



# "Asynchronous Programming and Concurrency"

---

## Lab Manual

Wincubate ApS

19-06-2020



## Table of Contents

Exercise types .....	3
Prerequisites.....	3
Module 01: "Threads" .....	4
Lab 01.1: "Basic Threading" (★) .....	4
Module 02: "Blocking Synchronization" .....	6
Lab 02.1: "Locking Static Members" (★) .....	6
Lab 02.2: "Singleton Application Instance" (★) .....	7
Lab 02.3: "Bounded Number of Application Instances" (★★) .....	8
Lab 02.4: "Graceful Worker Threads" (★★★) .....	9
Module 3: "Signaling with Events and Handles" .....	11
Lab 03.1: "Graceful Waitable Worker Threads with Results" (★★) .....	11
Module 4: "Nonblocking Synchronization" .....	13
Lab 04.1: "Much Simpler Object Counters" .....	13
Module 5: "The Thread Pool" .....	14
Lab 05.1: "Speeding Up ASCII Art using the Thread Pool" (★★) .....	14
Module 6: "Data Parallelism in TPL" .....	16
Lab 06.1: "Speeding Up ASCII Art using the Parallel Class" (★) .....	16
Module A: "Case Study: WPF and Threads" .....	17
Lab A.1: "Making Threaded Updates Thread-safe in WPF" (★★) .....	17
Module 7: "Tasks in TPL" .....	19
Lab 07.1: "Basic Task with Result" (★) .....	19
Module 8: "Async, Await, and Task Combinators" .....	20
Lab 08.1: "Tasks" (★★) .....	20
Lab 08.2: "Handling Multiple Tasks" (★★★) .....	21
Module 9: "Concurrent Collections" .....	22
Lab 09.1: "Counting Words with ConcurrentDictionary" (★) .....	22
Module B: "Case Study: Concurrent Updates in WPF" .....	23
Lab B.1: "Sharing Information Between Clients" (★★★) .....	23
Lab B.2: "Processing All Exceptions in the V4 Solution" (★★★) .....	25

## Exercise types

The exercises in the present lab manual differs in type and difficulty. Most exercises can be solved by applying the techniques from the presentations in the slides in a more or less direct manner. Such exercises are not categorized further.

However, the remaining exercises differs slightly in the sense that they are not necessarily easily solvable. These are categorized as follows:



Labs marked with a single star denote that the corresponding exercises are a bit more loosely specified.



Labs marked with two stars denote that the corresponding exercises contain only a few hints (or none at all!) or might be a bit more difficult or nonessential. They might even require additional searches for information elsewhere than in the slide presentations.



Labs marked with three stars denote that the corresponding exercises are not expected in any way to be solved. These are difficult, tricky, or mind-bending exercises for the interested participants – mostly for fun! 😊

## Prerequisites

The present labs require the course files accompanying the course to be extracted in some directory path, e.g.

C:\Wincubate\90377

with Visual Studio 2019 (or later) installed on the PC.

We will henceforth refer to the chosen installation path containing the lab files as *PathToCourseFiles* .

## Module 01: “Threads”

### Lab 01.1: “Basic Threading” (★)

In this lab you will create a basic thread processing test fragments and start it.

- Open the existing console application in  
C:\Wincubate\90377\Module 01\Lab 01.1\Starter .

When you investigate the included project files, you will see that the DifficultWords.txt contain a huge text fragment in html scraped from a web page containing hundreds of identically-structured difficult English words.

As an example, the fragment for the word “amorphous” is contained in the follow block of text:

```
<li class="entry learnable" id="entry22"
lang="en" word="amorphous" freq="9002.76" >

<a class="word dynamictext"
href="/dictionary/amorphous">amorphous</a>

<div class="definition">having no definite form or distinct
shape</div>

<div class="details">


</div>

</li>
```

Your job is now to create a thread processing such fragments.

- Augment the existing program of the console application to create a thread
  - reading the file contents from the specified filename
  - extracting each word, e.g. “amorphous” using a regular expression
  - prints each word to the console as it is extracted.

Hint: If you find regular expressions a pain in the a\*\*e, you can use this following very simple expression:

```
Regex reg = new Regex(@"word\s*=\s*(?:'"'(?<1>[^\s"]*)"'|(?<1>\S+))",
    RegexOptions.IgnoreCase | RegexOptions.Compiled
);
```

Before shutting down there are a few things to discuss:

- When does the program terminate?
- Why?

## Module 02: “Blocking Synchronization”

### Lab 02.1: “Locking Static Members” (★)

This lab investigates thread safety of seemingly trivial programs.

- Open the existing console application in  
C:\Wincubate\90377\Module 02\Lab 02.1\Starter .

The program makes essential use of constructors and destructors to produce a running count of the number of instances of a given class.

- Inspect the program and make sure you realize how the program works and what it does.
- It is thread-safe? Why (not)?

Finally,

- Modify the ObjectCounter program above to ensure that it is thread-safe.

## Lab 02.2: "Singleton Application Instance" (★)

In this exercise we will investigate how synchronization primitives can be used to ensure critical sections across process boundaries.

- Open the existing WPF application in  
C:\Wincubate\90377\Module 02\Lab 02.2\Starter .

It contains a very simple WPF application which doesn't really do anything interesting.

- Modify the code in the WPF application such that it ensures, that at most one instance of the application is running at any given time.
- Which synchronization primitive did you use?

Note: Simplicity is your friend here! Don't go overboard with nice design and technicalities (just yet 😊).

### Lab 02.3: "Bounded Number of Application Instances" (☆☆)

Pretty satisfied with your solution to Lab 02.2: "Singleton Application Instance" (★), you show it off to your boss during the lunch break. She is very impressed with the idea.

However, she gets another idea! What if we could enforce a maximum count of how many application instances would be permitted to run simultaneously? So essentially the solution to Lab 02.2: "Singleton Application Instance" (★) is a special case, where the maximum count is 1.

- How would you modify your solution to Lab 02.2: "Singleton Application Instance" (★) (or the Starter located below in)  
C:\Wincubate\90377\Module 02\Lab 02.3\Starter  
to loosen the constraints such that at most 3 applications are permitted to run simultaneously.
- Is it possible to solve with the same synchronization primitive? Why (not)?



## Lab 02.4: "Graceful Worker Threads" (☆☆☆)

This exercise will investigate a well-known historical best practices of wrapper the Thread objects is a type-safe wrapper which allows a graceful shutdown of the underlying thread.

If you consider the solution to Lab 01.1: "Basic Threading" (☆☆) earlier, it was not immediately possible to shutdown the thread before the computation ran to completion. This is due to two factors:

- most threads are foreground threads keeping the application up and running
- it does not check for termination in its processing loop.

Also, most threads are not properly encapsulated a single class but merely exist a function "loose" function somewhere in the middle of your code.

For all these reasons, developers have historically often defined worker thread to expose a uniform interface, e.g. `IWorkerThread<TInput>`

```
interface IWorkerThread<TInput> : IDisposable
{
    void Start(TInput input);
    void Kill();
}
```

The `Dispose()` method would then simply invoke `Kill()`.

When implementing the interface for multiple thread classes it was helpful to create a common base class containing all the commonly used functionality. Such a reusable, abstract, generic `WorkerThreadBase<TInput>` class would be the essential weapon in remedying all the deficiencies outlined above.

This class would then manage a `System.Threading.Thread` object properly encapsulated.

The signature of the class was usually something along the lines of:

```
abstract class WorkerThreadBase<TInput> : IWorkerThread<TInput>
{
    public void Start(TInput input) { ... }
    protected abstract void Work(TInput input);
    public void Kill() { ... }
}
```

Long story short;

- Open the existing console application in  
C:\Wincubate\90377\Module 02\Lab 02.4\Starter .  
which is the solution from Lab 01.1: "Basic Threading" (☆☆).

**Note:** To force the thread to run a little longer (with the purpose of giving the user time to kill the thread manually), we have inserted a small delay loop inside the code of the word processor extraction method.

Now

- Implement `WorkerThreadBase<TInput> : IWorkerThread<TInput>` following the requirements discussed above.
- Rewrite the existing thread code into a new class `ExtractThread` deriving from the worker thread base class in the proper fashion.

You should make your `Main()` method implementation be as follows:

```
string htmlFileName = @"DifficultWords.txt";

ExtractThread workerThread = new ExtractThread();
workerThread.Start(htmlFileName);

Console.ReadLine();

workerThread.Kill(); // <-- Or could use the C# using construct
```

- Test your program!
  - When you run your program, the program should start displaying the extracted words, but terminate when you hit Enter.

## Module 3: “Signaling with Events and Handles”

### Lab 03.1: “Graceful Waitable Worker Threads with Results” (☆☆)

This lab continues our explorations into best practices which were initiated in Lab 02.4: “Graceful Worker Threads” (☆☆☆).

Occasionally threads compute some sort of result of their computation, which the program controlling the thread needs to use when the thread has run to completion – and hence computed the result. Consequently, it makes sense to produce threads with a `Result` property getter such that the result of the computation can be retrieved.

An appropriate interface might be

```
interface IWorkerThreadWithResult<TInput, TResult> : IWorkerThread<TInput>
{
    WaitHandle Completed { get; }
    TResult Result { get; }
}
```

The spirit of the interface should be as follows:

- i. The wait handle `Completed` is signaled when the thread has successfully run to completion.
- ii. The `Result` property returns the computed result (when completed)
  - a. If the `Result` property getter is invoked without the thread having run to completion, it will block – and wait for the thread to run to completion.

Your objective is now to implement an appropriately extended base class:

```
abstract class WorkerThreadWithResultBase<TInput, TResult> :
    WorkerThreadBase<TInput>
```

You will then modify `ExtractThread` accordingly.

- Open the existing console application in  
C:\Wincubate\90377\Module 03\Lab 03.1\Starter .  
which is the solution from Lab 02.4: “Graceful Worker Threads” (☆☆☆).

You are of course, once again, allowed to use your own solution, if you prefer.

Now

- Implement `WorkerThreadWithResultBase<TInput, TResult>` following the requirements discussed above.
- Rewrite the existing `ExtractThread` class to derive from the new worker thread base class in the proper fashion.

You should make your `Main()` method implementation be similar to the following:

```
string htmlFileName = @"DifficultWords.txt";

ExtractThread workerThread = new ExtractThread();
workerThread.Start(htmlFileName);

Console.WriteLine("Waiting for results to be completed...");

// Use workerThread.Completed.WaitOne(); to wait for completion, or:

foreach (string word in workerThread.Result)
{
    Console.WriteLine(word);
}
```

- Test your program!
  - When you run your program, the program should start displaying the extracted words exactly when they have all been computed (and then terminate gracefully).

## Module 4: “Nonblocking Synchronization”

### Lab 04.1: “Much Simpler Object Counters”

This lab makes the solutions to Lab 02.1: “Locking Static Members” (★) much lighter-weight and readable.

- Open the existing console application in  
C:\Wincubate\90377\Module 04\Lab 04.1\Starter ,

which is the solution to Lab 02.1: “Locking Static Members” (★).

Your objective now is easy to state:

- Use the `Interlocked` class (nonblocking synchronization) in the specified program instead of the `lock` keyword (blocking synchronization).

Needless to say, your program should of course still be thread-safe (but now much lighter-weight) 😊.

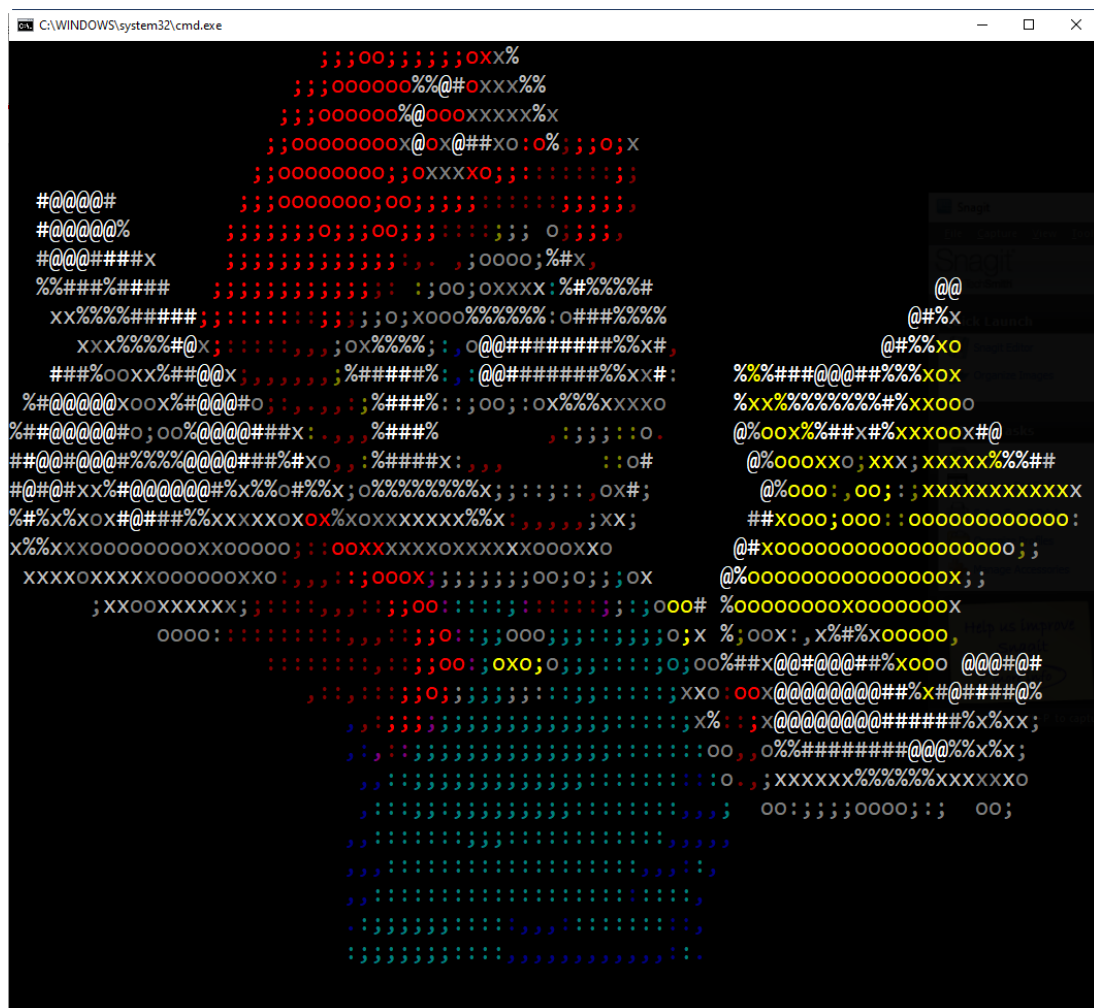
## Module 5: "The Thread Pool"

### Lab 05.1: "Speeding Up ASCII Art using the Thread Pool" (★★)

In this exercise we will speed up colored ASCII art using the thread pool.

- Open the existing console application in  
C:\Wincubate\90377\Module 05\Lab 05.1\Starter.
- Run the program and verify its console output.

Once the program has run to completion, you will see it load a bitmap file and produce a colored ASCII art picture similar to the following:



**Note:** Due to the fragileness of the console, you should probably not resize the console window while the program is painting... 😊

The console application roughly works as follows:

- It loads a bitmap from file
- It sets the width and height properties of the Console from the loaded image.

- iii. It traverses through all the pixels of the loaded bitmap and
  - a. Converts the intensity of the pixel to a corresponding character.
  - b. Inspects the pixel color and attempts to compute the “nearest” of the (very few) `ConsoleColor` values.
  - c. It then writes the character of a) in the `ConsoleColor` from b) to the Console.

These computations are performed in the **Wincubate.ColorClub** library referenced from the main console application. Fortunately, you don’t need to know any details of these hairy computations, because they’re encapsulated in the `AsciiArtHelper` class.

- Inspect the code of `Program.cs` to get an idea of how the library is used.

The computation is painfully slow (mostly due to the fact that it is deliberately slowed for instructional purposes within the inner workings of the `AsciiArtHelper` class 😊)...!

Somewhat frustrated, however, you do notice that the computation for each pixel is *exactly* the same – it only varies with its inputs. You get the idea that we might “parallelize” the overall computation by using the thread pool. Great idea!

- Use the thread pool to queue and run each single pixel computation independently
- A few hints:
  - Define a new type, e.g. `Input`, which contains the inputs required for each computation.
    - Why is that?
  - You can invoke `Console.SetCursorPosition()`
  - Is any locking necessary?

Finally;

- How does your new program compare to the original one?

## Module 6: “Data Parallelism in TPL”

### Lab 06.1: “Speeding Up ASCII Art using the Parallel Class” (★)

This lab investigates the use of the `Parallel` class to solve computational problems involving data parallelism.

- Open the existing console application in  
C:\Wincubate\90377\Module 06\Lab 06.1\Starter ,

which is the solution to Lab 05.1: “Speeding Up ASCII Art using the Thread Pool” (★★).

We will refactor the solution (using the thread pool) to instead use the Task Parallel Library.

- Refactor the console application to use `Parallel.For()` instead of the thread pool.

**Be warned!** Usually, you would just replace any for-loops with `Parallel.For()`, but this time you need to “prepare” the parallel computations in a bit more laborious manner.

It turns out that `Bitmap.GetPixel()` is not thread-safe – even though it merely reads the pixel color without modifying it. This is an inherent limitation of that class, which we need to work around.

Consequently, you need to

- Compute an array of all input instances sequentially first, *and*
- Then invoke `Parallel.For()` on the array of inputs performing the computations concurrently afterwards.

Run your program and test the result.

- What are the notable differences compared to using the thread pool?
  - Order?
  - Waiting for termination?
  - Easy of use?
  - ...



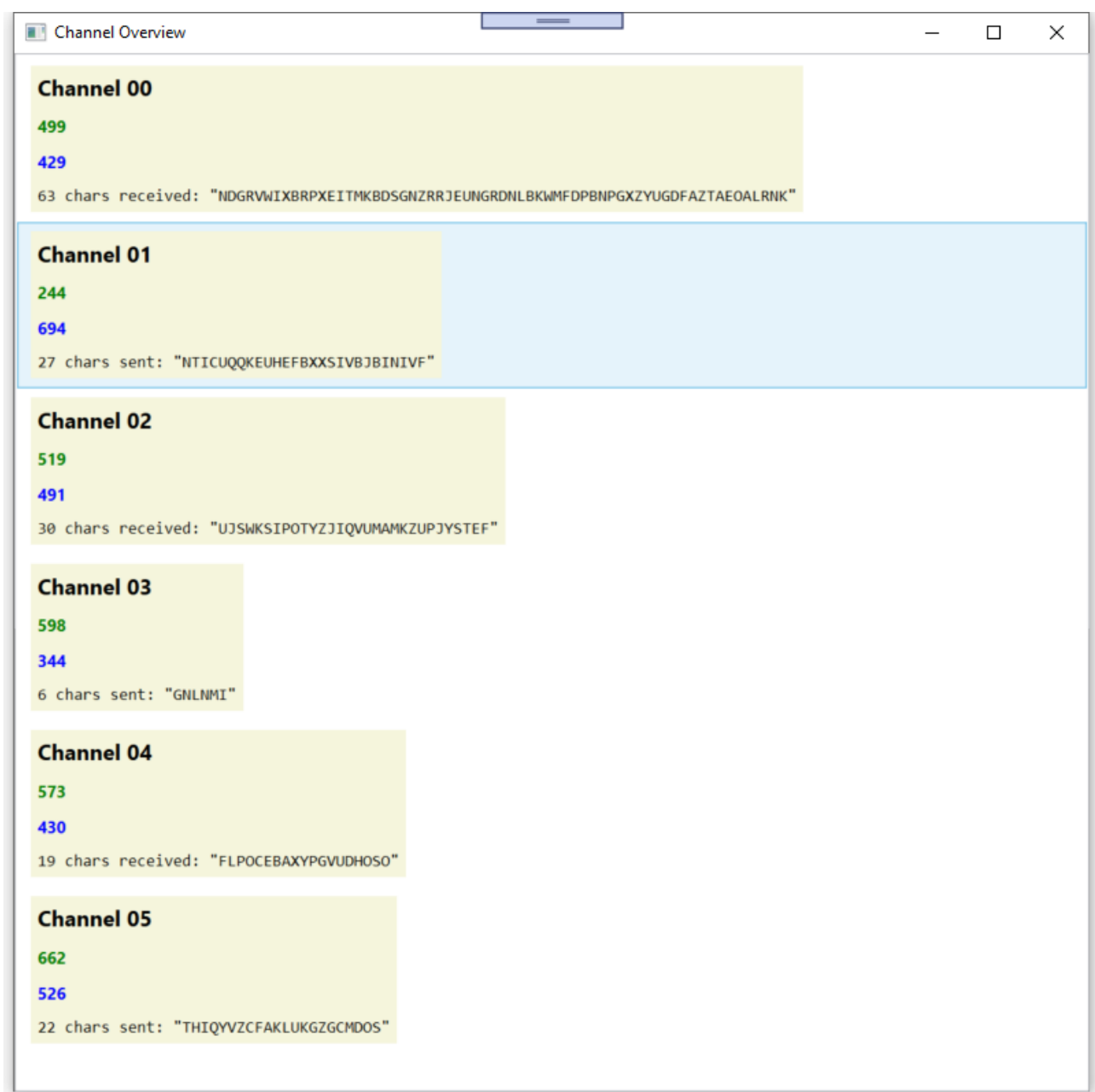
## Module A: “Case Study: WPF and Threads”

### Lab A.1: “Making Threaded Updates Thread-safe in WPF” (☆☆)

This exercise investigates how to make a WPF UI both thread-safe and “update-safe” providing detailed insight into how both updates to collections as well as to single elements are handled.

- Open the existing WPF application located in  
C:\Wincubate\90377\Case Study A\Lab A.1\Starter ,  
and get an overview of the existing classes, views, and view models.

The solution contains an application skeleton, where you need to process the updates of `ChannelMessage` instances from a dedicated worker thread to the UI such that your UI will resembles the following:



The application skeleton provided for you above for each incoming channel display a status consisting of:

- i. **Channel Description**
- ii. **Number of chars received**
- iii. **Number of chars sent**
- iv. Latest status

The models, views, and view models are already provided to you and are wired up appropriately. The only missing piece is the `MainViewModel.OnNewChannelMessage()` which needs to be completed:

```
// This will be called on a non-Dispatcher thread
internal void OnNewChannelMessage(ChannelMessage channelMessage)
{
    // TODO: Handle updates from worker thread correctly...
}
```

- You should complete the method in a manner such that
  - When the first `ChannelMessage` for a specified Index is encountered, a new, the view is updated with a new `ChannelViewModel` appropriately initialized with the data in the `channelMessage`.
  - When a `ChannelMessage` for a `ChannelViewModel` already existing in the UI is encountered, the existing `ChannelViewModel` is updated using `ChannelViewModel.UpdateWith()`
- Note: This needs to be done in a manner which is
  - **update-safe**, i.e the UI updates properly without exceptions when
    - new elements are added to the UI (the underlying collection)
    - properties of existing elements in the UI are updated (a single element)
  - **thread-safe**, i.e. if other sources (perhaps button presses or timers) also update the UI, they updates in `MainViewModel.OnNewChannelMessage()` are not “disturbed” in any way.

## Module 7: “Tasks in TPL”

### Lab 07.1: “Basic Task with Result” (★)

The purpose of this exercise is to create a simple task for the same computation for which we created our first thread.

- Open the existing console application in  
C:\Wincubate\90377\Module 07\Lab 07.1\Starter ,

which is essentially the solution to Lab 01.1: “Basic Threading” (★) (albeit with a small idle delay added for each iteration for instructional purposes).

- Refactor the program to create a `Task<IEnumerable<string>>` producing the word strings as the Result instead of printing the word individually to the console.
- Modify the main program to
  - Create and run the task
  - Retrieve the task’s result
  - Print all the word strings of the result.

You have essentially produced a task version of Lab 03.1: “Graceful Waitable Worker Threads with Results” (★★).

## Module 8: “Async, Await, and Task Combinators”

### Lab 08.1: “Tasks” (☆☆)

This exercise addresses the concept of authoring tasks, asynchronous methods, as well as using await. We will construct a program starting three tasks – each fetching the contents of a newspaper web page – and calculates the accumulated lengths.

- v. Open the program skeleton for the WPF application located in  
C:\Wincubate\90377\Module 08\Lab 08.1\Starter .
- vi. Firstly, implement the method  
`Task<string> CreateFetchTaskAsync( string url )`  
which creates an instance of WebClient and downloads the string located at the specified URL using `WebClient.DownloadStringTaskAsync()`.
- vii. Then implement  
`async Task<int> ComputeLengthSumAsync()`  
by filling out the TODO-part (adding an async modifier!), such that the method constitutes a task, which – when all subtasks complete – prints the total lengths of the strings downloaded by tasks t1, t2, and t3.
- viii. Finally, conclude by implementing  
`void OnComputeClick( object sender, RoutedEventArgs e )`  
such that it invokes `ComputeLengthSum()`, awaits that the result has finished computing, and then updates the UI.

## Lab 08.2: "Handling Multiple Tasks" (☆☆☆)

This lab investigates how to combine task combinators.

- ix. Open the WPF application located in  
C:\Wincubate\90377\Module 08\Lab 08.2\Starter.

The solution contains a project with a user interface with

- i. A button
- ii. Status text field.

Moreover, `MainWindow.xaml.cs` contains a method called `HeavyWork()`, emulating a lengthy computation with a duration between 0 and 10 seconds. When the button is clicked, the `OnClick()` method is executed.

- x. Open `MainWindow.xaml.cs` and complete the code in the various TODOs
  - TODO 1: Start the incarnations of `HeavyWork()`, such that they run simultaneously.
  - TODO 2: In the event that one (or more) of these tasks after 7 seconds is still running, use the status text field to show an appropriate string indicating that "tasks are still running".
  - TODO 3: When all tasks of TODO 1 have completed, use the status text field to display that "tasks are all complete".
- xi. Note: You should only use the classes `Task`, `Task<T>`, and the methods on these classes. 😊

## Module 9: “Concurrent Collections”

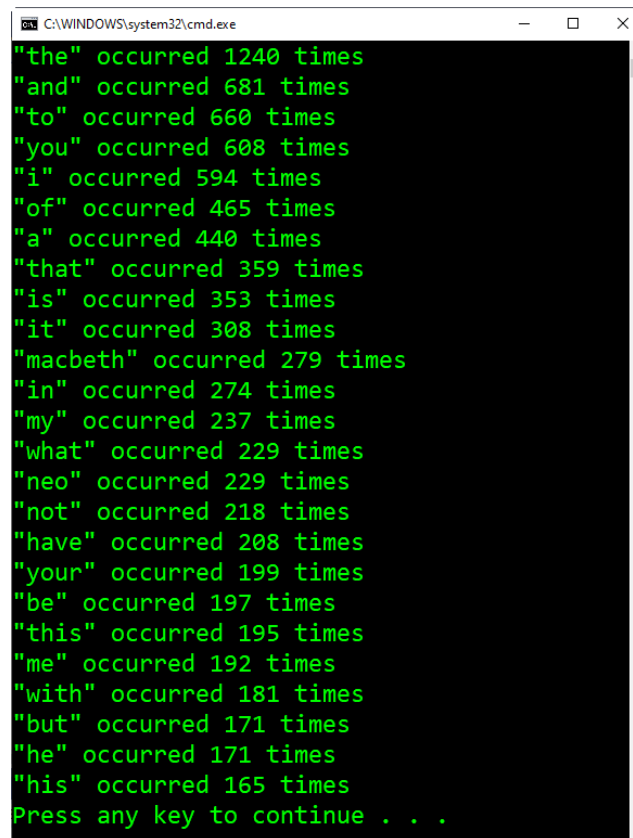
### Lab 09.1: “Counting Words with ConcurrentDictionary” (★)

This exercise deals with the concurrent collections which are a part of the TPL. We will use the [ConcurrentDictionary](#) to count occurrences of words in a number of different scripts.

- xii. Open the program skeleton for the console application located in  
C:\Wincubate\90377\Module 09\Lab 09.1\Starter .

This program contains some text-files containing various movie and play scripts with very many words. Moreover, the program contains a skeleton for a [WordCounter](#) class as well as the main program.

It contains a numbers of TODO which – when completed – will print out the 25 most frequently occurring words in the files:



```
C:\WINDOWS\system32\cmd.exe
"the" occurred 1240 times
"and" occurred 681 times
"to" occurred 660 times
"you" occurred 608 times
"i" occurred 594 times
"of" occurred 465 times
"a" occurred 440 times
"that" occurred 359 times
"is" occurred 353 times
"it" occurred 308 times
"macbeth" occurred 279 times
"in" occurred 274 times
"my" occurred 237 times
"what" occurred 229 times
"neo" occurred 229 times
"not" occurred 218 times
"have" occurred 208 times
"your" occurred 199 times
"be" occurred 197 times
"this" occurred 195 times
"me" occurred 192 times
"with" occurred 181 times
"but" occurred 171 times
"he" occurred 171 times
"his" occurred 165 times
Press any key to continue . . .
```

- xiii. Locate all the TODO's in the program skeleton and complete the program using [ConcurrentDictionary](#)
- to count the frequency of words in all scripts
  - using three tasks simultaneously running while concurrently updating the count in [ConcurrentDictionary](#)
- xiv. Note: Make sure that this is done in a thread-safe manner!

## Module B: “Case Study: Concurrent Updates in WPF”

### Lab B.1: “Sharing Information Between Clients” (☆☆☆)

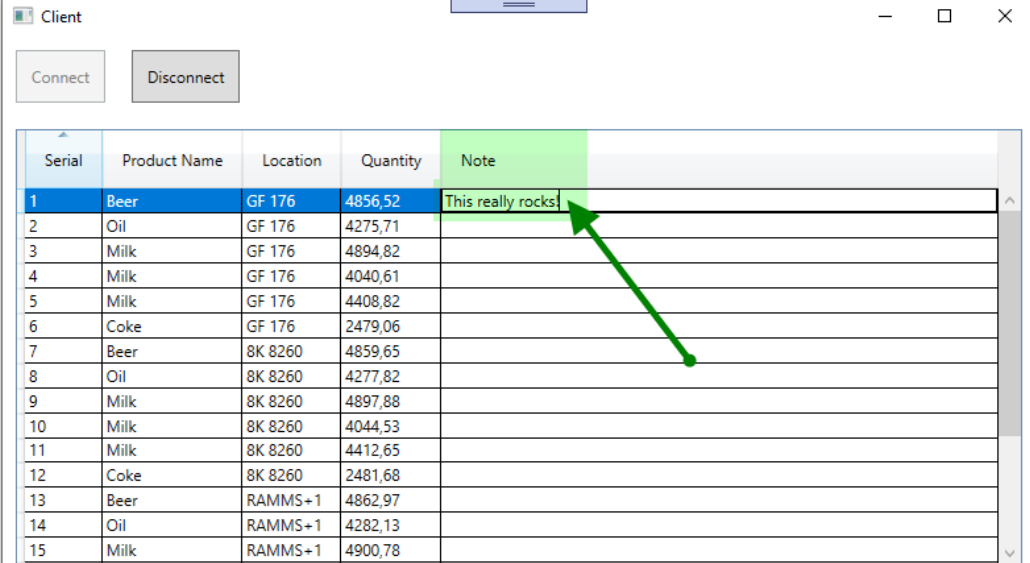
The provided solution in V4 of the presentation established a system with both a contract and a callback contract between Clients and Server. **You will probably need some knowledge of WPF (and MVVM) to complete this exercise.**

Both contracts contain methods for sending a Note between a Client to the Server such that the Server can relay this Note to all other Client instances.

- xv. Open the entire V4 system solution located in  
C:\Wincubate\90377\Case Study B\Lab B.1\Starter ,  
which is the V4 solution to Case Study B.

Note: Remember to run Visual Studio as an Administrator in order to be able to open ports for the WCF services. Also, consider setting multiple start-up project to start both **InstrumentApp**, **ServerApp**, and **ClientApp**.

- xvi. The ReagentsView.xaml file already contains some markup for a Note column in the [DataGrid](#). Remove comments for this code to activate the Note column.
- xvii. Complete all the remaining bits and pieces of the system to let the users edit notes in the Note column as depicted in the green shaded part below:



The screenshot shows a WPF application window titled "Client". It has two buttons at the top: "Connect" and "Disconnect". Below the buttons is a DataGrid with 5 columns: "Serial", "Product Name", "Location", "Quantity", and "Note". The "Note" column is highlighted in green. The first row of the DataGrid has the text "This really rocks!" in the "Note" column. A green arrow points to this cell.

Serial	Product Name	Location	Quantity	Note
1	Beer	GF 176	4856,52	This really rocks!
2	Oil	GF 176	4275,71	
3	Milk	GF 176	4894,82	
4	Milk	GF 176	4040,61	
5	Milk	GF 176	4408,82	
6	Coke	GF 176	2479,06	
7	Beer	8K 8260	4859,65	
8	Oil	8K 8260	4277,82	
9	Milk	8K 8260	4897,88	
10	Milk	8K 8260	4044,53	
11	Milk	8K 8260	4412,65	
12	Coke	8K 8260	2481,68	
13	Beer	RAMMS+1	4862,97	
14	Oil	RAMMS+1	4282,13	
15	Milk	RAMMS+1	4900,78	

- xviii. When the user edits the note, it should automatically be transferred by other connected clients and updated in their UI.
- xix. Here are some slightly simplifying assumptions that you should take:
  - In order to avoid reentrancy issues, just send the newly edited to all other clients except the originating client itself.
  - Don't worry about the order of notes if many are sent simultaneously
    - This means that if you trigger the sending of the note from the setter of the Note property of the [ReagentViewModel](#), just don't have to await the [Task](#).

- If you insist on doing this perfectly in the current setup, you would need at least Stephen Cleary's Mvvm.Async library or even resort to internal queues.
- Feel free to use a MVVM Framework with a Messenger component (such as MVVM Light or similar) to communicate between view models if you don't want to manually pass references upon construction.



## Lab B.2: "Processing All Exceptions in the V4 Solution" (☆☆☆)

The provided solution in V4 of the presentation illustrated a simple solution to make the Server App thread-safe. However, if we're ultimately pedantic about it, the solution provided there only logs the first (of theoretically many) task exceptions which can occur when sending a reagent data update to many clients simultaneously.

- xx. Open the entire V4 system solution located in  
C:\Wincubate\90377\Case Study B\Lab B.2\Starter ,  
which is the V4 solution to Case Study B.
- xxi. Modify the code to inspect all task exceptions which may have occurred when invoking client callbacks in the [ServiceMediator.OnReagentUpdatedAsync\(\)](#) method
  - Silently ignore `ObjectDisposedException` instances.
  - Log all other exceptions.

