Asynchronous Programming

# Asynchronous Programming

Asynchronous programming refers to the occurrence of events independent of the main program flow and ways to deal with such events. These may be

1. "Outside" events such as the arrival of signals.
2. Actions instigated by a program that take place concurrently with program execution without the program getting blocked to wait for results.

example: Asynchronous I/O, where programs issue commands to storage or network devices that service these requests while the processor continues executing the program. Parallelism is implemented this way.

## ****Examples of Asynchronous Program:****

### Multithreading

Applying Multi-Threading to Single Threaded Applications [STAs] where the order of operations is non deterministic.

### Ajax

**Ajax** is a set of client-side web technologies used on to create asynchronous I/O web applications.

### Task Based Asynchronous Programming (TAP)

A method to run a procedure concurrently, a lightweight alternative to Threads. (async...await using Task<TResult>).

### Asynchronous Method Dispatch (AMD)

**A** data communication method used when there is a need for the server side to handle a large number of long lasting client requests. Using synchronous method dispatch (SMD), this scenario may turn the server into an unavailable busy state resulting in a connection failure response caused by a network connection request timeout. The servicing of a client request is immediately dispatched to an available thread from a pool of threads and the client is put in a blocking state. Upon the completion of the task, the server is notified by a callback. The server unblocks the client and transmits the response back to the client. In case of thread starvation, clients are blocked waiting for threads to become available.

# Asynchronous Patterns

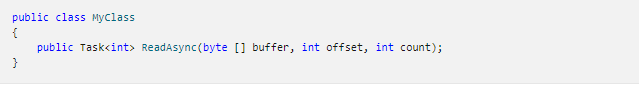
Consider a Read method that reads a specified amount of data into a provided buffer starting at a specified offset:

## 

## Task Based Asynchronous Patterns (TAP)

Uses a single method to represent the initiation and completion of an asynchronous operation. TAP was introduced in .NET Framework 4. It's the recommended approach to asynchronous programming in .NET. The async and await keywords in C# and the Async and Await operators in Visual Basic add language support for TAP.

The TAP counterpart of this method would expose the following single **ReadAsync()**:



## Asynchronous Programming Model (APM)

Asynchronous Programming Model (APM) pattern (also called the IAsyncResult pattern), which is the legacy model that uses the IAsyncResult interface to provide asynchronous behavior. In this pattern, synchronous operations require Begin and End methods (for example, BeginWrite and EndWrite to implement an asynchronous write operation). This pattern is no longer recommended for new development.

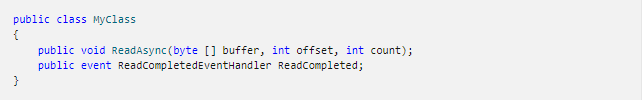
The APM counterpart would expose the BeginRead and EndRead methods:

# 

## Event-based Asynchronous Pattern (EAP)

This is the Event-based legacy model for providing asynchronous behavior. It requires a method that has the Async suffix and one or more events, event handler delegate types, and EventArg-derived types. EAP was introduced in .NET Framework 2.0. It's no longer recommended for new development. For more information, see Event-based Asynchronous Pattern (EAP).

The EAP counterpart would expose the following set of types and members:



# ThreadPool

# Task Based Asynchronous Patterns (TAP)

The async and await keywords, along with the Task<TResult> provides the backbone for TAP in C#. The same is implemented with Async and Await operators in Visual Basic.

1. The Task-related classes belong to **System.Threading.Tasks** namespace and the **System.Runtime.dll.**
2. The Task<TResult> class represents an Asynchronous operation that can return a value.
3. Work performed by a Task<TResult> object typically executes asynchronously in a ThreadPool rather than synchronously on the main application thread.

## Task<TResult> instance creation procedure.

There are basically two ways through which you can instantiate a task – 1) Run() and 2) StartNew().

### ****Task.Run()****

Simple way to start a TASK by using default values and without acquiring additional parameters.****

### ****TaskFactory**.**StartNew**()**

****

## Task<TResult> Properties

|  |  |
| --- | --- |
| Property | Description |
| status |  |
| IsCompleted |  |
| IsCancelled |  |
| IsFaulted |  |

## The “async … await” model

1. 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 async keyword only enables the await keyword (and manages the method results).
2. The beginning of an async method is executed just like any other SYNC method. That is, it runs synchronously until it hits an “await” (or throws an exception).
3. The “await” keyword is where things can get asynchronous. Await is like a unary operator: it takes a single argument, an awaitable (an “awaitable” is an asynchronous operation). Await examines that awaitable to see if it has already completed; if the awaitable has already completed, then the method just continues running (synchronously, just like a regular method).

## Cancelling a Task<T>

1. Create and start a cancelable task.
2. Pass a cancellation token to your user delegate and optionally to the task instance.
3. Notice and respond to the cancellation request in your user delegate.
4. Optionally notice on the calling thread that the task was canceled.

There are 2 important things in this –

1. **CancellationToken** – a structure used by listeners to monitor token current state.
2. **CancellationTokenSource** – an object responsible for creating a cancellation token and sending a cancellation request to all copies of that token.

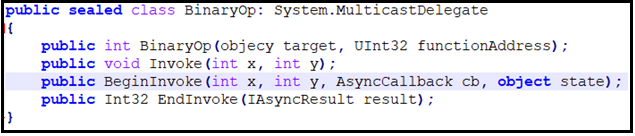
# Asynchronous Callbacks Programming Model Callbacks (IAsyncResult)

This Asynchronous Program Model is based on the concept of **Asynchronous Callbacks** with the support of the **IAsyncResult**. A mechanism to invoke a delegate on a separate thread.

Consider the delegate shown below:



When this delegate is compiled, the internally he compiler generates a code like this -



## Approach #1

Poll the delegate to check for its completion through the IAsyncResult.IsCompleted property. Once it is set to true, carry on with the subsequent things.

## Approach #2

Don't poll the delegate to know the completion status and waste some time towards polling (through the IAsyncResult.IsCompleted = true);

A better approach is to have the secondary thread inform the primary thread when the task is finished. as part of this, pass an instance of System.AsyncCallback delegate to BeginInvoke().

## ****Invoke()****

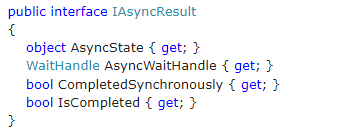
The Invoke method is used to invoke the methods maintained by a delegate object in a Synchronous manner. The **Invoke()** method does not need to be directly called in code, but can be triggered indirectly under the hood when applying "normal" method invocation syntax.

## ****BeginInvoke()****

## ****EndInvoke()****

## IAsyncResult()

The IAsyncResult compatible object returned from BeginInvoke() is basically a coupling mechanism that allows the calling thread to obtain the asynchronous method invocation at a later time via EndInvoke().



# Timer Class

The C# Timer is used to implement a timer in C#. The Timer class in C# represents a Timer control that executes a code block at a specified interval of time repeatedly. For example, backing up a folder every 10 minutes, or writing to a log file every second. The method that needs to be executed is placed inside the event of the timer.

|  |  |
| --- | --- |
| Property | Description |
| Tick | This event occurs when the Interval has elapsed. |
| Start | Starts raising the Tick event by setting Enabled to true. |
| Stop | Stops raising the Tick event by setting Enabled to false. |
| Close | Releases the resources used by the Timer. |
| AutoReset | Indicates whether the Timer raises the Tick event each time the specified Interval has elapsed or whether the Tick event is raised only once after the first interval has elapsed. |
| Interval | Indicates the interval on which to raise the Tick event. |
| Enabled | Indicates whether the Timer raises the Tick event. |

# References

1. Asynchronous Patterns -

<https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/>

1. Delegates and Async Programming

<https://www.c-sharpcorner.com/uploadfile/vendettamit/delegates-and-async-programming/>