2Driver</td>
</tr>
</tbody>
</table>

`Class.forName()` invocation must be inside Java's try-catch block.

**Example.** The following code loads Oracle's JDBC driver or, if unsuccessful, reports an error.

```java
try{
    Class.forName("oracle.jdbc.OracleDriver");
} catch(ClassNotFoundException ex)
{
    System.out.println("Driver not found");
};
```
Establishing a Connection

JDBC package contains a Connection class representing client-server connections between the client Java applications and the DBMS servers. An instance of the Connection class will be created via the following driver manager call:

```java
Connection conn = DriverManager.getConnection(url, userId, password);
```

Here,

- `url` is a connection URL specifying location and connection port for the database. See below for syntax.
- `userId` is the DBMS user login account.
- `password` is the password for the DBMS account of the user `userId`.

Connection URL

Connection url has the following syntax:

```plaintext
<driver>:<dbms>:<connection-type>@<host>:<port>:<server-name>
```

For CSL machines, use the following line value for the connection url:

```java
jdbc:oracle:thin:@hercules.csc.calpoly.edu:1522:ora10g
```

Example. The following code establishes the connection to our Oracle server for one of the student accounts:

```java
Connection conn = null;
String url = "jdbc:oracle:thin:@hercules.csc.calpoly.edu:1522:ora10g";
String user = "ST44";
String password="empty";
try {
    conn = DriverManager.getConnection(url, user, password);
} catch (Exception ex) {
    System.out.println("Could not open connection");
}
```

Statements

Work with a Connection object within a Java program is straightforward: SQL statements are created and passed via the connection, results are received in return. There are three classes for SQL statements:

- **Statement**: general use statement class. Used for creation and execution of SQL statements, typically, once during the run of the program.
- **PreparedStatement**: statement class to be used in the following cases:
  - an sequence of similar SQL statements, different only in values of some parameters needs to be executed;
a single time-consuming SQL statement needs to be executed, possibly multiple times.

SQL statements represented by instances of PreparedStatement class are pre-compiled and thus may be more efficiently executed.

CallableStatement: statement class for execution of stored SQL (PL/SQL) procedures.

Instances of each class are obtained by invoking (see below) methods from the Connection class.

JDBC distinguishes two types of SQL statements:

Non-Query SQL statements: All DDL and DML statements, which do NOT return relational tables.

Queries: SELECT statements and their combinations, which return relational tables.

Because queries return tables while non-queries return only exit status, different methods are used to pass these two types of SQL statements.

Class Statement

Obtaining Instances. Instances of class Statement can be obtained by invoking the createStatement() method of the Connection class:

```java
Statement s = conn.createStatement();
```

Executing non-queries. Non-queries (a.k.a. updates) are executed using method executeUpdate() of class Statement. While, several method signatures exist, the method call to be used under most standard circumstances is:

```java
int executeUpdate(String sql) throws SQLException
```

The method returns the number of rows affected by the update, 0 if a DDL statement (CREATE TABLE, etc.) was executed.

For example, the following sequence executes two statements: a table Employees is created and a record is inserted into it.

```java
String sql = "CREATE TABLE Employee" +
            "(Id INT PRIMARY KEY, " +
            "Name CHAR(30), Salary INT)";
try{
    s.executeUpdate(sql);
    s.executeUpdate("INSERT INTO Employee VALUES(1,'John Smith', 30000");
} catch (SQLException e) { }
```

Note, that the Statement instance is reusable. Generally speaking, in order to execute a sequence of SQL statements, you only need to create one Statement instance.
Executing SQL queries. Use method `executeQuery()` of class `Statement`. The method returns an instance of the class `ResultSet`, which is discussed below.

```java
try{
    ResultSet result = s.executeQuery("SELECT * FROM Employees WHERE Name='Jones'");
} catch (SQLException e) { }
```

Class `PreparedStatement`

`PreparedStatement` should be used when the same query with possibly different parameters is to be executed multiple times in the course of a program.

Instances of `PreparedStatement` are created with an SQL statement, possibly with parameter placeholders associated with them, and that association cannot change.

Obtaining instances. Instances of class `Statement` can be obtained by invoking the `preparedStatement()` method of the `Connection` class:

```java
String sql = "INSERT INTO Employee VALUES(2, 'Bob Brown', 40000)"
PreparedStatement s = conn.createStatement(sql);
```

This code creates a prepared SQL statement associated with the SQL statement `INSERT INTO Employee VALUES(2, 'Bob Brown', 40000)`.

Parameterized Prepared Statements. The text of the SQL code for the `PreparedStatement` instance can contain `'?'` symbols: one symbol per input parameter to the query. For example, in order to create a generic parameterized `INSERT` statement for the `Employee` table, we can do the following:

```java
String sql1 = "INSERT INTO Employee VALUES(?,?,?)"
PreparedStatement s1 = conn.createStatement(sql1);
```

`createStatement()` method parses the input SQL string and identifies locations of all parameters. Each parameter will get a number (starting with 1).

Setting Parameter Values. `PreparedStatement` uses the following methods to set values for parameters (note: all methods are `void` and throw `SQLException`):

<table>
<thead>
<tr>
<th>Method</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setNull(int parIndex, int jdbcType)</code></td>
<td>sets parameter <code>parIndex</code> to null</td>
</tr>
<tr>
<td><code>setBoolean(int parIndex, boolean x)</code></td>
<td>sets parameter <code>parIndex</code> to a boolean value <code>x</code></td>
</tr>
<tr>
<td><code>setByte(int parIndex, byte x)</code></td>
<td>sets parameter <code>parIndex</code> to a byte value <code>x</code></td>
</tr>
<tr>
<td><code>setInt(int parIndex, int x)</code></td>
<td>sets parameter <code>parIndex</code> to an integer value <code>x</code></td>
</tr>
<tr>
<td><code>setLong(int parIndex, long x)</code></td>
<td>sets parameter <code>parIndex</code> to a long integer value <code>x</code></td>
</tr>
<tr>
<td><code>setFloat(int parIndex, float x)</code></td>
<td>sets parameter <code>parIndex</code> to a floating point value <code>x</code></td>
</tr>
<tr>
<td><code>setDouble(int parIndex, double x)</code></td>
<td>sets parameter <code>parIndex</code> to a double precision value <code>x</code></td>
</tr>
<tr>
<td><code>setString(int parIndex, String x)</code></td>
<td>sets parameter <code>parIndex</code> to a string value <code>x</code></td>
</tr>
<tr>
<td><code>setDate(int parIndex, java.sql.Date x)</code></td>
<td>sets parameter <code>parIndex</code> to a date value <code>x</code></td>
</tr>
<tr>
<td><code>setTime(int parIndex, java.sql.Time x)</code></td>
<td>sets parameter <code>parIndex</code> to a time value <code>x</code></td>
</tr>
<tr>
<td><code>clearParameters()</code></td>
<td>clears the values of all parameters</td>
</tr>
</tbody>
</table>
Executing non-query statements. PreparedStatement class has the executeUpdate() method to execute non-query SQL statements. This method takes no input arguments (since the SQL statement is already prepared).

The following example shows how parameters are set up and prepared updates are executed:

```java
try{
    s1.setInt(1,3); //set first column value for the INSERT statement (ID)
    s1.setString('Mary Williams'); //set second column value (Name)
    s1.setInt(45000); //set third column value (Salary)
    s1.executeUpdate(); //execute INSERT INTO Employee VALUES(3,'Mary Williams',45000')
} catch (SQLException e) {} 
```

Executing queries. To execute queries, use executeQuery() method, which also does not take any arguments. This method returns an instance of ResultSet.

The following code fragment shows the preparation and execution of SELECT statements which select rows of the Employee table by salary.

```java
try{
    String sql2 = "SELECT * FROM Employee WHERE Salary > ?";
    PreparedStatement s2 = conn.prepareStatement(sql2);
    s2.setString(1,35000);
    ResultSet result = s2.executeQuery(); //execute SELECT * FROM Employee WHERE Salary > 35000
    s2.clearParameters(); // clear all parameters
    s2.setString(1,42000);
    result = s2.executeQuery(); //execute SELECT * FROM Employee WHERE Salary > 42000
} catch (SQLException e) {} 
```

Working with output: Class ResultSet

Results of SELECT statements (and other SQL statements that return tables) are stored in instances of the ResultSet class.

An instance of the ResultSet maintains a cursor which points to the currently observed record (tuple) in the returned table. The following methods can be used to navigate a ResultSet object:

<table>
<thead>
<tr>
<th>Method</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean next()</td>
<td>move cursor to the next record.</td>
</tr>
<tr>
<td>boolean previous()</td>
<td>move cursor to the previous record.</td>
</tr>
<tr>
<td>boolean first()</td>
<td>move cursor in front of the first record.</td>
</tr>
<tr>
<td>boolean last()</td>
<td>move cursor to the last row of the cursor.</td>
</tr>
<tr>
<td>boolean absolute(int row)</td>
<td>move cursor to the record number row.</td>
</tr>
<tr>
<td>boolean isLast()</td>
<td>true if the cursor is on the last row.</td>
</tr>
<tr>
<td>boolean relative(int rows)</td>
<td>move cursor rows records from the current position.</td>
</tr>
<tr>
<td>void close()</td>
<td>close the cursor, release JDBC resources.</td>
</tr>
<tr>
<td>boolean wasNull()</td>
<td>true if the last column read had null value</td>
</tr>
<tr>
<td>int findColumn(String columnName)</td>
<td>returns the column number given the name of the column</td>
</tr>
</tbody>
</table>

In addition to these methods, a collection of get methods is associated with ResultSet class. Each get method retrieves one value from the current record (tuple) in the cursor. There are two families of get methods: one family retrieves values by column number, the other — by column name.
get by Column Number | get by Column Name | Explanation
---|---|---
String getString(int colIndex) | String getString(String colName) | retrieve a string value
boolean getBoolean(int colIndex) | boolean getBoolean(String colName) | retrieve a boolean value
byte getByte(int colIndex) | byte getByte(String colName) | retrieve a byte value
short getShort(int colIndex) | short getShort(String colName) | retrieve a short integer value
int getInt(int colIndex) | int getInt(String colName) | retrieve an integer value
float getFloat(int colIndex) | float getFloat(String colName) | retrieve a floating point value
double getDouble(int colIndex) | double getDouble(String colName) | retrieve a double precision value
java.sql.Date getDate(int colIndex) | java.sql.Date getDate(String colName) | retrieve a string value

**Example.** The code fragment below prints out the values of the Name column from the Employee table returned from the query.

```java
Statement query = conn.createStatement();
ResultSet result = query.executeQuery("SELECT * FROM Employee WHERE Salary > 27000");

boolean f = result.next(); // original position of the cursor - before first record
while (f)
{
    String s = result.getString("Name");
    System.out.println(s);
    f=result.next();
}
```

**Types of ResultSet instances**

ResultSet instances can be of one of three types:

- **TYPE_FORWARD_ONLY**: the result set is non-scrollable, the cursor can be moved using only the next() and last() methods (no methods that go back can be used) (default).

- **TYPE_SCROLL_INSENSITIVE**: the result set is scrollable, i.e., the cursor can be moved both forward (next(), last()) and backwards (previous(), first()). Also, the result set typically does not change in response to changes in the database.

- **TYPE_SCROLL_SENSITIVE**: the result set is scrollable, i.e., the cursor can be moved both forward (next(), last()) and backwards (previous(), first()). Also, the result set typically changes if the data in the underlying database changes.

In addition, the result set may, or may not be updatable. This is controlled by the concurrency setting:

- **CONCUR_READ_ONLY**: the result set is read-only, no programmatic updates are allowed (default).

- **CONCUR_UPDATABLE**: the result set can be updated programmatically.

The type of the ResultSet instance to be returned by executeQuery() statements can be selected at the creation time of the SQL statement object:

- Class Statement: default result set type set by the createStatement() method is TYPE_FORWARD_ONLY, and the concurrency setting is CONCUR_READ_ONLY.

To create a statement with a different type of result set use
createStatement(int Scrollable, int Concur)

Here, Scrollable is the scrollability type (one of TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, TYPE_SCROLL_SENSITIVE) and Concur is the concurrency setting (one of CONCUR_READ_ONLY, CONCUR_UPDATABLE)\(^1\).

**JDBC Types**

JDBC package contains a number of **JDBC type** constants that are used to pass to the DBMS server the information about the type of various arguments. All constant declarations are made as as below:

```
public static final int NULL = 0;
```

The full list of types names, their ids and the appropriate Java types (for type conversion purposes) is below:

<table>
<thead>
<tr>
<th>JDBC type</th>
<th>Constant Value</th>
<th>matching Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>OTHER</td>
<td>1111</td>
<td>N/A</td>
</tr>
<tr>
<td>BIT</td>
<td>-7</td>
<td>boolean</td>
</tr>
<tr>
<td>TINYINT</td>
<td>-6</td>
<td>byte</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>5</td>
<td>short</td>
</tr>
<tr>
<td>INTEGER</td>
<td>4</td>
<td>int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>-5</td>
<td>long</td>
</tr>
<tr>
<td>FLOAT</td>
<td>6</td>
<td>double</td>
</tr>
<tr>
<td>REAL</td>
<td>7</td>
<td>float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>CHAR</td>
<td>1</td>
<td>String</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>12</td>
<td>String</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>-1</td>
<td>String</td>
</tr>
<tr>
<td>DATE</td>
<td>91</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>TIME</td>
<td>92</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>93</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>BINARY</td>
<td>-2</td>
<td>byte[]</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>-3</td>
<td>byte[]</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>-4</td>
<td>byte[]</td>
</tr>
<tr>
<td>JAVA_OBJECT</td>
<td>2000</td>
<td>underlying Java class</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>2001</td>
<td>N/A</td>
</tr>
<tr>
<td>STRUCT</td>
<td>2002</td>
<td>Struct</td>
</tr>
<tr>
<td>ARRAY</td>
<td>2003</td>
<td>Array</td>
</tr>
<tr>
<td>BLOB</td>
<td>2004</td>
<td>Blob</td>
</tr>
<tr>
<td>CLOB</td>
<td>2005</td>
<td>Clob</td>
</tr>
<tr>
<td>REF</td>
<td>2006</td>
<td>Ref</td>
</tr>
<tr>
<td>DATALINK</td>
<td>70</td>
<td>java.net.URL</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>16</td>
<td>boolean</td>
</tr>
</tbody>
</table>

---

\(^1\)For the purposes of this course, we can live with the default concurrency setting, and do not even need to know about it. However, **Connection** class does not have a version of `createStatement()` that sets only the scrollability setting, hence, a brief description of the second argument is needed.