Assignment 2: Programming with Oracle using the JDBC API

Code shown bellow might not run as is on csoracle, and might need some minor changes to compile and run. It is included for illustration of the JDBC API use.

Introduction

This section presents Oracle JDBC programming through three complete applications. The first example illustrates basic query processing via `PreparedStatement` object in Java. The query result has at most one answer. The second example illustrates basic query processing using `Statement` object. In this example, the query may have more than one answer. The final example is a more involved one in which recursive query as well as aggregate query processing is discussed.

To be able to program with Oracle databases using Java/JDBC, one must have access to an Oracle database installation with a listener program for JDBC connections. One must also have access to the JDBC drivers (`ojdbc7.jar,` google it, then download it from Oracle website). The driver archive file must be included in the Java `CLASSPATH` variable. Any standard Java environment is sufficient to compile and run these programs, but our assignment, the program should run on `csoracle.utdallas.edu`.

The `COMPANY` database of the Elmasri/Navathe text is used in each of the three illustrative examples that follow.

Example 1: A simple Java/JDBC program that prints the last name and salary of an employee given his or her social security number is shown below:

```
import java.sql.*;
import java.io.*;
class getEmpInfo {
  public static void main (String args [])
      throws SQLException, IOException {
  // Load Oracle's JDBC Driver
    try {
      Class.forName
("oracle.jdbc.driver.OracleDriver");
    } catch (ClassNotFoundException e) {
        System.out.println ("Could not load the
driver");
}

  //Connect to the database
    String user, pass;
    user = readEntry("userid  : ");
    pass = readEntry("password: ");
    Connection conn =
      DriverManager.getConnection

("jdbc:oracle:thin:@csoracle.utdallas.edu:1521:stu
dent",
user,pass);

  //Perform query using PreparedStatement object

  //by  providing SSN at run time

  String query =  "select LNAME,SALARY from
EMPLOYEE where SSN = ?";

    PreparedStatement p = conn.prepareStatement
(query);
    String ssn = readEntry("Enter a Social
Security Number: ");
    p.clearParameters();
    p.setString(1,ssn);
    ResultSet r = p.executeQuery();
  //Process the ResultSet
```

```
    if (r.next ()) {
      String lname = r.getString(1);
      double salary = r.getDouble(2);
      System.out.println(lname + "  " + salary);
}

  //Close objects
p.close();

    conn.close();
  }
 //readEntry function -- to read input string
  static String readEntry(String prompt) {
     try {
       StringBuffer buffer = new StringBuffer();
       System.out.print(prompt);
       System.out.flush();
       int c = System.in.read();
       while(c != '\n' && c != -1) {
         buffer.append((char)c);
         c = System.in.read();
}

       return buffer.toString().trim();
     } catch (IOException e) {
return "";

} }

}
```

In the above program, the user is prompted for a database user id and password. The program uses this information to connect[3] to the database. The user is then prompted for the social security number of an employee. The program supplies this social security number as a parameter to an SQL query used by a `PreparedStatement` object. The query is executed and the resulting `ResultSet` is then processed and the last name and salary of the employee is printed to the console.

The JDBC API is a very simple API to learn as there are an few important classes and methods that are required to manipulate relational data. After the driver is loaded and a connection is made (quite standard code to do this), queries and updates are submitted to the database via one of three objects: `Statement`, `PreparedStatement`, and `CallableStatement`.

The `PreparedStatement` method is illustrated in Example 1 where the SQL statement is sent to the database server with possible place-holders for pluggable values for compilation using the `prepareStatement` method of the `Connection` object. After this is done, the same SQL statement may be executed a number of times with values provided for the place holder variables using the `setString` or similar methods. In Example 1, the prepared statement is executed only once.

The `Statement` method is more commonly used and is illustrated in the following example.

Example 2: Given a department number, the following Java/JDBC program prints the last name and salary of all employees working for the department.

```
import java.sql.*;
import java.io.*;
class printDepartmentEmps {
  public static void main (String args [])
      throws SQLException, IOException {
  //Load Oracle's JDBC Driver
    try {
      Class.forName
("oracle.jdbc.driver.OracleDriver");
    } catch (ClassNotFoundException e) {
        System.out.println ("Could not load the
driver");
      }
  //Connect to the database
    String user, pass;
    user = readEntry("userid  : ");
    pass = readEntry("password: ");
    Connection conn =
    DriverManager.getConnection
("jdbc:oracle:thin:@csoracle.utdallas.edu:1521:stu
dent", user,pass);
  //Perform query using Statement object
    String dno = readEntry("Enter a Department
Number: ");
    String query =
      "select LNAME,SALARY from EMPLOYEE where DNO
= " + dno;
    Statement s = conn.createStatement();
    ResultSet r = s.executeQuery(query);
  //Process ResultSet
    while (r.next ()) {
      String lname = r.getString(1);
      double salary = r.getDouble(2);
      System.out.println(lname + "   " + salary);
}

  //Close objects
s.close();

    conn.close();
  }
```

The query is executed via a `Statement` object by including the department number as part of the query string at run time (using string concatenation). This is in contrast to the `PreparedStatement` method used in Example 1. The main difference between the two methods is that in the `PreparedStatement` method, the query string is first sent to the database server for syntax checking and then for execution subsequently. This method may be useful when the same query string is executed a number of times in a program with only a different parameter each time. On the other hand, in the `Statement` method, syntax checking and execution of the query happens at the same time.

The third method for executing SQL statements is to use the CallableStatement object. This is useful only in situations where the Java program needs to call a stored procedure or function.

Example 3: In this next program, the user is presented with a menu of 3 choices:

. (a) **Find supervisees at all levels:** In this option, the user is prompted for the last name of an employee. If there are several employees with the same last name, the user is presented with a list of social security numbers of employees with the same last name and asked to choose one. The program then proceeds to list all the supervisees of the employee and all levels below him or her in the employee hierarchy.

. (b) **Find the top 5 highest paid employees**: In this option, the program finds five employees who rank in the top 5 in salary and lists them.

. (c) **Find the top 5 highest worked employees:** In this option, the program finds five employees who rank in the top 5 in number of hours worked and lists them.

A sample interaction with the user is shown below:

```
$ java example3
Enter userid: book
Enter password: book
        QUERY OPTIONS
(a) Find Supervisees at all levels.
(b) Find Highest paid workers.
(c) Find the most worked workers.
(q) Quit.

Type in your option: a
Enter last name of employee : King
King,Kate 666666602
King,Billie 666666604
King,Ray 666666606
Select ssn from list : 666666602
     SUPERVISEES
FNAME          LNAME          SSN
----------------------------------------
Gerald         Small
Arnold         Head
Helga          Pataki
```

```
Naveen          Drew
Carl            Reedy
Sammy           Hall
Red             Bacher
666666607
666666608
666666609
666666610
666666611
666666612
666666613
        QUERY OPTIONS
(a) Find Supervisees at all levels.
(b) Find Highest paid workers.
(c) Find the most worked workers.
(q) Quit.
Type in your option: b
    HIGHEST PAID WORKERS
--------------------------------------------------
666666600 Bob
222222200 Evan
444444400 Alex
111111100 Jared
     QUERY OPTIONS
(a) Find Supervisees at all levels.
(b) Find Highest paid workers.
(c) Find the most worked workers.
(q) Quit.
Type in your option: c
    MOST WORKED WORKERS
--------------------------------------------------
666666613 Red
222222201 Josh
333333301 Jeff
555555501 Nandita
111111100 Jared
Bacher           50.0
Zell             48.0
```

```
Chase               46.0
Ball                44.0
James               40.0
          QUERY OPTIONS
(a) Find Supervisees at all levels.
(b) Find Highest paid workers.
(c) Find the most worked workers.
(q) Quit.
Type in your option: q
$
```

The program for option (a) is discussed now. Finding supervisees at all levels below an employee is a recursive query in which the data tree needs to be traversed from the employee node all the way down to the leaves in the sub-tree. One common strategy to solve this problem is to use a temporary table of social security numbers. Initially, this temporary table will store the next level supervisees. Subsequently, in an iterative manner, the supervisees at the "next" lower level will be computed with a query that involves the temporary table. These next supervisees are then added to the temporary table. This iteration is continued as long as there are new social security numbers added to the temporary table in any particular iteration. Finally, when no new social security numbers are found, the iteration is stopped and the names and social security numbers of all supervisees in the temporary table are listed.

The section of the main program that (a) creates the temporary table, (b) calls the `findSupervisees` method, and (c) drops the temporary table is shown below:

```
  /* create new temporary table called tempSSN */
    String sqlString = "create table tempSSN (" +
                       "ssn char(9) not null, " +
                       "primary key(ssn))";
    Statement stmt1 = conn.createStatement();
    try {
       stmt1.executeUpdate(sqlString);
    } catch (SQLException e) {
        System.out.println("Could not create
tempSSN table");
```

```
        stmt1.close();
        return;
}

...  ...

        case 'a': findSupervisees(conn);
  ...
...

}
```

The `findSupervisees` method is divided into four stages.

Stage 1: The program prompts the user for input data (last name of employee) and queries the database for the social security number of employee. If there are several employees with same last name, the program lists all their social security numbers and prompts the user to choose one. At the end of this stage, the program would have the social security number of the employee whose supervisees the program needs to list. The code for this stage is shown below.

```
  private static void findSupervisees(Connection
conn)
       throws SQLException, IOException {
    String sqlString = null;
    Statement stmt = conn.createStatement();
  // Delete tuples from tempSSN from previous request.
    sqlString = "delete from tempSSN";
    try {
      stmt.executeUpdate(sqlString);
    } catch (SQLException e) {
        System.out.println("Could not execute
Delete");
        stmt.close();
        return;
}

  /* Get the ssn for the employee */
/* drop table called tmpSSN */
```

```
sqlString = "drop table tempSSN";
try {
   stmt1.executeUpdate(sqlString);
} catch (SQLException e) {
sqlString = "select lname, fname, ssn " +
            "from   employee " +
            "where  lname = '";
String lname =
   readEntry( "Enter last name of employee :
").trim();
sqlString += lname;
sqlString += "'";
ResultSet rset1;
try {
  rset1 = stmt.executeQuery(sqlString);
} catch (SQLException e) {
    System.out.println("Could not execute Query");
    stmt.close();
    return;
}
String samelName[] = new String[40];
String fName[] = new String[40];
String empssn[] = new String[40];
String ssn;
int nNames = 0;
while (rset1.next()) {
  samelName[nNames] = rset1.getString(1);
  fName[nNames] = rset1.getString(2);
  empssn[nNames] = rset1.getString(3);
  nNames++;
}
if (nNames == 0) {
  System.out.println("Name does not exist in
database.");
  stmt.close();
  return;
}
else if (nNames > 1) {
  for(int i = 0; i < nNames; i++) {
    System.out.println(samelName[i] + "," +
                       fName[i] + " " +
empssn[i]);
  }
  ssn = readEntry("Select ssn from list : ");
  ResultSet r = stmt.executeQuery(
       "select ssn from employee where ssn = '" +
ssn + "'");
  if( !r.next()) {
    System.out.println("SSN does not exist in
database.");
    stmt.close();
    return;
} }

else {
  ssn = empssn[0];
}
```

Stage 2: In the second stage, the `findSupervisees` method finds the immediate supervisees, i.e. supervisees who directly report to the employee. The social security numbers of immediate supervisees are then inserted into the temporary table.

```
  /* Find immediate supervisees for that employee */
    sqlString =
      "select distinct ssn from employee where
superssn = '";
    sqlString += ssn;
    sqlString += "'";
    try {
      rset1 = stmt.executeQuery(sqlString);
    } catch (SQLException e) {
        System.out.println("Could not execute
query");
        stmt.close();
        return;
}
```

/* Insert result into tempSSN table*/
```
    Statement stmt1 = conn.createStatement();
    while (rset1.next()) {
      String sqlString2 = "insert into tempSSN
values ('";
      sqlString2 += rset1.getString(1);
      sqlString2 += "')";
      try {
        stmt1.executeUpdate(sqlString2);
      } catch (SQLException e) {
} }
```

```
  select employee.ssn
  from   employee, tempSSN
  where  superssn = tempSSN.ssm;
```

The results of this query are inserted back into the `tempSSN` table to prepare for the next iteration. A boolean variable, called `newrowsadded`, is used to note if any new tuples were added to the `tempSSN` table in any particular iteration. Since the `tempSSN` table's only column (`ssn`) is also defined as a primary key, duplicate social security numbers would cause an SQLException which is simply ignored in this program. The code for this stage is shown below:

In the third stage, the `findSupervisees` method iteratively calculates supervisees at the next lower level using the query:

Stage 3:

/* Recursive Querying */
```
    ResultSet rset2;
    boolean newrowsadded;
    sqlString = "select employee.ssn from
employee, tempSSN " +
                "where superssn = tempSSN.ssn";
    do {
      newrowsadded = false;
      try {
        rset2 = stmt.executeQuery(sqlString);
      } catch (SQLException e) {
          System.out.println("Could not execute
Query");
          stmt.close();
          stmt1.close();
          return;
      }
      while ( rset2.next()) {
        try {
          String sqlString2 = "insert into tempSSN
values ('";
          sqlString2 += rset2.getString(1);
          sqlString2 += "')";
          stmt1.executeUpdate(sqlString2);
          newrowsadded = true;
        } catch (SQLException e) {
          }
      }
    } while (newrowsadded);
    stmt1.close();
```

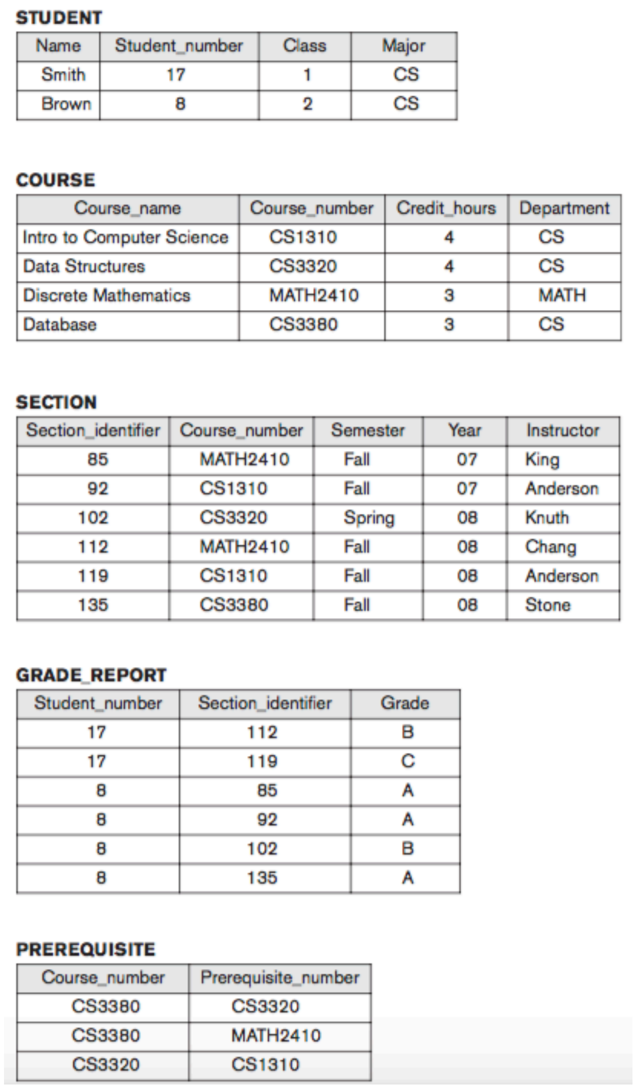
Stage 4: In the final stage, the `findSupervisees` method prints names and social security numbers of all employees whose social security number is present in the `tempSSN` table. The code is shown below:

```
  /* Print Results */
    sqlString = "select fname, lname, e.ssn from "
+ "employee e, tempSSN t where e.ssn = t.ssn";
    ResultSet rset3;
    try {
      rset3 = stmt.executeQuery(sqlString);
    } catch (SQLException e) {
        System.out.println("Could not execute
Query");
        stmt.close();
        return;
     }
     System.out.println("     SUPERVISEES ");
     System.out.print("FNAME");
     for (int i = 0; i < 10; i++)
       System.out.print(" ");
     System.out.print("LNAME");
     for (int i = 0; i < 10; i++)
       System.out.print(" ");
     System.out.print("SSN");
     for (int i = 0; i < 6; i++)
       System.out.print(" ");
     System.out.println("\n-----------------------
-----------\n");
     while(rset3.next()) {
       System.out.print(rset3.getString(1));
       for (int i = 0;
            i < (15 -
rset3.getString(1).length()); i++)
         System.out.print(" ");
       System.out.print(rset3.getString(2));
       for (int i = 0;
            i < (15 -
rset3.getString(2).length()); i++)
         System.out.print(" ");
       System.out.println(rset3.getString(3));
     }
     stmt.close();
  }
```

This concludes the discussion of the `findSupervisees` method. The code to implement options (b) and (c) is quite straightforward and is omitted. The entire code for all the examples is available along with this lab manual.

Assignment:

Consider the `UNIVERSITY` database shown bellow:



STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

. 1- Create the database tables in Oracle using the `SQL*Plus` interface, after deciding on primary and foreign key constrains. You will produce a script called `university_schema.sql`. (If foreign key constrains defines in separate script, document it).

. 2- Ignore the data shown in the above diagram for the University database above, and create comma separated data files containing data for at least 3 departments (CS, Math, and Biology), 20 students, and 20 courses. You may evenly distribute the students and the courses among the three departments. You may also choose to assign minors to some of the students. For a subset of the courses, create sections for the Fall 2006 and Spring 2007 terms. Make sure that you assign multiple sections for some courses. Assuming that the grades are available for Fall 2006 term, add enrollments of students in sections and assign numeric grades for these enrollments. Do not add any enrollments for the Spring 2007 sections. You should be able to use `SQL*Loader to` load the data created into the database.

3- Write and test a Java program that performs the functions illustrated in the following terminal session:

```
   $ java p1
   Student Number: 1234
   Semester: Fall
   Year: 2005
Main Menu (4) Exit

. (1)  Add a class

. (2)  Drop a class

. (3)  See my schedule

Enter your choice: 1
Course Number: CSC 1010
Section: 2
Class Added
Main Menu

(1) Add a class (2) Drop a class (3) See my
schedule (4) Exit

Enter your choice: 1
Course Number: MATH 2010
Section: 1
Class Added
Main Menu

(1) Add a class (2) Drop a class (3) See my
schedule (4) Exit

Enter your choice: 3
```

```
Your current schedule is:
CSC 1010 Section 2, Introduction to Computers, Instructor:
Smith MATH 2010 Section 1, Calculus I, Instructor: Jones

Main Menu

(1) Add a class (2) Drop a class (3) See my
schedule (4) Exit

Enter your choice: 2
Course Number: CSC 1010
Section: 2
Class dropped
Main Menu

(1) Add a class (2) Drop a class (3) See my
schedule (4) Exit

Enter your choice: 3

Your current schedule is: MATH 2010 Section 1, Calculus
I, Instructor: Jones

Main Menu

(1) Add a class (2) Drop a class (3) See my
schedule (4) Exit

  Enter your choice: 4
  $
```

As the terminal session shows, the student signs into this program with an id number and the term and year of registration. The `Add a class` option allows the student to add a class to his/her current schedule and the `Drop a class` option allows the student to drop a class in his/her current schedule. The `See my schedule` option lists the classes in which the student is registered.

What to Submit

In a zip file properly named, have:

- SQL script(s) for creating the schema
- Coma separate files with data
- Shell script executing sqlldr, loading the coma separated files
- Java source files (no class files), with “compile” shell script for compiling, and “run” shell script for executing.
- Driver file.
- README.txt, explaining steps to perform to get the project running