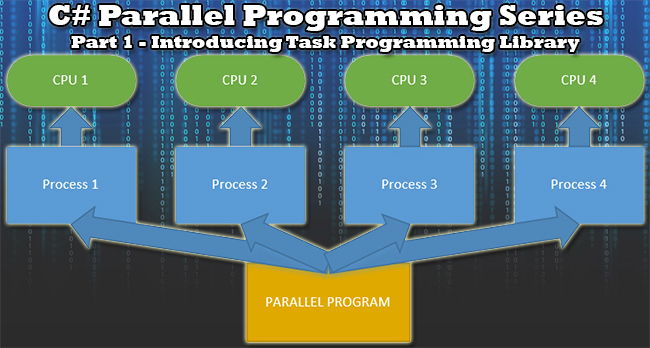
Introducing Task Programming Library



## **Task vs Thread Differences in C#**

When we execute things on multiple threads, it’s not guaranteed that the threads are separated across multiple processors.

Task is a lightweight object for managing a parallelizable unit of work. It can be used whenever you want to execute something in parallel. Parallel means the work is spread across multiple processors to maximize computational speed. Tasks are tuned for leveraging multicores processors.

Task provides following powerful features over thread.

1. If system has multiple tasks then it make use of the CLR thread pool internally, and so do not have the overhead associated with creating a dedicated thread using the Thread. Also reduce the context switching time among multiple threads.
2. Task can return a result. There is no direct mechanism to return the result from thread.
3. Wait on a set of tasks, without a signaling construct.
4. We can chain tasks together to execute one after the other.
5. Establish a parent/child relationship when one task is started from another task.
6. Child task exception can propagate to parent task.
7. Task support cancellation through the use of cancellation tokens.
8. Asynchronous implementation is easy in task, using’ async’ and ‘await’ keywords.

## **Beginning With Task Programming Library**

The support of parallel programming within the .Net framework is not new since it is supported from its very first version 1.0. We refer to this as a **classic threading model**. Even though it works really well, managing all the parallel aspects is complicated, so many times the applications end with unexpected results.

On the other hand, the TPL is **built on the foundation of the classic threading features** and manages many aspects for you, so you will need to write less code to achieve the same behavior. Actually, the reduction of the amount of code is huge.

We start with the basics of the **Task** class that can be considered to be the heart of the entire library. Please note that as long as you want to rely on TPL's features, you need to reference the proper namespace in your project.

1. **using** System.Threading.Tasks;

### **Creating And Starting New Task**

In the simplest scenarios to create and start a task, you just need to provide its body that represents the **workload you want to run in parallel** by passing in a System.Action delegate. There are several ways to declare the task's body. These are listed below and demonstrated in the first example.

* Using Action delegate
* Using anonymous function
* Using lambda function

For the sake of simplicity, our pieces of workload running concurrently will be represented by a simple HelloConsole method that will print one line to the console.

1. **static** **void** HelloConsole()
2. {
3. Console.WriteLine("Hello Task");
4. }

After creating a new instance of the Task class and passing the workload you want to perform in the constructor argument, you just need to call the instance **Start()** method to begin with the execution.

The following example shows the three options for declaring the Task object along with the console output.

1. **static** **void** Main(**string**[] args)
2. {
3. //Action delegate
4. Task task1 = **new** Task(**new** Action(HelloConsole));
6. //anonymous function
7. Task task2 = **new** Task(**delegate**
8. {
9. HelloConsole();
10. });
12. //lambda expression
13. Task task3 = **new** Task(() => HelloConsole());
15. task1.Start();
16. task2.Start();
17. task3.Start();
19. Console.WriteLine("Main method complete. Press any key to finish.");
20. Console.ReadKey();
21. }



**Image 1**: Creating and running simple tasks

**Note:** If you have some simple and short-living tasks, you can start them directly using the Task.Factory.StartNew() static method without having to explicitly create the object.

1. Task.Factory.StartNew(() => {
2. HelloConsole()
3. });

### **Setting Task State**

If you need to perform the **same workload** on a **different set of data** or just need to **provide some parameter** to the task, you need to pass in a System.Action<object> and an object representing these data/parameters. This process is very similar to supplying your console application with command line arguments. The following example shows this process by providing a simple string argument that will be printed to the console during the workload execution.

1. **static** **void** Main(**string**[] args)
2. {
3. //Action delegate
4. Task task1 = **new** Task(**new** Action<**object**>(HelloConsole), "Task 1");
6. //anonymous function
7. Task task2 = **new** Task(**delegate**(**object** obj)
8. {
9. HelloConsole(obj);
10. }, "Task 2");
12. //lambda expression
13. Task task3 = **new** Task((obj) => HelloConsole(obj), "Task 3");
15. task1.Start();
16. task2.Start();
17. task3.Start();
19. Console.WriteLine("Main method complete. Press any key to finish.");
20. Console.ReadKey();
21. }

We have also slightly altered the HelloConsole method that now accepts an object argument that will be printed to the console.

1. **static** **void** HelloConsole(**object** message)
2. {
3. Console.WriteLine("Hello: {0}", message);
4. }



**Image 2:** Setting state/supplying a parameter

Please note that the **order of the task's completion might differ** on your machine since it depends on how fast each task is executed.

### **Getting A Task's Result**

To get a result from a Task, you need to create an instance of Task<T> instead of just a pure Task. T represents the **type of the result that will be returned**. Returning the desired result is identical to other C# methods, so you use the "return" keyword. Finally, to fetch the result, you need to call the Result property. Note that reading this property will wait until its task has completed.

1. **static** **void** Main(**string**[] args)
2. {
3. //creating the task
4. Task<**int**> task1 = **new** Task<**int**>(() =>
5. {
6. **int** result = 1;
8. **for** (**int** i = 1; i < 10; i++)
9. result \*= i;
11. **return** result;
12. });
14. //starting the task
15. task1.Start();
17. //waiting for result - printing to the console
18. Console.WriteLine("Task result: {0}", task1.Result);
20. Console.WriteLine("Main method complete. Press any key to finish.");
21. Console.ReadKey();
22. }



**Image 3**:  Getting a result from a task

### **Cancelling A Task**

If we have more complex tasks that take some time to complete, we undoubtedly need a way how to cancel them before they finish if needed. For this purpose, the TPL introduced **cancellation tokens**that are used to cancel the given tasks. To be able to cancel a started task, we need to provide an instance of a CancellationToken in the task's constructor.

1. Task task = **new** Task(() =>
2. {
3. //task's body
4. }, token);

Acquiring this token is a **two-step process**:

First, we need to create an instance of CancellationTokenSource:

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

Next, to get the required CancellationToken instance, we call the CancellationTokenSource.Tokenproperty:

* 1. CancellationToken token = cancellationTokenSource.Token;

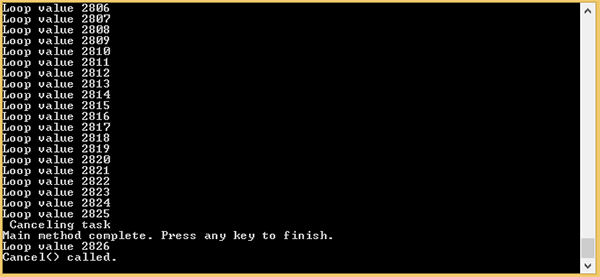
Finally, when the token is acquired and passed to the task's constructor, we simply call the **Cancel() method** of CancellationTokenSource to cancel it.

1. cancellationTokenSource.Cancel();

Calling the Cancel() method **won't cancel the task immediately**. Therefore, in the body of a given task you need to **monitor the token** whether a cancellation was requested by checking the token's IsCancellationRequested property. Once set to true, a cancellation was requested and you can cancel it either by calling "return" or throwing an OperationCanceledException.

The following example shows a basic use of cancellation tokens to cancel a running task.

1. **static** **void** Main(**string**[] args)
2. {
3. //creating the cancelation token
4. CancellationTokenSource cancellationTokenSource = **new** CancellationTokenSource();
5. CancellationToken token = cancellationTokenSource.Token;
7. //creating the task
8. Task task = **new** Task(() =>
9. {
10. **for** (**int** i = 0; i < 100000; i++)
11. {
12. **if** (token.IsCancellationRequested)
13. {
14. Console.WriteLine("Cancel() called.");
15. **return**;
16. }
18. Console.WriteLine("Loop value {0}", i);
19. }
20. }, token);
22. Console.WriteLine("Press any key to start task");
23. Console.WriteLine("Press any key again to cancel the running task");
24. Console.ReadKey();
26. //starting the task
27. task.Start();
29. //reading a console key
30. Console.ReadKey();
32. //canceling the task
33. Console.WriteLine("Canceling task");
34. cancellationTokenSource.Cancel();
36. Console.WriteLine("Main method complete. Press any key to finish.");
37. Console.ReadKey();
38. }



**Image 4**: Cancelling a task

## **Summary**

The **Task Programming Library** is built on the classic threading model and greatly **simplifies** the management of concurrent workloads. As a result, it greatly reduces the amount of code we need to write thus helps to prevent typical problems that are associated with the older threading concept.

## **Waiting For Tasks**

In the previous part of the series, we have used the Task.Result method that **stops the execution flow**until the given task has completed. However, there are other three methods that allow us to wait for a task that doesn't return any result or for a set of tasks that is useful for achieving some coordination among them.

The following is a brief summary of these additional methods:

* The **Call()** method on the Task instance is used to wait until the task has completed. You can optionally set a maximum waiting duration or CancellationToken to enable the task cancellation.
* The static**Task.WaitForAll()** method waits for all the tasks in the supplied task array to complete. Yet again you can add the maximum duration and the cancellation support.
* The static **Task.WaitForAny()** waits for the first task of a set of tasks to complete. Setting the duration and cancellation support are optional via the method arguments.

### **Waiting For A Single Task To Complete**

By calling the instance **Wait()** method you can **wait until a single task has completed**. Note that the task is considered to be completed not only when it executes all its workload, but **also when it has been cancelled or it has thrown an exception**.

There are several overloaded methods available that allow you to add an instance of a **CancellationToken**to enable the task's cancellation or add some specific waiting time duration in the form of a number of milliseconds or a **TimeSpan**.

The following example demonstrates the use of the **Wait()** method. We start by creating a static Workload method that represents the workload. In this case, we will just print the iteration turn to the console and put the task to sleep for 1 second.

1. **static** **void** Workload()
2. {
3. **for** (**int** i = 0; i < 5; i++)
4. {
5. Console.WriteLine("Task - iteration {0}", i);
7. //sleeping for 1 second
8. Thread.Sleep(1000);
9. }
10. }

We will now create two simple tasks and use **Wait()**to wait until they have completed. Note that we have used an **overloaded version** of the method with the second task and restricted it to wait only 2000 milliseconds (2 seconds).

1. **static** **void** Main(**string**[] args)
2. {
3. //creating and starting a simple task
4. Task task = **new** Task(**new** Action(Workload));
5. task.Start();
7. //waiting for the task
8. Console.WriteLine("Waiting for task to complete.");
9. task.Wait();
10. Console.WriteLine("Task Completed.");
12. //creating and starting another task
13. task = **new** Task(**new** Action(Workload));
14. task.Start();
15. Console.WriteLine("Waiting 2 secs for task to complete.");
16. task.Wait(2000);
17. Console.WriteLine("Wait ended - task completed.");
19. Console.WriteLine("Main method complete. Press any key to finish.");
20. Console.ReadKey();
21. }



**Image 1:** Waiting for a single task to complete

### **Waiting For Several Tasks To Complete**

The static **Task.WaitAll()**method is used to wait for a number of tasks to complete, so it will not return until all the given tasks will either complete, throw an exception or be cancelled. This method uses the **same overloading pattern** as the **Wait()** method.

For the sake of demonstration, we have created two tasks and will wait for both until they complete. Note that the second task will print just one line to the console, so it will finish almost immediately. Still, the **WaitAll()** method won't return until they both have completed.

1. **static** **void** Main(**string**[] args)
2. {
3. //createing the tasks
4. Task task1 = **new** Task(() =>
5. {
6. **for** (**int** i = 0; i < 5; i++)
7. {
8. Console.WriteLine("Task 1 - iteration {0}", i);
10. //sleeping for 1 second
11. Thread.Sleep(1000);
12. }
13. Console.WriteLine("Task 1 complete");
14. });
16. Task task2 = **new** Task(() =>
17. {
18. Console.WriteLine("Task 2 complete");
19. });
21. // starting the tasks
22. task1.Start();
23. task2.Start();
25. //waiting for both tasks to complete
26. Console.WriteLine("Waiting for tasks to complete.");
27. Task.WaitAll(task1, task2);
28. Console.WriteLine("Tasks Completed.");
30. Console.WriteLine("Main method complete. Press any key to finish.");
31. Console.ReadKey();
32. }



**Image 2:** Waiting for several tasks to complete

**Waiting For One Of Many Tasks To Complete**

The static**Task.WaitAny()** method is very similar to the method above (**WaitAll**), but instead of waiting for all the tasks to complete, it **waits only for the first one** that either has completed, was cancelled or has thrown an exception. Moreover, it returns the array index of the first completed task.

In the following example, we are starting two tasks and waiting for the first one to finish. Because the workload of the second task is quite simple, it completes first. As a result, the **WaitAny()** method returns 1 since that is the array index of the first completed task.

1. **static** **void** Main(**string**[] args)
2. {
3. //creating the tasks
4. Task task1 = **new** Task(() =>
5. {
6. **for** (**int** i = 0; i < 5; i++)
7. {
8. Console.WriteLine("Task 1 - iteration {0}", i);
9. //sleeping for 1 second
10. Thread.Sleep(1000);
11. }
12. Console.WriteLine("Task 1 complete");
13. });
15. Task task2 = **new** Task(() =>
16. {
17. Console.WriteLine("Task 2 complete");
18. });
20. //starting the tasks
21. task1.Start();
22. task2.Start();
24. //waiting for the first task to complete
25. Console.WriteLine("Waiting for tasks to complete.");
26. **int** taskIndex = Task.WaitAny(task1, task2);
27. Console.WriteLine("Task Completed - array index: {0}", taskIndex);
29. Console.WriteLine("Main method complete. Press any key to finish.");
30. Console.ReadKey();
31. }



**Image 3:** Waiting for one of many tasks to complete

**Note:** When the **WaitAny()**method returns, all the other started tasks will continue executing.

## **Exceptions Handling In Tasks**

No matter whether you write just a sequential application or add in some concurrency, you still **must handle exceptions**. Otherwise, the application will crash when an exception is thrown, leaving a poor use experience behind. In parallel programming, it is even more important since the application could behave unpredictably. Thankfully, TPL provides a **consistent model** for handling exceptions that are thrown during a task's execution.

When an exception occurs within a task (and is not caught), it is not thrown immediately. Instead, it is squirreled away by the .Net framework and thrown when some trigger member method is called, such as **Task.Result, Task.Wait(), Task.WaitAll() or Task.WaitAny().** In this case, an instance of **System.AggregateException** is thrown that acts as a wrapper around one or more exceptions that have occurred. This is important for methods that coordinate multiple tasks like **Task.WaitAll()** and **Task.WaitAny()**, so the **AggregateException**is able to wrap all the exceptions within the running tasks that have occurred.

In the following example, we are creating and starting three tasks, two of which throw different exceptions. After starting these tasks, the main calling thread calls the **WaitAll()**method and catches the **AggregateException**. Finally, it iterates through the **InnerExceptions property**and prints out the details regarding the thrown exceptions. This property is the wrapper holding all the information about the aggregated exceptions.

1. **static** **void** Main(**string**[] args)
2. {
3. //creating the tasks
4. Task task1 = **new** Task(() =>
5. {
6. NullReferenceException exception = **new** NullReferenceException();
7. exception.Source = "task1";
8. **throw** exception;
9. });
11. Task task2 = **new** Task(() =>
12. {
13. **throw** **new** IndexOutOfRangeException();
14. });
16. Task task3 = **new** Task(() =>
17. {
18. Console.WriteLine("Task 3");
19. });
21. //starting the tasks
22. task1.Start();
23. task2.Start();
24. task3.Start();
26. //waiting for all the tasks to complete
27. **try**
28. {
29. Task.WaitAll(task1, task2, task3);
30. }
31. **catch** (AggregateException ex)
32. {
33. //enumerate the exceptions
34. **foreach** (Exception inner **in** ex.InnerExceptions)
35. {
36. Console.WriteLine("Exception type {0} from {1}", inner.GetType(), inner.Source);
37. }
38. }
40. Console.WriteLine("Main method complete. Press any key to finish.");
41. Console.ReadKey();
42. }



**Image 4:** Exceptions handling in tasks

**Summary**

In order to coordinate tasks, we use the instance method **Wait()**or one from the two static methods **WaitAll()/WaitAny()**. These methods allow the application either to wait until all the given tasks have completed or wait until any of the tasks completes first.

An exception that occurs during a task execution is not thrown until some trigger member is called. We use and handle the **AggregateException**that aggregates all the tasks exceptions that have been thrown.