[CHICAGO] CODER CONFERENCE

INNOVATION. THOUGHT LEADERSHIP. TRAINING.

Demystifying Async-Await

Understanding what is really going on and how to use it properly

or

"I do not think it means what you think it means"

David Allen

BLUEMETAL

An Insight Company

Microsoft Mobile App Partner of the Year 2017
Microsoft IoT Partner of the Year 2016
Xamarin Partner

BLUEMETAL
An Insight Company

June 26, 2017 • Room #204

Xamarin
Microsoft
.net

Asynchronous Programming

Asynchronous programming allows CPU time to be shared across multiple processes.

- This is vital where a process needs to be responsive or meet certain timing requirements, and where multiple ‘simultaneous’ actions need to be supported.
- In .Net the asynchronous processing control elements are:
  - Thread
  - ThreadPool
  - Task

Threads

Threads are the smallest element of processing control

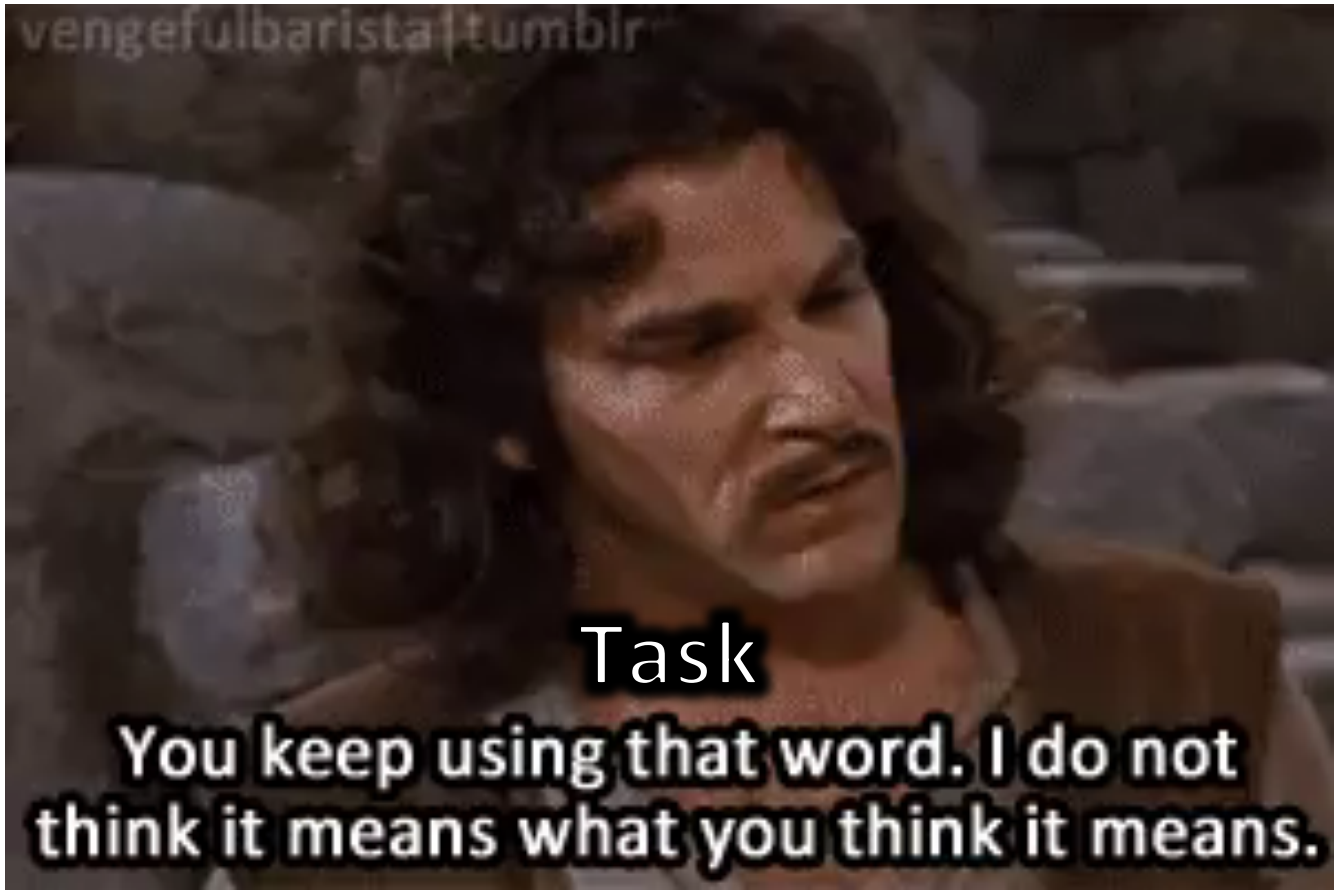
- Represent & maintain the actual OS resources required to run processes:
  - Stack
  - Kernel resources etc.
- Allows most atomic level of control:
  - Start, Stop, Abort, Suspend, Resume etc.
  - Observe state
  - Set properties
- Threads are costly
  - Consumes memory for Stack, Heap etc.
  - CPU overhead for context switching etc.
  - Takes time to instantiate a Thread

ThreadPools

ThreadPools address the issue of resources associated with Threads

- A collection, or 'pool' of pre-created Threads maintained by the CLR
- When running a process on a ThreadPool, it provides one of its managed Threads on which to run the code
  - Avoids overhead of dynamically creating Threads
  - Avoids overhead associated with too many Threads
  - Very little control - can control size of the pool, but little else
  - Submitting too many long running items can cause new items to be blocked
- No easy way to get results back from a process run via a ThreadPool or directly on a Thread

Tasks


vengefulbarista.tumblr
Task
You keep using that word. I do not think it means what you think it means.

Tasks

DO NOT

- Create or store Threads
- Tasks do not directly schedule code
  - The code is scheduled and managed by a TaskScheduler.

Tasks

DO

- Hold information
- Can pass back a result. Task has a Generic version Task<TResult> which can pass back a result of Type TResult from the asynchronously run code
- Include the following Properties (amongst others):
  - Result - contains the returned result of Type Tresult (only for Task<TResult>)
  - Status - contains a TaskStatus enumerable representing the Task's current state: Created, Running, RanToCompletion, Cancelled, Faulted, WaitingToRun, WaitingForActivation, WaitingForChildrenToComplete
  - Exception - contains an AggregateException that caused the Task to end prematurely or null if there is no Exception

Tasks

- There are two categories of Task
  - Delegate Tasks contain a reference to code that will be run asynchronously.
  - Promise Tasks do not have their own code, but represent other code or events

- Delegate Tasks may be Cancelled by passing a CancellationToken

Tasks

- There
  - Del ously.
  - Pro r events
- Deleg n

Tasks

- There are two categories of Task
  - Delegate Tasks contain a reference to code that will be run asynchronously.
  - Promise Tasks do not have their own code, but are ‘Virtual Tasks’ that represent other code or events

- Delegate Tasks may be Cancelled by passing a CancellationToken
  - This actually cancels the scheduling only, not the code to be run
  - To cancel the code to be run, the code (delegate) must explicitly observe the Cancelation Token

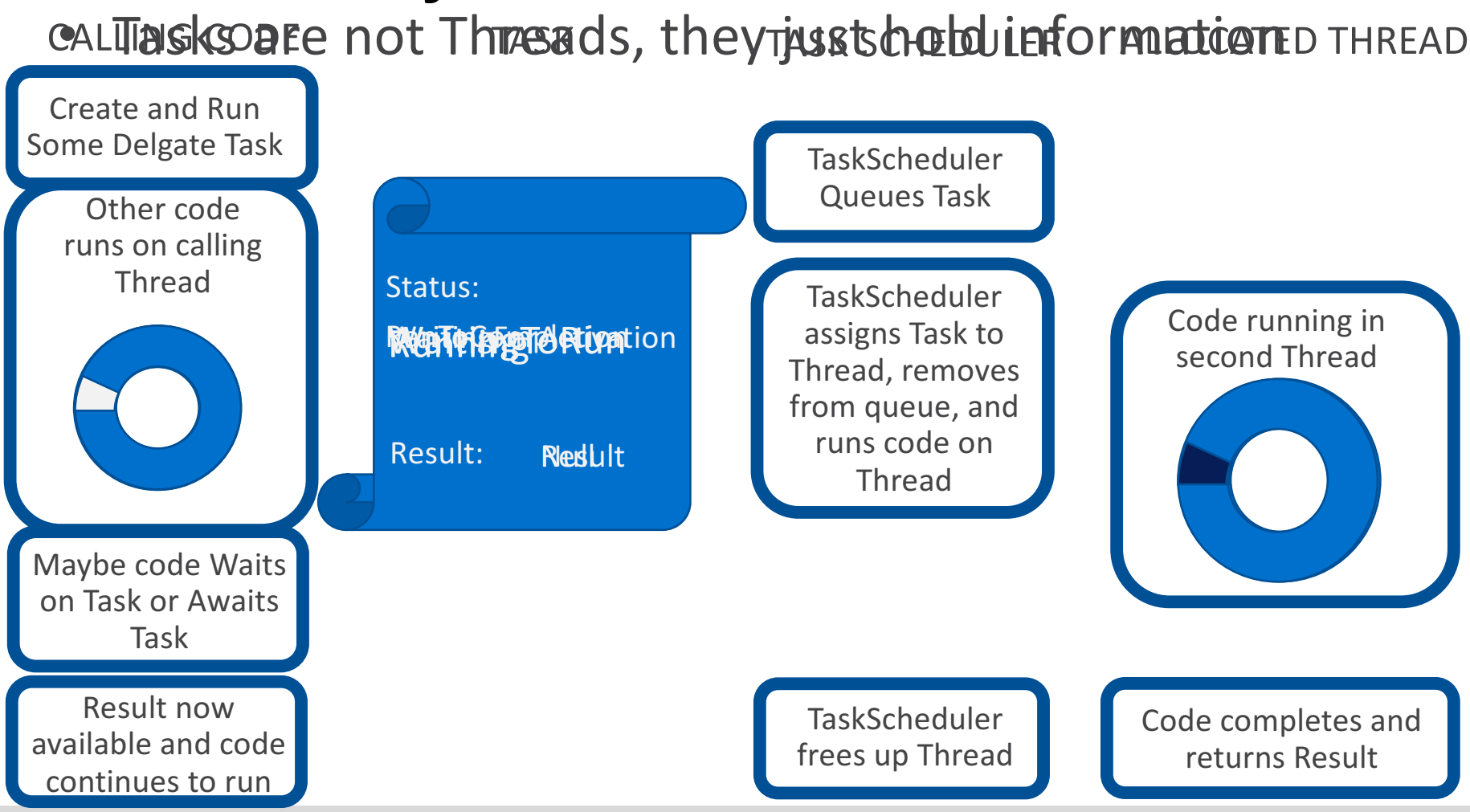
- Task provides a number of Continuation Methods which control the behaviour of the code that follows the Task definition, the context in which it runs, and the behaviour of the calling Thread

Task Scheduler

- The .Net default TaskScheduler:
  - Utilizes the Threadpool to run the delegated code
  - Maintains global and local queues of Tasks, which are used to queue related Tasks onto the same Threads reducing context switching.
  - When the item on a Thread completes, the next item (Task) in its queue runs
  - Implements 'Work Stealing' to maximize Thread use.
  - Very sophisticated and suitable in vast majority of scenarios
- The .Net CLR also provides alternative TaskScheduler that executes Tasks on the synchronization context of a specified target
- Can also create and schedule Tasks using a Custom TaskScheduler
- Tasks using default TaskScheduler shouldn't be used for long-running code
  - Task provides an option to create a new, dedicated Thread on which to run its referenced code - this should be used for long running code

Tasks and Asynchronous Behaviour

- Tasks are not Threads, they just hold information



Creating Tasks

- Don’t use Task Constructors
  - There are almost no circumstances where they are needed
- The Task is almost always needed to be scheduled immediately
- Use Task.Run or Task.Factory.StartNew to create a Delegate Task
  - Task.Run is the preferred mechanism
    - Uses default TaskScheduler
    - Is async aware
  - For more control, use Task.Factory.StartNew
    - By default uses the Current TaskScheduler, but can specify a different one
    - Not async aware
    - If you have an async delegate, it will return Task<Task<TResult>> rather than Task<TResult>
    - Can’t await Task<Task> have to use task.Unwrap() or use await await

Creating Tasks

```
Task<Task> task = Task.Factory.StartNew(async () =>
{
   while (IsEnabled)
   {
      await FooAsync();
      await Task.Delay(TimeSpan.FromSeconds(10));
   }
}, TaskCreationOptions.LongRunning);

Task actualTask = task.Unwrap();
await actualTask;
```

Creating Tasks

- If you're using async-await, always use Task.Run if you can
- If you're wrapping another asynchronous API or event, use Task.Factory.FromAsync or TaskCompletionSource<TResult>
  - Use to wrap old style asynchronous processes
- Usually you'll use the async keyword to create, or reference, a virtual Promise Task.
  - Rarely need to explicitly create Promise Task
  - Task.Delay is the most common scenario for creating a Promise Task

Task Creation Options

When creating a Task with Task.Factory.StartNew() and Task.FromAsync you can specify creation options

- LongRunning
- PreferFairness
- HideScheduler
- RunContinuationsAsynchronously
- AttachToParent
- DenyChildAttach
- None

Should not use LongRunning option in async-await world

- Consumes resources with no benefit

Task Waiting Methods

Tasks has a number of Synchronous Waiting methods

- Blocks the calling thread until condition met
- Do not use with Promise Tasks or awaited Tasks
- Common cause of deadlocks and async methods not apparently completing

Do NOT use with async-await

- Available Metods:
  - Wait – waits until Task complete
  - WaitAll – waits until all Tasks in a collection have completed
  - WaitAny – waits until one of the Tasks in a collection has completed
  - Result – has same effect as Wait but returns the Result, wraps exception
  - GetAwaiter().GetResult() – same as Result but doesn't wrap exceptions

Tasks Continuation Methods

Attaches a delegate that runs after a Task has completed

- task.ContinueWith - attaches code or delegate as a continuation to a Task to run more code once it has completed. Returns Task or Task<TResult>
- Task.Factory.ContinueWhenAny - executes single continuation when any of a collection of Tasks completes. Returns Task.
- Task.Factory.ContinueWhenAll - executes single continuation when all of a collection of Tasks complete. Returns Task
- Task.WhenAll - returns a task that completes when all of a set of tasks have completed. Async aware. Returns Task or Task<TResult[]>
- Task.WhenAny - returns a task that completes when any of a set of tasks has completed. Async aware. Returns the Task that completed

Async - Await

Await

- Await is an operator that takes an awaitable expression
- Task and Task<T> are awaitables. They can be awaited
- Do not need the async keyword to make them awaitable
- Can construct custom awaitables
- Awaitables must implement
  - GetAwaiter()
    - Must return an object the implements INotifyCompletion
    - Returned object must also expose
      - bool IsCompleted { get; }
      - void OnCompleted(Action continuation)
      - TResult GetResult()

Async - Await

Await

- Await examines the awaitable object
- If completed, immediately returns and method continues running
- If not completed:
  - Schedules the remainder of the method to run when awaitable completes
  - Returns from the current method to the calling code
  - When awaitable does complete, runs the remainder of the method
  - Behaves analogously to wrapping the remainder of the method in a ContinueWith, but returns control to the calling thread and implements a callback to execute when awaitable completes
- Default awaitables (Tasks) capture Synchronization Context and the remainder of the method will execute on that context when it runs
- Await unwraps the result from a completed generic awaitable

Async - Await

Async

- Async is just syntactical candy for the compiler to act on a method
  - Forces the return type of a method to be Task, Task<TResult>, or void
  - Allows the method to contain await statements
  - Causes a compilation error if there is an await statement in a method without the async keyword
  - Flags compiler warning of an async marked method does not contain an await statement
  - Wraps the returned type in a Task
- Beginning of async method is executed just like any other method
  - Flags compiler warning of an async marked method does not contain an await statement
- Convention to append “Async” suffix to method name

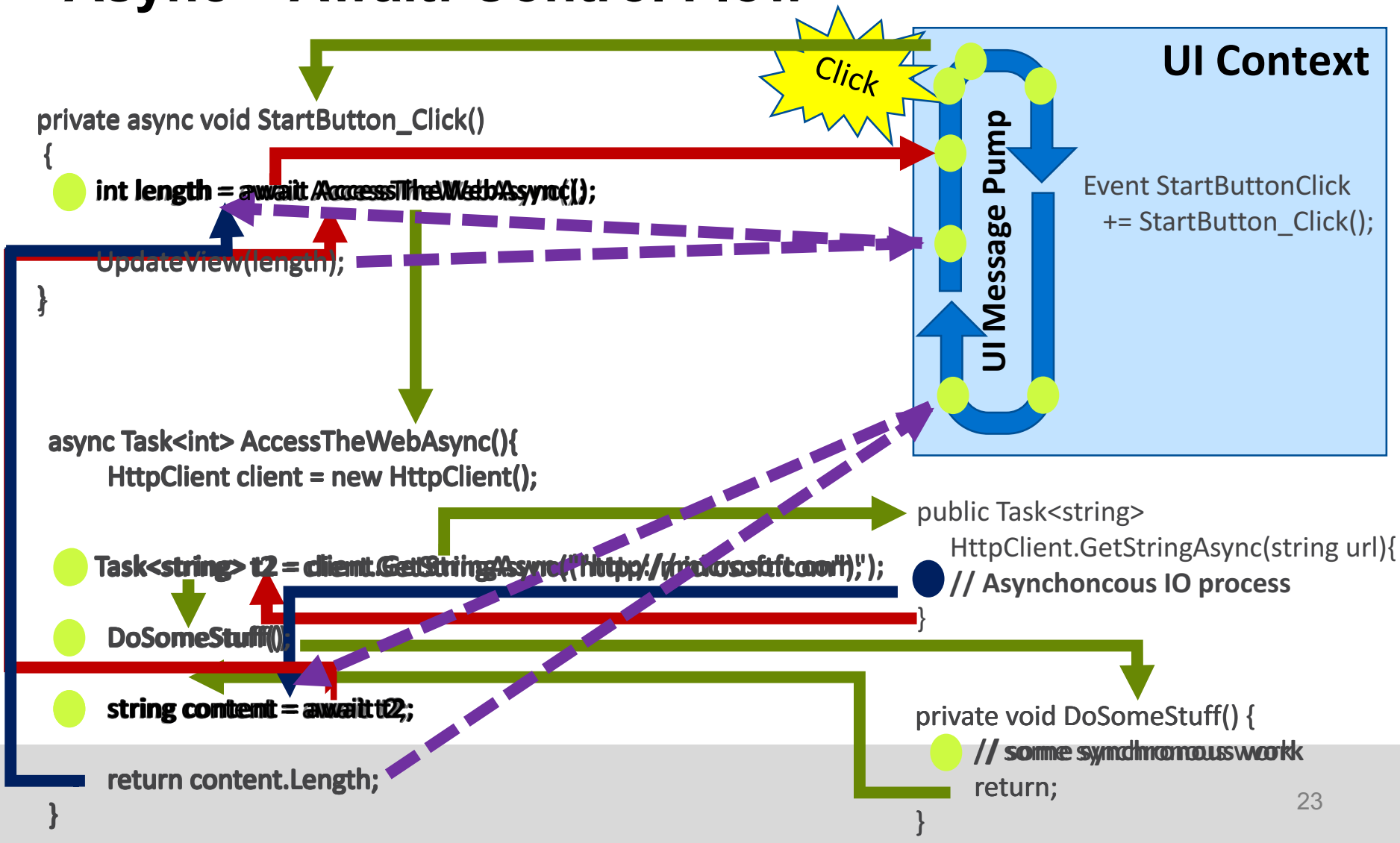
Async - Await

Async

- The supported return types are:
  - Task
  - Task<TResult>
  - void
- Task and Task<TResult> can be awaited, void can not
- Return Task if no value to be returned
- Return Task<TResult> to return a value
- Only return void for high level event handlers

DO NOT USE VOID with async-await pattern

Async – Await: Control Flow



Creating Async Methods From Synchronous Methods

- .Net framework and libraries provides many async methods
- If you need to create your own
  - Wrap the code in an Asynchronous Task
  - Refactor code to use less resources if possible
  - Use Task.Run to wrap the code where possible

```
public async Task<TResult> NewAsyncMethodAsync()
{
    return await Task.Run(() =>
    {
        return OldSyncMethod();
    });
}
```

```
public TResult OldSyncMethod()
{
    ……..
}
```

Synchronization Context

- Built in awaitables (Task) capture the Synchronization Context – the context in which the code is running
- All Task are executed on a Synchronization Context
- Synchronization Context is a collection of information that defines the environment on which code is executed. It could reference:
  - Thread
  - ThreadPool
  - TaskScheduler
- Current Synchronization Context is exposed as a static property of the SynchronizationContext class:
  - SynchronizationContext.Current
  - Not all Threads have a current Synchronization Context in which case it is null

Synchronization Context

- UI Thread is Synchronization Context for UI code
- A new Delegate Task will use default TaskSheduler and the Synchronization Context used by the delegate will reference a ThreadPool Thread
  - Running UI code on this will cause an exception
- Could create Task using alternative TaskScheduler and pass current Synchronization Context
  - All delegate code will run on the UI Thread, so its as if it was synchronous
- Could capture current SynchronizationContext (UI Thread) before the Task is created and run any UI updates on that context. .Net frameworks provide convenience methods to do this:
  - .Net: Invoke Post on the captured context
  - Xamarn.Forms: Device.BeginOnMainThread
  - iOS: InvokeOnMainThread
  - Android: RunOnUIThread

Synchronization Context

- If current Synchronization Context is null, then awaiting a Task will create a new Synchronization Context on the ThreadPool.
- In most cases there is no need to sync back to the Synchronization Context of the calling thread.
- This can be controlled by controlling how the awaitable captures the Synchronization Context
- Task has a ConfigureAwait(bool captureContext) method
  - If set to true, it will act in the default manner and capture the current context
  - If set to false, it will create a new one as if the current one was null
- Unless you have a reason to capture the context and sync back, it is good practice to set ConfigureAwait to false:
  - var result = await GetDataAsyc(stringRef).ConfigureAwait(false);

Creating Async Methods Revisited

- Once you have used ConfigureAwait(false) at some point within a method, it is good practice to use it for every awaited method from that point on
- As you don't know the context in which Library methods will be used, you should configure any awaits contained to false.
- If you wrap a synchronous method, its good practice to ConfigureAwait(false)

```
public async Task<TResult> NewAsyncMethodAsync()
{
    return await Task.Run(() =>
    {
        return OldSyncMethod();
    }).ConfigureAwait(false);
}
```

```
public TResult OldSyncMethod()
{
    ……..
}
```

Synchronization Context

CODE DEMO

See James Clancey's talk at Xamarin Evolve 2016 for more complete example of code demo
https://evolve.xamarin.com/session/56e1fe9ebad314273ca4d811

Awaiting Multiple Tasks

Use Task.WhenAll or Task.WhenAny to await completion of multiple Tasks

- Task.WhenAll - returns a task that completes when all of a set of passed tasks have completed. Returns Task or Task<TResult[]>
  - Task = Task.WhenAll(params Task []);
  - Task = Task.WhenAll(IEnumerable<Task>);
  - Task<TResult []> = Task<TResult>.WhenAll(params Task<TResult> []);
  - Task<TResult []> = Task<TResult>.WhenAll(Ienumerable<Task<TResult>>);

```
var client = new HttpClient();
Task<string> t1 = client.GetStringAsync("http://example.com");
Task<string> t2 = client.GetStringAsync("http://microsoft.com");
string[] results = await Task.WhenAll(t1, t2);
Console.WriteLine("Result0 = " + results[0] + "  and Result1 = " + results[1]);
```

Awaiting Multiple Tasks

- Task.WhenAny - returns a task that completes when any one of a set of passed tasks have completed. Returns Task or Task<TResult[]>
  - Task = Task.WhenAny(params Task []);
  - Task = Task.WhenAny(IEnumerable<Task>);
  - Task<TResult []> = Task<TResult>.WhenAny(params Task<TResult> []);
  - Task<TResult []> = Task<TResult>.WhenAny(Ienumerable<Task<TResult>>);

```
var client = new HttpClient();
Task<string> t1 = client.GetStringAsync("http://example.com");
Task<string> t2 = client.GetStringAsync("http://microsoft.com");
Task<string> resultTask = Task.WhenAny(t1,t2);
string result = await resultTask;
Console.WriteLine("The first result to be returned was " + result );
```

Awaiting Multiple Tasks

- You can use LINQ with Task.WhenAll and TaskWhenAny Task.WhenAll
  - Pass an IEnumerable as a LINQ statement

```
IEnumerable<string> urls = …..

var client = new HttpClient();
string[] results = await Task.WhenAll(urls.Select(url => client.GetStringAsync(url)));
Console.WriteLine("Result0 = " + results[0] + "  and Result1 = " + results[1]);
```

Task Completion Source

- If you need to create something such as an Asynchronous Event or Async Queue, you can use TaskCompletionSource
- TaskCompletionSource<TResult> wraps a Task and allows its state to be manually set:
  - Create the TaskCompletionSource<TResult>
  - Run some asynchronous code within which we set the result of the TaskCompletionSource (which sets it status)
  - Return the Task the TaskCompletionSource wraps, like any other Task
- When TaskCompletionSource is instantiated, the status of its Task is WaitingForActivation
  - Can call SetResult, SetCancelled, and SetException (or use Try versions) on the TCS
    - Appropriately sets the Tasks status and corresponding properties

Asynchronous Event Handler

- Can make an Event Async by wrapping it in a TaskCompletionSource

```
public Task<float> GetSignalStrengthAsync ()
{
   var tcs = new TaskCompletionSource<float> ();

   var centralManager = new CBCentralManager(DispatchQueue.CurrentQueue);

   centralManager.DiscoveredPeripheral += (object sender, CBDiscoveredPeripheralEventArgs e)
      => {tcs.SetResult(e.RSSI.FloatValue);
   };

   centralManager.FailedToConnectPeripheral += (object sender, CBPeripheralErrorEventArgs e)
      => {tcs.TrySetException(new Exception("Failed to connect to device"));
}; };

   return tcs.Task;
}
```

```
var bluetoothService =  new BlueToothService();
float bluetoothStrength = await bluetoothService.GetSignalStrength();
```

Cheat Sheets – How To

Objective	Synchronous approach	Async Approach
Get the result of a completed Task	task.Result	await task
Wait for a Task to complete	task.Wait	await task
Wait for one of a collection of Tasks to complete and retrieve result	Task.WaitAny or Task.Factory.WaitAny	await await Task.WhenAny
Wait for all of a collection of Tasks to complete and retrieve the results	Task.WaitAll or Task.Factory.WaitAll	await Task.WhenAll
Wait a period of time	Thread.Sleep	await Task.Delay
Create a Task	Task constructor	Task.Run or Task.Factory.StartNew

Cheat Sheets – How To

Problem	Solution
Create a task wrapper for an operation or event	TaskFactory.FromAsync or TaskCompletionSource<T>
Support cancellation	CancellationTokenSource and CancellationToken
Report progress	IProgress<T> and Progress<T>
Handle streams of data	TPL Dataflow or Reactive Extensions
Synchronize access to a shared resource	SemaphoreSlim
Asynchronously initialize a resource	AsyncLazy<T> * - nitoasyncex.codeplex.com
Async-ready producer/consumer structures	TPL Dataflow or AsyncCollection<T>

References

James Clancey's talk at Xamarin Evolve 2016
https://evolve.xamarin.com/session/56e1fe9ebad314273ca4d811

Any of Stephen Toub's blogs for Microsoft on TPL and Async-Await - especially for advanced topics
https://blogs.msdn.microsoft.com/pfxteam/2013/01/28/psychic-debugging-of-async-methods/

Any of Stephen Cleary's blogs on TPL and Async-Await
http://blog.stephencleary.com/2013/11/there-is-no-thread.html

Any of John skeets blogs on TPL and Async-Await
https://codeblog.jonskeet.uk/2010/10/30/c-5-async-investigating-control-flow/

Microsoft documentation
https://msdn.microsoft.com/en-us/library/dd449174(v=vs.110).aspx

http://www.michaelridland.com/xamarin/taskcompletionsource-xamarin-beautiful-async/

THANK YOU!

David Allen

Demystifying Async-Await

.Net Track • 10:00am • Room #204

Microsoft Mobile App Partner of the Year 2017
Microsoft IoT Partner of the Year 2016
Xamarin Partner

Social

/in/David1Allen

Email

david.allen@bluemetal.com