# C# Await Tutorial

Good await tutorial

<https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/concepts/async/>

## Control flow in asynchronous programming

public async Task DoSomethingAsync() {

    Console.WriteLine("Start");

    await Task.Delay(1000);

    Console.WriteLine("End");

}

public async Task Main() {

    Console.WriteLine("Before");

    await DoSomethingAsync();

    Console.WriteLine("After");

}

Result:

Before

Start

End

After

7 ways to start a task in C#

<https://www.spaceclick.com/blog/7-ways-to-create-tasks-in-dot-net-c-sharp/>

## Create a task without starting it right away

Using Lambda with a method:

Task task = new Task( () => MyMethod() );

task.Start();

## Create a task and start it right away

Using Run (.NET 4.5)

public async Task DoStuff()

{

await Task.Run(() => MyMethod());

}

If you don't need to wait for the task in the DoStuff()

public void DoStuff()

{

   Task.Run(() => MyMethod());

}

## Get return value from Task.Run

string urlContents = await Task.Run(() => RunLongTask(i.ToString(CultureInfo.InvariantCulture)));

The string result is stored in the task that represents the completion of the RunLongTask method. The await operator retrieves the result from the task. The assignment statement assigns the retrieved result to urlContents.

private async Task<string> ABCAsync(string someParam)

{

    //adding .ConfigureAwait(false) may NOT be what you want but google it.

    return await Task.Run(() => SomeObj.SomeMethodAsync(someParam)).ConfigureAwait(false);

}

ABCAsync returns a [Task(Of TResult)](https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1) where TResult is a string.

The task represents the ongoing process for the call to ABCAsync, with a commitment to produce an actual string value when the work is complete.

You can work on other independent works after starting a task, and await it to finish when you need its return value to continue the program.

Task<string> LongTask = new Task( () => MyMethod() );

LongTask.Start();

//You can continue with other work that doesn't depend on the final result from LongTask.

DoIndependentWork();

//When you have run out of work that you can do without a result from LongTask.

//Await it to finish

string urlcontent = await LongTask

# Wait for multiple tasks : [WaitAll vs WhenAll](https://stackoverflow.com/questions/6123406/waitall-vs-whenall)

Task.WaitAll blocks the current thread until everything has completed.

Task.WhenAll returns a *task* which represents the action of waiting until everything has completed.

That means that from an async method, you can use:

await Task.WhenAll(tasks);

... which means your method will continue when everything's completed, but you won't tie up a thread to just hang around until that time.

## Task.WaitAll example:

public async Task UnresultTaskMethod()

{

   var task1 = AsyncMethod1();

   var task2 = AsyncMethod2();

   // This is not awaitable, you're blocking the current thread

   Task.WaitAll(task1, task2);

   // Here you are sure both task1 and task2 are completed

}

## Task.WhenAll example:

private async Task ABCAsync(string someParam)

{

    //Show the marquee to indicate it's waiting for the process to finish

    var box = new Form2();

    Form2.Show();

    //Do the process

    var BroadCategory = new List<string>();

    BroadCategory.Add("123");

    BroadCategory.Add("1234");

    // ///////////////////Replace parallel for in kannsihhanntei btn event with this structure,

    // //only Replace the Heavy process part would do the work////////////////////////

    var taskArray = new Task[BroadCategory.Count];

    for (int i = 0, i <= BroadCategory.Count - 1;  i++)

    {

        taskArray[i] = Task.Run(() =>

        {

                // Heavy process

            Console.WriteLine("in tasks thread ID:" + Environment.CurrentManagedThreadId);

            System.Threading.Thread.Sleep(2000);

        });

    }

    // Asynchronously wait for all tasks to finish so that the caller function can continue its work.

    await(Task.WhenAll(taskArray));

    //Close the marquee window to indicate the process is done.

    Form2.Close();

}

## What I observed:

1. The code after await will be run by another thread. As the result, the main thread will not be interrupted when the task is done.
2. Main thread doesn’t return immediately when encountering await. It runs the task until it meets another await If it is “await AsyncFunction”.

 static void Main(string[] args)

        {

            Console.WriteLine($"In Main Before KDAsync,Thread ID:{System.Environment.CurrentManagedThreadId}");

            KDAsync();

            Console.WriteLine($"In Main Return from KD Await All Done KDAsync,Thread ID:{System.Environment.CurrentManagedThreadId}");

            Thread.Sleep(10000);

            Console.WriteLine($"In Main After All Done KDAsync,Thread ID:{System.Environment.CurrentManagedThreadId}");

            Console.Read();

        }

public static async Task KI()

        {

            await Task.Delay(3000);

            Thread.Sleep(10000);

            Console.WriteLine($"In KI Before,Thread ID:{System.Environment.CurrentManagedThreadId}");

            Console.WriteLine($"In KI After,Thread ID:{System.Environment.CurrentManagedThreadId}");

        }

public static async void KDAsync()

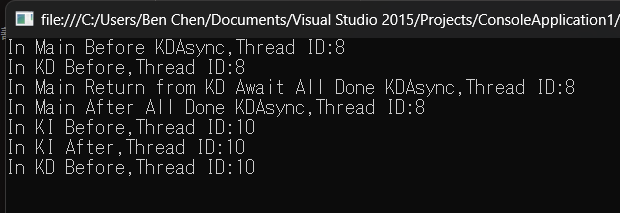
        {

            Console.WriteLine($"In KD Before,Thread ID:{System.Environment.CurrentManagedThreadId}");

            await KI();

            Console.WriteLine($"In KD Before,Thread ID:{System.Environment.CurrentManagedThreadId}");

        }



 static void Main(string[] args)

        {

            Console.WriteLine($"In Main Before KDAsync,Thread ID:{System.Environment.CurrentManagedThreadId}");

            KDAsync();

            Console.WriteLine($"In Main Return from KD Await All Done KDAsync,Thread ID:{System.Environment.CurrentManagedThreadId}");

            Thread.Sleep(10000);

            Console.WriteLine($"In Main After All Done KDAsync,Thread ID:{System.Environment.CurrentManagedThreadId}");

            Console.Read();

        }

public static async Task KI()

        {

            Thread.Sleep(10000);

            Console.WriteLine($"In KI Before,Thread ID:{System.Environment.CurrentManagedThreadId}");

            await Task.Delay(3000);

            Console.WriteLine($"In KI After,Thread ID:{System.Environment.CurrentManagedThreadId}");

        }

public static async void KDAsync()

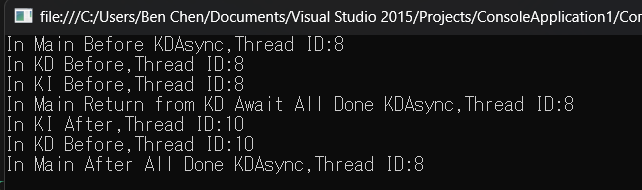
        {

            Console.WriteLine($"In KD Before,Thread ID:{System.Environment.CurrentManagedThreadId}");

            await KI();

            Console.WriteLine($"In KD Before,Thread ID:{System.Environment.CurrentManagedThreadId}");

        }



1. If it is “await Task.Run( )”, the main thread will return immediately.

static void Main(string[] args)

        {

            Console.WriteLine($"In Main Before KDAsync,Thread ID:{System.Environment.CurrentManagedThreadId}");

            KDAsync();

            Console.WriteLine($"In Main Return from KD Await All Done KDAsync,Thread ID:{System.Environment.CurrentManagedThreadId}");

            Thread.Sleep(10000);

            Console.WriteLine($"In Main After All Done KDAsync,Thread ID:{System.Environment.CurrentManagedThreadId}");

            Console.Read();

        }

public static void KI()

        {

            Console.WriteLine($"In KI Before,Thread ID:{System.Environment.CurrentManagedThreadId}");

            Thread.Sleep(10000);

            Console.WriteLine($"In KI After,Thread ID:{System.Environment.CurrentManagedThreadId}");

        }

public static async void KDAsync()

        {

            Console.WriteLine($"In KD Before,Thread ID:{System.Environment.CurrentManagedThreadId}");

            await Task.Run(() => KI());

            Console.WriteLine($"In KD Before,Thread ID:{System.Environment.CurrentManagedThreadId}");

        }

