CSC180 Assignment #06

**#1 -**Exercise that uses StreamReader, StreamWriter, Command-Line Arguments. Write a program that reads {**StreamReader**} a text file (**words.txt**) and inserts line numbers. Instead of displaying to the Console window, writes out {**StreamWriter**} to a text file (**words2.txt**). Your program runs on the command line like the following:  
      **Program.exe  Hello.cs  Hello2.cs**Useful resources: StreamReader sample code:

/\*

\* B.3 File input: S simple demo of reading a text file

\*/

using System;

using System.IO;

namespace ThinkSharp

{

public class ListFile

{

public static void Main()

{

processFile("words.txt");

}

private static void processFile(string filename)

{

string line;

int count = 1;

TextReader reader = new StreamReader(filename);

while (true)

{

line = reader.ReadLine();

if (line == null)

{

break;

}

Console.WriteLine(count + ": " + line);

count++;

}

reader.Close();

}

}

}

StreamWriter Class  
        <https://docs.microsoft.com/en-us/dotnet/api/system.io.streamwriter?view=net-5.0>    Command-Line Arguments  
        <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/main-and-command-args/command-line-arguments>

using System;

using System.IO;

namespace ThinkSharp

{

public class ListFile

{

public static void Main()

{

processFile("words.txt");

}

private static void processFile(string filename)

{

string line;

int count = 1;

TextReader reader = new StreamReader(filename);

TextWriter writer = new StreamWriter("words2.txt");

while (true)

{

line = reader.ReadLine();

if (line == null)

{

break;

}

writer.WriteLine(count + ": " + line);

count++;

}

reader.Close();

writer.Close();

}

}

}

C:Users\user\source\repos\ListFile\ListFile\bin\Debug>ListFile.exe words.txt words2.txt

**#2 - Exercise 9.3.**(pages 111-112) A rational number is a number that can be represented as the ratio of two integers. For example, 2/3 is a rational number, 355/133 is a good approximation of PI and you can think of 7 as a rational number with an implicit 1 in the denominator (7/1). For this assignment, you are going to write a class definition for rational numbers.   
**1.** Create a new program called Rational.cs that defines a class named Rational. A Rational object should have **two integer fields** to store the numerator and denominator.  
**2.** Write a default constructor, one that takes no arguments, and that sets the **numerator to 0 and denominator to 1.**  
**3.** Write a method called **WriteRational** that takes a Rational object as an argument and outputs it in some reasonable format.  
**4.** Write a **Main method** that creates a new object with type Rational, sets its fields to some values, and writes the object.  
**5.** At this stage, you have a minimal testable program. Test it and, if necessary, debug it.

**6.** Write a **parametric constructor** for your class that takes two arguments and that uses them to initialize the fields.

using System;

namespace Rational

{

class Rational

{

public int numerator, denominator;

public Rational ()

{

numerator = 0;

denominator = 1;

}

public Rational(int n, int d)

{

this.numerator = n;

this.denominator = d;

}

public static void WriteRational(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

class Program

{

public static void Main(string[] args)

{

Rational r1 = new Rational();

Rational.WriteRational(r1);

}

}

}

}

**7.** Write a method called **Negate** that reverses the sign of a rational number. This method should be a modifier, so it should return void. Add lines to Main to test the new method.

using System;

namespace Rational

{

class Rational

{

public int numerator, denominator;

public Rational ()

{

numerator = 0;

denominator = 1;

}

public Rational(int n, int d)

{

this.numerator = n;

this.denominator = d;

}

public static void WriteRational(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public static void WriteRational2(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public void WriteRational3()

{

Console.WriteLine($"{this.numerator}/{this.denominator}");

}

public static void Negate (Rational r)

{

r.numerator = r.numerator \* -1;

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

class Program

{

public static void Main(string[] args)

{

Rational r1 = new Rational();

Rational r2 = new Rational(1,2);

var r3 = new Rational(3,4);

Rational.WriteRational(r1);

Rational.WriteRational2(r2);

r3.WriteRational3();

Negate(r2);

}

}

}

}  
**8.** Write a method called **Invert** that inverts the number by swapping the numerator and denominator. Add lines to Main to test the new method.

using System;

namespace Rational

{

class Rational

{

public int numerator, denominator;

public Rational ()

{

numerator = 0;

denominator = 1;

}

public Rational(int n, int d)

{

this.numerator = n;

this.denominator = d;

}

public static void WriteRational(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public static void WriteRational2(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public void WriteRational3()

{

Console.WriteLine($"{this.numerator}/{this.denominator}");

}

public static void Negate (Rational r)

{

r.numerator = r.numerator \* -1;

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public static void Invert(Rational r)

{

Console.WriteLine($"{r.denominator}/{r.numerator}");

}

class Program

{

public static void Main(string[] args)

{

Rational r1 = new Rational();

Rational r2 = new Rational(1,2);

var r3 = new Rational(3,4);

Rational.WriteRational(r1);

Rational.WriteRational2(r2);

r3.WriteRational3();

Invert(r2);

}

}

}

}  
**9.** Write a method called **ToDouble** that converts the rational number to a double(floating-point number) and returns the result. This method is a pure function; it does not modify the object. As always, test the new method.

using System;

namespace Rational

{

class Rational

{

public int numerator, denominator;

public Rational ()

{

numerator = 0;

denominator = 1;

}

public Rational(int n, int d)

{

this.numerator = n;

this.denominator = d;

}

public static void WriteRational(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public static void WriteRational2(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public void WriteRational3()

{

Console.WriteLine($"{this.numerator}/{this.denominator}");

}

public static void Negate (Rational r)

{

r.numerator = r.numerator \* -1;

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public static void Invert(Rational r)

{

Console.WriteLine($"{r.denominator}/{r.numerator}");

}

public static void ToDouble(Rational r)

{

double deuce = ((double)r.numerator/(double)r.denominator);

Console.WriteLine(deuce);

}

class Program

{

public static void Main(string[] args)

{

Rational r1 = new Rational();

Rational r2 = new Rational(1,2);

var r3 = new Rational(3,4);

Rational.WriteRational(r1);

Rational.WriteRational2(r2);

r3.WriteRational3();

ToDouble(r2);

}

}

}

}  
**10.** Write a modifier named **Reduce** that reduces a rational number to its lowest terms by finding the greatest common divisor (GCD) of the numerator and denominator and dividing through. This method should be a pure function; it should not modify the fields of the object on which it is invoked. See Exercise 5.9 for the algorithm to find the GCD.  
using System;

namespace Rational

{

class Rational

{

public int numerator, denominator;

public Rational()

{

numerator = 0;

denominator = 1;

}

public Rational(int n, int d)

{

this.numerator = n;

this.denominator = d;

}

public static void WriteRational(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public static void WriteRational2(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public void WriteRational3()

{

Console.WriteLine($"{this.numerator}/{this.denominator}");

}

public static void Negate(Rational r)

{

r.numerator = r.numerator \* -1;

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public static void Invert(Rational r)

{

Console.WriteLine($"{r.denominator}/{r.numerator}");

}

public static void ToDouble(Rational r)

{

double deuce = ((double)r.numerator / (double)r.denominator);

Console.WriteLine(deuce);

}

public static int GCD(int a, int b)

{

int r = a % b;

if (r == 0)

return b;

else

return GCD(b, r);

}

public static Rational Reduce(Rational r)

{

int n = r.numerator / GCD(r.numerator, r.denominator);

int d = r.denominator / GCD(r.numerator, r.denominator);

return (new Rational(n, d));

}

class Program

{

public static void Main(string[] args)

{

Rational r1 = new Rational();

Rational r2 = new Rational(2, 4);

var r3 = new Rational(8, 16);

Rational r4 = new Rational(3, 9);

Rational.WriteRational(r1);

Rational.WriteRational2(r2);

r3.WriteRational3();

Rational.WriteRational(Rational.Reduce(r4));

Rational.WriteRational(Rational.Reduce(r3));

Rational.WriteRational(Rational.Reduce(r2));

}

}

}

}  
**11.** Write a method called **Add** that takes two Rational numbers as arguments and returns a new Rational object. The return object should contain the sum of the arguments. There are several ways to add fractions. You can use any one you want, but you should make sure that the result of the operation is reduced so that the numerator and denominator have no common divisor (other than 1).

using System;

namespace Rational

{

class Rational

{

public int numerator, denominator;

public Rational()

{

numerator = 0;

denominator = 1;

}

public Rational(int n, int d)

{

this.numerator = n;

this.denominator = d;

}

public static void WriteRational(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public static void WriteRational2(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public void WriteRational3()

{

Console.WriteLine($"{this.numerator}/{this.denominator}");

}

public static void Negate(Rational r)

{

r.numerator = r.numerator \* -1;

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public static void Invert(Rational r)

{

Console.WriteLine($"{r.denominator}/{r.numerator}");

}

public static void ToDouble(Rational r)

{

double deuce = ((double)r.numerator / (double)r.denominator);

Console.WriteLine(deuce);

}

public static int GCD(int a, int b)

{

int r = a % b;

if (r == 0)

return b;

else

return GCD(b, r);

}

public static Rational Reduce(Rational r)

{

int n = r.numerator / GCD(r.numerator, r.denominator);

int d = r.denominator / GCD(r.numerator, r.denominator);

return (new Rational(n, d));

}

public static Rational Add(Rational r2,Rational r4)

{

Rational retAdd = new Rational(0,0);

if (r2.denominator == r4.denominator)

{

retAdd.numerator = r2.numerator + r4.numerator;

retAdd.denominator = r2.denominator;

return(retAdd);

}

else

{

retAdd.numerator = r2.numerator \* r4.denominator;

retAdd.denominator = r2.denominator \* r4.denominator;

r4.numerator = r4.numerator \* r2.denominator;

r4.denominator = r4.denominator \* r2.denominator;

retAdd.numerator += r4.numerator;

int remain, num1, num2;

num1 = retAdd.numerator;

num2 = retAdd.denominator;

remain = num1 % num2;

while (remain != 0)

{

remain = num1 % num2;

num1 = num2;

num2 = remain;

}

int GCD = num1;

retAdd.numerator = retAdd.numerator / GCD;

retAdd.denominator = retAdd.denominator / GCD;

return(retAdd);

}

}

class Program

{

public static void Main(string[] args)

{

Rational r1 = new Rational();

Rational r2 = new Rational(2, 4);

var r3 = new Rational(8, 16);

Rational r4 = new Rational(3, 9);

Rational.WriteRational(Add(r2, r4));

}

}

}

}

**#3 - Exercise 13.4.** (page 158) This exercise is a continuation of Exercise 9.3. The purpose is to practice the syntax of instance methods and get familiar with the relevant error messages.  
**1.** Transform the methods in the Rational class from static methods to instance methods and make the necessary changes in Main. An instance method belongs to the instance, not the class. To define an instance method, omit static from the method heading and define the variables and methods internally.

using System;

namespace Rational

{

class Rational

{

int numerator, denominator;

Rational()

{

numerator = 0;

denominator = 1;

}

public Rational(int n, int d)

{

this.numerator = n;

this.denominator = d;

}

public void WriteRational(Rational r)

{

Console.WriteLine($"{this.numerator}/{this.denominator}");

}

public static void WriteRational2(Rational r)

{

Console.WriteLine($"{r.numerator}/{r.denominator}");

}

public void WriteRational3()

{

Console.WriteLine($"{this.numerator}/{this.denominator}");

}

class Program

{

public static void Main(string[] args)

{

Rational r1 = new Rational();

Rational r2 = new Rational(2, 4);

var r3 = new Rational(8, 16);

Rational r4 = new Rational(3, 9);

r1.WriteRational(r1);

Rational.WriteRational2(r2);

r3.WriteRational3();

}

}

}

}  
**2.** Make a few mistakes. Try invoking static methods as if they were instance methods and vice-versa. Try to get a sense for what is legal and what is not, and for the error messages that you get when you mess up. The error message states that an object reference is required for a non-static field, method, or property. I had to refer to the object(r1) instead of the method (Rational).

**3.** Think about the pros and cons of static and instance methods. Which is more concise (usually)? Which is a more natural way to express computation (or, maybe more fairly, what kind of computations can be expressed most naturally using each style)? In object-oriented programming, instance method definitions are more concise. They are lower-level definitions. Setting everything to pubic allows the objects or variables to be modified by another class. I believe it is still difficult to me because I learned C# with everything being public and static initially. I believe high-level OOP would be more naturally conducive to instance methods because it requires you to create an object. Static method is good for sharing methods in classes so mathematical formulas so you don’t have to constantly refer to an object directly every time.