C# Review and Assignment Format Overview
Cpt S 422 Homework Assignment
by Evan Olds

Assignment Instructions:

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

In this assignment you will build a C# class library (DLL) containing several classes that meet various requirements. Although most of the general requirements are described later, there are a few things to note right off the bat:

1. For each class that you add to your DLL, you will be told the name of the namespace and class. These must match EXACTLY. If you are told to create a class named MyClass, then creating a class named myClass, myclass, my_class, My_Class, or any other slight name variant will be marked as incorrect.
2. The target platform for testing is Linux. Although C# is generally pretty good with respect to consistent functionality across different platforms, you must test on Linux before submitting. If your app doesn't work on Linux during grading, you will not be given credit, even if the app worked fine on Windows or Mac OS. Ideally you'd test your code on all 3 platforms to ensure that you can write good, platform-independent code. But in the interest of keeping things simple, you'll be graded only on Linux.
3. Following from the previous point, you will need to install Mono in Linux. If your Linux distribution supports apt-get for package management, then the following terminal command will usually suffice for installing Mono:

   sudo apt-get install mono-complete

   - See this page for official and more detailed installation instructions: http://www.mono-project.com/docs/getting-started/install/linux/
4. You will submit only code files (.CS) for the DLL project. You can test your DLL in whatever way you see fit, such as writing a standalone application that loads the DLL or using the unit testing framework in some IDE. Just make sure that the test code is not submitted (unless it is specifically requested in the instructions). Submit ONLY the code files needed to compile the DLL.

Part 1: Implement the **NumberedTextWriter** class

Create a class named **NumberedTextWriter** in the CS422 namespace. The namespace CS422 must be a top-level namespace, meaning it is not nested within another namespace. As mentioned earlier, the names for classes and namespaces must match exactly, including casing!

This class will inherit from System.IO.TextWriter. This class will have been covered in one (or more) of your prerequisite courses. As a very brief reminder, a text writer just writes text to some location. This could be the console/terminal window, a file (exists already in .NET as a StreamWriter), a

string in memory (exists already in .NET as a StringWriter), or just about anywhere else that you can imagine.

The purpose of the NumberedTextWriter class will be to wrap around *another* TextWriter and simply pass strings through to it with a line number in front. So when you call WriteLine(someString) on the NumberedTextWriter it will add the current line number to the beginning of someString, and pass that on to the WriteLine function on the wrapped TextWriter. It will also incremement the current line number for each WriteLine call. Thus the member variables in the NumberedTextWriter class should be an integer for the current line number and a TextWriter object reference. You should not need more than these two member variables in the class.

Your NumberedTextWriter class must have two specific constructors. One takes only a TextWriter as a parameter and starts the line number at 1. The other takes a TextWriter as the first parameter and a starting line number as the second parameter. This allows the NumberedTextWriter to start at a desired offset that may not be 1. The signatures for these constructors are shown below:

NumberedTextWriter(TextWriter wrapThis)

NumberedTextWriter(TextWriter wrapThis, int startingLineNumber)

Override the **Encoding** property and the **WriteLine(string value)** function. For the encoding, return the encoding of the wrapped TextWriter object. For the WriteLine function, write the current line number, then a colon, then a space and then the actual value passed as the parameter. Increment the line number after this.

As an example of the behavior of this class, assume you constructed your NumberedTextWriter object with the single parameter constructor and used Console.Out as that parameter. If you then called WriteLine("Hello World!") you would see the string "1: Hello World!" on the screen. Calling WriteLine("Hi") next would result in the string "2: Hi" on screen.

Part 2: Implement the **IndexedNumsStream** class

In this course there will be the need for several custom classes that inherit from System.IO.Stream. Why this is the case will become apparent as the course progresses. Here we get a simple taste of one such stream. As with the previous class, you will implement **IndexedNumsStream** in the CS422 namespace.

The IndexedNumsStream is a read-only stream of procedurally-generated data. This means that the stream provides us with data when we call Read, even though it is not getting that data from memory, a file, or any other source outside the class. The stream generates the data during the Read call.

Each byte in the stream has a value equal to the stream position % 256. So the very first byte in the stream (position 0) has a value of 0 % 256 = 0. The next byte (position 1) has a value of 1 % 256 = 1. This continues on and when you get to position 256, 257, and 258 you'll see byte values of 0, 1, and 2 respectively. The table below gives a simple illustration of this.

Stream Position:	0	1	2	…	255	256	257	…	N
Byte Value:	0	1	2	…	255	0	1	…	N % 256

Have two member variables in the class. One will store the current stream position, and the other the stream length, in bytes. The IndexedNumsStream constructor will take a 64-bit signed integer (you should know the C# type for this) value for the length of the stream and will store this in the member variable. In addition to all of this, have your stream implementation obey the following rules:

- If the **Position** property of the stream is ever set to a negative value, "clamp" it to 0 instead. So as an example, if you set the **Position** to -10 and then read the value of **Position** on the next line of code, it would be 0.
- If the **Position** property of the stream is ever set to a value greater than the length of the stream, clamp it to the stream length instead.
- If a negative stream length is ever specified, either as the parameter to the constructor or the parameter to the **SetLength** member function, use 0 instead.

Final Notes:

- You'll need to extensively review online documentation to ensure that you're properly implementing your classes according to the rules of TextWriter and Stream. Develop many of your own test cases to test your code thoroughly.
- Although C# console/terminal applications run pretty consistently across different platforms, test your code in Linux, since this is the OS that will be used for grading.
- Make sure that your classes are public, so that they can be seen and accessed by applications that reference your DLL.
- Recall that you must only put .CS files in your .zip and you must be able to compile your code from the command line with: **mcs –target:library *.cs**
- Your zip file must not contain any nested folders. Zip from the command line to ensure this.
- Submit to the online system well before the due date, record the submission ID, and look up your grade. Deal with issues and resubmit if need be before the due date/time.

- This is a relatively simple assignment that everyone should be able to get full credit on. If you are having trouble, seek help early on or else all future assignments will be very difficult for you to do.