(T29)討論 Thread(執行緒)、Async、Await
CourseGUID: 29f1196a-1950-41a4-b9c1-dd13a9e92d92
==========
(T29)討論 Thread(執行緒)、Async、Await
==========

==========

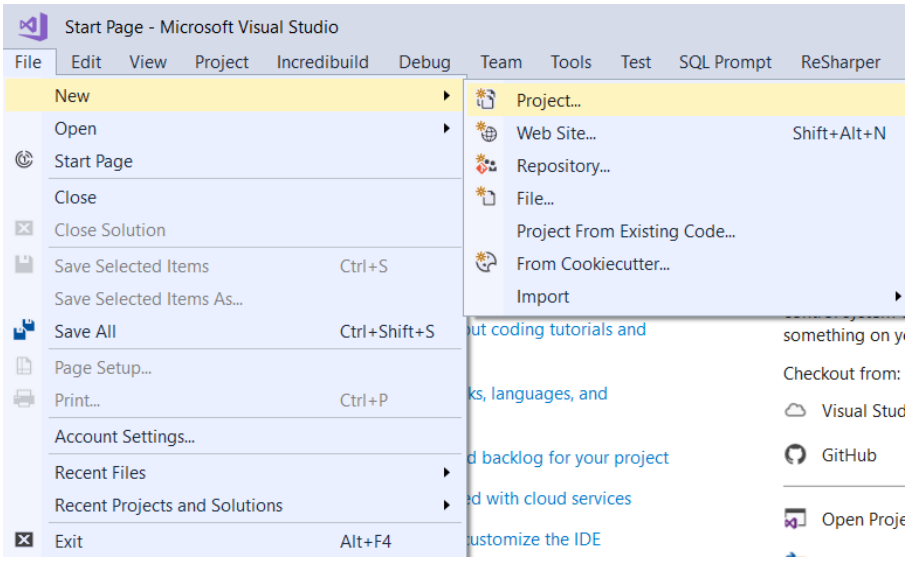
1. New Project

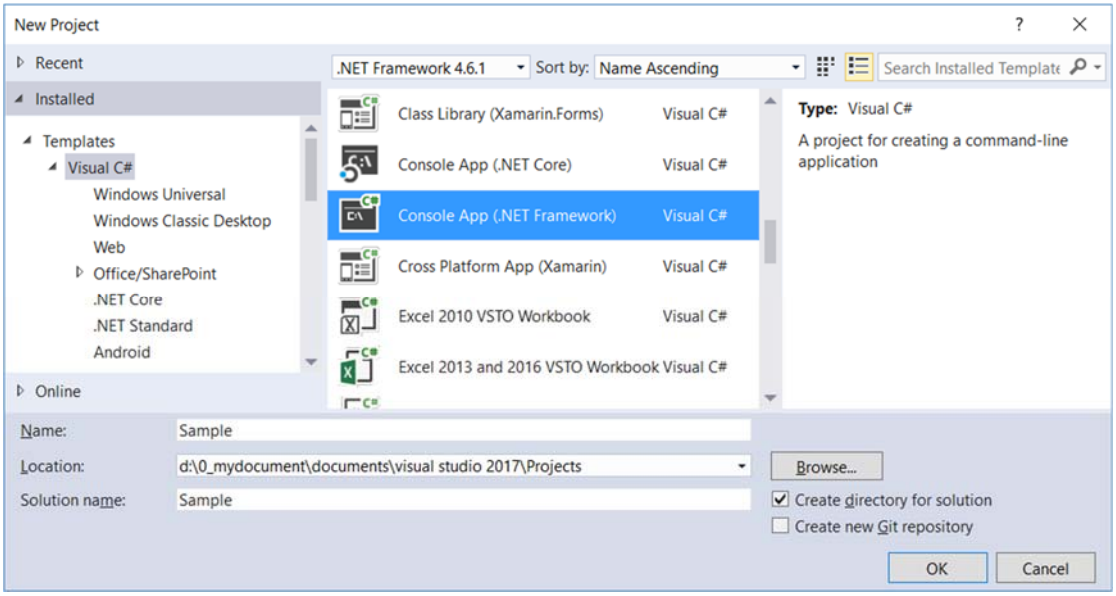
1.1. Create New Project : Sample

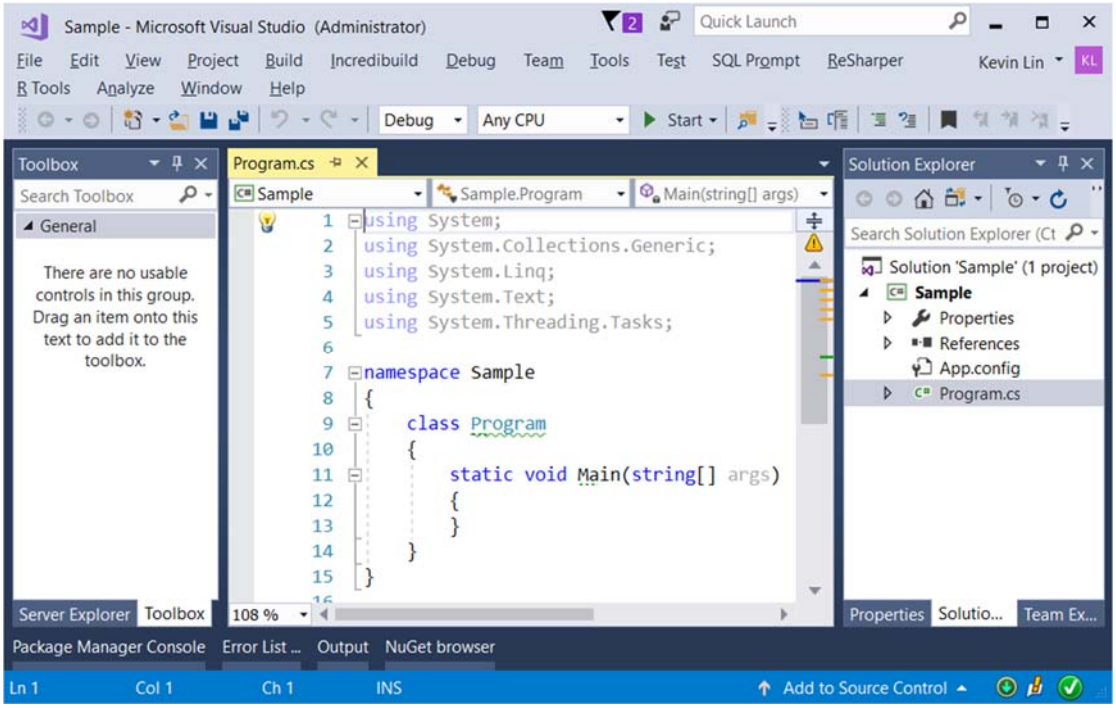
File --> New --> Project... -->

Visual C# --> **Console App (.Net Framework)** -->

Name: **Sample**







==========

2. Sample : Program.cs

```
using System;
usingSystem.Net.Http;
using System.Threading;
using System.Threading.Tasks;
namespace Sample
{
    class Program
    {
        static void Main(string[] args)
        {
            // 1. ========================================
```

```
            // scenario 1 - non-thread, non-task
            Console.WriteLine("1. scenario 1 - non-thread, non-task ================ ");
             Run();
            //// 2. ========================================
            //// scenario 2 - Thread
            //Console.WriteLine("2. scenario 2 - Thread ================ ");
            //RunThreads();
            //// 3. ========================================
            //// scenario 3 - Task<string>, async, await, CancellationTokenSource, IsCancellationRequested
            //Console.WriteLine("3. scenario 3 - Task<string>, async, await, CancellationTokenSource,
IsCancellationRequested ================ ");
            //try
            //{
            //    RunAsync().Wait();
            //}
            //catch (Exception e)
            //{
            //    Console.WriteLine(e);
            //}
            //// 4. ========================================
            //// scenario 4 - Task<string>, async, await, CancellationTokenSource
            ////Reference:
            //// https://stackoverflow.com/questions/10134310/how-to-cancel-a-task-in-await
            //// https://stackoverflow.com/questions/18738008/task-iscanceled-is-false-while-i-canceled
            //Console.WriteLine("4. scenario 4 - Task<string>, async, await, CancellationTokenSource
================ ");
            //try
            //{
            //    TryTask().Wait();
            //}
            //catch (Exception e)
            //{
            //    Console.WriteLine(e);
            //}
            Console.ReadLine();
        }

       // 1. ========================================
        // scenario 1 - non-thread, non-task
        /// <summary>
        /// scenario 1 - non-thread, non-task
        /// </summary>
        static void Run()
        {
            string googleSource = DownloadSource("http://google.com");
            string bingSource = DownloadSource("http://bing.com");
            Console.WriteLine($"Google: {googleSource.Length} Bing: {bingSource.Length}");
        }
        /// <summary>
        /// scenario 1 - non-thread, non-task
        /// scenario 2 - Thread
        /// </summary>
        /// <param name="url">The url</param>
        /// <returns>A string of source from the url.</returns>
```

```
static string DownloadSource(string url)
{
    using (HttpClient httpClient = new HttpClient())
    {
        Console.WriteLine($"Getting source for {url}");
        Task<string> theTask = httpClient.GetStringAsync(url);
        string source = theTask.Result;
        Console.WriteLine($"Finished getting source for {url}");
        return source;
    }
}
// 2. ========================================
// scenario 2 - Thread
/// <summary>
/// scenario 2 - Thread
/// </summary>
static void RunThreads()
{
    Thread googleThread = new Thread(() => DownloadSource("http://google.com"));
    Thread bingThread = new Thread(() => DownloadSource("http://bing.com"));
    Thread yahooThread = new Thread(() => DownloadSource("http://yahoo.com"));
    googleThread.Start();
    bingThread.Start();
    googleThread.Join();
    yahooThread.Start();
    bingThread.Join();
}
// 3. ========================================
// scenario 3 - Task<string>, async, await, CancellationTokenSource, IsCancellationRequested
private void Foo()
{
    // no return value.
}
private Task FooAsync1()
{
    // need to return a successfully completed task.
    return Task.FromResult(0);
}
private async Task FooAsync2()
{
    // if you have async, then you don't need to    return Task.FromResult(0);
    // it is just like a void method.
    // then you may use await in the method.
}
/// <summary>
/// scenario 3 - async, await
/// </summary>
/// <returns>a async task</returns>
static async Task RunAsync()
{
    CancellationTokenSource cts = new CancellationTokenSource(100);
    CancellationToken ct = cts.Token;
    //Task<string> googleTask = DownloadSourceAsync("http://google.com", ct);
    //Task<string> bingTask = DownloadSourceAsync("http://bing.com", ct);
    string googleSource = await DownloadSourceAsync("http://google.com", ct);
    string bingSource = await DownloadSourceAsync("http://bing.com", ct);
    Console.WriteLine($"Google: {googleSource.Length} Bing: {bingSource.Length}");
```

```
        }
        /// <summary>
        /// scenario 3 - async, await
        /// </summary>
        /// <param name="url">The url</param>
        /// <param name="ct">The cancelation token.</param>
        /// <returns>A string of source from the url</returns>
        static async Task<string> DownloadSourceAsync(string url, CancellationToken ct)
        {
            using (HttpClient httpClient = new HttpClient())
            {
                if (ct.IsCancellationRequested)
                {
                    // manually stop Task if receive the CancellationToken.
                    // the inter i parameter inside of the CancellationToken means
                    // after i million-second, stop the Task
                    Console.WriteLine($"DownloadSourceAsync for {url} is manually stop because of
CancellationToken.");
                    ct.ThrowIfCancellationRequested();
                }
                Console.WriteLine($"Getting source for {url}");
                string source = await httpClient.GetStringAsync(url);
                Console.WriteLine($"Finished getting source for {url}");
                return source;
            }
        }

       // 4. ========================================
        // scenario 4 - Task<string>, async, await, CancellationTokenSource
        //Reference:
        // https://stackoverflow.com/questions/10134310/how-to-cancel-a-task-in-await
        // https://stackoverflow.com/questions/18738008/task-iscanceled-is-false-while-i-canceled
        /// <summary>
        /// a task has no return, just like void.
        /// </summary>
        /// <returns>a async Task.</returns>
        static async Task TryTask()
        {
            CancellationTokenSource source = new CancellationTokenSource();
            // a CancellationTokenSource will make the task cancel after 1 second.
            source.CancelAfter(TimeSpan.FromSeconds(1));
            // Create a task in order to run a very slow function
            //which take the CancellationTokenSource
            //that will make the task cancel after 1 second.
            Task<int> task = Task.Run(() => slowFunc(1, 2, source.Token), source.Token);
            // run the task, A canceled task will raise an exception when awaited
            await task;
        }
        // you do not have to know what a and b means.
        // basically, it will make this function do something very slow.
        // in the middle of do something very slow,
        // CancellationToken will be throw when request. E.g.
cancellationToken.ThrowIfCancellationRequested();
        static int slowFunc(int a, int b, CancellationToken cancellationToken)
        {
            // do something very slow
```

```
            string someString = string.Empty;
            for (int i = 0; i < 200000; i++)
            {
                someString += "a";
                if (i % 1000 == 0)
                    cancellationToken.ThrowIfCancellationRequested();
            }
            return a + b;
        }
    }
}

/*
================================================================
1.
scenario 1 - non-thread, non-task
//static void Main(string[] args)
//{
//    Run();
//}
...
//static void Run()
//{
//    string googleSource = DownloadSource("http://google.com");
//    string bingSource = DownloadSource("http://bing.com");
//    Console.WriteLine($"Google: {googleSource.Length} Bing: {bingSource.Length}");
//}
...
//static string DownloadSource(string url)
//{
//    using (HttpClient httpClient = new HttpClient())
//    {
//        Console.WriteLine($"Getting source for {url}");
//        Task<string> theTask = httpClient.GetStringAsync(url);
//        string source = theTask.Result;
//        Console.WriteLine($"Finished getting source for {url}");
//        return source;
//    }
//}
1.1.
it will return
//Getting source for http://google.com
//Finished getting source for http://google.com
//Getting source for http://bing.com
//Finished getting source for http://bing.com
//Google: 45157 Bing: 106518
1.2.
//static string DownloadSource(string url)
//{
//    using (HttpClient httpClient = new HttpClient())
//    {
//        Console.WriteLine($"Getting source for {url}");
//        Task<string> theTask = httpClient.GetStringAsync(url);
//        string source = theTask.Result;
//        Console.WriteLine($"Finished getting source for {url}");
//        return source;
//    }
//}
1.2.1.
// using (HttpClient httpClient = new HttpClient()){...}
means run ... and then disposed HttpClient httpClient
HttpClient has .disposed(bool) method.
We do not need to know why we need to disposed HttpClient httpClient.
However, we better to dispose anything after use, which has disposed method.
1.2.2.
// Task<string> theTask = httpClient.GetStringAsync(url);
```

```
// string source = theTask.Result;
or
// string source = httpClient.GetStringAsync(url).Result;
-->
// Task<string> theTask = httpClient.GetStringAsync(url);  need await.
but
We do not make   //static string DownloadSource(string url)  become a async method.
Therefore, we can use   .Result
which means wait until this Task finished and get the result.
In ours case, get the string value of the result.
1.3.
//static void Run()
//{
//    string googleSource = DownloadSource("http://google.com");
//    string bingSource = DownloadSource("http://bing.com");
//    Console.WriteLine($"Google: {googleSource.Length} Bing: {bingSource.Length}");
//}
get the googleSource and bingSource and Console.WriteLine

============================================================
2.
scenario 2 - Thread
//static void Main(string[] args)
//{
//    RunThreads();
//}
...
//static void RunThreads()
//{
//    Thread googleThread = new Thread(() => DownloadSource("http://google.com"));
//    Thread bingThread = new Thread(() => DownloadSource("http://bing.com"));
//    googleThread.Start();
//    bingThread.Start();
//    googleThread.Join();
//    bingThread.Join();
//    Console.ReadLine();
//}
...
//static string DownloadSource(string url)
//{
//    using (HttpClient httpClient = new HttpClient())
//    {
//        Console.WriteLine($"Getting source for {url}");
//        Task<string> theTask = httpClient.GetStringAsync(url);
//        string source = theTask.Result;
//        Console.WriteLine($"Finished getting source for {url}");
//        return source;
//    }
//}
2.1.
It will return
//Getting source for http://google.com
//Getting source for http://bing.com
//Finished getting source for http://bing.com
//Finished getting source for http://google.com
OR
//Getting source for http://google.com
//Getting source for http://bing.com
//Finished getting source for http://google.com
//Finished getting source for http://bing.com
OR
//Getting source for http://bing.com
//Getting source for http://google.com
//Finished getting source for http://google.com
//Finished getting source for http://bing.com
OR
//Getting source for http://bing.com
```

```
//Getting source for http://google.com
//Finished getting source for http://bing.com
//Finished getting source for http://google.com
2.2.
//static string DownloadSource(string url)
//{
//    using (HttpClient httpClient = new HttpClient())
//    {
//        Console.WriteLine($"Getting source for {url}");
//        Task<string> theTask = httpClient.GetStringAsync(url);
//        string source = theTask.Result;
//        Console.WriteLine($"Finished getting source for {url}");
//        return source;
//    }
//}
2.2.1.
// using (HttpClient httpClient = new HttpClient()){...}
means run ... and then disposed HttpClient httpClient
HttpClient has .disposed(bool) method.
We do not need to know why we need to disposed HttpClient httpClient.
However, we better to dispose anything after use, which has disposed method.
2.2.2.
// Task<string> theTask = httpClient.GetStringAsync(url);
// string source = theTask.Result;
or
// string source = httpClient.GetStringAsync(url).Result;
-->
// Task<string> theTask = httpClient.GetStringAsync(url);  need await.
but
We do not make   //static string DownloadSource(string url)  become a async method.
Therefore, we can use   .Result
which means wait until this Task finished and get the result.
In ours case, get the string value of the result.
2.3.
//static void RunThreads()
//{
//    Thread googleThread = new Thread(() => DownloadSource("http://google.com"));
//    Thread bingThread = new Thread(() => DownloadSource("http://bing.com"));
//    googleThread.Start();
//    bingThread.Start();
//    googleThread.Join();
//    bingThread.Join();
//}
2.3.1.
// Thread googleThread = new Thread(() => DownloadSource("http://google.com"));
create a Thread googleThread which will do   () => DownloadSource("http://google.com")
The parameter of   new Thread(...), ... means a method which you are going to run.
2.3.2.
// googleThread.Start();
just start the googleThread, and do not wait it finish and run next thread/next line.
2.3.3.
// googleThread.Join();
googleThread.join() means you must wait  googleThread finish before you run next line.

============================================================
3.
scenario 3 - Task<string>, async, await, CancellationTokenSource, IsCancellationRequested
3.1.
//private void Foo()
//{
//    // no return value.
//}
//...
//private Task FooAsync1()
//{
//    // need to return a successfully completed task.
//    return Task.FromResult(0);
```

```
//}
//...
//private async Task FooAsync2()
//{
//    // if you have async, then you don't need to    return Task.FromResult(0);
//    // it is just like a void method.
//    // then you may use await in the method.
//}
3.1.1.
// System.Threading.Tasks.Task.FromResult<TResult>(TResult)
Creates a System.Threading.Tasks.Task`1 that's completed successfully with the specified result.
Parameters:
result: The result to store into the completed task.
Returns:
The successfully completed task.
-----------------------------------------------------------------
3.2.
//static void Main(string[] args)
//{
//     try
//     {
//            RunAsync().Wait();
//     }
//     catch (Exception e)
//     {
//            Console.WriteLine(e);
//     }
...
//     Console.ReadLine();
//}
//...
//static async Task RunAsync()
//{
//     CancellationTokenSource cts = new CancellationTokenSource(100);
//     CancellationToken ct = cts.Token;
//     //Task<string> googleTask = DownloadSourceAsync("http://google.com", ct);
//     //Task<string> bingTask = DownloadSourceAsync("http://bing.com", ct);
//     string googleSource = await DownloadSourceAsync("http://google.com", ct);
//     string bingSource = await DownloadSourceAsync("http://bing.com", ct);
//     Console.WriteLine($"Google: {googleSource.Length} Bing: {bingSource.Length}");
//}
...
//static async Task<string> DownloadSourceAsync(string url, CancellationToken ct)
//{
//     using (HttpClient httpClient = new HttpClient())
//     {
//            if (ct.IsCancellationRequested)
//            {
//                   // manually stop Task if receive the CancellationToken.
//                   // the inter i parameter inside of the CancellationToken means
//                   // after i million-second, stop the Task
//                   Console.WriteLine($"DownloadSourceAsync for {url} is manually stop because of
CancellationToken.");
//                   ct.ThrowIfCancellationRequested();
//            }
//            Console.WriteLine($"Getting source for {url}");
//            string source = await httpClient.GetStringAsync(url);
//            Console.WriteLine($"Finished getting source for {url}");
//            return source;
//     }
//}
---------------------------------------
3.2.1.
It will return
//Getting source for http://google.com
//Finished getting source for http://google.com
//DownloadSourceAsync for http://bing.com is manually stop because of CancellationToken.
```

```
//System.AggregateException: One or more errors occurred. --->
System.Threading.Tasks.TaskCanceledException: A task was canceled.
//   --- End of inner exception stack trace ---
//   at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean includeTaskCanceledExceptions)
//   at System.Threading.Tasks.Task.Wait(Int32 millisecondsTimeout, CancellationToken cancellationToken)
//   at System.Threading.Tasks.Task.Wait()
//   at AsyncAwaitTest.Program.Main(String[] args) in
D:\0_MyDocument\Desktop\AsyncAwaitTest\AsyncAwaitTest\AsyncAwaitTest\Program.cs:line 21
//---> (Inner Exception #0) System.Threading.Tasks.TaskCanceledException: A task was canceled.<---
That means When
//     CancellationTokenSource cts = new CancellationTokenSource(100);
it means both Task with this CancellationToken will automatically stop after 100 million seconds.
The 100 million seconds is not quick enough to stop the first Task, so it will return
//Getting source for http://google.com
//Finished getting source for http://google.com
However the 100 million seconds is enough to stop the second Task.
Therefore,
//DownloadSourceAsync for http://bing.com is manually stop because of CancellationToken.
---------------------------------------
3.2.2.
//static async Task<string> DownloadSourceAsync(string url, CancellationToken ct)
//{
//     using (HttpClient httpClient = new HttpClient())
//     {
//            if (ct.IsCancellationRequested)
//            {
//                   // manually stop Task if receive the CancellationToken.
//                   // the inter i parameter inside of the CancellationToken means
//                   // after i million-second, stop the Task
//                   Console.WriteLine($"DownloadSourceAsync for {url} is manually stop because of
CancellationToken.");
//                   ct.ThrowIfCancellationRequested();
//            }
//            Console.WriteLine($"Getting source for {url}");
//            string source = await httpClient.GetStringAsync(url);
//            Console.WriteLine($"Finished getting source for {url}");
//            return source;
//     }
//}
3.2.2.1.
//static async Task<string> DownloadSourceAsync(string url, CancellationToken ct)
3.2.2.1.1.
// async
means you may use await in the method.
await means you have to wait until this Task finished in order to run next Task or next line.
3.2.2.1.2.
// Task<string>
means this is a Task which will return a Task of string after finished.
3.2.2.1.3.
// static async Task<string> DownloadSourceAsync(string url, CancellationToken ct){
...
//if (ct.IsCancellationRequested)
//{
//     // manually stop Task if receive the CancellationToken.
//     // the inter i parameter inside of the CancellationToken means
//     // after i million-second, stop the Task
//     Console.WriteLine($"DownloadSourceAsync for {url} is manually stop because of
CancellationToken.");
//     ct.ThrowIfCancellationRequested();
//}
...
//CancellationTokenSource cts = new CancellationTokenSource(100);
//CancellationToken ct = cts.Token;
//Task<string> googleTask = DownloadSourceAsync("http://google.com", ct);
This means the task with the Cancellation token will automatic stop automatically 100 million seconds.
3.2.2.2.
//    using (HttpClient httpClient = new HttpClient())
```

```
//    {
//          ...
//    }
means run ... and then disposed HttpClient httpClient
HttpClient has .disposed(bool) method.
We do not need to know why we need to disposed HttpClient httpClient.
However, we better to dispose anything after use, which has disposed method.
3.2.2.3.
// string source = await httpClient.GetStringAsync(url);
// return source;
wait GetStringAsync Task finished and then return a string as source.

---------------------------------------------------------------
3.3.
//static async Task RunAsync()
//{
//     CancellationTokenSource cts = new CancellationTokenSource(100);
//     CancellationToken ct = cts.Token;
//     //Task<string> googleTask = DownloadSourceAsync("http://google.com", ct);
//     //Task<string> bingTask = DownloadSourceAsync("http://bing.com", ct);
//     string googleSource = await DownloadSourceAsync("http://google.com", ct);
//     string bingSource = await DownloadSourceAsync("http://bing.com", ct);
//     Console.WriteLine($"Google: {googleSource.Length} Bing: {bingSource.Length}");
//}
3.3.1.
// CancellationTokenSource cts = new CancellationTokenSource(100);
// CancellationToken ct = cts.Token;
...
//// Task<string> googleTask = DownloadSourceAsync("http://google.com", ct);
//string googleSource = DownloadSourceAsync("http://google.com", ct);
Task<string> googleTask
means a Task which will return string after finished.
new CancellationTokenSource(100)
means the task with the Cancellation token will automatic stop automatically 100 million seconds.
3.3.2.
//// Task<string> googleTask = DownloadSourceAsync("http://google.com", ct);
it means download the source code from http://google.com
//string googleSource = DownloadSourceAsync("http://google.com", ct);
it means wait until  Task<string> googleTask finished in order to run next line/next task
---------------------------------------------------------------
3.4.
// RunAsync().Wait();
RunAsync() is a async Task
Therefore, we need to .Wait(), to wait   RunAsync() finished in order to run next line/next task.
---------------------------------------------------------------
3.5.
When
// CancellationTokenSource cts = new CancellationTokenSource(10);
it will return
//DownloadSourceAsync for http://google.com is manually stop because of CancellationToken.
//System.AggregateException: One or more errors occurred. --->
System.Threading.Tasks.TaskCanceledException: A task was canceled.
//   --- End of inner exception stack trace ---
//   at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean includeTaskCanceledExceptions)
//   at System.Threading.Tasks.Task.Wait(Int32 millisecondsTimeout, CancellationToken cancellationToken)
//   at System.Threading.Tasks.Task.Wait()
//   at AsyncAwaitTest.Program.Main(String[] args) in
D:\0_MyDocument\Desktop\AsyncAwaitTest\AsyncAwaitTest\AsyncAwaitTest\Program.cs:line 21
//---> (Inner Exception #0) System.Threading.Tasks.TaskCanceledException: A task was canceled.<---
That means 10 million seconds is not enough to run the first Task.
---------------------------------------------------------------
3.6.
When
// CancellationTokenSource cts = new CancellationTokenSource(1000);
it will return
//Getting source for http://google.com
//Finished getting source for http://google.com
```

```
//Getting source for http://bing.com
//Finished getting source for http://bing.com
//Google: 47143 Bing: 141750
That means When
//     CancellationTokenSource cts = new CancellationTokenSource(1000);
it means both Task with this CancellationToken will automatically stop after 1000 million seconds.
The 1000 million seconds is enough to stop both Tasks.

------------------------------------------------------------
3.7.
Compare Thread and async Task
5.1.
async Task
// CancellationTokenSource cts = new CancellationTokenSource(100);
// CancellationToken ct = cts.Token;
...
//// Task<string> googleTask = DownloadSourceAsync("http://google.com", ct);
//string googleSource = DownloadSourceAsync("http://google.com", ct);
Task<string> googleTask
means a Task which will return string after finished.
new CancellationTokenSource(100)
means the task with the Cancellation token will automatic stop automatically 100 million seconds.
5.2.
// Thread googleThread = new Thread(() => DownloadSource("http://google.com"));
Thread and task are very similar.
We can run several Task<string> in the same time or we can run several Thread in the same time.
The different is that Thread has no return type.
Thread means keep ruining and do not return anything after finished.
Task<string> means keep ruining and return a string value after finished.
We normally can run Task<string> on top of Thread.
For example,
Thread a has Task a1, Task a2 ...etc.
Thread b has Task b1, Task b2 ...etc.
// 4. ========================================
scenario 4 - Task<string>, async, await, CancellationTokenSource
Reference:
https://stackoverflow.com/questions/10134310/how-to-cancel-a-task-in-await
https://stackoverflow.com/questions/18738008/task-iscanceled-is-false-while-i-canceled
//try
//{
//    TryTask().Wait();
//}
//catch (Exception e)
//{
//    Console.WriteLine(e);
//}
...
//// a task has no return, just like void.
//static async Task TryTask()
//{
//    CancellationTokenSource source = new CancellationTokenSource();
//    // a CancellationTokenSource will make the task cancel after 1 second.
//    source.CancelAfter(TimeSpan.FromSeconds(1));
//    // Create a task in order to run a very slow function which take the CancellationTokenSource that
will make the task cancel after 1 second.
//    Task<int> task = Task.Run(() => slowFunc(1, 2, source.Token), source.Token);
//    // run the task, A canceled task will raise an exception when awaited
//    await task;
//}
...
//// you do not have to know what a and b means.
//// basically, it will make this function do something very slow.
//// in the middle of do something very slow,
//// CancellationToken will be throw when request. E.g. cancellationToken.ThrowIfCancellationRequested();
//static int slowFunc(int a, int b, CancellationToken cancellationToken)
//{
//    // do something very slow
```

```
//    string someString = string.Empty;
//    for (int i = 0; i < 200000; i++)
//    {
//        someString += "a";
//        if (i % 1000 == 0)
//            cancellationToken.ThrowIfCancellationRequested();
//    }
//    return a + b;
//}
it will return
//System.AggregateException: One or more errors occurred. --->
System.Threading.Tasks.TaskCanceledException: A task was canceled.
//   --- End of inner exception stack trace ---
//   at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean includeTaskCanceledExceptions)
//   at System.Threading.Tasks.Task.Wait(Int32 millisecondsTimeout, CancellationToken cancellationToken)
//   at System.Threading.Tasks.Task.Wait()
//   at AsyncAwaitTest.Program.Main(String[] args) in
D:\0_MyDocument\Desktop\AsyncAwaitTest\AsyncAwaitTest\AsyncAwaitTest\Program.cs:line 31
//---> (Inner Exception #0) System.Threading.Tasks.TaskCanceledException: A task was canceled.<---
*/
```