WinForms / Avalonia “Notepad” Application / Fibonacci BigInt Text Reader
Cpt S 321 Homework Assignment
Washington State University

Submission Instructions:

- Create a branch called “Branch_HW3” and work in this branch for this assignment.
- Once you are done and before the deadline, tag the version that you would want us to grade with the assignment number (for example, “HW3”).
- On Canvas -> Assignments -> Submit the link to your repository (a link to the tag or the branch works) by the HW deadline.
- **IMPORTANT: The HW must be tagged by the due date and a link to that tag needs to be submitted via Canvas in order to receive a grade.**

Assignment Instructions:

Read all the instructions *carefully* before you write any code.

Using Visual Studio (or Rider if you are doing a cross-platform development), create a WinForms (Avalonia MVVM) application that can load and save plain text files. This will be much like a simplified version of the Notepad utility application that comes with Microsoft Windows. This app will also include the ability to load text from a non-file source (Fibonacci in-memory sequence generator).

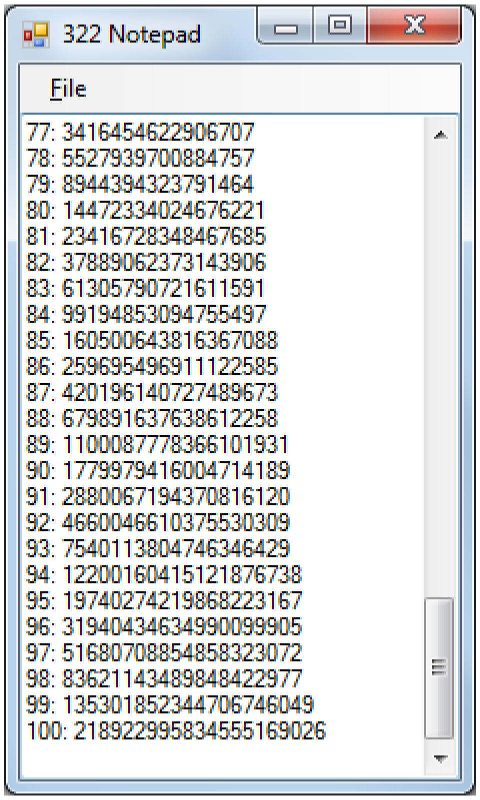
Interface:

You are free to design the interface the way you wish for the most part, but you must include the following:

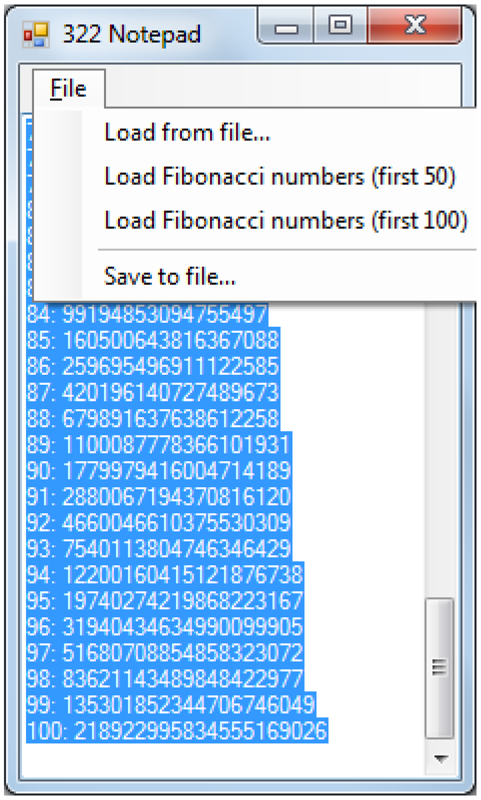
- a prominent text area that contains the text
    - Use a TextBox
    - Set the Multiline property to true for WinForms (multiline is the default for Avalonia)
    - For WinForms, set the ScrollBars property to Vertical or Both so that vertical scrolling can occur
- buttons or menu options to load and save files
    - You will need 3 options for loading text, one from file, two from another source. This is discussed more below.
    - For Avalonia, check the documentation on Menus here; we will use a static menu.
- selecting files to open/save
    - for WinForms, use OpenFileDialog/SaveFileDialog for selecting files to open/save. Can be created in code or at design time by dragging them in from the toolbox
    - for Avalonia, we need to manually bind the code to the user interface in the .axaml file. As we go, we will learn more about good practices for interactions between the user

interface and the rest of the code but for now, we will follow a template provided in addition to the instructions without worrying too much about why things are defined this way. Having said that, we will be also using the OpenFileDialog and SaveFileDialog controls implemented in Avalonia.

The image below shows an example of what the interface could look like:

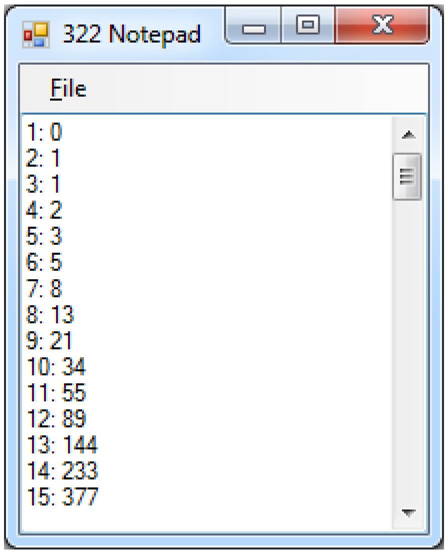


```
77: 3416454622906707
78: 5527939700884757
79: 8944394323791464
80: 14472334024676221
81: 23416728348467685
82: 37889062373143906
83: 61305790721611591
84: 99194853094755497
85: 160500643816367088
86: 259695496911122585
87: 420196140727489673
88: 679891637638612258
89: 1100087778366101931
90: 1779979416004714189
91: 2880067194370816120
92: 4660046610375530309
93: 7540113804746346429
94: 12200160415121876738
95: 19740274219868223167
96: 31940434634990099905
97: 51680708854858323072
98: 83621143489848422977
99: 135301852344706746049
100: 218922995834555169026
```


322 Notepad
File
Load from file...
Load Fibonacci numbers (first 50)
Load Fibonacci numbers (first 100)
Save to file...

```
84: 99194853094755497
85: 160500643816367088
86: 259695496911122585
87: 420196140727489673
88: 679891637638612258
89: 1100087778366101931
90: 1779979416004714189
91: 2880067194370816120
92: 4660046610375530309
93: 7540113804746346429
94: 12200160415121876738
95: 19740274219868223167
96: 31940434634990099905
97: 51680708854858323072
98: 83621143489848422977
99: 135301852344706746049
100: 218922995834555169026
```

Here's another image of the application scrolled up to the beginning numbers:



```
1: 0
2: 1
3: 1
4: 2
5: 3
6: 5
7: 8
8: 13
9: 21
10: 34
11: 55
12: 89
13: 144
14: 233
15: 377
```

Saving Functionality:

Your application needs to provide the ability to save the text to a text file. Provide a button or menu option that brings up a SaveFileDialog, prompting the user for a file name. If the user clicks OK in that dialog, which you'll determine by checking the return value of the ShowDialog method for WinForms or ShowAsync for Avalonia, then save the text in the text box to that file. You can save it using streams, text-writers, or any other method you see fit (it's the loading that's going to be more specific).

Loading Functionality:

1. Generic Loading Function
   Write a function called "LoadText" in your main form's .CS file for WinForm or in you MainWindowViewModel's .CS file for Avalonia. This will take a System.IO.TextReader object as a parameter. It should read all the text from the TextReader object and put it in the text box in your interface.
   a. Declare and implement the method mentioned above:
      private void **LoadText**(TextReader sr)
   b. Use either ReadToEnd or ReadLine (in a loop) to load all text from the reader and put it in the text box. This should replace any content already in the text box.
2. Loading from file
   Add the option to load text from a file. You can use a stream reader to open the file and pass it to your loading function. Remember that StreamReader inherits from TextReader and has a constructor that takes a file name as a parameter.
3. Loading from FibonacciTextReader
   Write a class named FibonacciTextReader that inherits from the System.IO.TextReader. Since you've implemented a loading function in part 1 that takes a text reader, you'll be able to pass this object to your loading function after creating an instance of it. This class must:
   a. Inherit from the TextReader class
   b. Make it have a constructor that takes an integer as a parameter indicating the maximum number of lines available (the maximum number of numbers in the sequence that you can generate).
   c. Override the ReadLine method which delivers the next number (as a string) in the Fibonaci sequence. You need logic in this function to handle the first two numbers as special cases. You must return null after the nth call, where n is the integer value passed to the constructor.
   d. Override the ReadToEnd method if you used it back in the implementation for part 1. Implement it such that it repeatedly calls ReadLine and concatenates all the lines together. You can use the System.Text.StringBuilder class to append the lines together into one string.
   e. Use the System.Numerics.BigInteger struct for the sequence numbers. The numbers get beyond what a 64-bit integer can store very quickly, so you need this to accurately compute the first 100 numbers in the sequence.

   f. Optional: put the line number before the sequence number as shown in the sample app screenshot.

4. Add three menu options (or buttons if you want) in your interface to allow for demonstration of the above functionality.
   a. There needs to be an option to load from a file. This should prompt the user for a file name (use OpenFileDialog for convenience) and then load it as described in #2 above.
   b. There needs to be an option to load the first 50 numbers of the Fibonacci sequence
   c. There needs to be an option to load the first 100 numbers of the Fibonacci sequence
   d. (note: parts b and c should have almost identical code, just with a different number passed to the FibonacciTextReader constructor)

<u>Point breakdown (the assignment is worth 10 points):</u>

- 2 points for correct saving functionality
- 3 points for correct loading function

And as usual:

- 1 point: For a "healthy" version control history, i.e., 1) the HW assignment should be built iteratively, 2) every commit should be a cohesive functionality, 3) the commit message should concisely describe what is being committed, 4) you should follow TDD – i.e., write and commit tests first and then implement and commit the functionality.
- 1 point: Code is clean, efficient and well organized.
- 1 point: Quality of identifiers.
- 1 point: Existence and quality of comments.
- 1 point: Existence and quality of test cases.

General Homework Requirements	
Quality of Version Control	● Homework should be built iteratively (i.e., one feature at a time, not in one huge commit). ● Each commit should have cohesive functionality. ● Commit messages should concisely describe what is being committed. ● TDD should be used (i.e, write and commit tests first and then implement and commit functionality). ● Include "TDD" in all commit messages with tests that are written before the functionality is implemented. ● Use of a .gitignore. ● Commenting is done as the homework is built (i.e, there is commenting added in each commit, not done all at once

	at the end).
Quality of Code	● Each file should only contain one public class. ● Correct use of access modifiers. ● Classes are cohesive. ● Namespaces make sense. ● Code is easy to follow. ● StyleCop is installed and configured correctly for all projects in the solution and all warnings are resolved. If any warnings are suppressed, a good reason must be provided. ● Use of appropriate design patterns and software principles seen in class.
Quality of Identifiers	● No underscores in names of classes, attributes, and properties. ● No numbers in names of classes or tests. ● Identifiers should be descriptive. ● Project names should make sense. ● Class names and method names use PascalCasing. ● Method arguments and local variables use camelCasing. ● No Linguistic Antipatterns or Lexicon Bad Smells.
Existence and Quality of Comments	● Every method, attribute, type, and test case has a comment block with a minimum of <summary>, <returns>, <param>, and <exception> filled in as applicable. ● All comment blocks use the format that is generated when typing "///" on the line above each entity. ● There is useful inline commenting in addition to comment blocks that explains how the algorithm is implemented.
Existence and Quality of Tests	● Normal, boundary, and overflow/error cases should be tested for each feature. ● Test cases should be modularized (i.e, you should have a separate test case for each thing you test - do not combine them into one large test case). ● *Note: In assignments with a GUI, we do not require testing of the GUI itself.*