**Task-based Asynchronous Programming in C#**

**Task-based Asynchronous Programming in C#**

In this article, I am going to discuss **Task-based Asynchronous Programming in C#** with some examples. Please read our [**multithreading articles**](https://dotnettutorials.net/lesson/multithreading-in-csharp/) before proceeding to this article. In C#.NET, the task is basically used to implement Asynchronous Programming i.e. executing operations asynchronously and it was introduced with .NET Framework 4.0.

Before understanding theory i.e. what is **Task**and **what are the benefits of using Task**, let us first discuss **how to create and use Task** in C#.

**Working with Task in C#:**

The Task-related classes belong to **System.Threading.Tasks** namespace. So the first and foremost step for you is to import the **System.Threading.Tasks** namespace in your program. Once you import the **System.Threading.Tasks** namespace, then you can create as well as access the task objects by using Task class.

**Note:** In general, the Task class will always represent a single operation and that operation will be executed asynchronously on a thread pool thread rather than synchronously on the main thread of the application. If this is not clear at the moment then don’t worry we will discuss this in practice.

**Example: Using the Task class and Start method in C#**

In the below example, we are creating the task object by using the Task class and then start executing it by calling the Start method on the Task object.

**using** *System;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *TaskBasedAsynchronousProgramming*

**{**

**class** Program

**{**

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

**{**

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

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}"**)**;

**}**

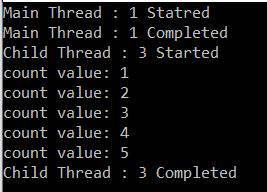
Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

**}**

**}**

**}**

In the above example, we created the task object i.e. task1 using the Task class and then call the Start method to start the task execution. Here, task object task1 will create a new child thread to execute the defined functionality asynchronously on a thread pool thread. So, when you run the above application, you will get the following output.



As you can see in the above output, two threads are used to execute the application code. The main thread and the child thread. And you can observe both threads are running asynchronously.

**Example: Creating a task object using Factory Property**

In the following example, we are creating the task object using the Factory property which will start automatically.

**using** *System;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *TaskBasedAsynchronousProgramming*

**{**

**class** Program

**{**

**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**()**;

**}**

**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"**)**;

**}**

**}**

**}**

It will give you the same output as the previous example. The only difference between the previous example and this example is here we creating and running the thread using a single statement.

**Example: Creating a Task object using the Run method**

In the following example, we are creating a task by using the Run method of the Task class.

**using** *System;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *TaskBasedAsynchronousProgramming*

**{**

**class** Program

**{**

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

**{**

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

Task task1 = Task.Run**(()** =**>** **{** PrintCounter**()**; **})**;

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"**)**;

**}**

**}**

**}**

So, we have discussed three different ways to create and start a task in C#. From a performance point of view, the Task.Run or Task.Factory.StartNew methods are preferable to create and schedule the tasks. But, if you want to the task creation and scheduling separately, then you need to create the task separately by using Task class and then call the Start method to schedule the task execution for a later time.

**Task using Wait in C#:**

As we already discussed, the tasks will run asynchronously on the thread pool thread and the thread will start the task execution asynchronously along with the main thread of the application. So far the examples we discussed in this article, the child thread will continue its execution until it finishes its task even after the completion of the main thread execution of the application.

If you want to make the main thread execution wait until all child tasks are completed, then you need to use the Wait method of the Task class. The Wait method of the Task class will block the execution of other threads until the assigned task has completed its execution.

In the following example, we are calling the Wait() method on the task1 object to make the program execution wait until task1 completes its execution.

**using** *System;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *TaskBasedAsynchronousProgramming*

**{**

**class** Program

**{**

**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**()**;

**}**

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

**{**

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

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

**{**

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

**}**

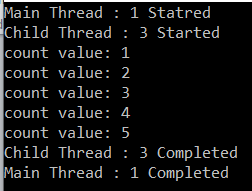
Console.WriteLine**(**$"Child Thread : {Thread.CurrentThread.ManagedThreadId} Completed"**)**;

**}**

**}**

**}**

As you can see in the above code, we are calling the Wait() method on the task object i.e. task1. So, the main thread execution will wait until the task1 object completes its execution. Now run the application and see the output as shown in the below image.



So as of now, we have discussed how to work with threads using different approaches. Now let us discuss what is Task and why should we use Task?

**What Is A Task In C#?**

A task in C# is used to implement Task-based Asynchronous Programming and was introduced with the .NET Framework 4. The Task object is typically executed asynchronously on a thread pool thread rather than synchronously on the main thread of the application.

A task scheduler is responsible for starting the Task and also responsible for managing it. By default, the Task scheduler uses threads from the thread pool to execute the Task.

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

A**Thread pool in C#** is a collection of **threads** that can be used to perform a number of tasks in the background. Once a thread completes its task, then again it is sent to the thread pool, so that it can be reused. This reusability of threads avoids an application to create a number of threads which ultimately uses less memory consumption.

**Why do we need to use a Task In C#?**

Tasks in C# basically used to make your application more responsive. If the thread that manages the user interface offloads the works to other threads from the thread pool, then it can keep processing user events which will ensure that the application can still be used.

That’s it for today. In the next article, I am going to discuss [**how to return a value from a task in C#**](https://dotnettutorials.net/lesson/return-a-value-from-a-task-in-csharp/) with some examples. Here, in this article, I try to explain Task-based Asynchronous Programming in C# using the Task class. I hope you understood how to create and use Task class objects in C#.

# C# Task Return Value

## ****C# Task Return Value with Examples****

In this article, I am going to discuss the **C# Task return value** in detail.Please read our previous article where we discussed [**how to create and use the task object in C#**](https://dotnettutorials.net/lesson/asynchronous-programming-in-csharp/)in different ways. At the end of this article, you will understand **How to Return a Value from a Task in C#** with examples.

The .NET Framework also provides a generic version of the Task class i.e. **Task<T>**. Using this **Task<T>** class we can return data or value from a task. In **Task<T>**, **T** represents the data type that you want to returns as a result of the task.

##### ****Example:****

In the following example, the CalculateSum method takes an input integer value and calculates the sum of the number starting from 1 to that number. Here the CalculateSum method returns a double value. As the return value from the CalculateSum method is of double type, so here we need to use Task<double> as shown in the below example.

**using** *System;*

**using** *System.Threading.Tasks;*

**namespace** *TaskBasedAsynchronousProgramming*

**{**

**class** Program

**{**

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

**{**

Console.WriteLine**(**$"Main Thread Started"**)**;

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

**{**

**return** CalculateSum**(**10**)**;

**})**;

Console.WriteLine**(**$"Sum is: {task1.Result}"**)**;

Console.WriteLine**(**$"Main Thread Completed"**)**;

Console.ReadKey**()**;

**}**

**static** **double** CalculateSum**(int** num**)**

**{**

**double** sum = 0;

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

**{**

sum += count;

**}**

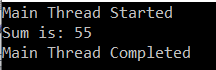
**return** sum;

**}**

**}**

**}**

**Output:**



**Note:** The Result property of the Task object blocks the calling thread until the task finishes its work.

##### ****Example:****

In the below example, we are writing the logic as part of the Anonymous method.

**using** *System;*

**using** *System.Threading.Tasks;*

**namespace** *TaskBasedAsynchronousProgramming*

**{**

**class** Program

**{**

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

**{**

Console.WriteLine**(**$"Main Thread Started"**)**;

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

**{**

**double** sum = 0;

**for** **(int** count = 1; count **<**= 10; count++**)**

**{**

sum += count;

**}**

**return** sum;

**})**;

Console.WriteLine**(**$"Sum is: {task1.Result}"**)**;

Console.WriteLine**(**$"Main Thread Completed"**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

It will also give you the same output as the previous example. So, whenever your logic is a few lines and that is going to be used only once, then it is always better to write the logic with the anonymous method.

##### ****Example: Returning Complex Type Value From a task****

In the below example, we are returning a Complex type.

**using** *System;*

**using** *System.Threading.Tasks;*

**namespace** *TaskBasedAsynchronousProgramming*

**{**

**class** Program

**{**

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

**{**

Console.WriteLine**(**$"Main Thread Started"**)**;

Task**<**Employee**>** task = Task**<**Employee**>**.Factory.StartNew**(()** =**>**

**{**

Employee employee = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Salary = 10000

**}**;

**return** employee;

**})**;

Employee emp = task.Result;

Console.WriteLine**(**$"ID: {emp.ID}, Name : {emp.Name}, Salary : {emp.Salary}"**)**;

Console.WriteLine**(**$"Main Thread Completed"**)**;

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

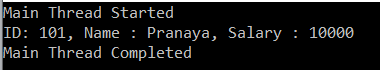
**public** string Name **{** **get**; **set**; **}**

**public** **double** Salary **{** **get**; **set**; **}**

**}**

**}**

###### **Output:**

****

**Chaining Tasks by Using Continuation Tasks**

**Chaining Tasks by Using Continuation Tasks in C#**

In this article, I am going to discuss **how to Chaining Tasks by Using Continuation Tasks in C#** with some examples. Please read our previous article where we discussed [**How to Return a Value from a Task in C#**](https://dotnettutorials.net/lesson/return-a-value-from-a-task-in-csharp/) with some examples.

While working with asynchronous programming, it is very common to invoke one asynchronous operation from another asynchronous operation passing the data once it completes its execution. This is called as continuations and in the traditional approach, this has been done by using the callback methods which is little difficult to understand.

But with the introduction of Task Parallel Library (TPL), the same functionality can be achieved very easily by using continuation tasks. In simple words, we can define a continuation task as an asynchronous task which is going to be invoked by another task (i.e. known as the antecedent).

**Creating a continuation for a single antecedent**

In C#, you can create a continuation by calling the ContinueWith method that is going to execute when its antecedent has completed its execution.

In the following example, the antecedent task i.e. task1 return an integer value. When it completes its executions, then it passes that value to the continuation task and that continuation task does some operation and returns the result as a string.

**using** *System;*

**using** *System.Threading.Tasks;*

**namespace** *TaskBasedAsynchronousProgramming*

**{**

**class** Program

**{**

**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**()**;

**}**

**}**

**}**

**Output:** The Square of 12 is : 144

**Scheduling Different Continuation Tasks**

The ContinueWith method has some overloaded versions that you can use to configure with multiple options when the continuation will run. In this way, you can add different continuation methods that will run when an exception occurred, when the Task is canceled, or the Task completed successfully. Let us see an example to understand this.

**using** *System;*

**using** *System.Threading.Tasks;*

**namespace** *TaskBasedAsynchronousProgramming*

**{**

**class** Program

**{**

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

**{**

Task**<int>** task = Task.Run**(()** =**>**

**{**

**return** 10;

**})**;

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**)**;

completedTask.Wait**()**;

Console.ReadKey**()**;

**}**

**}**

**}**