Student No: L00129014\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date of Submission:\_29/09/22\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Asynchronous Programming

C# Async and await

# Objectives of Tutorial 2

* **Objective 1: Introduction to C# Tasks and how to create Task objects**
* **Objective 2: Use the Wait method to block the Current Main Thread**
* **Objective 3: Use the Task.Result Property to view results of a Task function.**
* **Objective 4: Call our I/O Asynchronous Tasks/Methods at the beginning of the program execution and use an await() when we need to view the results.**

**What is Asynchronous Programming?**

Asynchronous programming is a form of parallel programming that allows a unit of work to run separately from the primary application thread. When the work is complete, it notifies the main thread. We use a Task Asynchronous Programming techniques(TAP) which allows us to create constructs which model work being done in the background.

**Objective 1: Show sample code of how to create Task objects**

Creating a new C# Console project called Week2\_Tutorial1.

We will create three static methods to do various functions as before. Type the code below.

Text

Description automatically generated

Add the following three static methods within the internal class Program.

Graphical user interface, text, application

Description automatically generated

In the lecture, we discussed the Task object. Can you describe what is the objective of the Task class?

Tasks are objects that represent a unit of work that can be started and then delayed until later. We can then return back to that task after a period of time to complete it. Tasks are a higher-level of abstraction.

We want to create Tasks explicitly in C# to execute these methods.

To create Task objects, we need the System.Threading.Tasks library.

Lets create a new task for each method, namely t1, t2, t3. Adjust code as below;

Graphical user interface, text, application

Description automatically generated

Next we need to start the tasks.

Text

Description automatically generated

Tasks will be assigned a thread from the thread pool. So, we will print the Thread ID of the thread assigned to our task. Save changes and run program. Show and discuss the output returned?

Text

Description automatically generated

The output I received shows which method was completed first and by which thread. Thread3 completed method1 1st, Thread5 completed method2 2nd and Thread4 completed method3 3rd. This demonstrates how each ThreadID was assigned to a method and if method1 had been a larger process, then it could have been completed after method2 and method3.

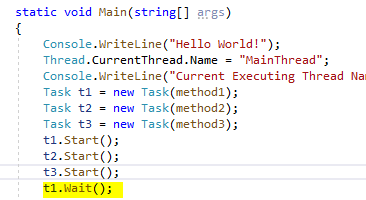
Notice that even though we call the tasks in order t1, t2, t3, all threads associated with the tasks are running asynchronously. In other words, the programs moves to another task before the previous one finishes.

Tasks allow us to use the Thread Pool efficiently. Once a task is complete, the thread is released back into the thread pool, where it can be reused by another task.

If you want to make the main thread execution wait until a child task is complete, then you need to use the ***Wait*** method of the Task class. The ***Wait*** method of the Task class will block the execution of the current threads until the assigned task has completed its execution.

**Objective 2: Use the Wait method to block the Current Main Thread**

Lets say we wanted the Main thread to wait until Task1 is completed. We would use the t1.wait() method to control this.



Run application and show output below.

Text

Description automatically generated

We can see now, that the Main Thread has waited on task1 to complete. Basically, the MainThread has been blocked! What if the MainThread was responsible for your UI? What would the consequences be?

If the UI was on the main thread then the UI would become unresponsive until it was complete.

*Wait is a synchronization method that causes the calling thread to wait until the current task has completed.*

If we change the code to below, what do you think the output will be?

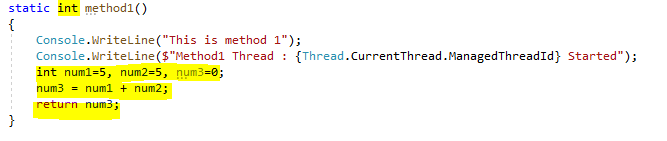
Text

Description automatically generated

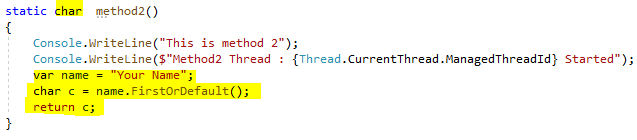
The wait method will cause the t2 thread to wait until the current thread is complete.

Lets now add some functionality to our methods.

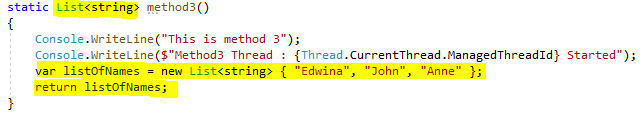
Method 1 will add two numbers together and return the result.



Method 2 will return the first letter of a string.



Method 3 will return a list of words.



So now, because the methods return different types, and we call the methods through a Task constructor, we must declare the Task objects of that specific type.

This time we will use the Task.Run() method to create and run a task. You should use Task.Run primarily not unless you need to explicitly create a Task and start it later. Adjust code as below;

Text

Description automatically generated

Run application. What happened?

When I ran the above code method3 did not print out the ThreadID from the WriteLine statement. Method2 either didn’t run at all or it simply did not print any WriteLine statements if it did run. The Stopwatch time was printed before the Method3 WriteLine was even printed to the console. This is because the main thread had completed before the methods were printed.

Text

Description automatically generated

As you can see, the code is running asynchronously. Change the code to ensure that the MainThread waits on t1 to complete. What changes did you make to ensure this? Show output of code to support your answer.

Graphical user interface, text

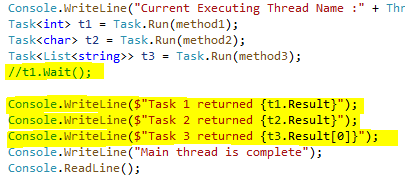
Description automatically generated

I added a Wait method to the t1 task object. I then added a long for loop that iterates through a large number and this is how I demonstrate that it waits for the task to be completed. If I remove the wait method, then it would complete before all the numbers were printed on screen.

We don’t see any results from the task during the execution of this program. We can use the Task.Result property for this.

**Objective 3: Use the Task.Result Property**

Next we need to view the results of our Tasks. To view the results of our Tasks t1, t2, t3, we will use .Result property. Change the code to resemble below. Make sure to comment out the t1.wait().



Run program and discuss the output?

The output shows the result of what was returned from the methods. I can see that Task1 holds the value 10, Task2 has the char “Y” and Task3 has “Edwina”. I can also see that even though I removed the Wait method from the main thread, it still waits for my previously long for loop to complete its iterations before moving on.

The results of the methods can be accessed via a Task.Result property. Have you noticed, that even though we removed the wait() method from Task1, the main thread is waiting for all tasks to complete before it completes.

The ***Result*** property of the Task object, blocks the calling thread until the task finishes its work. In other words, the calling thread (main thread) is blocked until all child threads have completed.

**Objective 4: Convert our Synchronous Methods to Asynchronous Tasks applying the Task Asynchronous Programming model.**

What we want is to use the TAP techniques to allow our I/O bound jobs to run in the background. To simulate long running I/O jobs, lets add a delay to each of our methods with a wait() operation applied.

Graphical user interface, text, application

Description automatically generated

So now, our 3 methods simulate long running I/O bound jobs. We will use Task.Run() to assign our I/O bound methods to a task and it should create a new thread for each task.

Our Main Method should resemble below;

Graphical user interface, text, application

Description automatically generated

Run application and show output. Can we discuss what is happening in our execution flow. What are the implications of Task.Run(methodname) and the wait()?

Text

Description automatically generated

I can see that all though I added a delay of 200ms, all the tasks are completing in the 216ms area. This demonstrates that each method is being assigned a Task and a Thread is being created for it.

An alternative to wait() is to use the ***await*** operation instead.

Await() is a keyword and is used to allow long running I/O bound jobs to run in the background.

Using ***await*** in a method means the current ***method*** is "paused" (its state is captured) and the method returns an incomplete task to its caller. Later, when the await expression completes, the remainder of the method is scheduled as a continuation.

The “async” keyword enables the “await” keyword in that method and changes how method results are handled.

So we need to implement TAP and use await() and async in our methods.

Step 1: First thing is to change the delay(200) function to use await instead of wait. This will allow other methods to execute statements while waiting on the delay to finish.

Text

Description automatically generated

Step 2: Add in the async keyword to the method signature and change the name of the method to append Async at the end.

Graphical user interface, text, application

Description automatically generated

Step 3 : Convert Return Type to Task<T>

When creating an async method, the return type must be of type Task<T>.

Tasks are constructs used to implement concurrent background operations. Therefore, we must change the return type to Task<T > as below.

Graphical user interface, text, application

Description automatically generated

Just to recap when creating an async method, perform the following changes to the method;

1. Append the word async to the end of the method name.
2. Add async keyword to the method signature
3. Use await keyword to allow background operation
4. Change the return type to Task<T>

Now change the name of the Tasks within Task.Run()

Text, letter

Description automatically generated

When we run this application, take note of the execution time. What are the implications of Task.Run(). How many threads were used?

The run time is now in the 221ms area which is higher than what it was before. I can see that 1 Thread is used for the main and then each method was assigned one unique Thread. The methods were completed in the following order. Method 1, Method 3, Method 2.

Now remove Task.Run() and just call the Tasks as below.

Text, letter

Description automatically generated

How many threads were used this time?

What are the implications of not using Task.Run(). Has the execution time improved? Show output and discuss.

Text

Description automatically generated

When not using Task.Run() I can see that each method was giving the same Thread. They also look to be completed in order, so Method 1, Method 2, Method 3. The completion time is staying quite steady in around the 222ms range so it hasn’t improved, I can see that the task could take 240ms sometimes.

**Objective 4:** **Call our I/O Asynchronous Tasks/Methods and use the await() when we need the result within our Main thread.**

In many .NET applications we will use a GUI to allow users to interact with databases and the web. When the user interacts with the web or a database, it is known as an I/O bound operation. While this operation is running, we want the GUI to remain responsive and these I/O bound operations to be done in the background..

In an ideal scenario, we could call all the required I/O bound operations simultaneously from the Main thread at the beginning of the application and then *await* their results. Only, when the results are required, we would allow the Main thread to compose the results and print results to screen.

Lets simulate that through our application and await each result as needed.

Text, application

Description automatically generated

Why did we have to change the Main method to return a Task object?

We changed it so that we could emulate what a interface with a GUI would be like. We needed the main as an async so that we could await the responses of the other tasks and this would stop the main from freezing if it were an app

What is the execution time?

226 milliseconds

How many threads are used in the execution of the program?

1

Task: Apply the TAP to the following program to make it as efficient as possible.

Graphical user interface, text, application

Description automatically generated

A picture containing text

Description automatically generated

Include the program code here:

static async Task Main(string[] args)

{

Task<int> t1 = DownloadPictureAsync();

Task<bool> t2 = GetConnectionToDBAsync();

Task<String> t3 = GetDataFromDBAsync();

int GetPictureResult = await t1;

bool GetConnectionResult = await t2;

String GetDataResult = await t3;

Console.WriteLine("Hello World");

Console.WriteLine($"{t1.Result} has been downloaded");

Console.WriteLine($"{t2.Result} - Database Connected");

Console.WriteLine($"{t3.Result} from DB ");

Console.WriteLine("This is the end of the main method.");

}

public async Task<bool> GetConnectionToDBAsync()

{

Console.WriteLine("Getting connection to database");

await Task.Delay(2000);

Console.WriteLine("Connection Up");

return true;

}

public async Task<String> GetDataFromDBAsync()

{

Console.WriteLine("Getting data from database");

await Task.Delay(2000);

string data = "Data Received";

Console.WriteLine(data);

return data;

}

public async Task<int> DownloadPictureAsync()

{

Console.WriteLine("Downloading Picture");

await Task.Delay(2000);

Console.WriteLine("Picture Downloaded");

return 100;

}

Can you justify your design decisions. Please elaborate.

I used an async approach so that the main thread would not become unresponsive as it did the background tasks of connecting to the database.

End of Tutorial.