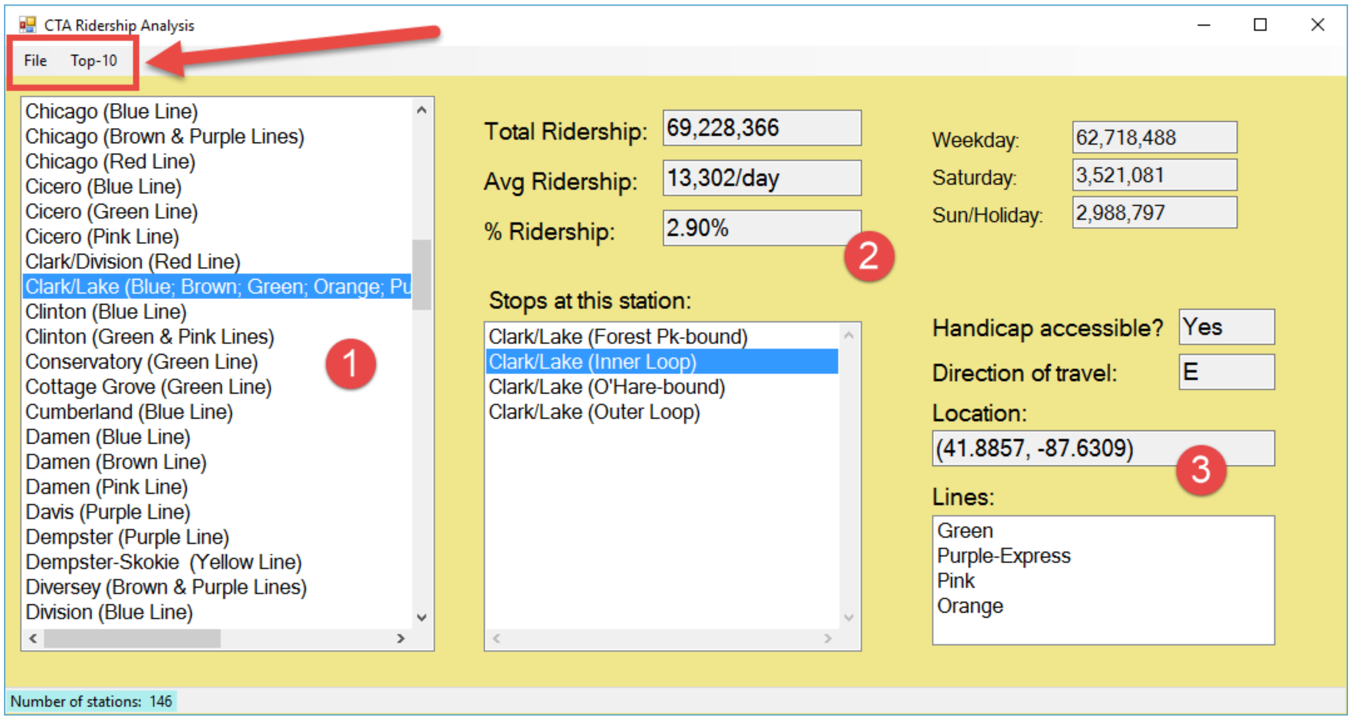
Project #07

Complete By: Monday, November 27th @ 11:59pm
Assignment: completion of CTA Ridership Analysis
Policy: Individual work only, late work *is* accepted
Submission: electronic submission via Blackboard

CTA Ridership Analysis

The assignment is to analyze a database of ridership about Chicago's CTA system. You'll be provided with a SQL Server database of CTA data, and you'll allow the user to investigate the stations and stops in the system, as well as ridership data. You are free to design the GUI as you see fit, as long as you provide the necessary functionality. Here's one possible design, but this UI is *not* required:



Chicago (Blue Line)
Chicago (Brown & Purple Lines)
Chicago (Red Line)
Cicero (Blue Line)
Cicero (Green Line)
Cicero (Pink Line)
Clark/Division (Red Line)
Clark/Lake (Blue; Brown; Green; Orange; Pu
Clinton (Blue Line)
Clinton (Green & Pink Lines)
Conservatory (Green Line)
Cottage Grove (Green Line)
Cumberland (Blue Line)
Damen (Blue Line)
Damen (Brown Line)
Damen (Pink Line)
Davis (Purple Line)
Dempster (Purple Line)
Dempster-Skokie (Yellow Line)
Diversey (Brown & Purple Lines)
Division (Blue Line)

Total Ridership: 69,228,366
Avg Ridership: 13,302/day
% Ridership: 2.90%

Weekday: 62,718,488
Saturday: 3,521,081
Sun/Holiday: 2,988,797

Stops at this station:
Clark/Lake (Forest Pk-bound)
Clark/Lake (Inner Loop)
Clark/Lake (O'Hare-bound)
Clark/Lake (Outer Loop)

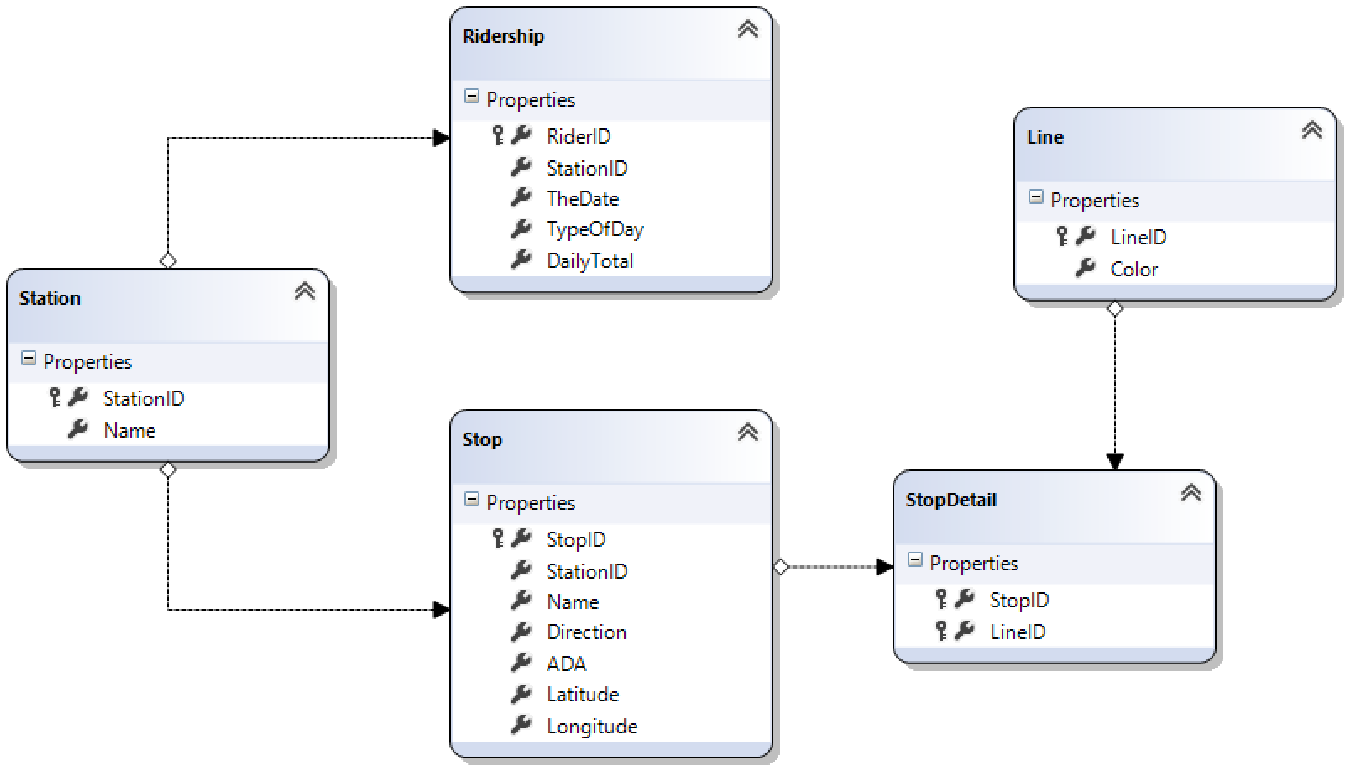
Handicap accessible? Yes
Direction of travel: E
Location: (41.8857, -87.6309)
Lines:
Green
Purple-Express
Pink
Orange

Number of stations: 146

When the application starts, the UI elements are empty. Step #1: the user selects File >> Load to load the **stations** in the left-most list box. Step #2: the user selects a station from the list, and the UI elements of #2 are populated — ridership data as well as stops. Step #3: the user selects a **stop**, and then info about this stop is displayed. The user also has the ability to display the top-10 stations in terms of ridership — station name and total ridership; in my design this is invoked through a menu command and displayed.

CTA Database

The CTA database consists of 5 tables: **Stations**, **Stops**, **StopDetails**, **Lines**, and **Riderships**. Here's the ER (Entity Relationship) diagram:



The database can be downloaded from the course web page: "Projects", "project07-files", and download "CTA_DB.zip" --- note that this is the same database you downloaded and installed for HW #12. Open, extract to a folder on your desktop, and then use Visual Studio's Server Explorer to connect, update, and explore the database. Here's a summary of the design:

Station:
- Denotes one station on the CTA system — a station can have one or more stops, e.g. "Clark/Lake" has 4 stops (see screenshot on page 1)
- Unique station id (primary key)
- Station's name

Stop:
- Denotes one stop on the CTA system — "Clark/Lake (inner loop)" is one of the stops at the "Clark/Lake" station
- Unique stop id (primary key)
- Contains station ID of station to which this stop is associated (foreign key)
- Stop's name
- Stop's direction (N, E, S, W)
- ADA — a boolean denoting whether stop is handicap accessible
- Position — latitude and longitude

Line:

- Denotes one CTA line, e.g. "Red" or "Blue"
- LineID is a unique ID (primary key)
- Color is the line's name

StopDetail:

- A stop may be on one or more CTA lines — e.g. "Clark/Lake (inner loop)" is on the Green, Purple-Express, Pink and Orange lines
- One StopDetail denotes one unique pair (stop id, line id). Example: "Clark/Lake (inner loop)" is on 4 lines, so there are 4 unique pairs and thus 4 StopDetails in this table
- Stop ID denotes the stop (foreign key)
- Line ID denotes the line (foreign key)
- The pair (StopID, LineID) forms a composite primary key

Ridership:

- Denotes how many customers went through the turnstile at this station in one day.
- RiderID is a unique id (primary key)
- StationID is the station id (foreign key)
- TheDate for this data collection
- TypeOfDay: a single character where 'W' denotes a weekday, 'A' denotes Saturday, and 'U' denotes Sunday or Holiday.
- DailyTotal is the total # of customers who went through the turnstile on this date

We'll talk more about this design in class, including primary keys, foreign keys, and the rationale behind "detail" tables like **ShopDetails**.

Getting Started

Download the database and learn more about the database design ("schema"). Be sure to open a connection using Visual Studio's Server Explorer so that the database files can be updated to the version of SQL Server you have installed. [*NOTE: this is the same database we used in HW #12, so you may have already downloaded and installed this database.*]

Next, think about your GUI design and start building out the functionality step by step. You need to include all the functionality shown in the screenshot on page 1, including the "top 10 stations by ridership" implied by the menu command. The top-10 should display both the stations name, and the total # of riders through that station, in descending order. Design whatever UI you like.

Here are some tips for building the UI… First, some of the ridership values are pretty large. Using commas to separate the digitsd --- as in "12,435,291" --- makes things much easier to read. This is easily done in C# using string.format. Here's an example of displaying the variable N in a text box with "," separator:

```
    this.textBox1.Text = string.Format("{0:#,##0}", N);
```

Speaking of large values, the total ridership may overflow SQL's default integer datatype. If that happens, convert to “bigint” when computing the sum, like this:

```
SELECT Sum(convert(bigint, DailyTotal)) AS ...
```

Listboxes are a handy design element in this project, and are pretty easy to work with. We saw how to load a listbox in the previous project, but in short think of a listbox as a string-based data structure, and you add to it. Whatever you add gets displayed:

```
for (int i=0; i<10; ++i)
{
   string msg = string.format("The integer {0}.", i);

   this.listBox1.Items.Add(msg);
}
```

Once a listbox is loaded, you can pre-select the first element by doing:

```
this.listBox1.SelectedIndex = 0;  // pre-select the first one:
```

To clear the contents of a listbox:

```
this.listBox1.Items.Clear();
this.listBox1.Refresh();
```

When the user selects an item in a listbox, the **SelectedIndexChanged** event is triggered. To implement this event, double-click on the listbox when in design mode, and Visual Studio will generate an event handler for you. To obtain the text that the user clicked on, code the event handler as follows:

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

Some of the station names have a ' in their name, e.g. “O’Hare (Blue Line)”. Think about what might happen if you build a query that searches for data using this name:

```
SELECT ...
FROM   ...
WHERE  Stations.Name = 'O'Hare (Blue Line)';
```

The SQL compiler will report an error because the string appears as 'O' and the rest --- **Hare (Blue Line)'** --- is invalid SQL. The solution is to “escape” all text values before building the SQL query:

```
string name = this.textBox1.Text;  // let’s suppose user enters the station name:

name = name.Replace("'", "''");    // replace each single ' with 2 in a row ' '
```

If you build the query using the **name** variable, the query now looks like this:

```
SELECT ...
FROM   ...
WHERE  Stations.Name = 'O''Hare (Blue Line)';
```

which is properly-formatted SQL.

Finally, a nice trick to making a more responsive application is to make sure SQL Server is up and running as the application starts up (recall that SQL Server is a separate program that has to be running in the background, and if not running, it has to be started before you can start executing queries). What I often do is open a connection to the database in the **Form_Load** event, an event that is triggered during application startup. Double-click on the background of the Form in Visual Studio, and you'll be taken to the Form_Load event so you can code the event handler. Code the event handler to simply open and close a database connection, which has the effect of starting SQL Server if it is not already running:

```
private void Form1_Load(object sender, EventArgs e)
{
  //
  // open-close connect to get SQL Server started:
  //
  SqlConnection db = null;

  try
  {
    string connectionInfo = "...";
    db = new SqlConnection(connectionInfo);
    db.Open();
  }
  catch
  {
    //
    // ignore any exception that occurs, goal is just to startup
    //
  }
  finally
  {
    // close connection:
    if (db != null && db.State == ConnectionState.Open)
      db.Close();
  }
}
```

Requirements

In order to execute SQL queries, you must use ADO.NET via C#, with your queries written as raw C# strings. Visual Studio and C# provide a number of other techniques for building database applications: LINQ, data-binding, drag-drop code generation, etc. NONE of these techniques may be used. You must write all the code to create the SQL query strings, execute them, and display the results. On the other hand, you are encouraged to use Server Explorer's query window to formulate your queries, and then copy-paste them into your C# program. Finally, use SQL joins when retrieving data involving multiple tables --- do not use separate queries when a single join would suffice. The type of join syntax is up to you (we discussed three different ways of writing joins in class, all of those are suitable).

Electronic Submission

First, add a header comment to the top of your C# source code file(s):

```
//
// CTA Ridership analysis.
//
// <<YOUR NAME HERE>>
// U. of Illinois, Chicago
// CS341, Fall 2017
// Project #07
//
```

Exit out of Visual Studio, find your project folder, drill down into your bin\Debug folder, and delete the database files. 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 "P07: CTA Analysis".

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 .