# Intro

* This doc is split into sections – some of section refer to example and exercise code which can be found in the following repo: <https://github.com/claresudbery/Async-experiments>
  + !! All the code is in the following sub-solution: Mvc4Async
  + !! In order to run the code, you first have to load up and run the web API, which is in the following sub-solution: WebAPI, in WebAPIpgw.sln
* Not all sections have examples, and some have examples but not exercises.
* The following links contain some useful descriptions:
  + Here: <https://blog.stephencleary.com/2012/02/async-and-await.html>
  + And here: <https://msdn.microsoft.com/en-gb/library/mt674882.aspx>
    - Note that this link also contains links to several MSDN walk-throughs / tutorials

# What Async and Await Actually Do

## Notes

* The “async” keyword enables the “await” keyword in that method and changes how method results are handled. That’s all the async keyword does!
* The “await” keyword is always used on a task, aka an “awaitable”. If “await” sees that the awaitable has not completed, then it tells the awaitable to run the remainder of the method when the task completes. At this point it returns from the async method.
  + The method has not completed at this point – for instance any **finally** clauses will not be executed until the awaitable has returned or an exception is thrown.
* Each async method will return a task. Each returned task represents ongoing work.
* When the async method eventually completes its work, the task is marked as completed and the result, if any, is stored in the task. Alternatively it will hold any exceptions raised.
* Async methods returning Task or void do not have a return value. Async methods returning Task<T> must return a value of type T:
* public async Task<int> CalculateAnswer()
* {
* await Task.Delay(100); // (Probably should be longer...)
* // Return a type of "int", not "Task<int>"
* return 42;
* }
* You do not have to create any kind of Task object.
* You can also return the result of another await statement.
  + Example:
  + public async Task CalculateAnswer()
  + {
  + await Task.Delay(100);
  + }
* An async method that has a void return type can’t be awaited, and the caller of a void-returning method can't catch any exceptions that the method throws. This return type is used primarily to define event handlers, where a void return type is required.

# Flow of Execution

## Examples

* ExamplesController.Flow\_Of\_Execution\_Example in the Async-Experiments repo

## Exercises

* See “What is the difference between the following two methods?” below
* See ExercisesController.FlowOfControlEx1Part1 in the Async-Experiments repo
* Also exercise 2: ExercisesController.FlowOfControlEx2Part1 in the Async-Experiments repo
* Also exercise 3: ExercisesController.FlowOfControlEx3Part1 in the Async-Experiments repo

## Notes

* There is a good diagram with arrows, followed by an explanation, describing the control flow here: <https://msdn.microsoft.com/en-gb/library/mt674882.aspx>
* Exercise: What is the difference between the following two methods?
  + public async Task AsyncMethod()
  + {
  + await Task.Delay(100);
  + }
  + public Task NonAsyncMethod()
  + {
  + return Task.Delay(100);
  + }
  + (Answer: In the first example, the await statement means that the method will not return until the task has completed. In the second example, the lack of an await statement means that the caller can do other processing before it waits for the task to complete).
* Description of execution flow:
  + When you call an async method it will return a task.
  + If you do not immediately await that task, then you can add more lines of code between the original call and your await call on that task.
  + When you call the async method, the first thing that happens is that it is executed – just like calling any other method.
  + BUT as soon as it hits an await statement, it will return to its caller. At THAT point, your original method will continue, and your lines of code after the call are executed. At this point, whatever is being awaited will also be running simultaneously.
  + As soon as your calling code decides to await the task returned by the async method, no more lines of code *in that method* will be executed. Instead, it will return to *its* caller.
* For an example, see ExamplesController.FlowOfExecutionExample in the Async-Experiments repo.

# Alternatives to Await

## Examples

* Async-Experiments repo:
  + ExamplesController.Marked\_Async\_With\_Empty\_Task\_And\_NOT\_Calling\_Await
  + ExamplesController.Marked\_Async\_With\_Empty\_Task\_And\_Calling\_Await
  + Read the comments in the code and watch your debug output window to see what happens

## Notes

* It is the type that is awaitable, not the method returning the type. In other words, you can await the result of an async method that returns Task … because the method returns Task, not because it’s async. So you can also await the result of a non-async method that returns Task
  + From here: <https://blog.stephencleary.com/2012/02/async-and-await.html>
* This means you do not need to mark your method async. In fact you will get a compiler error if you mark a method async without using the await keyword.
  + But watch out: In the case of Task, you could call some nested async code that also returns Task, and think that’s what’s being returned by your own async method. In fact – unless you call await – that’s not what’s happening.
  + See Async-Experiments for the following examples:
    - ExamplesController.Marked\_Async\_With\_Empty\_Task\_And\_NOT\_Calling\_Await
    - ExamplesController.Marked\_Async\_With\_Empty\_Task\_And\_Calling\_Await

# Mixing Async and non-Async Code

## Examples

* Async-Experiments repo: ExamplesController.Call\_Async\_Code\_In\_NonAsync\_Context
* Async-Experiments repo: ExamplesController.Dont\_await\_long\_running\_code
  + Watch your debug output window to see what happens

## Notes

* You should use async / await all the way up the stack right back to your entry point
  + …so that IIS will use the synchronization context and will keep a record of the entry point to come back in again when your task completes
  + …otherwise you are not really being async - you are actually just blocking the current thread while you wait for the task to complete
  + Example: ExamplesController.Call\_Async\_Code\_In\_NonAsync\_Context
  + The fact that you have async / await at the top level / entry point, means that your code will not complete until everything being awaited has terminated. BUT it also means that control will flow back to the application / UI context / request context while things are being awaited – and you will not lock the current thread.
* If you have async code being called from ASP.Net (for instance), from an endpoint which will return to the user, watch out for this:
  + It is possible to have a method marked as async, which calls another async method, *but does not await its results*.
    - This means that any task returned by the async method is being ignored.
  + Example: Async-Experiments repo: ExamplesController.Dont\_await\_long\_running\_code
    - All your async calls are bound to the request context. When the request completes, the context is lost, and therefore any remaining work is also lost.

# Wait vs Await

## Examples & Exercises

* See Deadlocks and the section on parallel code

## Notes

* Wait is blocking and await is non-blocking
* You can think of “await” as standing for “asynchronous wait”.
* Wait is used on tasks, and will block the thread (closing it to any further requests – if there is a UI, it will hang at this point). It will stay in the current context (synchronization context) until the task is complete.
* Await causes the calling function to return immediately
  + It keeps the thread open to continue execution
  + It also keeps the thread open to handle any new requests
  + It passes a callback marker to IIS
  + IIS can then use the same thread to re-enter the code at the point it left
    - This will happen when the thing being awaited has completed.
* More here: <http://stackoverflow.com/questions/9519414/whats-the-difference-between-task-start-wait-and-async-await>

# Deadlocks

## Examples

* Async-Experiments repo: ExamplesController.CreateADeadlock

## Exercises

* Async-Experiments repo: ExercisesController.DeadlockExercise
* See the comment above the controller method which explains what to do.

## Notes

* Deadlock example: <https://blog.stephencleary.com/2012/07/dont-block-on-async-code.html>
  + An equivalent can be found in the Async-Experiments repo: ExamplesController.CreateADeadlock
  + Basically what happens in this example is this:
    - an http request is awaited in an async method
    - That async method returns a task to the caller
    - The caller calls Task.Result on the returned task
    - Task.Result hides a Wait.
      * Wait is not the same as await!
      * Wait blocks the thread!
    - Because the thread is blocked, when the original http request completes, we cannot return to the code after the await statement.
      * It will wait for the context to be free, so that it can return to the same context
      * But the context never becomes free, because it has been blocked by Task.Result
    - Hence: deadlock.
* To avoid deadlocks: Don’t mix blocking and non-blocking code
  + For instance don’t use Task.Result inside some async code

# Synchronization Contexts and ConfigureAwait

## Examples

* See the following examples in the Async-Experiments repo:
  + ExamplesController.Leave\_The\_Request\_Context\_In\_Some\_Places\_But\_Not\_All
  + ExamplesController.Leave\_The\_Request\_Context\_In\_All\_Places
  + ExamplesController.Fire\_and\_forget

## Exercises

* See the following exercise in the Async-Experiments repo:
  + ExercisesController.ConfigureAwait\_Exercise
  + The comment above the controller method explains what to do.

## Notes

* + The awaitable will capture the current “synchronization context” and later apply it to the remainder of the async method. What exactly is that “synchronization context”?

1. If you’re on a UI thread, then it’s a UI context.
2. If you’re responding to an ASP.NET request, then it’s an ASP.NET request context.
3. Otherwise, it’s usually a thread pool context.
   * So this means that when you re-enter your method after the await,
     + For UI threads you will still have access to UI elements, eg resultTextBox.Text in WinForms
     + For ASP.Net threads you will still have access to the current request, so we can do things like Response.Write("File downloaded!");
   * An ASP.NET request context is not tied to a specific thread (like the UI context is), but it does only allow one thread in at a time.
   * Most of the time, you don’t need to sync back to the “main” synchronization context. In this case, you want to tell the awaiter to not capture the current context by calling ConfigureAwait and passing false, e.g.:

* var fileContent = await DownloadFile(fileName).ConfigureAwait(false);
  + A good rule of thumb is to use ConfigureAwait(false) unless you know you do need the context.
* When you use ConfigureAwait, you are still waiting for the code to complete before execution can continue, but that code will not hog the UI / request context, which means that other requests can be serviced simultaneously without having to share resources with your non-UI-dependent/non-request-dependent code.
* If you use ConfigureAwait(false) in a nested method, any code after that will be run in a separate context, but as soon as that method returns a task to a caller, any code after *that* await statement (if it doesn’t use ConfigureAwait) will go back to the default context.
* See the following examples in the Async-Experiments repo:
  1. ExamplesController.Leave\_The\_Request\_Context\_In\_Some\_Places\_But\_Not\_All
  2. ExamplesController.Leave\_The\_Request\_Context\_In\_All\_Places
* If you want your processing to continue after your request has completed, you can await an async call using ConfigureAwait, and then wrap this in an async call which is NOT awaited. Therefore your request will terminate, but your long-running code will continue in a separate synchronization context.
  + - * See ExamplesController.Fire\_and\_forget, in my Async-Experiments repo.
      * However, this is “fire and forget” – which is not recommended.
* It is not good practice to use ConfigureAwait to avoid deadlocks.
  + It would mean that ALL awaits – including third-party ones – would have to use ConfigureAwait(false)
  + Instead, you should just not mix blocking (Task.Result) and non-blocking (async) code.

# Async Progress Tracking

## Examples

* Async-Experiments repo: ExamplesController.ReportProgress

## Exercises

* See ExercisesController.Progress\_Exercise In Async-Experiments repo

## Notes

* Reporting progress:
  + For long-running async tasks, you can use callbacks to report progress.
  + To do this:
  + First, you need to implement a void method which takes a parameter of type T
    - (see ExamplesController.ReportProgress in the example code)
  + Then, create an instance of the Progress class, passing your void method into its constructor.
    - In the example, type T is an object of type ProgressIndicator:
    - var progressIndicator = new Progress<ProgressIndicator>(ReportProgress);
  + Now, your async method needs to take a parameter of type IProgress, with type T, ideally initialised to null:
    - public async Task<int> AsyncMethodWithProgress(IProgress<ProgressIndicator> progress = null)
  + Inside your async method, you can now call the progress method you created earlier:
    - if (progress != null)
    - {
    - progress.Report(new ProgressIndicator { Count = numSeconds, Total = totalSeconds });
    - }
  + Note that in the example code, it uses SignalR to gradually update a progress bar until the request is completed.

# Async Cancellation

## Examples

* Async-Experiments repo: ExamplesController.ThisActionCanBeCancelled
* Async-Experiments repo: ExamplesController.Cancel

## Exercises

* See ExercisesController.Cancel\_Exercise In Async-Experiments repo

## Notes

* Cancelling a long-running action:
  + First create a cancellation token:
    - static CancellationTokenSource \_cancellationToken = new CancellationTokenSource();
  + Now add a cancellation token to the parameters of your async method:
    - public async Task<int> AsyncMethodWithCancellation (CancellationToken cancellationToken,
  + Now add code inside your async method which will throw an exception if the cancellation token is marked as cancelled:
    - cancellationToken.ThrowIfCancellationRequested();
  + Now make your cancellation token available to your users, typically via a cancellation link on your front end.
  + In our example, we have a separate “Cancel” endpoint on the controller. When the user clicks Cancel, the following code is executed:
    - \_cancellationToken.Cancel();
    - !!NB Note that our member var \_cancellationToken is a static var, so that it is always available.

# Threads and async / parallel

## Examples

* See the following examples in the Async-Experiments repo:
  + ExamplesController.Parallel\_Synchronous
  + ExamplesController.Parallel\_Asynchronous

## Notes

* The TPL (Task Parallel Library) actually does a lot of stuff to optimise behind the scenes which means that it’s hard to predict what happens with threads.
* But basically new threads will only be started if you use blocking code and parallel tasks. With async, rather than doing things in separate threads, it uses the same thread.
  + Great analogy by Eric Lippert to explain this using toasters here: <http://stackoverflow.com/questions/9519414/whats-the-difference-between-task-start-wait-and-async-await>
* This means that async is NOT parallel processing.
* “The async and await keywords don't cause additional threads to be created. Async methods don't require multithreading because an async method doesn't run on its own thread. The method runs on the current synchronization context and uses time on the thread only when the method is active.”
  + From here: <https://msdn.microsoft.com/en-gb/library/mt674882.aspx>
* You can have parallel AND async though. Do this by using Task.WhenAll
  + Task.WhenAll is the asynchronous equivalent for Task.WaitAll.
  + Task.WaitAll blocks the calling thread until all parallel tasks are completed.
  + With Task.WhenAll, the calling thread becomes free again while the tasks are completing.
    - When the last task has completed, any code after Task.WhenAll will get executed.

# Task.Delay vs Thread.Sleep

* Task.Delay returns the thread immediately back to the thread-pool, whereas Thread.Sleep blocks it. Task.Delay is essentially the asynchronous version of Thread.Sleep

# Task.Yield

* Basically if you call await Task.Yield(); you are saying that you want your method to return to its caller at this point, and any subsequent code will be scheduled within the context, which means that the context can asynchronously prioritise other code being completed within the same context.
* However, the context will prioritise this kind of work over UI input, so don’t use Task.Yield to make your app more responsive to user input!
* More here: <https://msdn.microsoft.com/en-us/library/system.threading.tasks.task.yield(v=vs.110).aspx>

# Sample Code

* Basic Windows forms examples at the bottom of this page: <https://msdn.microsoft.com/en-gb/library/mt674882.aspx>
* Async and ASP.Net example here: <https://github.com/RickAndMSFT/Async-ASP.NET>
  + Uses MVC4, WebApi and Web Forms (three different solutions)