

## **Project #08: Netflix Database App, Part 1**

**Complete By:** Monday, April 30<sup>th</sup> @ 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 a **Netflix** database of movies and movie reviews. This application must be a Windows Forms Application that you'll design and build yourself. You have reasonable freedom to design the GUI as you see fit. The one restriction is that you must use C# and the ADO.NET database objects to access the database; you may not use any built-in controls that automate database access / display, nor may you use LINQ or other technologies that simplify database access. Your app must access the database, and display the results, using code you wrote or that we provide.

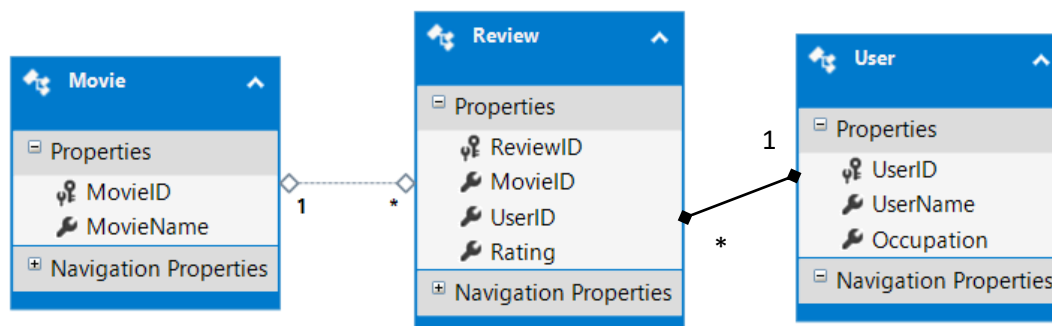
### The Netflix database

The Netflix database we'll be working with contains 3 tables: **Movies**, **Reviews**, and **Users**. There are 2 versions available: a smaller database with about 200K reviews, and a larger database with over 5MB reviews. The databases can be downloaded from the course web site: under "Projects", then "project08-files". Links:

Netflix-200K.zip: <https://www.dropbox.com/s/re6odetut22q38o/Netflix-200K.zip?dl=0>

Netflix-5MB.zip: <https://www.dropbox.com/s/nsbwi9rb5r7zcfg/Netflix-5MB.zip?dl=0>

Here's the ER diagram for the database:



After you download the databases, open the .zip, extract the database files to a folder (on the desktop?),

and discard the .zip files. Then upgrade the databases to your local version of SQL Server as follows:

1. Open up Visual Studio
2. View Server Explorer
3. Right-click on Data Connections, Add Connection...
4. Browse to the **netflix-200k.mdf** file, and select OK --- you should be asked to upgrade the database, select OK
5. Once the database is added as a data connection, right-click on the database and select "Detach database" --- this will allow you to copy the database file later.
6. Repeat steps 3-5 for the **netflix-5mb.mdf** database file...

At this point the database files are ready for use.

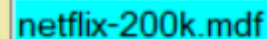
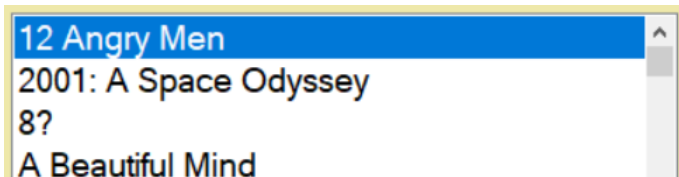
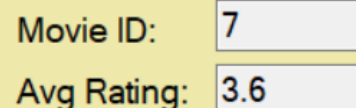
## The Assignment

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.

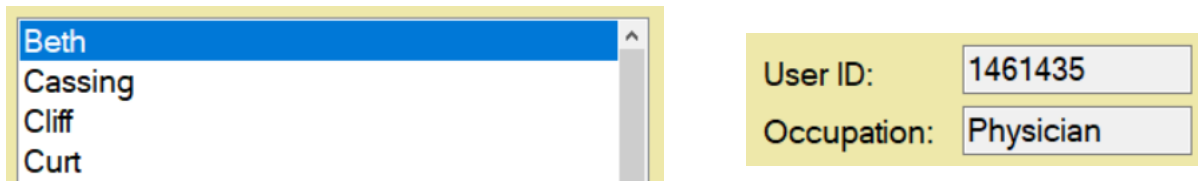
You may assume that all input is valid, in the sense that if you ask for an integer or string, the user will provide a properly formatted integer or a non-empty string. However, if the integer represents a movie id or a user id, or if the string represents a movie name, then there's no guarantee that 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 is the functionality your program must support. You can consider this in order of difficulty. Results shown are for the smaller **netflix-200k** database.

1. **Textbox on main form:** in the previous project (#07), the main form contained a text box where the name of the database file could be entered. The program always opened and connected to the database specified by this textbox. You must do the same here in this project. This allows you to run against different databases when testing, and enables the TAs to more easily test your final program. Write your program so the connection string is built against the filename entered in this text box. If you need help, see the code from project #07 for accessing the textbox and building the connection string.
2. **Movies:** view all movies in alphabetical order. Also provide a way to retrieve the movie's ID and average rating --- e.g. perhaps the user selects a movie from a listbox. The output is shown below using a listbox, but you may use any reasonable display mechanism (MessageBox.Show is *\*not\** reasonable due to the large # of movies):

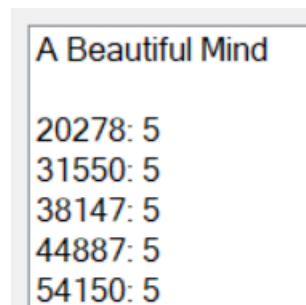
A screenshot of a text box with a yellow border, containing the text "netflix-200k.mdf".A screenshot of a listbox with a yellow border. It contains four items: "12 Angry Men" (highlighted in blue), "2001: A Space Odyssey", "8?", and "A Beautiful Mind".A screenshot of two input fields with yellow borders. The first field is labeled "Movie ID:" and contains the value "7". The second field is labeled "Avg Rating:" and contains the value "3.6".

3. **All Users:** view all Netflix users in alphabetical order by UserName, while also providing a way to view their UserID and Occupation. Note that Occupation can be NULL (i.e. have no value).



The screenshot shows a web application interface. On the left, there is a listbox containing the names of four users: Beth, Cassing, Cliff, and Curt. The name 'Beth' is currently selected and highlighted in blue. To the right of the listbox, there are two input fields. The first is labeled 'User ID:' and contains the value '1461435'. The second is labeled 'Occupation:' and contains the value 'Physician'.

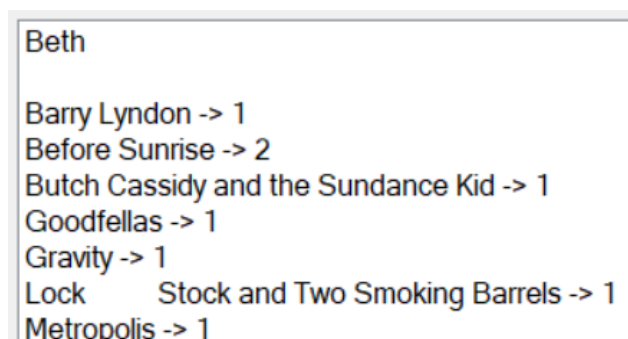
4. **Get Movie Reviews:** get all reviews for a particular movie, where the user either enters or selects a movie name. For example, if you have the movies listed in a listbox, the user could click on a movie name in order to retrieve all the reviews for that movie (displayed in a separate listbox). Or you can ask the user to enter a movie name into a text box, and then use a button to display all the reviews. When displaying the reviews for a movie, display the user id and rating. Display the results in descending order by rating, with secondary sort by user id in ascending order. Here's an example for the movie "A Beautiful Mind":



The screenshot shows a web application interface. At the top, there is a text box containing the movie name 'A Beautiful Mind'. Below the text box, there is a listbox displaying five reviews for the movie. Each review is formatted as 'UserID: Rating'. The reviews are: 20278: 5, 31550: 5, 38147: 5, 44887: 5, and 54150: 5.

If the movie does not exist in the database, say so. If the movie exists but there are no reviews, say so. Keep in mind the movie name might contain a ' --- e.g "Singin' in the Rain".

5. **Get User Reviews:** get all reviews entered by a particular Netflix user. You can list all the user names in a listbox and allow the user to select a name from the list, or you can have the user input a username into a textbox. When getting reviews for a user, display the movie name and rating for each review. Display the results in ascending order by movie name. Here's an example for the user "Beth":

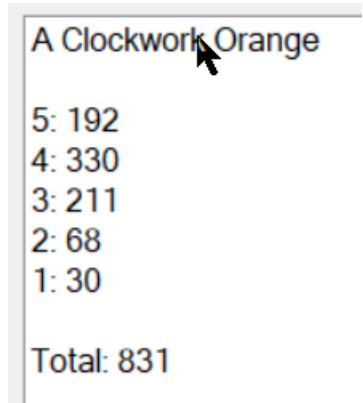


The screenshot shows a web application interface. At the top, there is a text box containing the user name 'Beth'. Below the text box, there is a listbox displaying seven reviews for the user. Each review is formatted as 'Movie Name -> Rating'. The reviews are: Barry Lyndon -> 1, Before Sunrise -> 2, Butch Cassidy and the Sundance Kid -> 1, Goodfellas -> 1, Gravity -> 1, Lock Stock and Two Smoking Barrels -> 1, and Metropolis -> 1.

6. **Insert Review:** insert a new review for an existing movie and an existing Netflix user. The user will select / input a movie name, select / input a user name, and a rating. The UI is up to you, as long as you check to make sure 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. After a review is inserted, it should be visible via “Get Movie Reviews” or “Get User Reviews” --- if not, something is wrong.

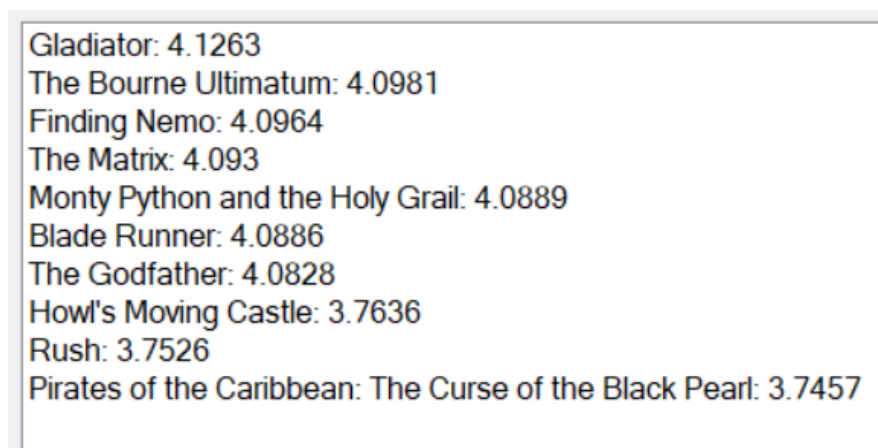
7. **Each Rating:** summarize the ratings for a movie, based on a movie name entered or selected by the user. Display the number of reviews for each rating, along with the total number of reviews. The example shown to the right is for the movie “A Clockwork Orange”. In other words, display the following 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.

8. **Top-N Movies by Average Rating:** display the top N movies by average rating, 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 is by movie name in ascending order. Display both the movie name and the average rating. For example, here are top-10 movies and their ratings:



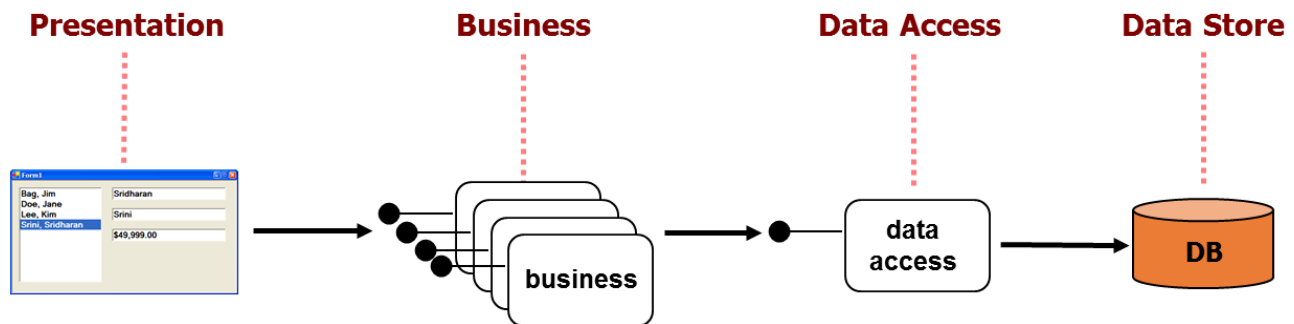
## Before you start programming...

In the previous project --- the *Chicago Crime Analysis* database app --- we used SQL to query the database and then C# to graph the results. If you look back over that code, you would find that much of it is repetitive: open the connection, create the SQL, execute, close the connection, and display. And do we really want SQL queries embedded throughout the user interface code? No...

So what you're going to do instead is call out to a provided set of **DLLs** (dynamic link libraries) to do most of the database access for you, while you focus here in project #08 on designing and building the user interface. Later, in the next project #09, you'll have a chance to implement the DLLs. This approach is based on the idea of **N-tier Design**.

## N-tier Design

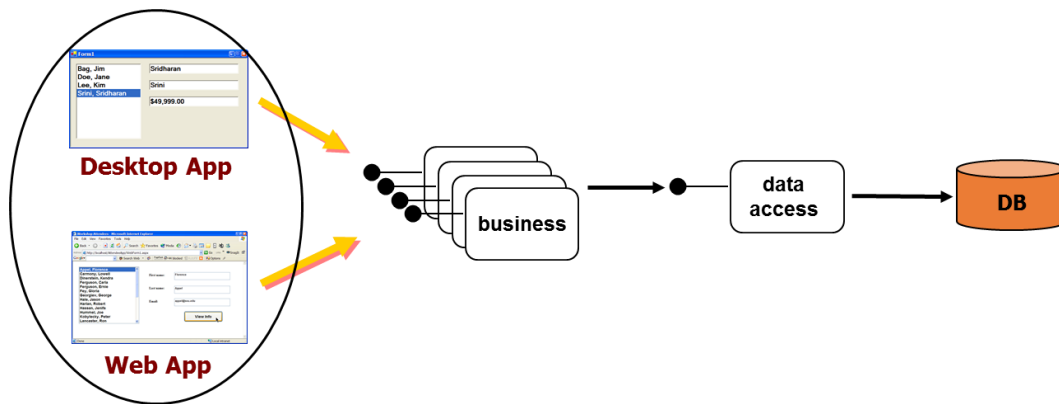
Data-driven applications, whether on a phone, laptop, or a web server, are generally built from at least 2 layers of software: the **front-end user interface** that interacts with the user, and the **back-end data access** code that interacts with the data store (typically a database). Since most organizations offer a variety of apps — imagine a bank that has customer-facing software as well as employee-facing software — the standard design consists of four layers or **tiers**:



Here's a summary of what each tier does:

1. **Presentation:** interacts with the user
2. **Business:** supports the security, business rules, and data processing needed by this particular application. [ *What's a business rule? Imagine a sales app; there would be rules about how to charge sales tax, who gets discounts, etc.*  ]
3. **Data Access:** interface between business tier and data store — the data access tier does not manage nor store data, it only facilitates access to the data
4. **Data Store:** actual data repository

This approach has numerous benefits. For example, if 2 different apps access the same data store, they can use the same Data Access Tier (DAT) to access the data store. As another example, suppose you need to support the same app on both a desktop and a mobile device. Then you should be able to reuse the Data Access and Business Tiers, and need only build 2 different Presentation Tiers:



That's the theory anyway :-) But most organizations practice this approach to data-driven application design.

## Getting Started

Start by creating a Windows Forms Application, naming the project **NetflixApp**; if you need a refresher on creating / building a WinForms app, see [HW #12](#). Build the program, minimize Visual Studio, and copy your **netflix-200k** and **netflix-5mb** database files into the project's bin\Debug sub-folder.

Start by implementing just requirements #1 and #2 as outlined on page #2: a textbox containing the database filename (start with **netflix-200k.mdf**), and perhaps a listbox for displaying all the movie names. Add a button, then code the button's Click event to open a connection, retrieve all the movie names in alphabetical order, and display in the list box. For help with displaying items in a list box, see page 5 of the ADO.NET code [handout](#). Since listboxes are a good UI element for this project, here are a few more tips:

1. To clear a listbox: `this.listBox1.Items.Clear();`
2. After a listbox is loaded, you can pre-select the first item: `this.listBox1.SelectedIndex = 0;`
3. When the user clicks on an item in a listbox, this triggers the event **SelectedIndexChanged**. To program this event to respond to a selection, do the following:
  - a. In design mode, double-click on the listbox
  - b. Visual Studio will generate a SelectedIndexChanged event handler
  - c. To access the text selected by the user: `this.listBox1.Text`

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    MessageBox.Show(this.listBox1.Text);
}
```

**Read on before implementing anything else...**

## Following N-tier Design

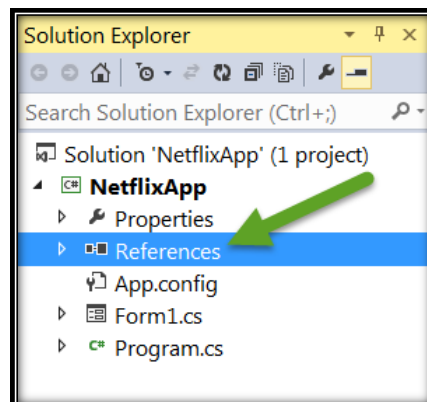
Now you want to add the required functionality #3 - #8 outlined on pages 3-4, but using the provided code and following N-tier Design. Browse to the course web page, open “Projects”, open “project08-files”, and you will see 2 DLLs listed. These denote the business and data access tiers of an N-tier design:

BusinessTier.dll: <https://www.dropbox.com/s/3gfuymjyd9m7sme/BusinessTier.dll?dl=0>

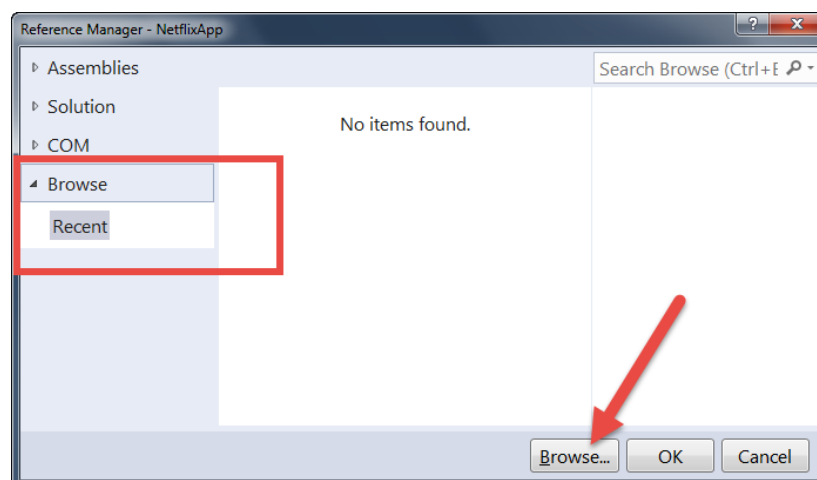
DataAccessTier.dll: <https://www.dropbox.com/s/0ypie22ttgbfaye/DataAccessTier.dll?dl=0>

Download these 2 DLLs, and place them into the bin\Debug sub-folder of your project — the same folder that contains your database files. If you also have a copy of the database in bin\Release, copy the 2 DLLs into bin\Release as well.

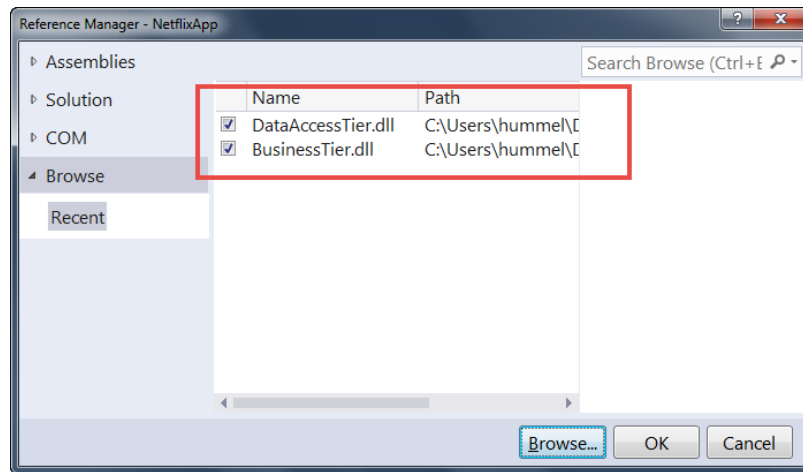
Back in Visual Studio, look at your Solution Explorer window (upper-right typically). If not visible, use View menu to expose. Expand your project, and you’ll see a small folder named “References”:



Right-click, select “Add Reference”, and in the dialog that opens, click on the “Browse” tab along the left. Now click the “Browse” button at the bottom:



The file dialog should start at the top-level of your project folder, so drill down to the bin\Debug sub-folder that contains the 2 DLLs that you copied earlier — BusinessTier.dll and DataAccessTier.dll. Select them both, and click the “Add” button. You should see this:



Make sure there are checkmarks next to both DLLs. Click “OK”, and your program will now have access to the compiled classes inside these DLLs. You’ll see the DLLs listed under the “References” folder in the Solution Explorer.

Run your program — it should still work, nothing has changed other than you have additional functionality you can use when you’re ready.

Now start to build out the rest of your application, and when you need data from the database, use the provided Business Tier (which in turn uses the provided Data Access Tier). DO NOT WRITE SQL unless you absolutely have to. For example, suppose the user selects / enters a movie name, and you want the movie’s ID and average rating. Call the Business Tier’s **GetMovie** method to retrieve the data for you:

```
string dbfilename = this.txtDatabase.Text; // get DB name from text box:

BusinessTier.Business biztier = new BusinessTier.Business(dbfilename);

string moviename = this.txtMovie.Text; // get user’s input:

BusinessTier.Movie movie = biztier.GetMovie(moviename); // obtain movie object:

if (movie == null) // not found:
{
    ...
}
else // found it!
{
    ...
}
```

On the next page you’ll find a complete reference of the functionality available in the Business Tier. You’ll notice that data is being returned in what are called *Business Objects*. These are also defined in the Business Tier: **Movie**, **Review**, **User**, **MovieDetail**, and **UserDetail**.



```

//
// TestConnection:
//
// Returns true if we can establish a connection to the database, false if not.
//
public bool TestConnection();

//
// GetUser:
//
// Retrieves User object based on USER ID; returns null if user is not
// found.
//
// NOTE: if the user exists in the Users table, then a meaningful name and
// occupation are returned in the User object. If the user does not exist
// in the Users table, then the user id has to be looked up in the Reviews
// table to see if he/she has submitted 1 or more reviews as an "anonymous"
// user. If the id is found in the Reviews table, then the user is an
// "anonymous" user, so a User object with name = "<UserID>" and no occupation
// ("") is returned. In other words, name = the user's id surrounded by < >.
//
public User GetUser(int UserID);

//
// GetNamedUser:
//
// Retrieves User object based on USER NAME; returns null if user is not
// found.
//
// NOTE: there are "named" users from the Users table, and anonymous users
// that only exist in the Reviews table. This function only looks up "named"
// users from the Users table.
//
public User GetNamedUser(string UserName);

//
// GetAllNamedUsers:
//
// Returns a list of all the users in the Users table ("named" users), sorted
// by user name.
//
// NOTE: the database also contains lots of "anonymous" users, which this
// function does not return.
//
public IReadOnlyList<User> GetAllNamedUsers();

```

```

//
// GetMovie:
//
// Retrieves Movie object based on MOVIE ID; returns null if movie is not
// found.
//
public Movie GetMovie(int MovieID);

//
// GetMovie:
//
// Retrieves Movie object based on MOVIE NAME; returns null if movie is not
// found.
//
public Movie GetMovie(string MovieName);

//
// AddReview:
//
// Adds review based on MOVIE ID, returning a Review object containing
// the review, review's id, etc. If the add failed, null is returned.
//
public Review AddReview(int MovieID, int UserID, int Rating);

//
// GetMovieDetail:
//
// Given a MOVIE ID, returns detailed information about this movie --- all
// the reviews, the total number of reviews, average rating, etc. If the
// movie cannot be found, null is returned.
//
public MovieDetail GetMovieDetail(int MovieID);

//
// GetUserDetail:
//
// Given a USER ID, returns detailed information about this user --- all
// the reviews submitted by this user, the total number of reviews, average
// rating given, etc. If the user cannot be found, null is returned.
//
// NOTE: this can be the ID of a named user, or an anonymous user.
//
public UserDetails GetUserDetail(int UserID);

```

```

//
// GetTopMoviesByAvgRating:
//
// Returns the top N movies in descending order by average rating. If two
// movies have the same rating, the movies are presented in ascending order
// by name. If N < 1, an EMPTY LIST is returned.
//
public IReadOnlyList<Movie> GetTopMoviesByAvgRating(int N);

//
// GetTopMoviesByNumReviews
//
// Returns the top N movies in descending order by number of reviews. If
// two movies have the same number of reviews, the movies are presented in
// ascending order by name. If N < 1, an EMPTY LIST is returned.
//
public IReadOnlyList<Movie> GetTopMoviesByNumReviews(int N);

//
// GetTopUsersByNumReviews
//
// Returns the top N users in descending order by number of reviews. If two
// users have the same number of reviews, the users are presented in ascending
// order by user id. If N < 1, an EMPTY LIST is returned.
//
// NOTE: not all user ids map to users in the Users table. User ids that don't
// map over are considered "anonymous" users, and returned with their name = to
// their userid ("<UserID>") and no occupation ("").
//
public IReadOnlyList<User> GetTopUsersByNumReviews(int N);

```

## Electronic Submission

First, add a header comment to the top of your main “Form1.cs” C# source code file, along the lines of:

```

//
// Netflix Database Application using N-Tier Design.
//
// <<YOUR NAME HERE>>
// U. of Illinois, Chicago
// CS341, Spring 2018
// Project 08
//

```

Exit out of Visual Studio, and find your **NetflixApp** project folder. Drill down to the “bin\Debug” sub-folder, and delete the database files to save space. Then create an archive (.zip) of this entire folder for submission:

right-click, Send To, and select “Compressed (zipped) folder”. Submit the resulting .zip file on Blackboard ( <http://uic.blackboard.com/> ) under the assignment “P08: Netflix DB App, Part 1”.

Your program should be readable with proper indentation and naming conventions; commenting is expected where appropriate. 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> .