CS 341
Programming Language Design and Implementation
Fall 2015

Homework #8

Complete By: Wednesday, November 4th @ 11:59pm
Assignment: completion of following exercise
Policy: Individual work only, late work *is* accepted
Submission: electronic submission via Blackboard

Background

You're going to write a database application to access the Netflix database we've been discussing in class. This application must be a Windows Forms Application, written using C#. You have reasonable freedom to design the app as you see fit, including the user interface and the database access code. The one restriction is that you must use the .NET ADO database objects to access the database; you may not use the built-in controls that automate database access and display, nor may you use LINQ or other technologies that simplify database access. You can use these tools to prototype and help you build the SQL queries, but the final app you submit for grading must access the database, and display the results, yourself using C# and ADO.NET.

Working with the Netflix database — using Visual Studio and C#

The pre-class assignment for Monday October 26th contained details on downloading the Netflix database, accessing via Visual Studio's solution explorer, and examples of how to write C# code to access the database programmatically. Additional examples are given at the end of the lecture notes for Monday, 10/26 (PDF, PPT). Use these examples as a guide for executing scalar, non-scalar, and action queries from C#.

For this assignment we are going to use an updated Netflix database that contains 3 tables: **Movies**, **Reviews**, and **Users**. Two versions of this database are being provided: one that contains approximately 200,000 reviews, and one that contains roughly 5,000,000 reviews. The files can be found on the course web page, open "Homeworks", and then open the sub-folder "hw8-files". Versions are provided for both VS 2013 and VS 2015. Here are direct links:

Visual Studio 2013:
- Netflix-200K.zip
- Netflix-5MB.zip

Visual Studio 2015:
- Netflix-200K.zip
- Netflix-5MB.zip

Work with the 200K database initially, and use the 5MB database for performance testing once you have a working application. The **Movies** and **Reviews** tables are identical to what has been discussed in class; the

Users table contains **UserID**, **UserName**, and **Occupation** fields. The UserID is the primary key; the UserName and Occupation are strings; Occupation can be NULL.

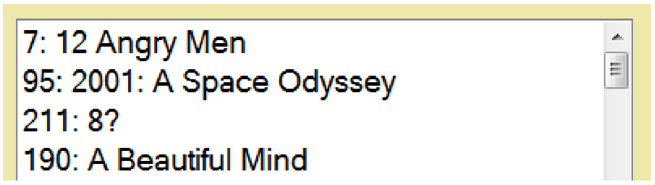
The Assignment

Your assignment is to provide the following functionality. You are free to design whatever user interface you want for collecting the necessary input, and displaying the results. However, you are required to build the SQL, execute the SQL, and display the results yourself — using only C#, ADO.NET, and basic UI controls (buttons, textboxes, listboxes, etc.). Visual Studio contains a wealth of technologies for making database programming easier, but those technologies cannot be used in this assignment: no LINQ, no data-binding, etc. You can use grid-like controls if you want, but you'll have to load the data manually — you cannot data-bind.

You may assume that all input is valid, in the sense that if you ask for an integer or string, the user will provide a properly formatted integer or a non-empty string. However, if the integer represents a movie id or a user id, or if the string represents a movie name, then there's no guarantee that the movie id or user id or movie name will exist in the database — your program must be prepared to handle the lack of a result. In other words, you may assume input data is in a valid format, but you must be prepared for searches that fail because the data is not there, or inserts that fail because of duplicate data, etc.

Here are the commands your program must support. You can consider this in order of difficulty. Results shown are for the smaller 200k Netflix database.

1. **<u>All Movies</u>**: view all movies in alphabetical order. For each movie, display both movie id and movie name. The output is shown below using a listbox, but you may use any reasonable output mechanism (MessageBox.Show is *not* reasonable due to the amount of data):

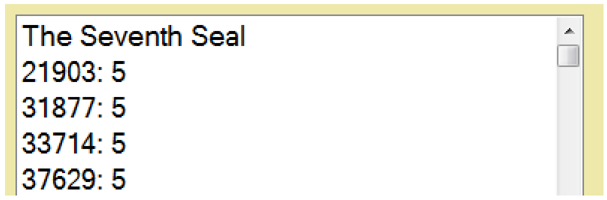


```
7: 12 Angry Men
95: 2001: A Space Odyssey
211: 8?
190: A Beautiful Mind
```

2. **<u>All Users</u>**: view all Netflix users in alphabetical order by UserName, along with their UserID and Occupation. One approach is to display the necessary information together in a listbox (much like MovieID and MovieName above). Another approach is to display just the UserNames in a listbox, and then when the user clicks on a name, display the user's UserID and Occupation in textboxes or labels. Note that Occupation can be NULL (i.e. have no value).

3. **<u>Get Movie Reviews</u>**: get all reviews for a particular movie. For example, if you have the movies listed in a listbox, perhaps clicking on a movie name yields all the reviews for that movie in a separate listbox. Or you can expect a movie id (or a movie name, your choice) to be input into a text box, and then use a button to display all the reviews. When getting reviews for a movie, display the movie name, and then each review (user id and rating). Display the results in <u>descending</u> order by rating, with secondary sort by user id in <u>ascending</u> order. Here's an example

for movie ID 124:



```
The Seventh Seal
21903: 5
31877: 5
33714: 5
37629: 5
```

If the movie id or name denotes a movie that doesn't exist in the DB, say so. If the movie exists but there are no reviews, say so.

4. **Get User Reviews**: get all reviews entered by a particular Netflix user. You can list all the user names in a listbox and then expect the user to click on a name, or you can expect a user id (or a user name, your choice) to be input into a text box followed by a button click. When getting reviews for a user, display the user name, and then each review (movie name and rating). Display the results in ascending order by movie name.

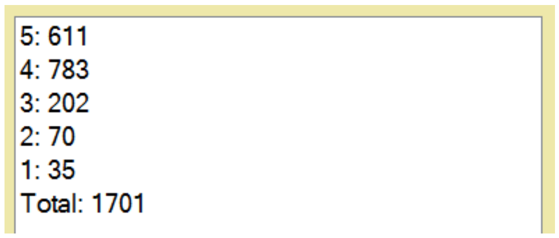
5. **Insert Review**: insert a new review for an existing movie and an existing Netflix user; ratings are integers 1, 2, 3, 4, or 5. Again, the UI is up to you, as long as you ensure that the movie exists, the Netflix user exists, and the rating is an integer in the range 1-5 inclusive. Report on the success or failure of the operation. Note that a unique ReviewID is automatically generated for you by the database, so you only need to insert the other values. [*Confirm your insert succeeded via "Get Movie Reviews" or "Get User Reviews"*.]

6. **Average Rating**: provide the average review for a movie. If the movie doesn't exist then say so, otherwise display the average rating out to 4 decimal places; use `ROUND(AVG( CAST(Rating AS Float) ), 4)`.

7. **Each Rating**: summarize the ratings for a movie. Display the number of reviews for each rating, along with the total number of reviews. In order words, display the following information in some form:

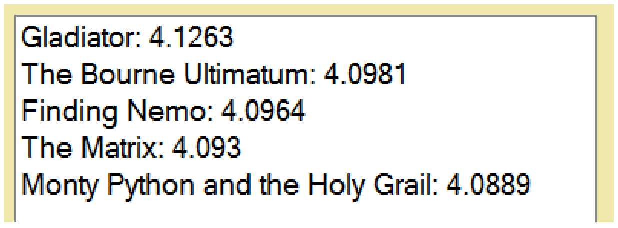
   5: # of reviews
   4: # of reviews
   3: # of reviews
   2: # of reviews
   1: # of reviews
   Total: # of reviews

   If the movie is not found, say so. Here's an example of the output for **Finding Nemo**:



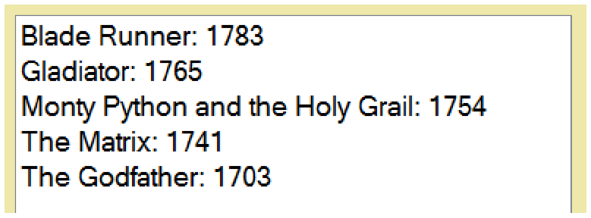
```
5: 611
4: 783
3: 202
2: 70
1: 35
Total: 1701
```

8. **Top-N Movies by Average Rating**: display the top N movies by average rating (computed to 4 decimal places), where N is a positive integer entered by the user (top 1, top 10, etc.). Display in descending order by average rating; the secondary sort should be by movie name in ascending order. You'll want to use SQL's "Group By" clause as discussed in the next section. Display both the movie name and the average rating. For example, here are top-5 movies and their ratings:



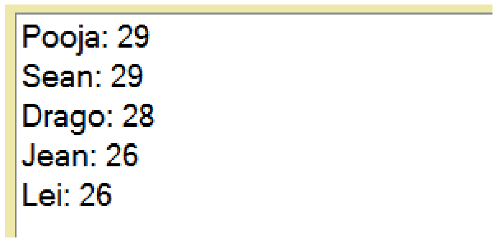
```
Gladiator: 4.1263
The Bourne Ultimatum: 4.0981
Finding Nemo: 4.0964
The Matrix: 4.093
Monty Python and the Holy Grail: 4.0889
```

9. **Top-N Reviewed Movies**: display the top N movies (N determined by the user), based on the # of reviews, in descending order; secondary sort by movie name in ascending order. Display both movie name and the total # of reviews of that movie. Here are top-5:



```
Blade Runner: 1783
Gladiator: 1765
Monty Python and the Holy Grail: 1754
The Matrix: 1741
The Godfather: 1703
```

10. **Top-N Prolific Users**: display the top N Netflix users (N determined by the user), based on the # of reviews submitted, in descending order; secondary sort by user name in ascending order. Display both user name and the total # of reviews submitted by that Netflix user. Here are top-5:



```
Pooja: 29
Sean: 29
Drago: 28
Jean: 26
Lei: 26
```

The Power of SQL "Group By"

One feature of SQL that you will find very helpful is "Group By". For example, here's an SQL query for finding the top-10 movies based on their average rating:

```
SELECT TOP 10 Reviews.MovieID,
              ROUND(AVG(CAST(Reviews.Rating AS Float)), 4) as AvgRating
FROM Reviews
GROUP BY Reviews.MovieID
ORDER BY AvgRating DESC;
```

The 2nd line computes the average rating to 4 decimal places, giving this result the name "AvgRating" in the result table. However, is this the average of all ratings, or what exactly is this the average of? The 4th line groups the data by MovieID, so that AVG will be applied to the ratings in each group — yielding the average of each movie. Finally, we sort these averages into descending order, and the "TOP 10" clause in line 1 limits the result set to the first 10. This yields the top-10 movies by average rating.

Command #8 (*Top-N Movies by Average Rating*) asks you to provide the movie name along with the average rating — the above query produces the movie id but not the name. To get the name, the best approach is to join the result of the above query with the Movies table. One way to do that is to nest one query inside the other, like this:

```
SELECT TOP 10 Movies.MovieName, Temp.AvgRating
FROM Movies
INNER JOIN
  (
    SELECT Reviews.MovieID,
           ROUND(AVG(CAST(Reviews.Rating AS Float)), 4) as AvgRating
    FROM Reviews
    GROUP BY Reviews.MovieID
  ) AS Temp
ON Temp.MovieID = Movies.MovieID
ORDER BY Temp.AvgRating DESC, Movies.MovieName ASC;
```

The inner query produces a temporary result table we call TEMP, and then TEMP is joined with the Movies table to obtain the movie name.

One final, incredibly helpful trick… When building multi-line strings in C#, you can use the escape character **@** character at the beginning of the string literal, like this:

```
string sql = @"SELECT TOP 10 Movies.MovieName, Temp.AvgRating
FROM Movies
INNER JOIN
.
.
.
ORDER BY Temp.AvgRating DESC, Movies.MovieName ASC;";
```

This allows you to build multi-line SQL query strings by copying-pasting from a query window (where you test the query) into your C# source file. [You are testing your queries beforehand in the query window right? ☺]

Electronic Submission

First, to save space and download time, please delete the database files (**netflix.mdf** and **netflix_log.ldf**) from your bin\Debug sub-folder before submission. Then create a .zip / compressed folder of your *entire* Visual Studio project folder. Finally, using Blackboard, submit this .zip / compressed file under the assignment "**HW8**". We expect your C# code to be commented, including a header comment at the top of one of the source files along the lines of

```
//
// Netflix Database Application
//
// <<YOUR NAME HERE>>
// U. of Illinois, Chicago
// CS341, Fall 2015
// Homework 8
//
```

You may submit as many times as you want before the due date, but we grade the last version submitted. This implies that if you submit a version before the due date and then another after the due date, we will grade the version submitted after the due date — we will *not* grade both and then give you the better grade. We grade the last one submitted. In general, do not submit after the due date unless you had a non-working program before the due date.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 25%. After 24 hours, no submissions will be accepted.

All work is to be done individually — group work is not allowed. While I encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates" or Piazza), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is described here:

http://www.uic.edu/depts/dos/docs/Student%20Disciplinary%20Policy.pdf .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from failure to expulsion from the university; cases are handled via the official student conduct process described at http://www.uic.edu/depts/dos/studentconductprocess.shtml .