Project #2

Complete By: Tuesday, July 3rd @ 11:59pm
Assignment: completion of following programming exercise
Policy: Individual work only, late work *is* accepted (see "Policy" section at end of doc for more details)
Submission: electronic submission of VS project folder to BB

Programming Exercise: BikeHike App

In project #1 you designed and populated a BikeHike database. Here in project #2, the goal is to build a Windows desktop GUI app that supports the basic operation of the BikeHike rental store. As discussed in class on Day 04 (Wed June 27th), you'll build a Windows Forms App that uses C# and ADO.NET to build and execute SQL dynamically against your BikeHike database.

A few notes before you start… First, if you find that your database design is inadequate, feel free to redesign your database and then use your app from Project #1 to rebuild and populate the new database. Second, the goal here is to learn SQL, not UI design --- a simple user interface is fine (buttons and list boxes are sufficient). Third, since the goal here is to learn SQL, do *not* use any of the tools in Visual Studio that generate SQL for you. For example, do not use LINQ to SQL, nor the Entity Framework, nor any of the drag-drop data-binding user interface controls. Build your SQL queries as strings and dynamically execute using C# and ADO.NET. [***Hint**: test your queries in a query window first, before executing in your C# app.*]

Requirements

The app should interact with the database to provide the following functionality, with all changes to the database being persistent. In other words, if customer A rents a bike, that rental should be reflected in the database. If the app is closed and then reopened, the bike is still rented. Functionality requirements:

1. Display a list of all the customers, in alphabetical order by last name. If 2 customers have the same last name, order by first name.

2. Select a customer from the list (or provide a search feature) to display customer's ID and email, and whether the customer is currently out on a rental. If the customer is with a rental, display how many bikes, and the expected return date and time.

3. Display a list of all bikes, in order by bike id.

4. Select a bike from the list (or provide a search feature) to display bike's year, type (the description, not the type id), and the rental price per hour. Also display whether the bike is currently out on a rental; if so, display the expected return date and time.

5. Display a list of bikes available for rent, by type. For bikes of the same type, list in order by newest bikes --- i.e. in descending order by year put into service.

6. Allow customer C to rent N > 0 bikes for H hours; the hours can be a real number, e.g. 3.5 hours. The UI can be as simple as selecting a customer and then inputting a set of bike ids into a text box, but the app must confirm that (a) the customer exists, (b) the customer is not already out on a rental, and (c) each of the bikes is available for rent. These conditions should be checked before the rental is entered into the database. If the rental is successful, display the rental ID (MessageBox.Show is fine).

7. Allow customer C to return from a rental, in which case all N bikes are returned at the same time. The app should display (MessageBox.Show is fine) the total cost of the rental based on the actual return time.

The current date and time is available in SQL by calling **GetDate( )**; e.g. this can be called as part of your SQL Insert statement when inserting the rental info. Alternatively, the current date and time is available from C# via **DateTime.Now**; the DateTime class in C# also provides functionality for working with dates and times, e.g. to subtract one from another.

For simplicity, you may assume the user will input valid data --- e.g. if you expect the user to input an integer, assume the user will input an integer. However, if the integer represents a customer or bike id, do *not* assume it's a valid ID --- the user could enter an id for which a customer or bike does not exist, or the id of a bike that is already rented.

When it comes time to dynamically execute the SQL, use ADO.NET as discussed in class on Wednesday, June 27th. The lecture notes are available on the course web page under Lectures, Day 04; the lecture recording is available on BB. Note that in Project #01, you were given a C# source file called "DataAccessTier.cs"; this file was used to help you execute the SQL. For example, to insert an entity in project #01, you were able to do:

```
DataAccessTier.Data data = new DataAccessTier.Data("BikeHike.mdf");

string sql = "INSERT … ";

data.ExecuteActionQuery(sql);
```

You are free to use "DataAccessTier.cs" in project #02 if you wish. To use, copy the .cs file from Project #01 and store this copy into the project 02 sub-folder that contains your other .cs file ("Program.cs", "Form1.cs", etc.). Then back in Visual Studio, use the project menu to "Add existing item…" to your project. When you are done, the file "DataAccessTier.cs" should appear in your Solution Explorer window.

Finally, when building SQL queries, it is more efficient --- and readable --- to use the C# function **String.Format** instead of string concatenation (+). Example:

```
string  songtitle = …;
```

```
int     yearpublished = …;
string  songduration = …;

string sql = String.Format(@"
INSERT INTO Songs(Title, YearPub, Duration)
VALUES('{0}', {1}, '{2}');
", songtitle, yearpublished, songduration);
```

The use of the @ symbol is also handy in that it allows a string constant to continue across multiple lines. This allows you to test your queries in a query window, and then copy-paste into your C# source file for execution.

Intro to GUI Programming

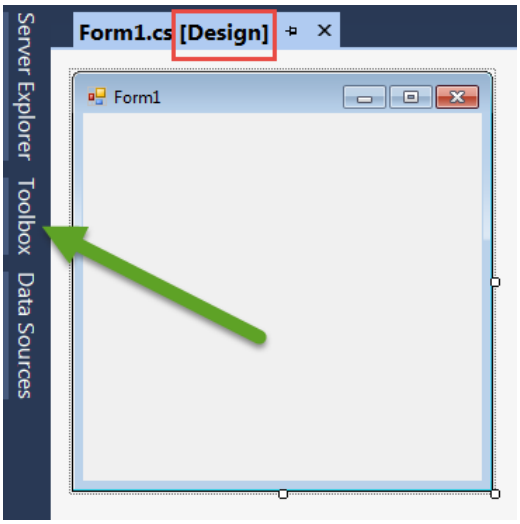
If you have taken CS 341 recently --- or know how to build a Windows Forms App --- you can skip this section. If you need an intro or refresher on GUI programming in Visual Studio and C#, read on…

GUIs are based on event-driven programming. The concept is very intuitive: you problem solve by waiting for things to happen, and then respond when they do. This paradigm maps naturally to the programming of user interfaces, which are also event-driven: keyboard press, mouse click, finger swipe. Here's a short article (a bit dated but otherwise okay) on event-driven programming:

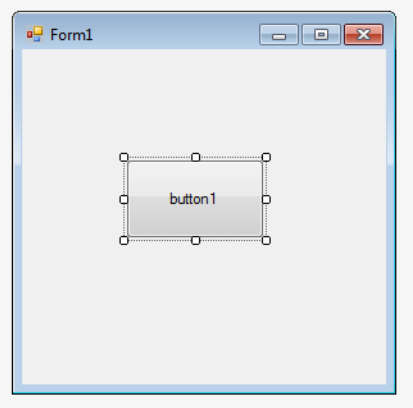
http://www.technologyuk.net/computing/software-development/event-driven-programming.shtml

Let's explore event-driven programming via Visual Studio. Startup Visual Studio, create a new project, and under "Templates" expand the options under **Visual C#**. Click on "Windows Classic Desktop", and then to the right select "Windows Forms App" (aka WinForms). Click OK to create the project. A new GUI-based app is created, and the main form (aka window) for your app is generated. Go ahead and run the program (F5). When you run, notice the main window is created and displayed, and that it automatically responds to many events: you can resize, you can grab the title bar and drag the window around, you can minimize and restore, and you can close. When you close the main window, the application halts. Go ahead and close the window.

Back in Visual Studio, let's do a little bit of event-driven programming. At this point you should be back in VS looking at the main window. Notice the title bar above the main window — Visual Studio is telling you the form is being viewed in "Design" mode — this mode is for designing the user interface. Click on "Toolbox" on the left side, expand "Common Controls", click-and-hold on "Button", and drag to the right until the Toolbox closes. Then drop the button anywhere onto the form…



Continued on next page…



Run the application and click the button — notice that nothing happens. This makes sense, because we haven't programmed the button to respond yet to "being clicked".

Back in Visual Studio, double-click on the button in Design mode, and Visual Studio will reveal the "code-behind" window — this mode is for programming. Here's what you should be seeing:

```
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    3 references
    public partial class Form1 : Form
    {
        1 reference
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {

        }
    }
}
```

When you double-clicked on the button in Design mode, Visual Studio did 2 things:

1. Creates an event handler — the function **button1_Click** — for responding to the click of this button
2. Hooks up this event handler so it's automatically called whenever the user clicks the button

Add the following line of code to the button's event handler method:

```
MessageBox.Show("click!");
```

Run, and click the button. A dialog box should open in response with the message "click!". Exciting I know :-)

In addition to buttons, the most common UI elements you'll need are text boxes (for user input) and list boxes (for displaying output). To retrieve the data from a textbox, use the object's Text property:

```
string input = this.textBox1.Text;
```

For working with list boxes, here are a few 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);
}
```

Have a question? Use Piazza, not email

Use Piazza for questions: http://piazza.com/uic/summer2018/cs480/home. As discussed in the syllabus, questions via email will be ignored, so post to our Piazza site. Remember the guidelines for using Piazza:

1. *Look before you post* — the main advantage of Piazza is that common questions are already answered, so search for an existing answer before you post a question. Posts are categorized to help you search, e.g. "Pre-class" or "HW".
2. Post publicly — only post privately when asked by the staff, or when it's absolutely necessary (e.g. the question is of a personal nature). Private posts defeat the purpose of piazza, which is answering questions to the benefit of everyone.
3. Ask pointed questions — do not post a big chunk of code and then ask "help, please fix this". Staff and other students are willing to help, but we aren't going to type in that chunk of code to find the error. You need to narrow down the problem, and ask a pointed question, e.g. "on the 3rd line I get this error, I don't understand what that means…".
4. Post a screenshot — sometimes a picture captures the essence of your question better than text. Piazza allows the posting of images, so don't hesitate to take a screenshot and post; see http://www.take-a-screenshot.org/ .
5. Don't post your entire answer — if you do, you just gave away the answer to the ENTIRE CLASS. Sometimes you will need to post code when asking a question --- in that case post only the fragment that denotes your question, and omit whatever details you can. If you must post the entire code, then do so privately --- there's an option to create a private post ("visible to staff only").

Electronic Submission

First, add a header comment at the top of your main C# source code file ("Program.cs"), something like this:

```
//
// Windows Form App to allow bike rentals via the BikeHike database.
//
// <<YOUR NAME HERE>>
// U. of Illinois, Chicago
// CS480, Summer 2018
// Project #2
//
```

To submit, exit out of Visual Studio, and find your project folder --- this should be the **top-level** folder such as "BikeHikeApp". Create an archive (.zip) of this entire folder: right-click, Send To, and select "Compressed (zipped) folder". Then submit the resulting .zip file on Blackboard: open "Projects" and submit under "Project 02: BikeHike App".

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 **72** hours after the deadline for a penalty of 25%. After **72** hours (i.e. after Friday, July 6th @ 11:59pm), 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. via 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:

https://dos.uic.edu/docs/Student%20Disciplinary%20Policy%2017-18%20(FINAL).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. 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.