# Asynchronous programming:

תיכנות א-סינכרוני נפוץ מאד בעבודה כיום. כשאנחנו קוראים מקובץ גדול, מבצעים קריאה לשרת, או מבצעים פעולה אחרת שלוקחת זמן ממושך, אנחנו לא מעוניינים שהאפליקציה תמתין לאותה פעולה.

במילים אחרות אם פעולה מסוימת לוקחת זמן בתכנות א-סינכרוני היא חוסמת את כל האפליקציה שמפסיקה להגיב.

תכנות א-סנכרוני הוא יעיל מאד במצב כזה. האפליקציה יכולה להמשיך ולא תלויה בביצוע משימה מסויימת.

# **C# Threadpool**

בדרך כלל לא ידוע מראש כמה threads, ירוצו בכל תוכנית. זה תלוי בכל מיני פרמטרים כגון:

* The number of CPU cores.
* The type of tasks that need to be executed.
* System performance.

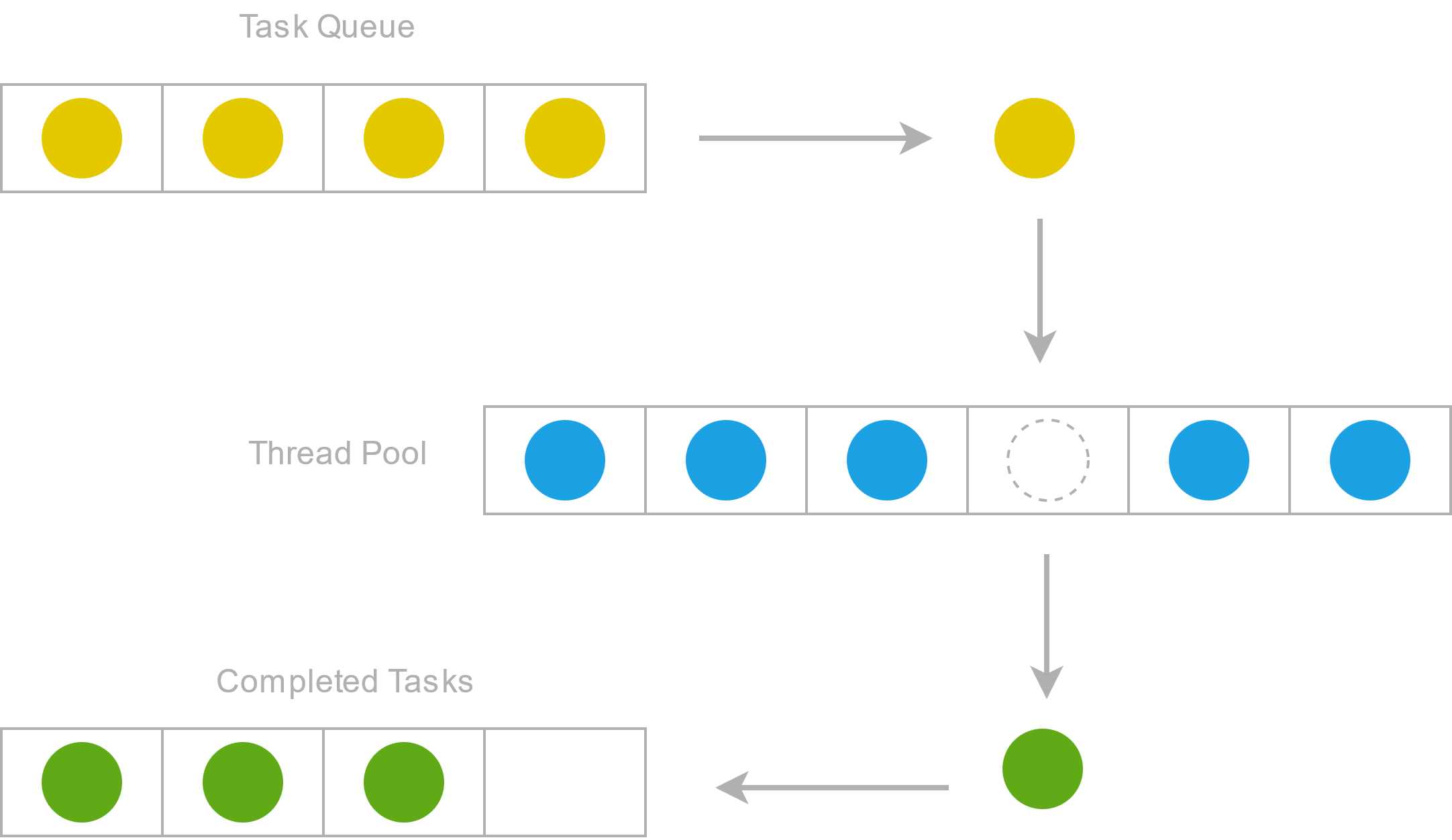
על המתכנת לשקול את כל הפרמטרים כדי לא לפגוע בביצועי האפליקציה.

הפתרון לאי הוודאות הוא thread pool.

בהגדרה thread pool הוא מאגר מנוהל של threads, שניתן להשתמש בהם כדי ליצור מקביליות. הthreads הקיימים במאגר נקראים גם worker threads. Threads אלה הם background threads.

**The thread pool works as follows:**

* מספר threads מיוצרים מראש.
* משימה עומדת בתור למאגר, עד שמתפנה thread לטפל בה.
* כאשר המשימה מסתיימת הthread חוזר למאגר ומחכה למשימה חדשה.

בשיטה זו יש מיחזור של thread וחיסכון בoverhead של יצירתם מחדש לכל משימה חדשה.

To create a thread pool, you use the ThreadPool class. The ThreadPool class provides you with the QueueUserWorkItem for submitting tasks to the thread pool.

The QueueUserWorkItem method queues a method for execution. In other words, the method executes when a thread in the thread pool is ready for execution.

## **C# Threadpool example**

How it works.

First, define a static method CheckHttpStatus to check the HTTP status code of an URL:

static void CheckHttpStatus(string url)

{

HttpClient client = new(); //new HttpClient();

var response = client.GetAsync(url).Result;

Console.WriteLine($"The HTTP status code of {url} is {response.StatusCode}");

}

List<string> urls = new(){

"https://www.google.com/",

"https://www.duckduckgo.com/",

"https://www.yahoo.com/",

};

foreach (var url in urls)

{

ThreadPool.QueueUserWorkItem((state) => CheckHttpStatus(url));

}

// wait for all thread to complete

// and press a key

Console.Read();}

Second, create a list of URLs to check:

Third, check the HTTP status code of each URL of the list by passing a lambda expression to the QueueUserWorkItem() method of the ThreadPool method:

Finally, add the Console.Read() that blocks the main thread to wait for all threads in the thread pool to be completed.

It’s important to note that you can wait for all the tasks to be completed but we’ll cover it in the subsequent tutorial.

## **Summary**

* Use a thread pool to improve the application’s performance by making efficient use of available resources and reducing the overhead of creating and destroying threads.
* Use the ThreadPool class to create a thread pool.
* Use the QueueUserWorkItem method of the ThreadPool class to submit tasks to the thread pool.

**TAP- Task-Based Asynchronous programming:**

TAP- Task-Based Asynchronous programming greatly simplifies asynchronous programming and makes it easier to write asynchronous code.

**TAP consists of the following key components:**

The Task class – represents an asynchronous operation.

The async / await keywords – define asynchronous methods and wait for the completion of asynchronous operations.

Task-based API – a set of classes that work seamlessly with the Task class and async/await keywords.

**TAP has the following advantages:**

Improved performance – TAP can improve an application’s performance by allowing it to perform I/O-bound operations asynchronously, freeing up the CPU for other tasks.

Simplified code – TAP allows you to write asynchronous code like synchronous code that makes it easy to understand.

Better resource management – TAP optimizes system resources by allowing applications to perform asynchronous operations without blocking threads.

**The Task class:**

מחלקת הTask היא הבסיס לTAP. היא מייצגת פעולות א-סינכרוניות שניתן לבצע בדרכים שונות.

פונקציה לדוגמא: המחזירה מספר אקראי. שימו לפעולה thread.sleep המיועדת לעכב את הפונקציה.

static int GetRandomNumber()

{

Thread.Sleep(1000);

int randomNumber = (new Random()).Next(1, 100);

Console.WriteLine($"The random number is {randomNumber}");

return randomNumber;

}

**Running a task**

To execute the GetRandomNumber() method asynchronously, you create a new Task object and call the GetRandomNumber() method in a lambda expression passed to the Task‘s constructor and start executing the task by calling the Start() method of the Task object:

var task = new Task(() => GetRandomNumber());

task.Start();

Put it all together:

static int GetRandomNumber()

{

Thread.Sleep(1000);

int randomNumber = (new Random()).Next(1, 100);

Console.WriteLine($"The random number is {randomNumber}");

return randomNumber;

}

var task = new Task(() => GetRandomNumber());

task.Start();

Console.WriteLine("Start the program...");

Console.ReadLine();

**Output:**

Start the program...

The random number is 65

Note that the task.Start() method doesn’t block the main thread therefore you see the following message first:

Start the program...before the random number:

The random number is 65

The Console.ReadLine() blocks the main thread until you press a key. It is used for waiting for the child thread scheduled by the Task object to complete.

If you don’t block the main thread, it’ll be terminated after the program displays the message “Start the program…”.

Behind the scenes, the program uses a thread pool for executing the asynchronous operation. The Start() method schedules the operation for execution.

To prove this, we can display the thread id and whether the thread belongs to the managed thread pool:

static int GetRandomNumber()

{

var threadId = Thread.CurrentThread.ManagedThreadId;

var threadPool = Thread.CurrentThread.IsThreadPoolThread;

Console.WriteLine($"The thread #{threadId}, use a thread pool {threadPool}");

Thread.Sleep(1000);

int randomNumber = (new Random()).Next(1, 100);

Console.WriteLine($"The random number is {randomNumber}");

return randomNumber;

}

var task = new Task(() => GetRandomNumber());

task.Start();

Console.WriteLine("Start the program...");

Console.ReadLine();

**Output:**

Start the program...

The thread #5, use a thread pool True

The random number is 97

The output shows that the thread id is 5 and the thread belongs to a thread pool. Note that you likely see a different number.

Since the code for creating a Task object and starting it are quite verbose, you can shorten it by using the Run() static method of the Task class:

Task.Run(() => GetRandomNumber());

The Run() method queues operation (GetRandomNumber) to the thread pool for execution.

Similarly, you can use the StartNew() method of the Factory object of the Task class to create a new task and schedule its execution:

Task.Factory.StartNew(() => GetRandomNumber());

**Getting the result from a task**

The Run() method returns a Task<TResult> object that represents the result of the asynchronous operation.

In our example, the GetRandomNumber() returns an integer, therefore, the Task.Run() returns the Task<int> object:

Task<int> task = Task.Run(() => GetRandomNumber());

To get the returned number of the GetRandomNumber() method, you use the Result property of the task object: task.Result

Put it all together:

static int GetRandomNumber()

{

Thread.Sleep(1000);

int randomNumber = (new Random()).Next(1, 100);

return randomNumber;

}

Task<int> task = Task.Run(() => GetRandomNumber());

Console.WriteLine($"The random number is {task.Result}");

**Async Await:**

We will get all the benefits of traditional Asynchronous programming with much less effort with the help of async and await keywords.

Suppose we are using two methods as Method1 and Method2 respectively, and both the methods are not dependent on each other, and Method1 takes a long time to complete its task. In Synchronous programming, it will execute the first Method1 and it will wait for the completion of this method, and then it will execute Method2. Thus, it will be a time-intensive process even though both methods are not depending on each other.

We can run all the methods parallelly by using simple thread programming, but it will block UI and wait to complete all the tasks. To come out of this problem, we have to write too many codes in traditional programming, but if we use the async and await keywords, we will get the solutions in much less code.

Also, we are going to see more examples, and if any third Method, as Method3 has a dependency of method1, then it will wait for the completion of Method1 with the help of await keyword.

Async and await in C# are the code markers, which marks code positions from where the control should resume after a task completes.

Let’s start with practical examples for understanding the programming concept.

## **Code examples of C# async await**

We are going to take a console application for our demonstration.

**Example 1**

In this example, we are going to take two methods, which are not dependent on each other.

**Code sample:**

Method1();

Method2();

Console.ReadKey();

public static async Task Method1()

{

await Task.Run(() =>

{

for (int i = 0; i < 100; i++)

{

Console.WriteLine(" Method 1");

// Do something

Task.Delay(100).Wait();

}

});

}

public static void Method2()

{

for (int i = 0; i < 25; i++)

{

Console.WriteLine(" Method 2");

// Do something

Task.Delay(100).Wait();

}

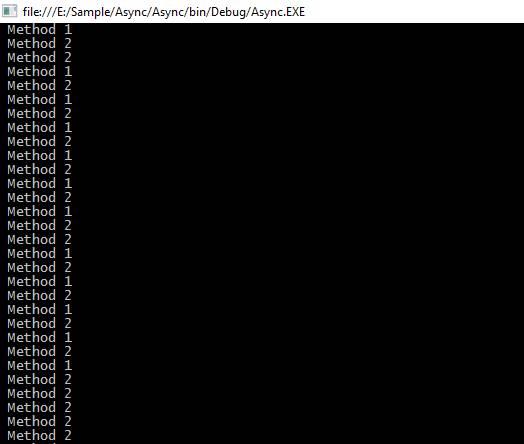
}

}

**Output**

in the code given above, Method1 and Method2 are not dependent on each other and we are calling from the Main method.

Here, we can clearly see Method1, and Method2 are not waiting for each other.



In this example, Method1 is returning the total length as an integer value and we are passing a parameter as a length in a Method3, which is coming from Method1.

Here, we have to use await keyword before passing a parameter in Method3 and for it, we have to use the async keyword from the calling method.

We are going to create a new method as callMethod and in this method, we are going to call our all Methods as Method1, Method2, and Method3, respectively.

callMethod();

Console.ReadKey();

public static async void callMethod()

{

Task<int> task = Method1();

Method2();

int count = await task;

Method3(count);

}

public static async Task<int> Method1()

{

int count = 0;

await Task.Run(() =>

{

for (int i = 0; i < 100; i++)

{

Console.WriteLine(" Method 1");

count += 1;

}

});

return count;

}

}

public static void Method2()

{

for (int i = 0; i < 25; i++)

{

Console.WriteLine(" Method 2");

}

}

public static void Method3(int count)

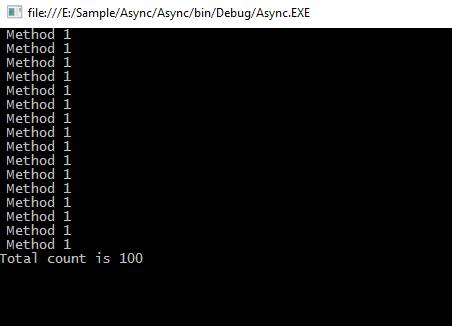
{

Console.WriteLine("Total count is " + count);

}

}

In the code given above, Method3 requires one parameter, which is the return type of Method1. Here, await keyword is playing a vital role for waiting of Method1 task completion.

**Output:**

Real-time example

There are some supporting API's from the .NET Framework 4.5 and the Windows runtime contains methods that support async programming.

We can use all of these in the real-time project with the help of async and await keyword for the faster execution of the task.

Some APIs that contain async methods are HttpClient, SyndicationClient, StorageFile, StreamWriter, StreamReader, XmlReader, MediaCapture, BitmapEncoder, BitmapDecoder etc.

In this example, we are going to read all the characters from a large text file asynchronously and get the total length of all the characters.

class Program

{

static void Main()

{

Task task = new Task(CallMethod);

task.Start();

task.Wait();

Console.ReadLine();

}

static async void CallMethod()

{

string filePath = "E:\\sampleFile.txt";

Task<int> task = ReadFile(filePath);

Console.WriteLine(" Other Work 1");

Console.WriteLine(" Other Work 2");

Console.WriteLine(" Other Work 3");

int length = await task;

Console.WriteLine(" Total length: " + length);

Console.WriteLine(" After work 1");

Console.WriteLine(" After work 2");

}

static async Task<int> ReadFile(string file)

{

int length = 0;

Console.WriteLine(" File reading is stating");

using (StreamReader reader = new StreamReader(file))

{

// Reads all characters from the current position to the end of the stream asynchronously

// and returns them as one string.

string s = await reader.ReadToEndAsync();

length = s.Length;

}

Console.WriteLine(" File reading is completed");

return length;

}

}

static async Task<int> ReadFile(string file)

{

int length = 0;

Console.WriteLine(" File reading is starting");

using (StreamReader reader = new StreamReader(file))

{

// Reads all characters from the current position to the end of the stream asynchronously

// and returns them as one string.

string s = await reader.ReadToEndAsync();

length = s.Length;

}

Console.WriteLine(" File reading is completed");

return length;

}

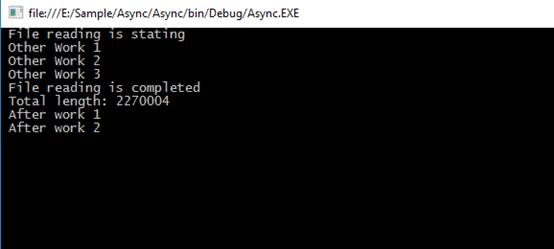
}

In the code given above, we are calling a ReadFile method to read the contents of a text file and get the length of the total characters present in the text file.

In our sampleText.txt, the file contains too many characters, so It will take a long time to read all the characters.

Here, we are using async programming to read all the contents from the file, so it will not wait to get a return value from this method and execute the other lines of code. Still it has to wait for the line of code given below because we are using await keywords, and we are going to use the return value.

**Output:**



**Best Practices in Async-Await Code in C#:**

1. Use and await asynchronous methods, if possible
2. Avoid async void method signatures:

If you need to call asynchronous code in your method, return Task or Task<T>.

1. Name your methods async (AddAsync):

If a method is asynchronous, name it accordingly.

1. Don’t await too much, return the task object instead:

int result = await SomeMethod();

public Task D4()

{

Console.Write("Enter the divisor: ");

var n = int.Parse(Console.ReadLine());

Console.WriteLine((24 / n).ToString());

return Task.FromResult(0);

}

**Exercise:**

1. Create a class that represents a student. Use ParallelFor to generate new students with random values and add it to a list;

In order to generate random string use:

public string RandomString(int size, bool lowerCase = false)

{

var builder = new StringBuilder(size);

// Unicode/ASCII Letters are divided into two blocks

// (Letters 65–90 / 97–122):

// The first group containing the uppercase letters and

// the second group containing the lowercase.

// char is a single Unicode character

char offset = lowerCase ? 'a' : 'A';

const int lettersOffset = 26; // A...Z or a..z: length=26

for (var i = 0; i < size; i++)

{

var @char = (char)\_random.Next(offset, offset + lettersOffset);

builder.Append(@char);

}

return lowerCase ? builder.ToString().ToLower() : builder.ToString();

}

Do the same with regular for.

Use watch in order to tell the difference in execution time.

var watch = new System.Diagnostics.Stopwatch();

watch.Start();

//Do Work

watch.Stop();

Console.WriteLine($"Execution Time: {watch.ElapsedMilliseconds} ms");

1. Create weather class with city, date and temperature properties

Create Weather forecast method in a static class called DummyWeatherProvider with this signature:

**public static async Task<Weather> GetWeather(string city)**

the method would be delayed using: await Task.Delay(10);

and then randomly generate temperature.

In main:

Call the method

Write to console the task status

Wait for the task result

Write to console the task status again

Write to console the weather

## **Differences Between Task And Thread**

Here are some differences between a task and a thread.

1. The Thread class is used for creating and manipulating a [thread](http://msdn.microsoft.com/en-us/library/windows/desktop/ms684841%28v=vs.85%29.aspx) in Windows. A [Task](http://msdn.microsoft.com/en-us/library/vstudio/system.threading.tasks.task) represents some asynchronous operation and is part of the [Task Parallel Library](http://msdn.microsoft.com/en-us/library/dd460717%28v=vs.110%29.aspx), a set of APIs for running tasks asynchronously and in parallel.
2. The task can return a result. There is no direct mechanism to return the result from a thread.
3. Task supports cancellation through the use of cancellation tokens. But Thread doesn't.
4. A task can have multiple processes happening at the same time. Threads can only have one task running at a time.
5. We can easily implement Asynchronous using ’async’ and ‘await’ keywords.
6. A new Thread()is not dealing with Thread pool thread, whereas Task does use thread pool thread.
7. A Task is a higher level concept than Thread.

**Task Cancellation:**

Asynchronous code is good for long running operation, and the provided task mechanism is plenty powerful. But sometimes we need to control the execution flow of this tasks. Why? We want observability into our tasks and not let some task hold the CPU and thread pool and hog down precious resources. Sometimes we want to have a difference between a task getting cancelled manually versus cancelling it due to an exception (or timeout?).

Why should we control the execution flow of the tasks?

* One reason is business logic or algorithm requirements, such as user request cancellation.
* Another reason comes from the idea that there should be no unobserved tasks. Such tasks may hold thread pool and CPU resources and therefore be dangerous.

**Cancel async tasks after a period of time:**

You can cancel an asynchronous operation after a period of time by using the [CancellationTokenSource.CancelAfter](https://learn.microsoft.com/en-us/dotnet/api/system.threading.cancellationtokensource.cancelafter) method if you don't want to wait for the operation to finish. This method schedules the cancellation of any associated tasks that aren't complete within the period of time that's designated by the CancelAfter expression.

CancellationTokenSource s\_cts = new CancellationTokenSource();

HttpClient s\_client = new HttpClient { MaxResponseContentBufferSize = 1\_000\_000 };

IEnumerable<string> s\_urlList = new string[]

{

"https://learn.microsoft.com", "https://learn.microsoft.com/aspnet/core",

"https://learn.microsoft.com/azure","https://learn.microsoft.com/azure/devops",

"https://learn.microsoft.com/dotnet" ,"https://learn.microsoft.com/dynamics365",

"https://learn.microsoft.com/education",

"https://learn.microsoft.com/enterprise-mobility-security",

"https://learn.microsoft.com/gaming","https://learn.microsoft.com/graph",

"https://learn.microsoft.com/microsoft-365",

"https://learn.microsoft.com/office","https://learn.microsoft.com/powershell",

"https://learn.microsoft.com/sql","https://learn.microsoft.com/surface",

"https://learn.microsoft.com/system-center",

"https://learn.microsoft.com/visualstudio",

"https://learn.microsoft.com/windows", "https://learn.microsoft.com/xamarin"};

try

{

s\_cts.CancelAfter(3500);

await SumPageSizesAsync();

}

catch (OperationCanceledException)

{

Console.WriteLine("\nTasks cancelled: timed out.\n");

}

finally

{

s\_cts.Dispose();

}

async Task SumPageSizesAsync()

{

var stopwatch = Stopwatch.StartNew();

int total = 0;

foreach (string url in s\_urlList)

{

int contentLength = await ProcessUrlAsync(url, s\_client, s\_cts.Token);

total += contentLength;

}

stopwatch.Stop();

Console.WriteLine($"\nTotal bytes returned: {total:#,#}");

Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");

}

static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)

{

HttpResponseMessage response = await client.GetAsync(url, token);

byte[] content = await response.Content.ReadAsByteArrayAsync(token);

Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

return content.Length;

}

**Actively Cancel Task:**

* Instantiate a CancellationTokenSource object
* Pass the token returned by the CancellationTokenSource.Token property to each task or thread that listens for cancellation
* Provide a mechanism for each task or thread to respond to this cancellation
* Call the CancellationTokenSource.Cancel method to provide a notification for cancellation