Overloading Operators

One of my favorite features in C# is the ability to overload operators. This is the process by which we create a custom class or struct and define for the compiler what mathematical and/or equality operators mean. For example, imagine you want to create an RPG that lets you combine like items to build stronger, better ones. We could overload the ‘+’ operator so the compiler and runtime will know what “sword1 + sword2” means.

In this assignment, you will make a tedious task (performing fraction math) simpler by creating a new struct that overloads the mathematical operators (excluding modulo) as well as the equality operators.

# Requirements

* You will be building a simple DLL for this assignment, NOT an executable application.
  + However, since you need to test, you should build a secondary executable application for your own testing purposes.
  + Your DLL MUST be named LastF\_OverloadingOperators, where “Last” is your last name and “F” is your first initial. Don’t forget the underscore.
    - Remember to change the assembly name using the project properties, NOT file explorer to manually rename the file after the fact. Do you remember why?
* Create a new struct (NOT a class) called Fraction
  + This struct MUST be in the namespace OverloadingOperators
* Fraction will have the following members:
  + Data Fields
    - A private int to hold the denominator; we need this later
  + Properties
    - A public int called WholeNumber
    - A public int called Numerator
    - A public int called Denominator
      * Adjust this property so that it uses the private data field for the denominator. This value must never be set to 0 for mathematical reasons. If the value passed in during a set call is 0, throw a new ArgumentException with a message stating the denominator can never be 0.
  + Constructor
    - public Fraction(int whole, int numerator, int denominator)
      * Use default parameters to make whole and numerator default to 0, but denominator default to 1
      * Use the arguments passed in to set all three properties, but you will also be required to set the private denominator data field explicitly. This is a consequence of using a struct. Make sure that you set the denominator field BEFORE setting the Denominator property. Besides the fact that the compiler forces this, why should you do this? What could happen if you don’t?
  + Static Methods
    - public int GCD(int m, int n)
      * This method needs to be static in order to be used in all the places where it is required. Additionally, since it does not mutate or directly access any instance data, there is no need for it to be an instance method.
      * GCD stands for “Greatest Common Divisor”, also known as greatest common factor. This method takes in two ints m and n and returns the largest int that can evenly divide both of them. You should use Euclid’s recursive algorithm to accomplish this.
    - Four methods to overload the +, -, \*, and / operators
      * These methods are meant to perform proper fraction math between two instances of Fraction. The resulting Fraction that is returned should be fully simplified (meaning both proper and reduced).
    - Two methods to overload the == and != operators
      * These two methods should compare the values found within instances of Fraction for mathematical equality, returning an appropriate bool value. This means that 1/3 is equal to 1/3, but it is also equal to 2/6 or 4/12 or 111/333 and so on. Similarly, a mixed fraction like 2 3/4 is equal to 11/4, 1 7/4 and so on.
  + Instance Methods
    - public void Reduce()
      * Reduces the fraction to its simplest terms. This does NOT make an improper fraction proper. Remember that an improper fraction is one where the numerator is greater than the denominator.
    - public void MakeProper()
      * This takes the current fraction values and ensures the Fraction is proper. This does NOT reduce the Fraction.
    - public void MakeImproper()
      * This takes the current fraction values and does the math to make the fraction improper. If the WholeNumber is 0, there will be no change in the Fraction’s state. This method does NOT reduce the Fraction.
    - public void Simplify()
      * Makes the Fraction both proper and reduced.
  + Override Methods
    - ToString()
      * Override the ToString method so that it creates a string representation of the value of the Fraction in the format W N/D. W is the WholeNumber value, N is Numerator, and D is Denominator. There is a space between W and N, and a forward slash between N and D. The output should also obey the following rules:
        + If WholeNumber is 0, don’t include it or the separating space character in the string output.
        + If Numerator is 0, ONLY provide the WholeNumber value (no space character, no fraction expression).
    - Equals() – OPTIONAL
      * Override the Equals method to use your == logic instead of just comparing memory addresses. This is optional since not overriding this method only causes a warning and not an error.
    - GetHashCode() - OPTIONAL
      * Override this method so that it simply returns base.GetHashCode(). There is no additional coding required here. This is optional since not overriding this method only causes a warning and not an error.
* It is requisite that NONE of your operator overloads mutate the original values of the Fractions in the operation.
  + Pro Tip: if you understand the difference between class and struct, this is way easier than you think.
* You have been provided with strict requirements above. It is expected that you will follow them to the tiniest detail. While you may feel free to add functionality you find useful, all names, parameter lists, expected return types, functionality, etc. as explained above are REQUIRED and non-negotiable.

# What you’ll need

* Understanding structs
* Overloading Operators

# Want a challenge?

* Implement proper detection of negative fractions. Adjust your operator overloads as needed

# Rubric

**Automatic Zero:** Your deliverable is not a class library project, your DLL is not named correctly, Fraction is not a struct, you change ANY of the provided names, parameter lists, or expected return types, or your DLL is otherwise not testable in any way.

This one is binary; either you met all the requirements, or you didn’t. No partial credit on this one.