Abraham Neuwirth
MCO 368 - Advanced Topics in OOP
January 17, 2016

Assignment 8

The program consists of a `Program` class which contains the `Main` method, and a static class called `Goldbach` which contains a method called `Composition` that returns a tuple of a Goldbach composition for the int that was passed in plus several helper methods

Goldbach Algorithm

The static `Composition` method relies on several helper methods to find a Goldbach composition. The helper methods are defined as follows:

```
public static bool AreNotPrime(int x, int y)
{
    return y.IsNotPrime() || x.IsNotPrime();
}

public static bool IsNotPrime(this int x)
{
    return !x.IsPrime();
}

public static bool IsPrime(this int x)
{
    for (var i = 2; i <= Math.Sqrt(x); i++)
    {
        if (x % i == 0)
            return false;
    }
    return true;
}
```

The `Composition` method is defined as follows:

```
public static Tuple<int, int> Composition(int x)
{
    if (x == 4) return new Tuple<int, int>(2, 2);
    for (int i = 3; i <= x/2; i += 2)
    {
        if (AreNotPrime(i, x-i)) continue;
        return new Tuple<int, int>(i, x - i);
    }
    return null;
}
```

The number 4 is a special case because for any other number we prefer to not even consider any multiple of two as possible candidate for a Goldbach composition. In fact, it can be proven that 4 is the only number for which 2 is part of its Goldbach composition.

The general algorithm is simple enough: we iterate over all the numbers from 3 up to half of x, where x is the number whose Goldbach composition we're trying to find. For each number we check to see whether it and the difference of x and it are prime. If they are, we deem it a Goldbach composition and we return. If the for loop terminates before finding a Goldbach composition null is returned, because, come on, the Goldbach conjecture ensures us no null will ever be returned (at least before the computer runs out of memory.)

The Program class

The class reles on small helper functions to simplify IO:

```
private static void Print(string s)
{
    Console.WriteLine(s);
}

private static string Read(string s)
{
    Print(s);
    return Console.ReadLine();
}

private static int ReadToInt(string s)
{
    return Read(s).ToInt();
}

private static int ToInt(this string s)
{
    return Convert.ToInt32(s);
}
```

To do the actual threading, four global variables are declared:

```
private static int _upTo;
private static int _n = 2;

private static readonly Dictionary<int, Tuple<int, int>> Goldbachs = new Dictionary<int, Tuple<int, int>>();
private static readonly object Lock = new object();
```

The `_upto` will hold the number up to which the user would like the program to confirm the Goldbach conjecture. `_n` is a global counter which we use to track which numbers still need to be confirmed. `Goldbachs` is a Dictionary for all the found compositions, and `Lock` is just an object for threads to lock on.

`_n` gets updated using a thread-safe property:

```
private static int NextEvenNumber => Interlocked.Add(ref _n, 2)
```

The `CalculateGoldachs` is the function that does the actual work of getting the compositions and storing them in the Dictionary.

```
private static void CalculateGoldbachs(CancellationToken signal)
{
    int x = NextEvenNumber;
    while (!signal.IsCancellationRequested && x <= _upTo)
    {
```

```
        var composition = Goldbach.Composition(x);
        lock (Lock) Goldbachs.Add(x, composition);
        x = NextEvenNumber;
    }
}
```

The function takes a `CancellationToken` that it will use to receive a halt signal from the outside world.

The code runs in a loop while two conditions are met: One, no cancellation has been requested, and two, the next even number is less then or equal to `_upTo`.

Inside the loop, we compute a Goldbach composition for the current number, lock on our `Lock` object and add it to the Dictionary. We then advance to the next even number and go back to the beginning of the loop.

For `Main`, two helper properties are defined:

```
private static bool EnterKeyPressed => Console.ReadKey().Key == ConsoleKey.Enter;

private static bool EnterKeyDetected => Console.KeyAvailable && EnterKeyPressed;
```

The `EnterKeyPressed` waits for the user to enter a key and checks if it was the Enter key. `EnterKeyDetected` checks if there is any input from the keyboard before actually blocking its thread to see what key was pressed.

The `Main` method sets up the `Task`s and reports the results back to the user:

```
static void Main(string[] args)
{
    _upTo = ReadToInt("Up to what number would you like to confirm the Goldbach conjecture?");
    var numberOfThreads = ReadToInt("How many threads would you like to use?");
    Print("You may quit at any time by pressing enter");

    var taskManager = new CancellationTokenSource();
    var cancelSignal = taskManager.Token;

    var tasks = new List<Task>();
    for (var i = 0; i < numberOfThreads; i++)
    {
        var t = Task.Factory.StartNew(() => CalculateGoldbachs(cancelSignal));
        tasks.Add(t);
    }


    while (!Task.WhenAll(tasks).IsCompleted)
        if (EnterKeyDetected)
            taskManager.Cancel();

    Print("Done. Press Enter to see results.");
    if (EnterKeyPressed)
        Goldbachs
            .OrderBy(t => t.Key)
            .ToList()
            .ForEach(t => Print($"{t.Key} = {t.Value.Item1} + {t.Value.Item2}"));
    Console.ReadKey();
}
```

Let's break it down. First we get the number up to which the program will run. Next we get the number of threads. Then we set up the `CancellationTokenSource` which is used to send the cancellation signal to the `Task`s and, the `Token` which is passed to the delegate inside the task which in turn uses it to intercept the cancellation signal.

A `List` of tasks is initialized, and then a loop initialized the number of `Task`s as specified by the user. To each task, we pass in the `CalculateGoldbachs` as an action, and to `CalculateGoldbachs` itself we pass in our `Token`.

Next, a while loop runs, waiting for all the tasks to complete. (`WhenAll` was used for this rather then `WaitAll`, because `WhenAll` creates a new `Task` which exposes a bool property we can use as the loop's conditinal.) All the while we're waiting for all the tasks to complete, we use `EnterKeyDetected` to check if the user entered a key. If the user enters one, it will block the thread to see if it's the Enter key, and if it is, the cancellation signal will be sent. In all likelihood, it will take a short while before all the tasks will all get the halt signal, so the while loop will continue to run, but soon enough it the condition `!Task.WhenAll(tasks).IsCompleted` will return false and the program will continue on. In the case the user doesn't press any key, the while loop will continue spinning till all the tasks are truly done, and it will then move on.

Finally, we ask for Enter to be pressed in order to show the results, then use Linq extension methods to present the results in sorted order.