**Introduction to Concurrency**

این قسمت حول محور بهره وری یا effiency بحث می کند. به طور مثال یک برنامه کاری را با 20 ثانیه زمان انجام می دهد و برنامه ای دیگر با 40 ثانیه بنابراین effiency برنامه دوم بهتر از برنامه اول است.

##### **What is Concurrency?**

به معنای انجام دادن چندین کار در یک زمان است. یعنی اگر چندین تسک داریم به جای انجام sequentialy می توانیم به صورت هم زمان آن ها را انجام دهیم و زمان اجرای برنامه را کاهش دهیم.

مفهوم انجام دادن مجموعه ای از task ها و تقسیم آن ها به part های مختلف به معنای همزمانی یا parallelism است. به طور مثال یک رستوران که چندین آشپز دارد که سفارش ها را آماده می کند.

##### **How we can Achieve Parallelism in Programming?**

در برنامه نویسی رسیدن به این هدف با استفاده از thread ها مسیر می باشد.

Thread : یک ترد دنباله ای از دستورات است که می تواند مستقل از کد های دیگر اجرا شود. از آن جایی که ترد ها مستقل هستند در یک پراسس بنابراین می توانیم چندین ترد را به صورت هم زمان اجرا کنیم. زمانی که چنیدن ترد را به صورت هم زمان اجرا می کنیم می توانیم بگوییم نام آن multithreading است.

Parallelism استفاده می کند از چندین thread برای انجام دادن چندین task یا کار به صورت همزمان. بنابراین چند نخی نیز گونه ای از parallelism است.

ما نیاز داریم که thread ها را پس از انجام کار release کنیم و انجام این کار با استفاده از asynchronous programing مسیر است.

Asynchrons به ما این اجازه را می دهد که تا از thread ها به صورت موثر یا efficiently استفاده کنیم و از block شدن فیر ضروری thread ها جلوگیری کنیم.

مثلا پیتزای را سفارش می دهد که 30 دقیقه زمان می برد تا آماده شود آیا 30 دقیقه بدون انجام کاری منتظر می مانید تا آماده شود یا کار های که دارید را انجام می دهید ؟

با استفاده از asynch می توانید درخواست های http بیشتری که به صرور ارسال می شود را با استفاده از multithreading پاسخ دهید و از block شدن نخ ها جلوگیری کنید.

Benefit اصلی parallel این است که در زمان صرفه جویی می شود از طریق استفاده ی حداکثری از resource های سیستم.

در سیستم های که داده های زیادی را پردازش می کنند این روش بسار پر کاربرد است.

نکته : مواقعی است که parallelism می تواند باعث کاهش سرعت کار شود.

### **Introduction to Asynchronous Programming**

برنامه نویسی ناهم زمان به ما این کمک را می کند که نخ های یک پردازش را به شکلی با بهره وری بیشتری مدیریت کنیم.

ایده این است که از block شدن thread در زمانی که منتظر پاسخ هستیم پرهیز کنیم.

در سی شارپ از async , await استفاده می کنیم. Async برای نشانه گذاری یک متد به عنوان asynchronous و wait برای انتظار جهت انجام عملیات بدون block شدن. متد async باید task را برگرداند.

**نکته معمولا برای برقراری ارتباط با سیستم های خارجی مناسب است. I/O Bound**

#### **CPU vs I/O Bound Operations:**

Io مشابه عملیات های مانند وب سرویس ها و دیتابیس و فایل سیستم ها است با استفاده از Async می توانیم سطح مقیاس پذیری سیستم خود را افزایش دهیم. زمانی که موجودیت خارجی را فراخوانی می کنیم لازم است که برای پاسخ آن صبر کنیم در این شرایط آزاد کردن تردی که عملیات را شروع کرده است می تواند برای انجام Task های دیگر موثر باشد.

Cpu

عملیات های که به قدرت پردازشی سیستم خود متکی است

**Sequential programming:** Sequential programming is the one in which the instructions are done one at a time. That is where there is no concurrency of any kind. One of the advantages of this programming model is that it is relatively easy to understand since it consists of following a series of steps in an orderly manner. The problem with this programming model is that sometimes it can be slow.

**Concurrency:** Concurrency means doing several things at the same time. This is the opposite of sequential programming. The term concurrency encompasses everything related to in one way or another doing several things at the same time. There are different forms of concurrency. We have seen a fundamental concept of threads. We remember that a thread is a sequence of instructions that can be executed independently of our code.

**Multithreading:** Multithreading is the ability to use multiple threads. It is important to clarify that multithreaded does not imply parallelism, since we can have a computer with a processor that is not multicore and I still can use multithreading. This is because an operating system can provide several threads and execute them sequentially without using parallelism

**Parallelism:** It is running several threads simultaneously. This requires a multicore processor. Since parallelism uses multiple threads, parallelism uses multithreading. However, as we said, we can have multithreading without parallelism. In this case, typically what we have is called multitasking.

**Multitasking:** With multitasking, we can have several tasks running in such a way that we execute their different threads sequentially, typically with some type of Task Execution System. This is handled at the operating system level. For example, if we have a program A with threads one and two and a program B with threads three and four, and we try to execute both programs at the same time, it could be that the system executes the threads in the order one three two, and four.

#### **Determinism vs Non-Determinism**

**Determinism به این معنی است که بر اساس ورودی می توانیم خروجی متد را حدس بزنیم متد جمع دو عدد میگیرد مثال 2 و 5 و خروجی که انتظار می رود عدد 7 است**

**Non determinism**

**متدی که خروجی آن غیر قابل پیش بینی باشد مثال متدی که عدد رندومی را برمیگرداند.**

**نکته مهم**

**برنامه نویسی parallelism به صورت non-determinism است به این معنی که شما اطمینان دارید که تمامی خطوط کد اجرا می شوند اما نمی توانید ترتیب اجرای کد ها را پیش بینی کنید. و اگر ترتیب مهم است parallelism مناسب نیست.**

**نکات مهم**

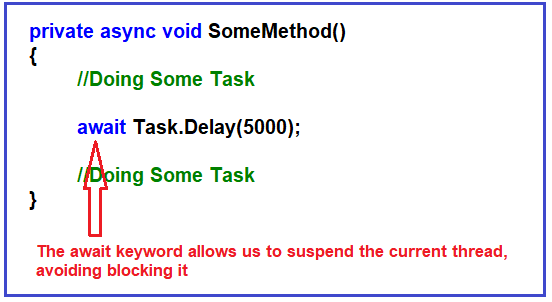
1. We saw that concurrency refers to, in one way or another, doing several things at the same time. The concept of concurrency encompasses parallel programming and asynchronous programming.
2. Parallel programming refers to the use of multiple threads simultaneously to solve a set of tasks. For this, we need processors with adequate abilities to perform several tasks at the same time. In general, we use parallel programming to gain speed.
3. Asynchronous programming refers to the efficient use of threads where we do not block a thread unnecessarily. But while we wait for the result of an operation, the thread gets to perform other tasks in the meantime. This increases vertical scalability and allows us to prevent the user interface from freezing during long tasks.
4. CPU-bound operations are those that depend entirely on the speed of our processors.
5. IO-bound operations are those that depend on communication with entities external to our application.
6. Deterministic refers to the fact that we can’t predict the result of something based on the initial conditions. For example, we can predict the result of a method from its input values. With parallel programming, we will not always be able to predict 100 percent the result of something, especially when we refer to the order of operations of a set of tasks, since we do not control the order of execution of the different threads of the application.

# Async and Await in C#

در افزایش بهره وری از منابع سیستم کنک می کند.

Task نشان دهنده اجرای یک متد است که در آینده به پایان می رسد. یک promise یا وعده اجرای متد. که امکان گزارش گیری فرایند اجرا را نیز دارد و می توان در طول اجرا آن را کنسل کرد.

نکته : await یک thrad را suspend می کند نه block



Notice : زمانی که از کلمه کلیدی await استفاده می کنیم اتفاقی که می افتد این است که Thread جاری را آزاد می کنیم از این که منتظر پایان کار task باشد و از بلاک شدن ترد جاری جلوگیری می کنیم که سبب می شود آن thread در سایر Task ها استفاده شود. بعد از این که task تکمیل شده Thread مجددا فراخوانی می شود به محل و ادامه کار را انجام می دهد.

نکته : اگر یک متد async داشته باشیم و آن را بدون await فراخوانی کنیم باعث می شود که به متد فراخواننده منتظر پاسخ نماند و ادامه عملیات را انجام دهد.

مثال پس از اجرای متد ادامه main اجرا می شود به دلیل این که ترد جاری آزاد است.

static void Main(string[] args)

{

Console.WriteLine("Main Method Started......");

SomeMethod();

Console.WriteLine("Main Method End");

Console.ReadKey();

}

public async static void SomeMethod()

{

Console.WriteLine("Some Method Started......");

//Thread.Sleep(TimeSpan.FromSeconds(10));

await Task.Delay(TimeSpan.FromSeconds(10));

Console.WriteLine("\n");

Console.WriteLine("Some Method End");

}

# How to Return a Value from Task in C#

**public** **static** **void** Main**(string[]** args**)**

**{**

Console.WriteLine**(**"Main Method Started......"**)**;

Console.WriteLine**(**"Enter the Name: "**)**;

**string** Name = Console.ReadLine**()**;

SomeMethod**(**Name**)**;

Console.WriteLine**(**"Main Method End"**)**;

Console.ReadKey**()**;

**}**

**public** **async** **static** **void** SomeMethod**(string** Name**)**

**{**

Console.WriteLine**(**"Some Method Started......"**)**;

var GreetingSMessage = **await** Greetings**(**Name**)**;

Console.WriteLine**(**$"\n{GreetingSMessage}"**)**;

Console.WriteLine**(**"Some Method End"**)**;

**}**

**public** **static** **async** Task**<string>** Greetings**(string** Name**)**

**{**

**string** message = **string**.Empty;

**using** **(**var client = new HttpClient**())**

**{**

client.BaseAddress = new Uri**(**"http://localhost:58937/"**)**;

HttpResponseMessage response = **await** client.GetAsync**(**$"api/greetings/{Name}"**)**;

message = **await** response.Content.ReadAsStringAsync**()**;

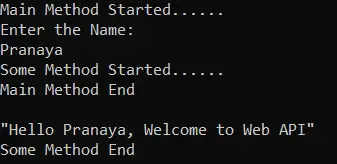
**}**

**return** message;

**}**

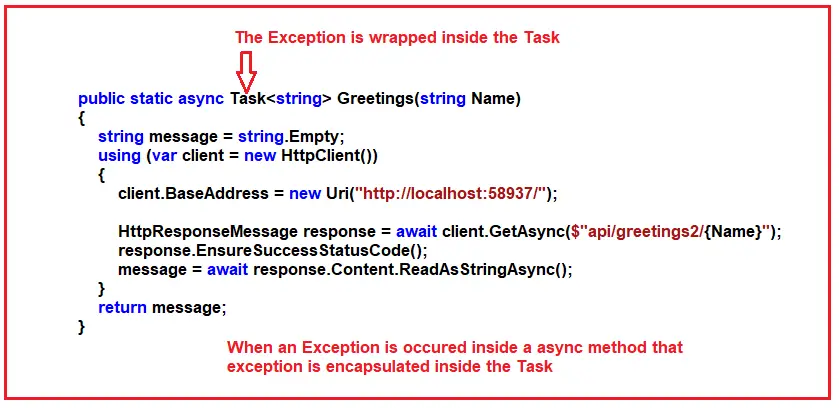
**}**

نکته مهم این است که در متد main چون await نکردیم پس منتظر پایان نمی ماند و رد می کند تا پایان متد اما در لایه ها داخلی چون await داریم تا دریافت پاسخ صبر می کند.



**نکته مهم : استفاده از کلمه کلیدی await به معنای suspend کردن current thread است و آن را آزاد می کنیم و آن thread می تواند در سایر قسمت های اپلیکیشن استفاده شود.**

**نکته مهم : اگر برای قطعه کدی که احتمال exception دارد await نگذاریم exception رخ داده در بلوک try , catch نمی رود. به دلیل این که زمانی که در یک متد Asynchronus یک Exception رخ می دهد آن در داخل task کپسوله می شود و بنا برا این به Exception اصلا دسترسی نخواهید داشت**

****

**در این حالت throw نمی شود.**

# How to Execute Multiple Tasks in C#

زمانی که بخواهیم چندین task را به صورت هم زمان اجرا کنیم از متد WhenAll مربوط به کلاس Task استفاده می کنیم که پس از اتمام اجرای تمامی task ها ادامه فرایند ها اجرا می شود.

مثال

**public** **static** **async** **void** ProcessCreditCards**(**List**<**CreditCard**>** creditCards**)**

**{**

var stopwatch = new Stopwatch**()**;

stopwatch.Start**()**;

var tasks = new List**<**Task**<string>>()**;

**foreach** **(**var creditCard in creditCards**)**

**{**

var response = ProcessCard**(**creditCard**)**;

tasks.Add**(**response**)**;

**}**

**await** Task.WhenAll**(**tasks**)**;

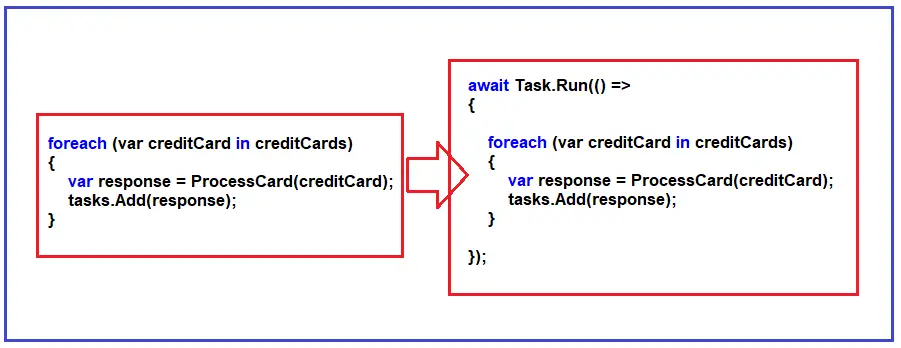
stopwatch.Stop**()**;

Console.WriteLine**(**$"Processing of {creditCards.Count} Credit Cards Done in {stopwatch.ElapsedMilliseconds / 1000.0} Seconds"**)**;

**}**

نکته : فراخوانی Task.WhenAll بیان گر این است که ما لیستی از task ها را داریم که می خواهیم صبر کنیم تا همه ی آن ها به اتمام برسند سپس ادامه فرآیند اجرای متد انجام شود و تمامی task ها به صورت simultaneously یا هم زمان اجرا شوند در این مثال 10 task داریم که همگی به صورت هم زمان اجرا می شوند.

نکته مهم : تفاوت 2 مثال برای اجرای 10 تسک یکی به صورت هم زمان و دیگری به صورت تک به تک خروجی یکی 1 ثانیه و دیگری 10 ثانیه می شود.



# How to Limit Number of Concurrent Tasks in C#

با استفاده از کلاس semaphoreSlim می توانیم تعداد task های که به صورت هم زمان در حال اجرا هستند را محدود کنیم.

شرایطی را فرض کنید که یک webApplication دارید که درخواست های http فراوانی به آن ارسال می شود مثال 10000 درخواست در ثانیه دریافت می کند در این حالت ممکن است سرور از پردازش task های ناتوان باشد یا تعداد زیاد باعث بروز block در سیستم یا down شدن آن شود

راه حل

می توانیم task ها را گروه بندی کنیم و دسته دسته آن ها ارسال کنیم یعنی تعداد task های که قرار است توسط whenAll اجرا شوند را محدود می کنیم.

کاری که قرار است انجام دهیم این است که به کلاس semaphore اعلام می کنیم که فقط مجاز است 3 task را در هر بار به صورت هم زمان اجرا کند.

**public** **static** **async** **void** ProcessCreditCards**(**List**<**CreditCard**>** creditCards**)**

**{**

var stopwatch = new Stopwatch**()**;

stopwatch.Start**()**;

var tasks = new List**<**Task**<string>>()**;

//Need to use async lambda expression

tasks = creditCards.Select**(async** card =**>**

**{**

//This will tell if we have more than 4000 tasks are running,

//we are going to wait and '

//we're going to wait until the semaphore gets released.

**await** semaphoreSlim.WaitAsync**()**;

//Need to use await operator here as we are using asynchronous WaitAsync

**try**

**{**

**return** **await** ProcessCard**(**card**)**;

**}**

**finally**

**{**

//Release the semaphore

semaphoreSlim.Release**()**;

**}**

**})**.ToList**()**;

//It will execute a maximum of 3 tasks at a time

**await** Task.WhenAll**(**tasks**)**;

stopwatch.Stop**()**;

Console.WriteLine**(**$"Processing of {creditCards.Count} Credit Cards Done in {stopwatch.ElapsedMilliseconds/1000.0} Seconds"**)**;

**}**

##### **Handle Response when Executing Multiple Tasks using Tasks.WhenAll**

//Need to use async lambda expression

tasks = creditCards.Select**(async** card =**>**

**{**

**await** semaphoreSlim.WaitAsync**()**;

**try**

**{**

**return** **await** ProcessCard**(**card**)**;

**}**

**finally**

**{**

semaphoreSlim.Release**()**;

**}**

**})**.ToList**()**;

**string[]** Responses = **await** Task.WhenAll**(**tasks**)**;

//var Responses = await Task.WhenAll(tasks);

**foreach** **(**var response in Responses**)**

**{**

Console.WriteLine**(**response**)**;

**}**

مثال دوم برای دریافت result

//Looping through the string array

**foreach** **(**var response in Responses**)**

**{**

//Here, the string is a JSON string

//Converting the JSON String to .NET Object (CreditCardResponse) using

//JsonConvert class DeserializeObject

CreditCardResponse creditCardResponse = JsonConvert.DeserializeObject**<**CreditCardResponse**>(**response**)**;

//Adding the .NET Object into the resposne collection

creditCardResponses.Add**(**creditCardResponse**)**;

**}**

# ****How to Cancel a Long-Running Task Using Cancellation Token****

زمانی که در حال انجام یک عملیات طولانی هستیم می توانیم این دسترسی را در اختیار کاربر قرار دهیم که در صورت نیاز task را کنسل کند. با استفاده از cancellationToken می توانی task در حال اجرا را متوقف کنیم.

کلاس CancellationTokenSource از رابط IDisposable پیروی کرده و آن را پیاده سازی کرده است.

##### **Constructors of CancellationTokenSource Class in C#:**

The CancellationTokenSource class provides the following three constructors to create an instance of the CancellationTokenSource class.

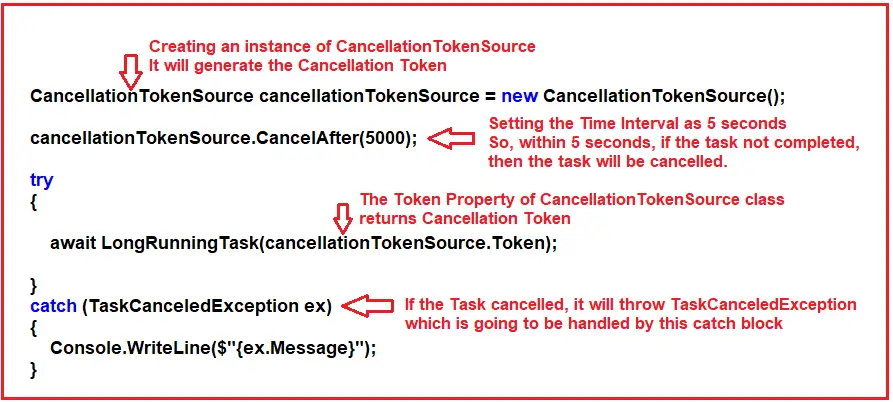
1. **CancellationTokenSource():** It Initializes a new instance of the CancellationTokenSource class.
2. **CancellationTokenSource(TimeSpan delay):** It initializes a new instance of the CancellationTokenSource class that will be canceled after the specified time span. Here, the parameter delay specifies the time interval to wait before canceling this CancellationTokenSource. It will throw ArgumentOutOfRangeException if delay.System.TimeSpan.TotalMilliseconds is less than -1 or greater than System.Int32.MaxValue.
3. **CancellationTokenSource(int millisecondsDelay):** It initializes a new instance of the CancellationTokenSource class that will be canceled after the specified delay in milliseconds. Here, the parameter millisecondsDelay specifies the time interval in milliseconds to wait before canceling this System.Threading.CancellationTokenSource. It will throw ArgumentOutOfRangeException if millisecondsDelay is less than -1.

##### **Properties of CancellationTokenSource class in C#:**

1. **public bool IsCancellationRequested { get; }:** It gets whether cancellation has been requested for this CancellationTokenSource. It returns true if the cancellation has been requested for this CancellationTokenSource; otherwise, false.
2. **public CancellationToken Token { get; }:** It gets the CancellationToken associated with the CancellationTokenSource. It returns the CancellationToken associated with this CancellationTokenSource. It will throw ObjectDisposedException if the token source has been disposed.

##### **Methods of CancellationTokenSource class in C#:**

1. **Cancel():** It communicates a request for cancellation.
2. **Cancel(bool throwOnFirstException):** It communicates a request for cancellation and specifies whether remaining callbacks and cancellable operations should be processed if an exception occurs. Here, the parameter throwOnFirstException specifies true if exceptions should immediately propagate; otherwise, false.
3. **CancelAfter(TimeSpan delay):** It schedules a cancel operation on the CancellationTokenSource after the specified time span. Here, the parameter delay specifies the time span to wait before canceling this CancellationTokenSource.
4. **CancelAfter(int millisecondsDelay):** It schedules a cancel operation on this CancellationTokenSource after the specified number of milliseconds. Here, the parameter millisecondsDelay specifies the time span to wait before canceling this System.Threading.CancellationTokenSource.
5. **Dispose():** It releases all resources used by the current instance of the CancellationTokenSource class.



**static** **void** Main**(string[]** args**)**

**{**

SomeMethod**()**;

Console.ReadKey**()**;

**}**

**private** **static** **async** **void** SomeMethod**()**

**{**

**int** count = 10;

Console.WriteLine**(**"SomeMethod Method Started"**)**;

CancellationTokenSource cancellationTokenSource = new CancellationTokenSource**()**;

cancellationTokenSource.CancelAfter**(**5000**)**;

**try**

**{**

**await** LongRunningTask**(**count, cancellationTokenSource.Token**)**;

**}**

**catch** **(**TaskCanceledException ex**)**

**{**

Console.WriteLine**(**$"{ex.Message}"**)**;

**}**

Console.WriteLine**(**"\nSomeMethod Method Completed"**)**;

**}**

**public** **static** **async** Task LongRunningTask**(int** count, CancellationToken token**)**

**{**

var stopwatch = new Stopwatch**()**;

stopwatch.Start**()**;

Console.WriteLine**(**"\nLongRunningTask Started"**)**;

**for** **(int** i = 1; i **<**= count; i++**)**

**{**

**await** Task.Delay**(**1000**)**;

Console.WriteLine**(**"LongRunningTask Processing...."**)**;

**if** **(**token.IsCancellationRequested**)**

**{**

**throw** new TaskCanceledException**()**;

**}**

**}**

stopwatch.Stop**()**;

Console.WriteLine**(**$"LongRunningTask Took {stopwatch.ElapsedMilliseconds / 1000.0} Seconds for Processing"**)**;

**}**

**}**

# ****How to Create Synchronous Method using Task in C#?****

مواقعی وجود دارد که متدی با امضای Async دارید که task برمیگرداند اما پیاده سازی آن به صورت sync است یکی از دلایل آن این است که باید interface یک task را به صورت async برگرداند.

یکی دلیل دیگر این است که باید در unit test نیاز دارید که تست خود به روش asynchronous انجام دهید و پیاده سازی به صورت sync است. برای حل این مسئله می توان از متد های FromException , FromResult , CompletedTask , FromCanceled استفاده کنید.

**نکات مهم**

1. With Task.CompletedTask method, we can implement a method that returns a task, but that is synchronous.
2. With Task.FromResult method, we can implement a method that is task<T>, but that it is synchronous. And of course, we can return a value that will be wrapped inside of a task.
3. With Task.FromException, we can create a task that completed with an error
4. With Task.FromCanceled, we can create a task that has been canceled.

**نکته مهم**

می توانیم متد های داشته باشیم که به صورت synchronous پیاده سازی شده اند اما یک task برمیگردانند.

**static** **void** Main**(string[]** args**)**

**{**

Console.WriteLine**(**"Main Method Started"**)**;

SomeMethod1**()**;

SomeMethod2**()**;

SomeMethod3**()**;

SomeMethod4**()**;

Console.WriteLine**(**"Main Method Completed"**)**;

Console.ReadKey**()**;

**}**

//Method returning Task but it is synchronous

**public** **static** Task SomeMethod1**()**

**{**

//Do Some Task

Console.WriteLine**(**"SomeMethod1 Executing, It does not return anything"**)**;

//When your method returning Task in synchronous, return Task.CompletedTask

**return** Task.CompletedTask;

**}**

//Synchronous Method returning Task<T>

**public** **static** Task**<string>** SomeMethod2**()**

**{**

**string** someValue = "";

someValue = "SomeMethod2 Returing a String";

Console.WriteLine**(**"SomeMethod2 Executing, It return a string"**)**;

//When your synchronous method returning Task<T>, use Task.FromResult

**return** Task.FromResult**(**someValue**)**;

**}**

//Synchronous Method returning Task with Exception

**public** **static** Task SomeMethod3**()**

**{**

Console.WriteLine**(**"SomeMethod3 Executing, It will throw an Exception"**)**;

//When your synchronous method returning Task with Exception use, Task.FromException

**return** Task.FromException**(**new InvalidOperationException**())**;

**}**

//Synchronous Method Cancelling a Task

**public** **static** Task SomeMethod4**()**

**{**

CancellationTokenSource cts = new CancellationTokenSource**()**;

cts.Cancel**()**;

Console.WriteLine**(**"SomeMethod4 Executing, It will Cancel the Task Exception"**)**;

//When your synchronous method cancelling a Task, Task.FromCanceled

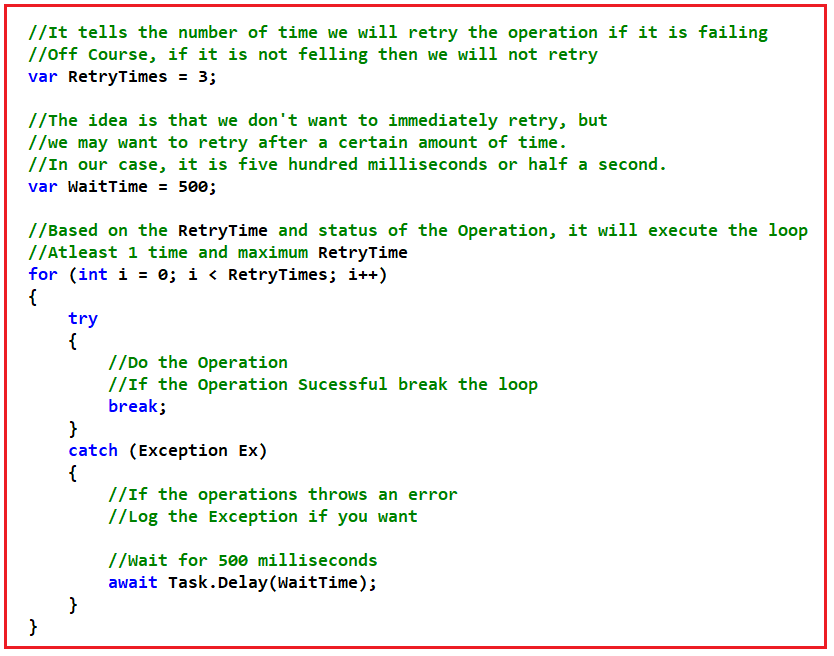
**return** Task.FromCanceled**(**cts.Token**)**;

**}**

**}**

# Retry Pattern in C#

مواقع وجود دارد که می خواهیم یک عملیات چندین بار تکرار شود اما نه صورت immediately بلکه بعد از مدت زمان مشخصی دوباره تکرار شود.



Generic version of pervious code

public static async Task Retry(Func<Task> fun, int RetryTimes = 3, int WaitTime = 500)

{

for (int i = 0; i < RetryTimes; i++)

{

try

{

//Do the Operation

//We are going to invoke whatever function the generic func delegate points to

await fun();

Console.WriteLine("Operation Successful");

break;

}

catch (Exception Ex)

{

//If the operations throws an error

//Log the Exception if you want

Console.WriteLine($"Retry {i + 1}: Getting Exception : {Ex.Message}");

//Wait for 500 milliseconds

await Task.Delay(WaitTime);

}

}

}

More complete sample

public static async Task<T> Retry<T>(Func<Task<T>> fun, int RetryTimes = 3, int WaitTime = 500)

{

//Reducing the for loop Exection for 1 time

for (int i = 0; i < RetryTimes - 1; i++)

{

try

{

//Do the Operation

//We are going to invoke whatever function the generic func delegate points to

//We will return from here if the operation was successful

return await fun();

}

catch (Exception Ex)

{

//If the operations throws an error

//Log the Exception if you want

Console.WriteLine($"Retry {i + 1}: Getting Exception : {Ex.Message}");

//Wait for 500 milliseconds

await Task.Delay(WaitTime);

}

}

//Final try to execute the operation

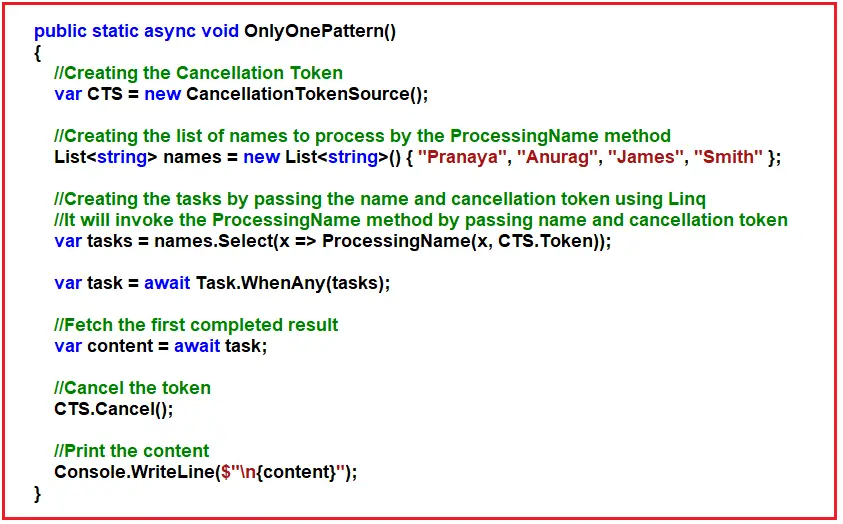
return await fun();

}

# Only One Pattern in C#

مواقعی وجود دارد که چندین task داریم که به ما اطلاعات یکسانی می دهد و ما فقط می خواهیم که یکی از آن ها تمام شود و مابقی را کنسل کنیم به این منظور از الگوی only one استفاده می کنیم با استفاده از cancellationToken

مثال:



Complete example

**static** **void** Main**(string[]** args**)**

**{**

OnlyOnePattern**()**;

Console.ReadKey**()**;

**}**

**public** **static** **async** **void** OnlyOnePattern**()**

**{**

//Creating the Cancellation Token

var CTS = new CancellationTokenSource**()**;

//Creating the list of names to process by the ProcessingName method

List**<string>** names = new List**<string>()** **{** "Pranaya", "Anurag", "James", "Smith" **}**;

Console.WriteLine**(**$"All Names"**)**;

**foreach** **(**var item in names**)**

**{**

Console.Write**(**$"{item} "**)**;

**}**

//Creating the tasks by passing the name and cancellation token using Linq

//It will invoke the ProcessingName method by passing name and cancellation token

var tasks = names.Select**(**x =**>** ProcessingName**(**x, CTS.Token**))**;

var task = **await** Task.WhenAny**(**tasks**)**;

//Fetch the first completed result

var content = **await** task;

//Cancel the token

CTS.Cancel**()**;

//Print the content

Console.WriteLine**(**$"\n{content}"**)**;

**}**

**public** **static** **async** Task**<string>** ProcessingName**(string** name, CancellationToken token**)**

**{**

//Creating Dynamic Waiting Time

//The following statement will generate a number between 1 and 10 dynamically

var WaitingTime = new Random**()**.NextDouble**()** \* 10 + 1;

**await** Task.Delay**(**TimeSpan.FromSeconds**(**WaitingTime**))**;

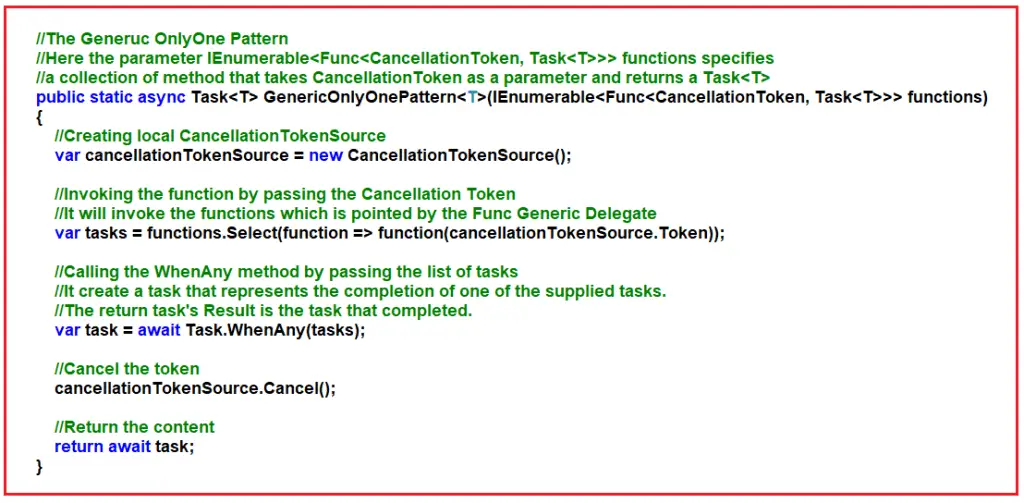
**string** message = $"Hello {name}";

**return** message;

**}**

**}**

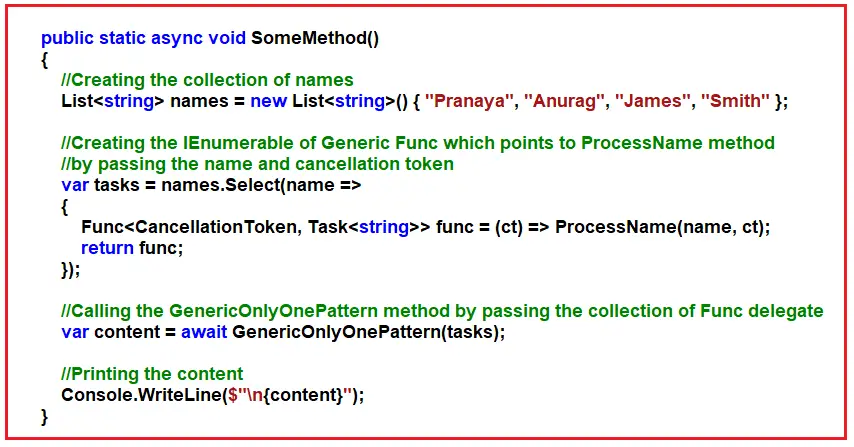
Generic only one pattern



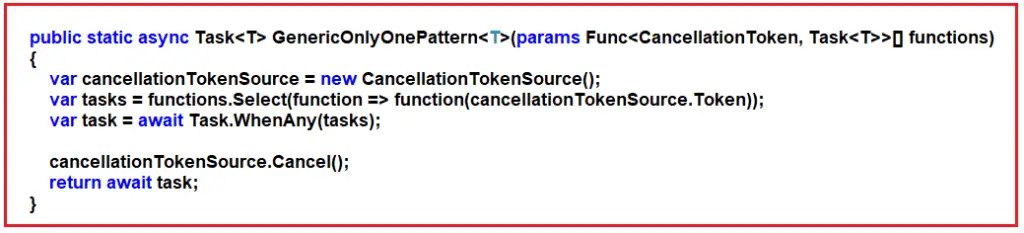
##### **Explanation of the above Code:**

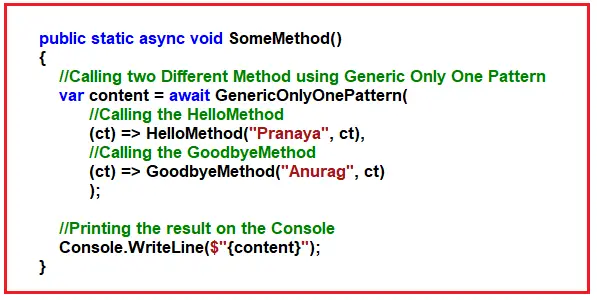
1. **IEnumerable<Func<CancellationToken, Task<T>>> functions:** A Func is a generic delegate that points to a method that returns something. Now, our OneOne Pattern will take multiple tasks. So, the parameter of Our Generic OnlyOne Pattern is going to be an IEnumerable of Func which take Cancellation Token as the input parameter and returns a Task of T i.e. IEnumerable<Func<CancellationToken, Task<T>>> and here we called this parameter as functions. So, here the parameter IEnumerable<Func<CancellationToken, Task<T>>> functions specify a collection of methods that takes CancellationToken as a parameter and returns a Task<T>.
2. **var cancellationTokenSource = new CancellationTokenSource():** Then we are creating a local CancellationTokenSource instance.
3. **var tasks = functions.Select(function => function(cancellationTokenSource.Token)):** Then we are invoking the function by passing the Cancellation Token. It will invoke the functions which are pointed by the Func Generic Delegate. Actually, at this point it will not invoke the methods, it will just create the list of tasks to be invoked when we call the WhenAll method.
4. **var task = await Task.WhenAny(tasks):** Then we call the WhenAny method by passing the list of tasks. The WhenAny method creates a task that represents the completion of one of the supplied tasks. The return task’s Result is the task that is completed.
5. **cancellationTokenSource.Cancel():** Once we get the result from the WhenAny method i.e. once the WhenAny method is completed, then we need to cancel the token.
6. **return await task:** Returning the completed task result.

Generic mode



##### **OnlyOne Pattern with Different Methods in C#:**

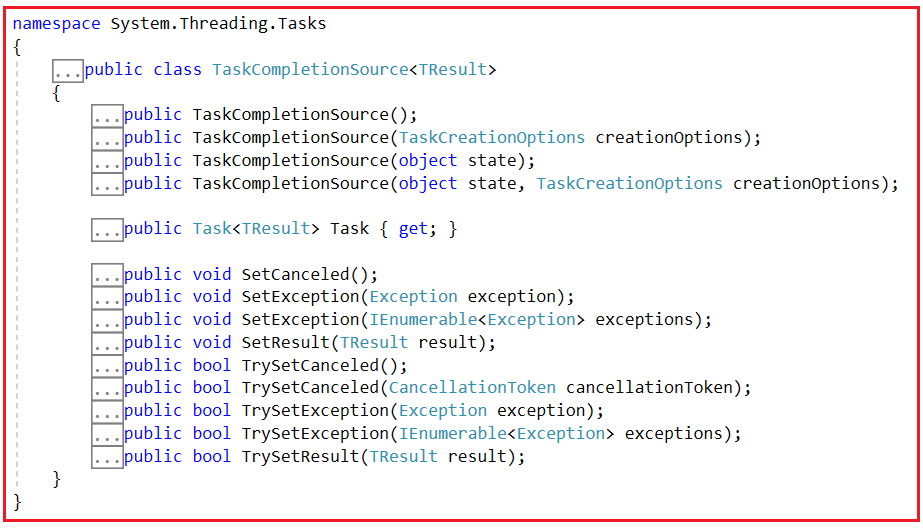




# How to Control the Result of a Task in C#

با استفاده از کلاس TaskCompletionSource می توانیم یک task ایجاد کنیم که بر وضعت آن status کنترل داریم در هر حالتی که باشد مثالا successful , canceled یا حتی دچار Exception شده باشد.

هم می تواند return داشته باشد هم task خالی باشد بدون generic بودن.



1. **TaskCompletionSource():** It creates a System.Threading.Tasks.TaskCompletionSource object.
2. **TaskCompletionSource(TaskCreationOptions creationOptions):** It creates a TaskCompletionSource with the specified options. Here, the parameter creationOptions specify the options to use when creating the underlying Task.
3. **TaskCompletionSource(object state):** It creates a TaskCompletionSource with the specified state. Here, the parameter state specifies the state to use as the underlying Task’s AsyncState.
4. **TaskCompletionSource(object state, TaskCreationOptions creationOptions):** It creates a TaskCompletionSource with the specified state and options. Here, the parameter state specifies the state to use as the underlying Task’s AsyncState and the parameter creationOptions specify the options to use when creating the underlying Task.

##### **Property of TaskCompletionSource class in C#:**

The TaskCompletionSource class in C# provides the following property.

1. **Task<TResult> Task { get; }:** It returns the System.Threading.Tasks.Task created by this TaskCompletionSource.

##### **Methods of TaskCompletionSource class in C#:**

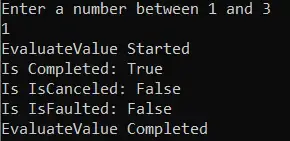
The TaskCompletionSource class in C# provides the following methods.

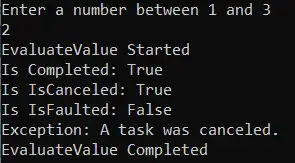
1. **SetCanceled():** This method is used to set the underlying Task into the Canceled state.
2. **SetException(Exception exception):** This method is used to set the underlying Task into the Faulted State and binds it to a specified exception. Here, the parameter exception specifies the exception to binding to this Task.
3. **SetException(IEnumerable<Exception> exceptions):** This method is used to set the underlying Task into the Faulted State and binds a collection of exception objects to it. Here, the parameter exception specifies the collection of exceptions to bind to this Task.
4. **SetResult(TResult result):** This method is used to set the underlying Task into the RanToCompletion State. Here, the parameter result specifies the result value to bind to this Task.

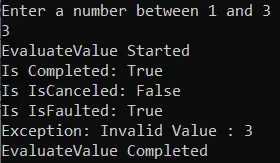
Properties:

1. **IsCompleted { get; }:** It returns true if the task has been completed; otherwise false.
2. **IsCanceled { get; }:** It returns true if the task has been completed due to being canceled; otherwise false.
3. **IsFaulted { get; }:** It returns true if the task has thrown an unhandled exception; otherwise false.
4. **static** **void** Main**(string[]** args**)**
5. **{**
6. Console.WriteLine**(**"Enter a number between 1 and 3"**)**;
7. **string** **value** = Console.ReadLine**()**;
8. SomeMethod**(value)**;
9. Console.ReadKey**()**;
10. **}**
11. **public** **static** **async** **void** SomeMethod**(string** **value)**
12. **{**
13. var task = EvaluateValue**(value)**;
14. Console.WriteLine**(**"EvaluateValue Started"**)**;
15. **try**
16. **{**
17. Console.WriteLine**(**$"Is Completed: {task.IsCompleted}"**)**;
18. Console.WriteLine**(**$"Is IsCanceled: {task.IsCanceled}"**)**;
19. Console.WriteLine**(**$"Is IsFaulted: {task.IsFaulted}"**)**;
20. **await** task;
21. **}**
22. **catch** **(**Exception ex**)**
23. **{**
24. Console.WriteLine**(**ex.Message**)**;
25. **}**
26. Console.WriteLine**(**"EvaluateValue Completed"**)**;
27. **}**
28. **public** **static** Task EvaluateValue**(string** **value)**
29. **{**
30. //Creates an object of TaskCompletionSource with the specified options.
31. //RunContinuationsAsynchronously option Forces the task to be executed asynchronously.
32. var TCS = new TaskCompletionSource**<object>(**TaskCreationOptions.RunContinuationsAsynchronously**)**;
33. **if** **(value** == "1"**)**
34. **{**
35. //Set the underlying Task into the RanToCompletion state.
36. TCS.SetResult**(null)**;
37. **}**
38. **else** **if(value** == "2"**)**
39. **{**
40. //Set the underlying Task into the Canceled state.
41. TCS.SetCanceled**()**;
42. **}**
43. **else**
44. **{**
45. //Set the underlying Task into the Faulted state and binds it to a specified exception.
46. TCS.SetException**(**new ApplicationException**(**$"Invalid Value : {value}"**))**;
47. **}**
48. //Return the task associted with the TaskCompletionSource
49. **return** TCS.Task;
50. **}**
51. **}**

خروجی با مقدار 1 و 2 و 3







در ادامه مثالی با دریافت مقدار بازگشتی خواهیم داشت.

**public** **static** **async** **void** SomeMethod**(string** **value)**

**{**

var task = EvaluateValue**(value)**;

Console.WriteLine**(**"EvaluateValue Started"**)**;

**try**

**{**

Console.WriteLine**(**$"Is Completed: {task.IsCompleted}"**)**;

Console.WriteLine**(**$"Is IsCanceled: {task.IsCanceled}"**)**;

Console.WriteLine**(**$"Is IsFaulted: {task.IsFaulted}"**)**;

var result = **await** task;

Console.WriteLine**(**$"Result: {result}"**)**;

**}**

**catch** **(**Exception ex**)**

**{**

Console.WriteLine**(**$"Exception: {ex.Message}"**)**;

**}**

Console.WriteLine**(**"EvaluateValue Completed"**)**;

**}**

**public** **static** Task**<string>** EvaluateValue**(string** **value)**

**{**

//Creates an object of TaskCompletionSource with the specified options.

//RunContinuationsAsynchronously option Forces the task to be executed asynchronously.

var TCS = new TaskCompletionSource**<string>(**TaskCreationOptions.RunContinuationsAsynchronously**)**;

**if** **(value** == "1"**)**

**{**

//Set the underlying Task into the RanToCompletion state.

TCS.SetResult**(**"Task Completed"**)**;

**}**

**else** **if(value** == "2"**)**

**{**

//Set the underlying Task into the Canceled state.

TCS.SetCanceled**()**;

**}**

**else**

**{**

//Set the underlying Task into the Faulted state and binds it to a specified exception.

TCS.SetException**(**new ApplicationException**(**$"Invalid Value : {value}"**))**;

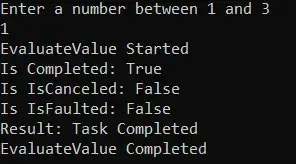
**}**

//Return the task associted with the TaskCompletionSource

**return** TCS.Task;

**}**

Result



# Task-Based Asynchronous Programming in C#

نکته : به صورت عمومی کلاس task برای انجام یک operation به صورت asynchroumusly و بر روی thread از جنس thread pool کاربرد دارد به جای عملیات sync بر روی application main thread

Task using example

**static** **void** Main**(string[]** args**)**

**{**

Console.WriteLine**(**$"Main Thread : {Thread.CurrentThread.ManagedThreadId} Statred"**)**;

Action actionDelegate = new Action**(**PrintCounter**)**;

Task task1 = new Task**(**actionDelegate**)**;

//You can directly pass the PrintCounter method as its signature is same as Action delegate

//Task task1 = new Task(PrintCounter);

task1.Start**()**;

Console.WriteLine**(**$"Main Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

Console.ReadKey**()**;

**}**

**static** **void** PrintCounter**()**

**{**

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Started"**)**;

**for** **(int** count = 1; count **<**= 5; count++**)**

**{**

Console.WriteLine**(**$"count value: {count}"**)**;

**}**

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

**}**

**}**

در زمان اجرای کد بالا 2 ترد در حال اجرا وجود دارد main و child

تقاوت میان task , task.factory این است که factory برای کار های است نیاز است به کنترل بیشتری بر روی task نیاز داریم مثلا می خواهیم یک longRunning task داشته باشیم.

**static** **void** Main**(string[]** args**)**

**{**

Console.WriteLine**(**$"Main Thread : {Thread.CurrentThread.ManagedThreadId} Statred"**)**;

Task task1 = Task.Factory.StartNew**(**PrintCounter**)**;

Console.WriteLine**(**$"Main Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

Console.ReadKey**()**;

**}**

.Wait() Method

این متد برای مواقعی که کاربرد دارد که بخواهیم اجرای سایر thread ها متوقف شود تا زمانی که کار مربوط به Task ما به پایان برسد کاربرد دارد. این متد سایر thread ها را به حالت block در می اورد.

**static** **void** Main**(string[]** args**)**

**{**

Console.WriteLine**(**$"Main Thread : {Thread.CurrentThread.ManagedThreadId} Statred"**)**;

Task task1 = Task.Run**(()** =**>**

**{**

PrintCounter**()**;

**})**;

task1.Wait**()**;

Console.WriteLine**(**$"Main Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

Console.ReadKey**()**;

**}**

How to use anonymous and lamda expression with task

**static** **void** Main**(string[]** args**)**

**{**

Console.WriteLine**(**$"Main Thread : {Thread.CurrentThread.ManagedThreadId} Statred"**)**;

#region Stat Method

//Creating Task using Method

Task task1 = new Task**(**PrintCounter**)**;

task1.Start**()**;

//Creating Task using Anonymous Method

Task task2 = new Task**(delegate** **()**

**{**

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Started"**)**;

Task.Delay**(**200**)**;

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

**})**;

task2.Start**()**;

//Creating Task using Lambda Expression

Task task3 = new Task**(()** =**>**

**{**

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Started"**)**;

Task.Delay**(**200**)**;

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

**})**;

task3.Start**()**;

#endregion

#region StartNew

//Creating Task using Method

Task task4 = Task.Factory.StartNew**(**PrintCounter**)**;

//Creating Task using Anonymous Method

Task task5 = Task.Factory.StartNew**(delegate** **()**

**{**

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Started"**)**;

Task.Delay**(**200**)**;

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

**})**;

//Creating Task using Lambda Expression

Task task6 = Task.Factory.StartNew**(()** =**>**

**{**

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Started"**)**;

Task.Delay**(**200**)**;

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

**})**;

#endregion

#region Run

//Creating Task using Method

Task task7 = Task.Run**(()** =**>** **{** PrintCounter**()**; **})**;

//Creating Task using Anonymous Method

Task task8 = Task.Run**(delegate** **()**

**{**

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Started"**)**;

Task.Delay**(**200**)**;

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

**})**;

//Creating Task using Lambda Expression

Task task9 = Task.Run**(()** =**>**

**{**

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Started"**)**;

Task.Delay**(**200**)**;

Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

**})**;

#endregion

Console.WriteLine**(**$"Main Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

Console.ReadKey**()**;

**}**

##### **What is a Thread Pool in C#?**

استخر مدیریت شده از thread ها است که توسط .net runtime ایجاد و مدیریت می شود برای انجام کار های async , parallel به صورت efficiently. این یک ویژگی بنیادی در .net برای افزایش بهره وری و کارامدی برنامه نویسی async , multithread است.

ThreaPool در واقع کاکلکشنی از thread ها است که برای انجام task های background کاربر دارد و زمانی که task تکمیل شد thread به threa pool بازمی گردد و آن thread دوباره قابل استفاده است.

##### **Advantages of Task-Based Asynchronous Programming in C#:**

Here are some key advantages of using TAP (Task-Based Asynchronous Programming):

###### **Simplified Code Structure:**

* **Readability:** Task-Based Asynchronous Programming allows for writing asynchronous code similar in structure to synchronous code, improving readability.
* **Maintainability:** The code is easier to maintain as it avoids the complexity of callbacks and manual thread management in older models.

###### **Improved Scalability and Performance:**

* **Efficient Resource Utilization:** Asynchronous operations free up the calling thread (typically UI or server threads) to handle other tasks. This improves resource utilization and responsiveness, particularly in UI applications or web services.
* **Scalability:** TAP can enhance the scalability of applications, especially those that handle many concurrent I/O-bound operations.

###### **Language and Framework Support:**

* **Language Integration:** Task-based asynchronous Programming is seamlessly integrated with C# language features, notably async and await keywords, making asynchronous programming more intuitive.
* **Framework Compatibility:** It’s fully supported across the .NET ecosystem, including newer versions of .NET Core and .NET 5/6, ensuring compatibility and ease of use.

###### **Exception Handling:**

* **Simplified Exception Handling:** Exceptions in asynchronous methods can be caught and handled using standard try-catch blocks, unlike older patterns that require more complex handling.

###### **Composability and Flexibility:**

* **Composable Operations:** Tasks can be easily combined and composed. For instance, you can await multiple tasks concurrently using Task.WhenAll, or await the first task to complete using Task.WhenAny.
* **Cancellation Support:** Task-Based Asynchronous Programming supports cancellation using the CancellationToken class, allowing for responsive cancellation of asynchronous operations.

###### **Unified Model for Asynchronous Operations:**

* **Consistency:** TAP provides a unified approach for all asynchronous operations, whether CPU-bound or I/O-bound, creating consistency in how asynchronous code is written and understood across different applications.

###### **Progress Reporting:**

* **Progress Feedback:** The model supports progress reporting out of the box, which is particularly useful in UI applications where you need to update the UI to reflect the progress of an asynchronous operation.

##### **Disadvantages of Task-Based Asynchronous Programming in C#:**

* **Complexity in Error Handling:** Asynchronous programming can make error handling more complex. Exceptions thrown in asynchronous methods are captured and placed on the Task, and they need to be handled using await or by examining the Task object. Unobserved exceptions can lead to unhandled exceptions.
* **Potential for Deadlocks:** Misusing async and await, especially with Task.Result or Task.Wait(), can lead to deadlocks, particularly in UI applications or when blocking on asynchronous code.
* **Resource Management:** Asynchronous operations can lead to more complex resource management scenarios. Ensuring that resources are properly disposed of or that certain operations are thread-safe adds complexity to the code.
* **Scalability Issues:** Asynchronous programming is great for scalability; improper use (like creating too many tasks or not using I/O-bound asynchronous APIs correctly) can consume many resources and degrade performance.
* **Debugging Difficulty:** Debugging asynchronous code can be more difficult than synchronous code. The execution flow is not linear, making it harder to follow and understand, especially when dealing with multiple concurrent asynchronous operations.
* **Overhead:** Some overhead is associated with managing the state and context of asynchronous operations. For very small operations, the overhead of setting up the asynchronous operation might outweigh its benefits.
* **Improper Usage:** It’s easy to misuse asynchronous programming by applying it where it’s not needed, leading to unnecessary complexity. Not every operation benefits from being made asynchronous.

# Chaining Tasks by Using Continuation Tasks

Task های که اجرای آن ها توسط یک دیگر انجام می شود به این معنی که یک task می تواند task دیگری را اجرا کند.

ContiueWith()

زمانی که task جاری خود به صورت کامل اجرا شود سپس Task فراخوانی شده در این متد اجرا می شود.

نکته : به مقدار return شده توسط task قبل خود نیز دسترسی داریم.

**static** **void** Main**(string[]** args**)**

**{**

Task**<string>** task1 = Task.Run**(()** =**>**

**{**

**return** 12;

**})**.ContinueWith**((**antecedent**)** =**>**

**{**

**return** $"The Square of {antecedent.Result} is: {antecedent.Result \* antecedent.Result}";

**})**;

Console.WriteLine**(**task1.Result**)**;

Console.ReadKey**()**;

**}**

Result : The square of 12 is : 144

این متد دارای overload های است که می توانیم مشخص کنیم چه زمانی فراخوانی شود مثل زمان exception یا cancel شدن و ...

task.ContinueWith**((**i**)** =**>**

**{**

Console.WriteLine**(**"TasK Canceled"**)**;

**}**, TaskContinuationOptions.OnlyOnCanceled**)**;

task.ContinueWith**((**i**)** =**>**

**{**

Console.WriteLine**(**"Task Faulted"**)**;

**}**, TaskContinuationOptions.OnlyOnFaulted**)**;

var completedTask = task.ContinueWith**((**i**)** =**>**

**{**

Console.WriteLine**(**"Task Completed"**)**;

**}**, TaskContinuationOptions.OnlyOnRanToCompletion**)**;

# How to Attached Child Tasks to a Parent Task in C#

یک child task یا nested task تسکی است که ساخته می شود توسط یک تسک دیگر که به آن parent task می گوییند.

یک child task می تواند attached , detached شود.

##### **What are Detached or Attached Child Tasks in C#?**

Detached task : تسکی است که اجرا می شود به صورت مستقل از parent خود. در این حالت هر دو Task پدر و فرزند به صورت مستقل از هم اجرا می شوند و parent منتظر پایان تسک فرزند خود نمی ماند و اگر exception توسط child رخ دهد باید توسط parent مدیریت شود و در نهایت status تسک پدر وابسته به child نیست.

Attached task : تسک فرزند متصل تسکی است که ایجاد شده با TaskCreationOption.AttachedToParent این تنظیم در این موارد parent منتظر child خود خواهد ماند تا عملیات آن تکمیل شود و اگر exception توسط child رخ دهد باید توسط parent خود مدیریت شود و parent status وابسته با child status است.

نکته: یک task می تواند دارای تعدادی از attached , detached تسک داشته باشد.

نکته: **طبق مستندات ماکروسافت پیشنهاد می شود که در بیشتر سناریو ها از detached task استفاده شود به عنوان child به دلیل این که ارتباط آن با parent خود کمتر complex است. به همین دلیل است که به صورت پیشفرض تسک ایجاد شده از نوع detached است و در صورت نیاز باید explicitly اعلام شود**

**TaskCreationOptions.AttachedToParent**

**نکته ای که باید در نظر گرفته شود این است که در هر بار اجرا ممکن است تسک فرزند اجرا نشود به دلیل این که idempotent است و به صورت مستقل اجرا می شود.**

**Example**

public static void Main()

{

Console.WriteLine("Main Method Started");

//Creating the Parent Task

var parentTask = Task.Factory.StartNew(() => {

Console.WriteLine("Outer Task Started");

//Creating the Child Task

var childTask = Task.Factory.StartNew(() => {

Console.WriteLine("Child Task Started.");

Thread.Sleep(5000);

Console.WriteLine("Child Task Completed");

});

Console.WriteLine("Outer Task Completed");

});

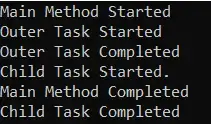
//Waiting for the Parent Task to completed. Not the Child Task

parentTask.Wait();

Console.WriteLine("Main Method Completed");

Console.ReadKey();

}



##### **How Parent task will wait for the detached task to complete its execution?**

اگر یک child task که task<T> برمیگرداند و ما بخواهیم که parent آن تسک منتظر اتمام فرآیند انجام تسک فرزند باشد باید از دستور .**Result** استفاده کنیم که سبب می شود حتی در حالتی که تسک detached است تسک **پدر block** شود تا زمانی که تسک فرزند کار خود را به اتمام برساند.

در ادامه مثالی را مشاهده خواهیم کرد.

public static void Main()

{

Console.WriteLine("Main Method Started");

//Creating the Parent Task

var parentTask = Task<string>.Factory.StartNew(() => {

Console.WriteLine("Outer Task Started");

//Creating the Child Task

var childTask = Task<string>.Factory.StartNew(() => {

Console.WriteLine("Child Task Started");

Thread.Sleep(5000);

Console.WriteLine("Child Task Completed");

return "Task Completed";

});

// Parent Task will wait for detached Child Task to complete its execution

return childTask.Result;

});

##### **Example to Understand Attached Child Tasks in C#:**

برای ایجاد این نوع تسک ها لازم است که فقط option مربوطه را ست کنیم

public static void Main()

{

Console.WriteLine("Main Method Started");

//Creating the Parent Task

var parentTask = Task.Factory.StartNew(() => {

Console.WriteLine("Outer Task Started");

//Creating the Child Task

var childTask = Task.Factory.StartNew(() => {

Console.WriteLine("Child Task Started");

Thread.Sleep(5000);

Console.WriteLine("Child Task Completed");

}, TaskCreationOptions.AttachedToParent);

Console.WriteLine("Outer Task Completed");

});

/Waiting for the Parent Task to completed.

parentTask.Wait();

##### **How to Prevent Parent tasks from attached Child Tasks in C#?**

با یک تنظیم می توانیم جلوگیری کنیم از این که child task به parent task متصل شود حتی اگر تسک فرزند تنظیم شده باشد برای اتصال.

**TaskCreationOptions.DenyChildAttach**

Console.WriteLine("Main Method Started");

//Creating the Parent Task with DenyChildAttach Option

var parentTask = Task.Factory.StartNew(() => {

Console.WriteLine("Outer Task Started");

//Creating the Child Task with AttachedToParent

var childTask = Task.Factory.StartNew(() => {

Console.WriteLine("Child Task Started");

Thread.Sleep(5000);

Console.WriteLine("Child Task Completed");

}, TaskCreationOptions.**AttachedToParent**);

Console.WriteLine("Outer Task Completed");

}, **TaskCreationOptions.DenyChildAttach**);

**نکته مهم : در صورتی که از task.run استفاده کرده باشیم امکان اتصال attachedTask وجود ندارد و از آن جلوگیری می کند.**

##### **Exceptions in Detached Child Task in C#**

اگر تسک از نوع detached باشد و یک exception توسط تسک فرزند throw شود آن ex توسط تسک پدر دریافت نمی شود و مدیریت نمی شود. و فرآیند اجرای طبیعی ادامه پیدا می کند.

public static void Main()

{

Console.WriteLine("Main Method Started");

try

{

//Creating the Parent Task using Task.Run Method

var parentTask = Task.Factory.StartNew(() =>

{

Console.WriteLine("Outer Task Started");

//Creating the Child Task with AttachedToParent

var childTask = Task.Factory.StartNew(() =>

{

Console.WriteLine("Child Task Started");

int x = 10, y = 0;

int z = x / y; //It will throw an Exception

Console.WriteLine("Child Task Completed");

});

Console.WriteLine("Outer Task Completed");

});

//Waiting for the Parent Task to completed.

parentTask.Wait();

}

catch (Exception ex)

{

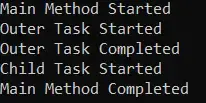
Console.WriteLine($"Exception Occurred: {ex.Message}");

}

Console.WriteLine("Main Method Completed");

Console.ReadKey();

}



##### **Exceptions in Attached Child Task in C#**

اگر یک attached task یک ex پرتاب کند آن ex به صورت خودکار توسط parent task مدیریت می شود یا تردی از .result استفاده کرده است با استفاده از attached task می توان exception ها را در یک نقطه مدیریت کرد در مثال زیر توسط parent اکسپشن رخ داده مدیریت می شود.

public static void Main()

{

Console.WriteLine("Main Method Started");

try

{

//Creating the Parent Task using Task.Run Method

var parentTask = Task.Factory.StartNew(() =>

{

Console.WriteLine("Outer Task Started");

//Creating the Child Task with AttachedToParent

var childTask = Task.Factory.StartNew(() =>

{

Console.WriteLine("Child Task Started");

int x = 10, y = 0;

int z = x / y; //It will throw an Exception

Console.WriteLine("Child Task Completed");

},TaskCreationOptions.AttachedToParent);

Console.WriteLine("Outer Task Completed");

});

//Waiting for the Parent Task to completed.

parentTask.Wait();

}

catch (Exception ex)

{

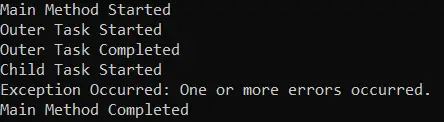
Console.WriteLine($"Exception Occurred: {ex.Message}");

}

Console.WriteLine("Main Method Completed");

Console.ReadKey();

}



# ValueTask in C#

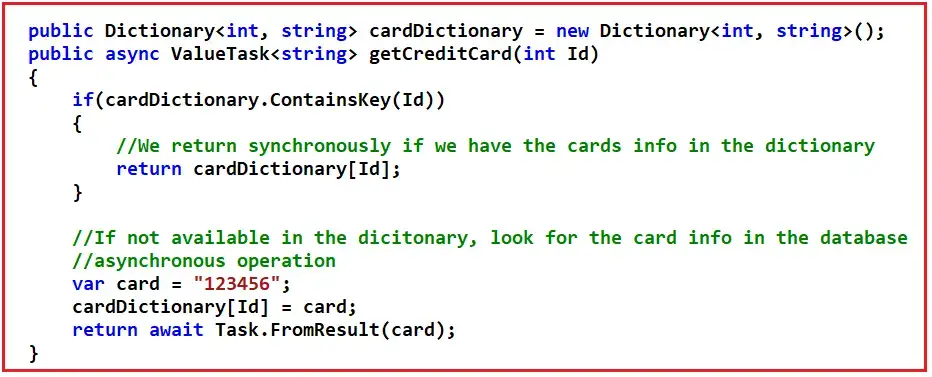
بسیار شبیه به Task , Task<> است با این تقاوت که هدف اصلی آن performance است و از نوع struct است و value type بوده برعکس task که reference type است.

Where to use

1. **Condition1**: When the result of the operation is most likely available synchronously.
2. **Condition2**: When the operation is used so frequently that the cost of using Task or Task<T> is significant.

جواب سریع در دسترس باشد یا این که مکررا به آن نیاز داشته باشیم. مثلا کش شده باشد.

سناریو ی اول شرایطی است مه متدری async داریم که به صورت sync نیز اجرا می شود خیلی کاربرد دارد.



ترجیح ما استفاده از task است اما اگر خواستید استفاده کنید حتما بررسی کنید که واقعا تاثیر گذار باشد.

##### **Limitations of ValueTask in C#:**

1. The first one is that they cannot be Cache.
2. You cannot await it multiple times.
3. It does not support multiple continuations.
4. It is not thread-safe with an arbitrary number of threads capable of concurrently registering continuations.
5. They do not support a blocking model where the asynchronous method can block the current thread instead of releasing it using await operator.

# How to Cancel a Non-Cancellable Task in C#

منظور از task های non-cancellable تسک های است که cancellation token دریافت نمی کنند. کاری که می کنیم با استفاده از TaskCompletionSource است برای ساخت یک task که می توانیم با token آن را کنسل کنیم.

مناسب زمانی است که نمی خواهیم از shchedule time out استفاده کنیم اما می خواهیم قابلیت cancel داشته باشیم.

ایجاد متد Cancellation برای تسک های غیر قابل کنسل به صورت extension متد.

public static async Task<T> WithCancellation<T>(this Task<T> task, CancellationToken cancellationToken)

{

var TCS = new TaskCompletionSource<object>(TaskCreationOptions.RunContinuationsAsynchronously);

using (cancellationToken.Register(state =>

{

((TaskCompletionSource<object>)state).TrySetResult(null);

},TCS))

{

var resultTask = await Task.WhenAny(task, TCS.Task);

if(resultTask == TCS.Task)

{

throw new OperationCanceledException(cancellationToken);

}

return await task;

};

}

}

استفاده از متد

var result = await Task.Run(async () =>

{

await Task.Delay(TimeSpan.FromSeconds(5));

Console.WriteLine("Operation was Successful");

return 7;

}).WithCancellation(cancellationTokenSource.Token);

# Asynchronous Streams in C#

از سی شارپ نسخه 8 به بعد ما محدودیتی برای برگرداندن single value از یک asynchronous متد نداریم می توانیم sequence از value ها را که به صورت dynamic ایجاد می شوند را برگردانیم.

##### **IEnumerable Interface and yield keyword in C#**

رابط IEnumerable به ما این امکان را می دهد که بر روی یک type عملیات iterate را انجام دهیم.

Yield » به ما این امکان را می دهد که مقادیر با به صورت one by one جنریت کنیم.

Asynchronous stream

یک stream که به ما این امکان را می دهد که مقادیر را به صورت یک به یک برگردانیم.

**Example**

static void Main(string[] args)

{

foreach (var name in GenerateNames())

{

//You can do anything with the name

//for example printing the name on the console

Console.WriteLine(name);

}

Console.ReadKey();

}

//This method is going to generate names over a period of time

private static IEnumerable<string> GenerateNames()

{

yield return "Anurag";

yield return "Pranaya";

yield return "Sambit";

}

اگر مثال بالا را اجرا کنید می بینید که به صورت one by one مقادیر بر روی صفحه نمایش داده می شوند به دلیل استفاده از stream با استفاده از yield.

کاری که yield انجام می دهد این است که با هر بار return دفعه ی بعد مقدار قبلی را نه بلکه مقدار جدید را برمیگرداند.

##### **Stream with Asynchronous Programming in C#**

برای تبدیل مثال قبل به async از روش زیر استفاده می کنیم.

static void Main(string[] args)

{

foreach (var name in GenerateNames())

{

Console.WriteLine(name);

}

Console.ReadKey();

}

private static async Task<IEnumerable<string>> GenerateNames()

{

yield return "Anurag";

yield return "Pranaya";

await Task.Delay(TimeSpan.FromSeconds(3));

yield return "Sambit";

}

کد بالا با خطا مواجه می شود به دلیل این که مقدار برگشتی متد از نوع task است و قابل استفاده در foreach نیست.

Solution

IAsyncEnumreable<>

static async Task Main(string[] args)

{

**await** foreach(var name in GenerateNames())

{

Console.WriteLine(name);

}

Console.ReadKey();

}

private static async **IAsyncEnumerable**<string> GenerateNames()

{

yield return "Anurag";

yield return "Pranaya";

await Task.Delay(TimeSpan.FromSeconds(3));

yield return "Sambit";

}

نکته » متد delay سبب بلاک شدن thread نمی شود.

برای زمانی که بلافاصله بعد از آماده شدن داده می خواهیم دسترسی پیدا کنیم و آن را در memory نگه داری نکنیم.

# How to Cancel Asynchronous Stream in C#

With using **break** keyword

static async Task Main(string[] args)

{

await foreach (var name in GenerateNames())

{

Console.WriteLine(name);

//Some condition to break the asynchronous stream

if (name == "Pranaya")

{

break;

}

}

Console.ReadKey();

}

Using **Cancellation token**

static async Task Main(string[] args)

{

//Create an instance of CancellationTokenSource

var CTS = new CancellationTokenSource();

//Set the time when the token is going to cancel the stream

CTS.CancelAfter(TimeSpan.FromSeconds(5));

try

{

//Pass the Cancelllation Token to GenerateNames method

await foreach (var name in GenerateNames(CTS.Token))

{

Console.WriteLine(name);

}

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

finally

{

//Dispose the CancellationTokenSource

CTS.Dispose();

CTS = null;

}

Console.ReadKey();

}

//This method accepts Cancellation Token as input parameter

//Set its value to default

private static async IAsyncEnumerable<string> GenerateNames(CancellationToken token = default)

{

//Check if request comes for Token Cancellation

//if(token.IsCancellationRequested)

//{

// token.ThrowIfCancellationRequested();

//}

//But here we just need to pass the token to Task.Delay method

yield return "Anurag";

await Task.Delay(TimeSpan.FromSeconds(3), token);

yield return "Pranaya";

await Task.Delay(TimeSpan.FromSeconds(3), token);

yield return "Sambit";

await Task.Delay(TimeSpan.FromSeconds(3), token);

yield return "Rakesh";

}

}

##### **Canceling Through IAsyncEnumerable – EnumeratorCancellation in C#:**

مثال های کامل در لینک زیر

https://dotnettutorials.net/lesson/how-to-cancel-asynchronous-stream-in-csharp/