WORKSHOP 2 – MORE C# AND ASYNC TASKS

PURPOSE OF THE WORKSHOP

The purpose of this week is to give you some more practice using C# and visual studio. Next week we will be starting the development of an actual compiler for a specific target language and virtual machine. You may have already realised that the techniques we are looked at last week and this week will be used in our compilers.

We will also this week look at the dotnet Command Line Interface (CLI) tools. If you are doing this on your own machine, you will need to make sure that .NET core (https://www.microsoft.com/net/core) is installed. The example I will show you from now on will largely be in visual studio, but you are free to use any editor you like. A word of warning though, many organisations use Visual Studio and may want you to have experience of this!

.NET core applications will run on any OS with the .NET core SDK installed. After this week it is up to you if you want to take the visual studio or CLI approach.

PART 1: CREATING A C# CONSOLE PROJECT FROM COMMAND LINE

Open a command prompt and create a new directory for your application in your **H: drive** (mkdir on most systems). Make that the current directory. Type the command **`dotnet new console`** at the command prompt. This creates the starter files for a basic "Hello World" application.

So what did we just do? The dotnet command runs the NuGet package manager (like npm for node and pip for python). Type **`dotnet restore`** at the command prompt. This tells NuGet package to download any of the missing dependencies for your project. As this is a new project, none of the dependencies are in place, so the first run will download the .NET Core framework. You will only need to run dotnet restore when you add new dependent packages, or update any of your dependencies.

Now run **`dotnet build`**. This executes the build engine and creates your application executable. Finally, you execute **`dotnet run`** to run your application.

PART 2: BETTER FILE READING

You can open the project in Visual Studio (or other editor) and see that all your code is in **program.cs.**

The first line uses the

```
using System;
```

remember from last week that this lets our code use anything from the C# systems namespace. You can see that the program itself enclosed in the **ConsoleApplication** namespace. That’s not a very descriptive name, so change it to

```
namespace TeleprompterConsole
```

you will see why in a minute.

Last week you developed one way of reading a file and printing out the contents, this week we are going to use threads to create a more efficient way of doing this. Download the sampleQuotes.txt file from moodle and put this in your project directory. This will act as the input script for your application.

Now, as you did last week, create a new class file called **QuoteReader**. Your class should be in the same **namespace** as your **program.cs.** It should have a constructor, which takes no parameters.

Finally add the method below underneath your constructor.

```
IEnumerable<string> ReadFrom(string file)
{
    string line;
    using (StreamReader reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

This method uses types from two new namespaces. For this to compile you'll need to add the following two lines to the top of your QuoteReader.cs file:

```
using System.Collections.Generic;
using System.IO;
```

The IEnumerable<T> interface is defined in the System.Collections.Generic namespace. The File class is defined in the System.IO namespace.

This method is a special type of C# method called an Enumerator. Enumerator methods return sequences that are evaluated lazily. That means each item in the sequence is generated as it is requested by the code consuming the sequence. Enumerator methods are methods that contain one or more yield return statements. The object returned by the **ReadFrom** method contains the code to generate each item in the sequence. In this example, that involves reading the next line of text from the source file, and returning that string. Each time the calling code requests the next item from the sequence, the code reads the next line of text from the file and returns it. When the file has been completely read, the sequence indicates that there are no more items.

Ok, now in your constructor for the QuoteReader add the following code to make the whole thing work.

```
IEnumerable<string> lines = ReadFrom("sampleQuotes.txt");
foreach (string line in lines)
{
    Console.WriteLine(line);
}
```

You can now run your code, either by using the **dotnet build** then **dotnet run** commands at the console, or by hitting the run button in visual studio.

PART 2: ADDING DELAYS

What you have is being displayed far too fast to read aloud. Now you need to add the delays in the output. As you start, you'll be building some of the core code that enables asynchronous processing. However, these first steps will follow a few anti-patterns. The anti-patterns are pointed out in comments as you add the code, and the code will be updated in later steps.

There are two steps to this section. First, you'll update the iterator method to return single words instead of entire lines. That's done with these modifications. Replace the `yield return line;` statement with the following code:

```
String[] words = line.Split(' ');
foreach (string word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

Next, you need to modify how you consume the lines of the file, and add a delay after writing each word. Replace the `Console.WriteLine(line)` statement in the Main method with the following block:

```
Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    Task pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

The **Task** class is in the **System.Threading.Tasks** namespace, so you need to add that using statement at the top of file.

Run the sample, and check the output. Now, each single word is printed, followed by a 200 ms delay. However, the displayed output shows some issues because the source text file has several lines that have more than 80 characters without a line break. That can be hard to read while it's scrolling by. That's easy to fix. You'll just keep track of the length of each line, and generate a new line whenever the line length reaches a certain threshold. Declare a variable after the declaration of words that holds the line length.

Then, add the following code after the `yield return word + " ";` statement (before the closing brace):

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Run the sample, and you'll be able to read aloud at its pre-configured pace.

PART 3: ASYNC TASKS

In this final step, you'll add the code to write the output asynchronously in one task, while also running another task to read input from the user if they want to speed up or slow down the text display. This has a few steps in it and by the end, you'll have all the updates that you need. The first step is to create an asynchronous Task returning method that represents the code you've created so far to read and display the file.

At the minute you are using your QuoteReader constructor to call the ReadFrom method. We want this to ask as an async task, so remove the code currently sitting in your constructor and create a new method called ShowTeleprompter. As you can see most of the required code you already have.

```
private async Task ShowTeleprompter()
{
    IEnumerable<string> words = ReadFrom("sampleQuotes.txt");
    foreach (var line in words)
    {
        Console.Write(line);
        if (!string.IsNullOrWhiteSpace(line))
        {
            await Task.Delay(200);
        }
    }
}
```

There are two small changes; First, in the body of the method, instead of calling Wait() to synchronously wait for a task to finish, this version uses the await keyword. In order to do that, you need to add the async modifier to the method signature. This method returns a Task. Notice that there are no return statements that return a Task object. Instead, that Task object is created by code the compiler generates when you use the await operator. You can imagine that this method returns when it reaches an await. The returned Task indicates that the work has not completed. The method resumes when the awaited task completes. When it has executed to completion, the returned Task indicates that it is complete. Calling code can monitor that returned Task to determine when it has completed.

Now in your constructor put the following line to call the task `ShowTeleprompter().Wait();`

Next, you need to write the second asynchronous method to read from the Console and watch for the '<' and '>' keys. Here's the method you add for that task:

```
private async Task GetInput()
{
    int delay = 200;
    Action work = () =>
    {
        do {
            ConsoleKeyInfo key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
        } while (true);
    };
    await Task.Run(work);
}
```

This creates a lambda expression (a special function that you can pass as a parameter, more info here https://www.codeproject.com/Tips/298963/Understand-Lambda-Expressions-in-minutes) to represent an Action delegate that reads a key from the Console and modifies a local variable representing the delay when the user presses the '<' or '>' keys. This method uses ReadKey() to block and wait for the user to press a key.

To finish this feature, you need to create a new async Task returning method that starts both of these tasks (GetInput and ShowTeleprompter), and also manages the shared data between these two tasks.

It's time to create a class that can handle the shared data between these two tasks. This class contains two public properties: the delay, and a flag to indicate that the file has been completely read. Create a third class called TelePrompterConfig.cs

```
namespace TeleprompterConsole
{
    internal class TelePrompterConfig
    {
        private object lockHandle = new object();
        public int DelayInMilliseconds { get; private set; } = 200;

        public void UpdateDelay(int increment) // negative to speed up
        {
            int newDelay = Min(DelayInMilliseconds + increment, 1000);
            newDelay = Max(newDelay, 20);
            lock (lockHandle)
            {
                DelayInMilliseconds = newDelay;
            }
        }
    }
}
```

Put that class in a new file, and enclose that class in the TeleprompterConsole namespace as shown above. You'll also need to add a using static statement so that you can reference the Min and Max method without the enclosing class or namespace names. A using static statement imports the methods from one class. This is in contrast with the using statements used up to this point that have imported all classes from a namespace.

```
using static System.Math;
```

The other language feature that's new is the lock statement. This statement ensures that only a single thread can be in that code at any given time. If one thread is in the locked section, other threads must wait for the first thread to exit that section. The lock statement uses an object that guards the lock section. This class follows a standard idiom to lock a private object in the class.

Next, you need to update the ShowTeleprompter and GetInput methods to use the new config object. Write one final Task returning async method to start both tasks and exit when the first task finishes:

```
private async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);
    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

The one new method here is the WhenAny(Task[]) call. That creates a Task that finishes as soon as any of the tasks in its argument list completes.

Next, you need to update both the ShowTeleprompter and GetInput methods to use the config object for the delay:

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    IEnumerable<string> words = ReadFrom("sampleQuotes.txt");
    foreach (string line in words)
    {
        Console.Write(line);
        if (!string.IsNullOrWhiteSpace(line))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}
```

then:

```
private static async Task GetInput(TelePrompterConfig config)
{

    Action work = () =>
    {
        do {
            ConsoleKeyInfo key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
        } while (!config.Done);
    };
    await Task.Run(work);
}
```

This new version of ShowTeleprompter calls a new method in the TeleprompterConfig class. Now, you need to update Main to call RunTeleprompter instead of ShowTeleprompter:

```
RunTeleprompter().Wait();
```

To finish, you'll need to add the SetDone method, and the Done property to the TelePrompterConfig class:

```
public bool Done => done;

private bool done;

public void SetDone()
{
    done = true;
}
```

Now try running your code and see what happens.