**Asynchronous Programming**

Async programming is a key technique that makes it straightforward to handle blocking I/O and concurrent operations on multiple cores.

**Why Write Async Code?**

Modern apps make extensive use of file and networking I/O. I/O APIs traditionally block by default, resulting in poor user experiences and hardware utilization unless you want to learn and use challenging patterns.

**Advantages of Async Code**

* Handles more server requests by yielding threads to handle more requests while waiting for I/O requests to return.
* Enables UIs to be more responsive by yielding threads to UI interaction while waiting for I/O requests and by transitioning long-running work to other CPU cores.
* Many of the newer .NET APIs are asynchronous.
* It's easy to write async code in .NET!

**Async in depth**

Writing I/O- and CPU-bound asynchronous code is straightforward using the .NET Task-based async model. The model is exposed by the Task and Task<T> types and the async and await keywords in C#

**Task and Task<T>**

Task and Task<T> works on Promise Model of Concurrency – Promise that your task will be executed later point of time.

* Task represents a single operation which does not return a value.
* Task<T> represents a single operation which returns a value of type T.

By default, tasks execute on the current thread and delegate work to the Operating System, as appropriate. Optionally, tasks can be explicitly requested to run on a separate thread via the Task.Run API.

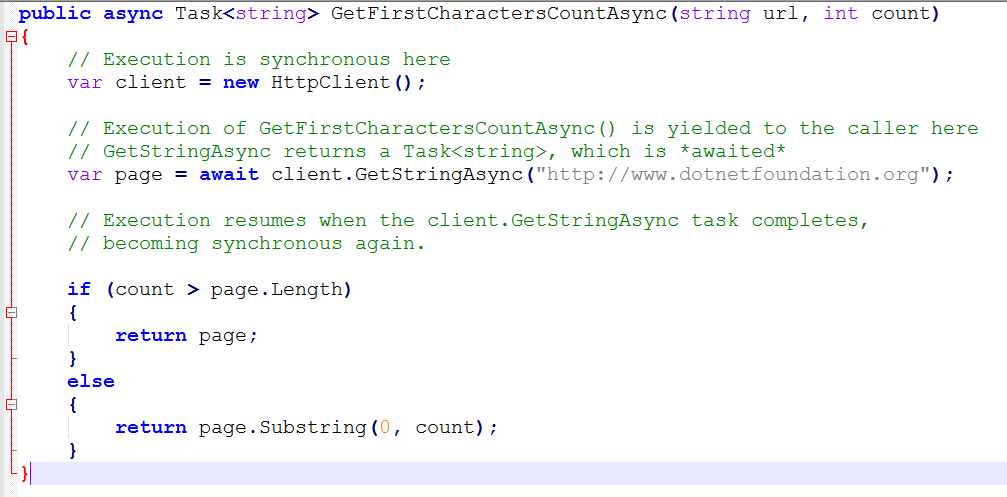
Language integration, with the await keyword, provides a higher-level abstraction for using tasks.

Using await allows your application or service to perform useful work while a task is running by yielding control to its caller until the task is done. Your code does not need to rely on callbacks or events to continue execution after the task has been completed

**Task – Input Output Bound Operation**

For the Input Output Operation, The Task execution is handed over to Operating System level and CPU is 100% free to take over another activities available.

***Input Output Operation Example***



There is no thread is dedicated to running the task. Although work is executed in some context (that is, the OS does have to pass data to a device driver and respond to an interrupt), there is no thread dedicated to waiting for data from the request to come back. This allows the system to handle a much larger volume of work rather than waiting for some I/O call to finish.

**Example in Server Scenario**

Consider two servers: one that runs async code, and one that does not. For the purpose of this example, each server only has 5 threads available to service requests.

Assume both servers receive 6 concurrent requests. Each request performs an I/O operation. The server without async code has to queue up the 6th request until one of the 5 threads have finished the I/O-bound work and written a response. At the point that the 20th request comes in, the server might start to slow down, because the queue is getting too long.

The server with async code running on it still queues up the 6th request, but because it uses async and await, each of its threads are freed up when the I/O-bound work starts, rather than when it finishes. By the time the 20th request comes in, the queue for incoming requests will be far smaller (if it has anything in it at all), and the server won't slow down.

**Client Scenario**

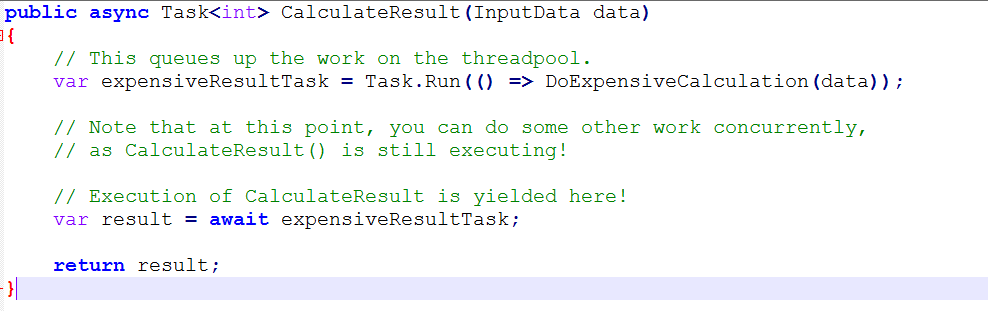
The biggest gain for using async and await for a client app is an increase in responsiveness. Although you can make an app responsive by spawning threads manually, the act of spawning a thread is an expensive operation relative to just using async and await. Especially for something like a mobile game, impacting the UI thread as little as possible where I/O is concerned is crucial.

More importantly, because I/O-bound work spends virtually no time on the CPU, dedicating an entire CPU thread to perform barely any useful work would be a poor use of resources.

**Task – CPU Operation**

The use of async and await provides you with a clean way to interact with a background thread and keep the caller of the async method responsive.

***CPU Operation Example***



CalculateResult() executes on the thread it was called on. When it calls Task.Run, it queues the expensive CPU-bound operation, DoExpensiveCalculation(), on the thread pool and receives a Task<int>handle. DoExpensiveCalculation() is eventually run concurrently on the next available thread, likely on another CPU core. It's possible to do concurrent work while DoExpensiveCalculation() is busy on another thread, because the thread which called CalculateResult() is still executing.