

# Task-based asynchronous programming

03/30/2017 • 38 minutes to read •  +9

## In this article

[Creating and running tasks implicitly](#)

[Creating and running tasks explicitly](#)

[Task ID](#)

[Task creation options](#)

[Tasks, threads, and culture](#)

[Creating task continuations](#)

[Creating detached child tasks](#)

[Creating child tasks](#)

[Waiting for tasks to finish](#)

[Composing tasks](#)

[Handling exceptions in tasks](#)

[Canceling tasks](#)

[The TaskFactory class](#)

[Tasks without delegates](#)

[Custom schedulers](#)

[Related data structures](#)

[Custom task types](#)

[Related topics](#)

[See also](#)

The Task Parallel Library (TPL) is based on the concept of a *task*, which represents an asynchronous operation. In some ways, a task resembles a thread or [ThreadPool](#) work item, but at a higher level of abstraction. The term *task parallelism* refers to one or more independent tasks running concurrently. Tasks provide two primary benefits:

- More efficient and more scalable use of system resources.

Behind the scenes, tasks are queued to the [ThreadPool](#), which has been enhanced with algorithms that determine and adjust to the number of threads and that provide

load balancing to maximize throughput. This makes tasks relatively lightweight, and you can create many of them to enable fine-grained parallelism.

- More programmatic control than is possible with a thread or work item.

Tasks and the framework built around them provide a rich set of APIs that support waiting, cancellation, continuations, robust exception handling, detailed status, custom scheduling, and more.

For both of these reasons, in the .NET Framework, TPL is the preferred API for writing multi-threaded, asynchronous, and parallel code.

## Creating and running tasks implicitly

The [Parallel.Invoke](#) method provides a convenient way to run any number of arbitrary statements concurrently. Just pass in an [Action](#) delegate for each item of work. The easiest way to create these delegates is to use lambda expressions. The lambda expression can either call a named method or provide the code inline. The following example shows a basic [Invoke](#) call that creates and starts two tasks that run concurrently. The first task is represented by a lambda expression that calls a method named `DoSomeWork`, and the second task is represented by a lambda expression that calls a method named `DoSomeOtherWork`.

### ⓘ Note

This documentation uses lambda expressions to define delegates in TPL. If you are not familiar with lambda expressions in C# or Visual Basic, see [Lambda Expressions in PLINQ and TPL](#).

C#



```
Parallel.Invoke(() => DoSomeWork(), () => DoSomeOtherWork());
```

### ⓘ Note

The number of [Task](#) instances that are created behind the scenes by [Invoke](#) is not necessarily equal to the number of delegates that are provided. The TPL may employ various optimizations, especially with large numbers of delegates.

For more information, see [How to: Use Parallel.Invoke to Execute Parallel Operations](#).

For greater control over task execution or to return a value from the task, you have to work with [Task](#) objects more explicitly.

## Creating and running tasks explicitly

A task that does not return a value is represented by the [System.Threading.Tasks.Task](#) class. A task that returns a value is represented by the [System.Threading.Tasks.Task<TResult>](#) class, which inherits from [Task](#). The task object handles the infrastructure details and provides methods and properties that are accessible from the calling thread throughout the lifetime of the task. For example, you can access the [Status](#) property of a task at any time to determine whether it has started running, ran to completion, was canceled, or has thrown an exception. The status is represented by a [TaskStatus](#) enumeration.

When you create a task, you give it a user delegate that encapsulates the code that the task will execute. The delegate can be expressed as a named delegate, an anonymous method, or a lambda expression. Lambda expressions can contain a call to a named method, as shown in the following example. Note that the example includes a call to the [Task.Wait](#) method to ensure that the task completes execution before the console mode application ends.

C#



```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        Thread.CurrentThread.Name = "Main";

        // Create a task and supply a user delegate by using a lambda expression.
        Task taskA = new Task( () => Console.WriteLine("Hello from taskA."));
        // Start the task.
        taskA.Start();

        // Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                        Thread.CurrentThread.Name);

        taskA.Wait();
    }
}
```

```
// The example displays output like the following:  
//     Hello from thread 'Main'.  
//     Hello from taskA.
```

You can also use the [Task.Run](#) methods to create and start a task in one operation. To manage the task, the [Run](#) methods use the default task scheduler, regardless of which task scheduler is associated with the current thread. The [Run](#) methods are the preferred way to create and start tasks when more control over the creation and scheduling of the task is not needed.

C#

 Copy

```
using System;  
using System.Threading;  
using System.Threading.Tasks;  
  
public class Example  
{  
    public static void Main()  
    {  
        Thread.CurrentThread.Name = "Main";  
  
        // Define and run the task.  
        Task taskA = Task.Run( () => Console.WriteLine("Hello from taskA."));  
  
        // Output a message from the calling thread.  
        Console.WriteLine("Hello from thread '{0}'.",  
                          Thread.CurrentThread.Name);  
  
        taskA.Wait();  
    }  
}  
// The example displays output like the following:  
//     Hello from thread 'Main'.  
//     Hello from taskA.
```

You can also use the [TaskFactory.StartNew](#) method to create and start a task in one operation. Use this method when creation and scheduling do not have to be separated and you require additional task creation options or the use of a specific scheduler, or when you need to pass additional state into the task that you can retrieve through its [Task.AsyncState](#) property, as shown in the following example.

C#

 Copy

```
using System;  
using System.Threading;  
using System.Threading.Tasks;
```

```

class CustomData
{
    public long CreationTime;
    public int Name;
    public int ThreadNum;
}

public class Example
{
    public static void Main()
    {
        Task[] taskArray = new Task[10];
        for (int i = 0; i < taskArray.Length; i++) {
            taskArray[i] = Task.Factory.StartNew( (Object obj ) => {
                CustomData data = obj as
CustomData;

                if (data == null)
                    return;

                data.ThreadNum =
Thread.CurrentThread.ManagedThreadId;

            },
            new CustomData() {Name = i,
CreationTime = DateTime.Now.Ticks} );
        }
        Task.WaitAll(taskArray);
        foreach (var task in taskArray) {
            var data = task.AsyncState as CustomData;
            if (data != null)
                Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                                data.Name, data.CreationTime, data.ThreadNum);
        }
    }
}

// The example displays output like the following:
//      Task #0 created at 635116412924597583 on thread #3.
//      Task #1 created at 635116412924607584 on thread #4.
//      Task #3 created at 635116412924607584 on thread #4.
//      Task #4 created at 635116412924607584 on thread #4.
//      Task #2 created at 635116412924607584 on thread #3.
//      Task #6 created at 635116412924607584 on thread #3.
//      Task #5 created at 635116412924607584 on thread #4.
//      Task #8 created at 635116412924607584 on thread #4.
//      Task #7 created at 635116412924607584 on thread #3.
//      Task #9 created at 635116412924607584 on thread #4.

```

[Task](#) and [Task<TResult>](#) each expose a static [Factory](#) property that returns a default instance of [TaskFactory](#), so that you can call the method as `Task.Factory.StartNew()`. Also,

in the following example, because the tasks are of type

[System.Threading.Tasks.Task<TResult>](#), they each have a public [Task<TResult>.Result](#)

property that contains the result of the computation. The tasks run asynchronously and may complete in any order. If the [Result](#) property is accessed before the computation finishes, the property blocks the calling thread until the value is available.

C#

 Copy

```
using System;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        Task<Double>[] taskArray = { Task<Double>.Factory.StartNew(() =>
DoComputation(1.0)),
                                     Task<Double>.Factory.StartNew(() =>
DoComputation(100.0)),
                                     Task<Double>.Factory.StartNew(() =>
DoComputation(1000.0)) };

        var results = new Double[taskArray.Length];
        Double sum = 0;

        for (int i = 0; i < taskArray.Length; i++) {
            results[i] = taskArray[i].Result;
            Console.WriteLine("{0:N1} {1}", results[i],
                               i == taskArray.Length - 1 ? "= " : "+ ");
            sum += results[i];
        }
        Console.WriteLine("{0:N1}", sum);
    }

    private static Double DoComputation(Double start)
    {
        Double sum = 0;
        for (var value = start; value <= start + 10; value += .1)
            sum += value;

        return sum;
    }
}

// The example displays the following output:
//      606.0 + 10,605.0 + 100,495.0 = 111,706.0
```

For more information, see [How to: Return a Value from a Task](#).

When you use a lambda expression to create a delegate, you have access to all the variables that are visible at that point in your source code. However, in some cases, most notably within loops, a lambda doesn't capture the variable as expected. It only captures the final value, not the value as it mutates after each iteration. The following example illustrates the problem. It passes a loop counter to a lambda expression that instantiates a `CustomData` object and uses the loop counter as the object's identifier. As the output from the example shows, each `CustomData` object has an identical identifier.

C#

 Copy

```
using System;
using System.Threading;
using System.Threading.Tasks;


class CustomData
{
    public long CreationTime;
    public int Name;
    public int ThreadNum;
}

public class Example
{
    public static void Main()
    {
        // Create the task object by using an Action(Of Object) to pass in the
        // loop counter. This produces an unexpected result.
        Task[] taskArray = new Task[10];
        for (int i = 0; i < taskArray.Length; i++) {
            taskArray[i] = Task.Factory.StartNew( (Object obj) => {
                var data = new CustomData()
                {Name = i, CreationTime = DateTime.Now.Ticks};
                data.ThreadNum =
                Thread.CurrentThread.ManagedThreadId;
                Console.WriteLine("Task #{0}
                created at {1} on thread #{2}.",
                                data.Name,
                                data.CreationTime, data.ThreadNum);
            },
            i );
        }
        Task.WaitAll(taskArray);
    }
}

// The example displays output like the following:
//      Task #10 created at 635116418427727841 on thread #4.
//      Task #10 created at 635116418427737842 on thread #4.
```

```
// Task #10 created at 635116418427737842 on thread #4.  
// Task #10 created at 635116418427737842 on thread #4.  
// Task #10 created at 635116418427737842 on thread #4.  
// Task #10 created at 635116418427737842 on thread #4.  
// Task #10 created at 635116418427727841 on thread #3.  
// Task #10 created at 635116418427747843 on thread #3.  
// Task #10 created at 635116418427747843 on thread #3.  
// Task #10 created at 635116418427737842 on thread #4.
```

You can access the value on each iteration by providing a state object to a task through its constructor. The following example modifies the previous example by using the loop counter when creating the `CustomData` object, which, in turn, is passed to the lambda expression. As the output from the example shows, each `CustomData` object now has a unique identifier based on the value of the loop counter at the time the object was instantiated.

C#	 Copy
<pre>using System; using System.Threading; using System.Threading.Tasks;  class CustomData {     public long CreationTime;     public int Name;     public int ThreadNum; }  public class Example {     public static void Main()     {         // Create the task object by using an Action(Of Object) to pass in custom data         // to the Task constructor. This is useful when you need to capture outer variables         // from within a loop. Task[] taskArray = new Task[10]; for (int i = 0; i &lt; taskArray.Length; i++) {     taskArray[i] = Task.Factory.StartNew( (Object obj ) =&gt; { CustomData data = obj as CustomData; if (data == null)     return; data.ThreadNum = Thread.CurrentThread.ManagedThreadId; }}</pre>	



```

        Console.WriteLine("Task #{0}
        created at {1} on thread #{2}.",
                                data.Name,
                                data.CreationTime, data.ThreadNum);
    },
    new CustomData() {Name = i,
CreationTime = DateTime.Now.Ticks} );
    }
    Task.WaitAll(taskArray);
}
}
// The example displays output like the following:
//      Task #0 created at 635116412924597583 on thread #3.
//      Task #1 created at 635116412924607584 on thread #4.
//      Task #3 created at 635116412924607584 on thread #4.
//      Task #4 created at 635116412924607584 on thread #4.
//      Task #2 created at 635116412924607584 on thread #3.
//      Task #6 created at 635116412924607584 on thread #3.
//      Task #5 created at 635116412924607584 on thread #4.
//      Task #8 created at 635116412924607584 on thread #4.
//      Task #7 created at 635116412924607584 on thread #3.
//      Task #9 created at 635116412924607584 on thread #4.

```

This state is passed as an argument to the task delegate, and it can be accessed from the task object by using the [Task.AsyncState](#) property. The following example is a variation on the previous example. It uses the [AsyncState](#) property to display information about the `CustomData` objects passed to the lambda expression.

C#



```

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

class CustomData
{
    public long CreationTime;
    public int Name;
    public int ThreadNum;
}

public class Example
{
    public static void Main()
    {
        Task[] taskArray = new Task[10];
        for (int i = 0; i < taskArray.Length; i++) {
            taskArray[i] = Task.Factory.StartNew( (Object obj ) => {
                CustomData data = obj as

```

```

CustomData;

        if (data == null)
            return;

        data.ThreadNum =
Thread.CurrentThread.ManagedThreadId;

        },
        new CustomData() {Name = i,
CreationTime = DateTime.Now.Ticks} );
    }
    Task.WaitAll(taskArray);
    foreach (var task in taskArray) {
        var data = task.AsyncState as CustomData;
        if (data != null)
            Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                data.Name, data.CreationTime, data.ThreadNum);
    }
}
}
// The example displays output like the following:
//      Task #0 created at 635116412924597583 on thread #3.
//      Task #1 created at 635116412924607584 on thread #4.
//      Task #3 created at 635116412924607584 on thread #4.
//      Task #4 created at 635116412924607584 on thread #4.
//      Task #2 created at 635116412924607584 on thread #3.
//      Task #6 created at 635116412924607584 on thread #3.
//      Task #5 created at 635116412924607584 on thread #4.
//      Task #8 created at 635116412924607584 on thread #4.
//      Task #7 created at 635116412924607584 on thread #3.
//      Task #9 created at 635116412924607584 on thread #4.

```

## Task ID

Every task receives an integer ID that uniquely identifies it in an application domain and can be accessed by using the [Task.Id](#) property. The ID is useful for viewing task information in the Visual Studio debugger **Parallel Stacks** and **Tasks** windows. The ID is lazily created, which means that it isn't created until it is requested; therefore, a task may have a different ID every time the program is run. For more information about how to view task IDs in the debugger, see [Using the Tasks Window](#) and [Using the Parallel Stacks Window](#).

## Task creation options

Most APIs that create tasks provide overloads that accept a [TaskCreationOptions](#) parameter. By specifying one of these options, you tell the task scheduler how to schedule the task on the thread pool. The following table lists the various task creation options.

## TaskCreationOptions

parameter value	Description
-----------------	-------------

None	The default when no option is specified. The scheduler uses its default heuristics to schedule the task.
------	--

PreferFairness	Specifies that the task should be scheduled so that tasks created sooner will be more likely to be executed sooner, and tasks created later will be more likely to execute later.
----------------	---

LongRunning	Specifies that the task represents a long-running operation.
-------------	--

AttachedToParent	Specifies that a task should be created as an attached child of the current task, if one exists. For more information, see <a href="#">Attached and Detached Child Tasks</a> .
------------------	--

DenyChildAttach	Specifies that if an inner task specifies the <code>AttachedToParent</code> option, that task will not become an attached child task.
-----------------	---

HideScheduler	Specifies that the task scheduler for tasks created by calling methods like <code>TaskFactory.StartNew</code> or <code>Task&lt;TResult&gt;.ContinueWith</code> from within a particular task is the default scheduler instead of the scheduler on which this task is running.
---------------	---

The options may be combined by using a bitwise **OR** operation. The following example shows a task that has the [LongRunning](#) and [PreferFairness](#) option.

C#	 Copy
<pre>var task3 = new Task(() =&gt; MyLongRunningMethod(),                     TaskCreationOptions.LongRunning                       TaskCreationOptions.PreferFairness); task3.Start();</pre>	

## Tasks, threads, and culture

Each thread has an associated culture and UI culture, which is defined by the [Thread.CurrentCulture](#) and [Thread.CurrentUICulture](#) properties, respectively. A thread's culture is used in such operations as formatting, parsing, sorting, and string comparison. A thread's UI culture is used in resource lookup. Ordinarily, unless you specify a default culture for all the threads in an application domain by using the

[CultureInfo.DefaultThreadCurrentCulture](#) and [CultureInfo.DefaultThreadCurrentUICulture](#) properties, the default culture and UI culture of a thread is defined by the system culture. If you explicitly set a thread's culture and launch a new thread, the new thread does not inherit the culture of the calling thread; instead, its culture is the default system culture. The task-based programming model for apps that target versions of the .NET Framework prior to .NET Framework 4.6 adhere to this practice.

### ❗ Important

Note that the calling thread's culture as part of a task's context applies to apps that *target* the .NET Framework 4.6, not apps that *run under* the .NET Framework 4.6. You can target a particular version of the .NET Framework when you create your project in Visual Studio by selecting that version from the dropdown list at the top of the **New Project** dialog box, or outside of Visual Studio you can use the [TargetFrameworkAttribute](#) attribute. For apps that target versions of the .NET Framework prior to the .NET Framework 4.6, or that do not target a specific version of the .NET Framework, a task's culture continues to be determined by the culture of the thread on which it runs.

Starting with apps that target the .NET Framework 4.6, the calling thread's culture is inherited by each task, even if the task runs asynchronously on a thread pool thread.

The following example provides a simple illustration. It uses the [TargetFrameworkAttribute](#) attribute to target the .NET Framework 4.6 and changes the app's current culture to either French (France) or, if French (France) is already the current culture, English (United States). It then invokes a delegate named `formatDelegat` that returns some numbers formatted as currency values in the new culture. Note that whether the delegate as a task either synchronously or asynchronously, it returns the expected result because the culture of the calling thread is inherited by the asynchronous task.

C#

 Copy

```
using System;
using System.Globalization;
using System.Runtime.Versioning;
using System.Threading;
using System.Threading.Tasks;

[assembly:TargetFramework(".NETFramework,Version=v4.6")]

public class Example
{
```

```

public static void Main()
{
    decimal[] values = { 163025412.32m, 18905365.59m };
    string formatString = "C2";
    Func<String> formatDelegate = () => { string output =
String.Format("Formatting using the {0} culture on thread {1}.\n",
CultureInfo.CurrentCulture.Name,
Thread.CurrentThread.ManagedThreadId);

                                foreach (var value in values)
                                    output += String.Format("{0}
", value.ToString(formatString));

                                output += Environment.NewLine;
                                return output;
                                };

    Console.WriteLine("The example is running on thread {0}",
        Thread.CurrentThread.ManagedThreadId);
    // Make the current culture different from the system culture.
    Console.WriteLine("The current culture is {0}",
        CultureInfo.CurrentCulture.Name);
    if (CultureInfo.CurrentCulture.Name == "fr-FR")
        Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
    else
        Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");

    Console.WriteLine("Changed the current culture to {0}.\n",
        CultureInfo.CurrentCulture.Name);

    // Execute the delegate synchronously.
    Console.WriteLine("Executing the delegate synchronously:");
    Console.WriteLine(formatDelegate());

    // Call an async delegate to format the values using one format string.
    Console.WriteLine("Executing a task asynchronously:");
    var t1 = Task.Run(formatDelegate);
    Console.WriteLine(t1.Result);

    Console.WriteLine("Executing a task synchronously:");
    var t2 = new Task<String>(formatDelegate);
    t2.RunSynchronously();
    Console.WriteLine(t2.Result);
}
}
// The example displays the following output:
//      The example is running on thread 1
//      The current culture is en-US
//      Changed the current culture to fr-FR.

```

```
//  
//      Executing the delegate synchronously:  
//      Formatting using the fr-FR culture on thread 1.  
//      163 025 412,32 €    18 905 365,59 €  
//  
//      Executing a task asynchronously:  
//      Formatting using the fr-FR culture on thread 3.  
//      163 025 412,32 €    18 905 365,59 €  
//  
//      Executing a task synchronously:  
//      Formatting using the fr-FR culture on thread 1.  
//      163 025 412,32 €    18 905 365,59 €  
// If the TargetFrameworkAttribute statement is removed, the example  
// displays the following output:  
//      The example is running on thread 1  
//      The current culture is en-US  
//      Changed the current culture to fr-FR.  
//  
//      Executing the delegate synchronously:  
//      Formatting using the fr-FR culture on thread 1.  
//      163 025 412,32 €    18 905 365,59 €  
//  
//      Executing a task asynchronously:  
//      Formatting using the en-US culture on thread 3.  
//      $163,025,412.32    $18,905,365.59  
//  
//      Executing a task synchronously:  
//      Formatting using the fr-FR culture on thread 1.  
//      163 025 412,32 €    18 905 365,59 €
```

If you are using Visual Studio, you can omit the [TargetFrameworkAttribute](#) attribute and instead select the .NET Framework 4.6 as the target when you create the project in the **New Project** dialog.

For output that reflects the behavior of apps the target versions of the .NET Framework prior to .NET Framework 4.6, remove the [TargetFrameworkAttribute](#) attribute from the source code. The output will reflect the formatting conventions of the default system culture, not the culture of the calling thread.

For more information on asynchronous tasks and culture, see the "Culture and asynchronous task-based operations" section in the [CultureInfo](#) topic.

## Creating task continuations

The [Task.ContinueWith](#) and [Task<TResult>.ContinueWith](#) methods let you specify a task to start when the *antecedent task* finishes. The delegate of the continuation task is passed a

reference to the antecedent task so that it can examine the antecedent task's status and, by retrieving the value of the [Task<TResult>.Result](#) property, can use the output of the antecedent as input for the continuation.

In the following example, the `getData` task is started by a call to the [TaskFactory.StartNew<TResult>\(Func<TResult>\)](#) method. The `processData` task is started automatically when `getData` finishes, and `displayData` is started when `processData` finishes. `getData` produces an integer array, which is accessible to the `processData` task through the `getData` task's [Task<TResult>.Result](#) property. The `processData` task processes that array and returns a result whose type is inferred from the return type of the lambda expression passed to the [Task<TResult>.ContinueWith<TNewResult>\(Func<Task<TResult>,TNewResult>\)](#) method. The `displayData` task executes automatically when `processData` finishes, and the [Tuple<T1,T2,T3>](#) object returned by the `processData` lambda expression is accessible to the `displayData` task through the `processData` task's [Task<TResult>.Result](#) property. The `displayData` task takes the result of the `processData` task and produces a result whose type is inferred in a similar manner and which is made available to the program in the [Result](#) property.

C#

 Copy

```
using System;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        var getData = Task.Factory.StartNew(() => {
            Random rnd = new Random();
            int[] values = new int[100];
            for (int ctr = 0; ctr <=
values.GetUpperBound(0); ctr++)
                values[ctr] = rnd.Next();

            return values;
        });

        var processData = getData.ContinueWith((x) => {
            int n = x.Result.Length;
            long sum = 0;
            double mean;

            for (int ctr = 0; ctr <=
x.Result.GetUpperBound(0); ctr++)
                sum += x.Result[ctr];
```

```

        mean = sum / (double) n;
        return Tuple.Create(n, sum,
mean);
    } );
    var displayData = processData.ContinueWith((x) => {
        return String.Format("N=
{0:N0}, Total = {1:N0}, Mean = {2:N2}",
x.Result.Item1, x.Result.Item2,
x.Result.Item3);
    } );
    Console.WriteLine(displayData.Result);
}
}
// The example displays output similar to the following:
//    N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82

```

Because [Task.ContinueWith](#) is an instance method, you can chain method calls together instead of instantiating a [Task<TResult>](#) object for each antecedent task. The following example is functionally identical to the previous example, except that it chains together calls to the [Task.ContinueWith](#) method. Note that the [Task<TResult>](#) object returned by the chain of method calls is the final continuation task.

C#

 Copy

```

using System;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        var displayData = Task.Factory.StartNew(() => {
            Random rnd = new Random();
            int[] values = new int[100];
            for (int ctr = 0; ctr <=
values.GetUpperBound(0); ctr++)
                values[ctr] = rnd.Next();

            return values;
        } ).
        ContinueWith((x) => {
            int n = x.Result.Length;
            long sum = 0;
            double mean;

            for (int ctr = 0; ctr <=

```



```

x.Result.GetUpperBound(0); ctr++)

                sum += x.Result[ctr];

                mean = sum / (double) n;
                return Tuple.Create(n, sum, mean);
            } ).
            ContinueWith((x) => {
                return String.Format("N={0:N0}, Total =
{1:N0}, Mean = {2:N2}",
                                x.Result.Item1,
                                x.Result.Item2,
                                x.Result.Item3);
            } );
        Console.WriteLine(displayData.Result);
    }
}
// The example displays output similar to the following:
//     N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82


```

The [ContinueWhenAll](#) and [ContinueWhenAny](#) methods enable you to continue from multiple tasks.

For more information, see [Chaining Tasks by Using Continuation Tasks](#).

## Creating detached child tasks

When user code that is running in a task creates a new task and does not specify the [AttachedToParent](#) option, the new task is not synchronized with the parent task in any special way. This type of non-synchronized task is called a *detached nested task* or *detached child task*. The following example shows a task that creates one detached child task.

C#	 Copy
<pre> var outer = Task.Factory.StartNew(() =&gt; {     Console.WriteLine("Outer task beginning.");      var child = Task.Factory.StartNew(() =&gt;     {         Thread.SpinWait(5000000);         Console.WriteLine("Detached task completed.");     }); });  outer.Wait(); Console.WriteLine("Outer task completed."); // The example displays the following output: </pre>	

```
// Outer task beginning.  
// Outer task completed.  
// Detached task completed.
```

Note that the parent task does not wait for the detached child task to finish.

## Creating child tasks

When user code that is running in a task creates a task with the [AttachedToParent](#) option, the new task is known as a *attached child task* of the parent task. You can use the [AttachedToParent](#) option to express structured task parallelism, because the parent task implicitly waits for all attached child tasks to finish. The following example shows a parent task that creates ten attached child tasks. Note that although the example calls the [Task.Wait](#) method to wait for the parent task to finish, it does not have to explicitly wait for the attached child tasks to complete.

C#

 Copy

```
using System;  
using System.Threading;  
using System.Threading.Tasks;  
  
public class Example  
{  
    public static void Main()  
    {  
        var parent = Task.Factory.StartNew(() => {  
            Console.WriteLine("Parent task beginning.");  
            for (int ctr = 0; ctr < 10; ctr++) {  
                int taskNo = ctr;  
                Task.Factory.StartNew((x) => {  
                    Thread.SpinWait(5000000);  
                    Console.WriteLine("Attached  
child #{0} completed.",  
                                     x);  
                },  
                taskNo,  
                TaskCreationOptions.AttachedToParent);  
            }  
        });  
  
        parent.Wait();  
        Console.WriteLine("Parent task completed.");  
    }  
}  
  
// The example displays output like the following:
```

```
//      Parent task beginning.  
//      Attached child #9 completed.  
//      Attached child #0 completed.  
//      Attached child #8 completed.  
//      Attached child #1 completed.  
//      Attached child #7 completed.  
//      Attached child #2 completed.  
//      Attached child #6 completed.  
//      Attached child #3 completed.  
//      Attached child #5 completed.  
//      Attached child #4 completed.  
//      Parent task completed.
```

A parent task can use the [TaskCreationOptions.DenyChildAttach](#) option to prevent other tasks from attaching to the parent task. For more information, see [Attached and Detached Child Tasks](#).

## Waiting for tasks to finish

The [System.Threading.Tasks.Task](#) and [System.Threading.Tasks.Task<TResult>](#) types provide several overloads of the [Task.Wait](#) methods that enable you to wait for a task to finish. In addition, overloads of the static [Task.WaitAll](#) and [Task.WaitAny](#) methods let you wait for any or all of an array of tasks to finish.

Typically, you would wait for a task for one of these reasons:

- The main thread depends on the final result computed by a task.
- You have to handle exceptions that might be thrown from the task.
- The application may terminate before all tasks have completed execution. For example, console applications will terminate as soon as all synchronous code in `Main` (the application entry point) has executed.

The following example shows the basic pattern that does not involve exception handling.

C#

 Copy

```
Task[] tasks = new Task[3]  
{  
    Task.Factory.StartNew(() => MethodA()),  
    Task.Factory.StartNew(() => MethodB()),  
    Task.Factory.StartNew(() => MethodC())  
};
```

```
//Block until all tasks complete.  
Task.WaitAll(tasks);  
  
// Continue on this thread...
```

For an example that shows exception handling, see [Exception Handling](#).

Some overloads let you specify a time-out, and others take an additional [CancellationToken](#) as an input parameter, so that the wait itself can be canceled either programmatically or in response to user input.

When you wait for a task, you implicitly wait for all children of that task that were created by using the [TaskCreationOptions.AttachedToParent](#) option. [Task.Wait](#) returns immediately if the task has already completed. Any exceptions raised by a task will be thrown by a [Task.Wait](#) method, even if the [Task.Wait](#) method was called after the task completed.

## Composing tasks

The [Task](#) and [Task<TResult>](#) classes provide several methods that can help you compose multiple tasks to implement common patterns and to better use the asynchronous language features that are provided by C#, Visual Basic, and F#. This section describes the [WhenAll](#), [WhenAny](#), [Delay](#), and [FromResult](#) methods.

### Task.WhenAll

The [Task.WhenAll](#) method asynchronously waits for multiple [Task](#) or [Task<TResult>](#) objects to finish. It provides overloaded versions that enable you to wait for non-uniform sets of tasks. For example, you can wait for multiple [Task](#) and [Task<TResult>](#) objects to complete from one method call.

### Task.WhenAny

The [Task.WhenAny](#) method asynchronously waits for one of multiple [Task](#) or [Task<TResult>](#) objects to finish. As in the [Task.WhenAll](#) method, this method provides overloaded versions that enable you to wait for non-uniform sets of tasks. The [WhenAny](#) method is especially useful in the following scenarios.

- Redundant operations. Consider an algorithm or operation that can be performed in many ways. You can use the [WhenAny](#) method to select the operation that finishes first and then cancel the remaining operations.

- Interleaved operations. You can start multiple operations that must all finish and use the [WhenAny](#) method to process results as each operation finishes. After one operation finishes, you can start one or more additional tasks.
- Throttled operations. You can use the [WhenAny](#) method to extend the previous scenario by limiting the number of concurrent operations.
- Expired operations. You can use the [WhenAny](#) method to select between one or more tasks and a task that finishes after a specific time, such as a task that is returned by the [Delay](#) method. The [Delay](#) method is described in the following section.

## Task.Delay

The [Task.Delay](#) method produces a [Task](#) object that finishes after the specified time. You can use this method to build loops that occasionally poll for data, introduce time-outs, delay the handling of user input for a predetermined time, and so on.

## Task(T).FromResult

By using the [Task.FromResult](#) method, you can create a [Task<TResult>](#) object that holds a pre-computed result. This method is useful when you perform an asynchronous operation that returns a [Task<TResult>](#) object, and the result of that [Task<TResult>](#) object is already computed. For an example that uses [FromResult](#) to retrieve the results of asynchronous download operations that are held in a cache, see [How to: Create Pre-Computed Tasks](#).

# Handling exceptions in tasks

When a task throws one or more exceptions, the exceptions are wrapped in an [AggregateException](#) exception. That exception is propagated back to the thread that joins with the task, which is typically the thread that is waiting for the task to finish or the thread that accesses the [Result](#) property. This behavior serves to enforce the .NET Framework policy that all unhandled exceptions by default should terminate the process. The calling code can handle the exceptions by using any of the following in a `try/catch` block:

- The [Wait](#) method
- The [WaitAll](#) method
- The [WaitAny](#) method
- The [Result](#) property

The joining thread can also handle exceptions by accessing the [Exception](#) property before the task is garbage-collected. By accessing this property, you prevent the unhandled exception from triggering the exception propagation behavior that terminates the process when the object is finalized.

For more information about exceptions and tasks, see [Exception Handling](#).

## Canceling tasks

The [Task](#) class supports cooperative cancellation and is fully integrated with the [System.Threading.CancellationTokenSource](#) and [System.Threading.CancellationToken](#) classes, which were introduced in the .NET Framework 4. Many of the constructors in the [System.Threading.Tasks.Task](#) class take a [CancellationToken](#) object as an input parameter. Many of the [StartNew](#) and [Run](#) overloads also include a [CancellationToken](#) parameter.

You can create the token, and issue the cancellation request at some later time, by using the [CancellationTokenSource](#) class. Pass the token to the [Task](#) as an argument, and also reference the same token in your user delegate, which does the work of responding to a cancellation request.

For more information, see [Task Cancellation](#) and [How to: Cancel a Task and Its Children](#).

## The TaskFactory class

The [TaskFactory](#) class provides static methods that encapsulate some common patterns for creating and starting tasks and continuation tasks.

- The most common pattern is [StartNew](#), which creates and starts a task in one statement.
- When you create continuation tasks from multiple antecedents, use the [ContinueWhenAll](#) method or [ContinueWhenAny](#) method or their equivalents in the [Task<TResult>](#) class. For more information, see [Chaining Tasks by Using Continuation Tasks](#).
- To encapsulate Asynchronous Programming Model `BeginX` and `EndX` methods in a [Task](#) or [Task<TResult>](#) instance, use the [FromAsync](#) methods. For more information, see [TPL and Traditional .NET Framework Asynchronous Programming](#).

The default [TaskFactory](#) can be accessed as a static property on the [Task](#) class or [Task<TResult>](#) class. You can also instantiate a [TaskFactory](#) directly and specify various options that include a [CancellationToken](#), a [TaskCreationOptions](#) option, a [TaskContinuationOptions](#) option, or a [TaskScheduler](#). Whatever options are specified when you create the task factory will be applied to all tasks that it creates, unless the [Task](#) is created by using the [TaskCreationOptions](#) enumeration, in which case the task's options override those of the task factory.

## Tasks without delegates

In some cases, you may want to use a [Task](#) to encapsulate some asynchronous operation that is performed by an external component instead of your own user delegate. If the operation is based on the Asynchronous Programming Model Begin/End pattern, you can use the [FromAsync](#) methods. If that is not the case, you can use the [TaskCompletionSource<TResult>](#) object to wrap the operation in a task and thereby gain some of the benefits of [Task](#) programmability, for example, support for exception propagation and continuations. For more information, see [TaskCompletionSource<TResult>](#).

## Custom schedulers

Most application or library developers do not care which processor the task runs on, how it synchronizes its work with other tasks, or how it is scheduled on the [System.Threading.ThreadPool](#). They only require that it execute as efficiently as possible on the host computer. If you require more fine-grained control over the scheduling details, the Task Parallel Library lets you configure some settings on the default task scheduler, and even lets you supply a custom scheduler. For more information, see [TaskScheduler](#).

## Related data structures

The TPL has several new public types that are useful in both parallel and sequential scenarios. These include several thread-safe, fast and scalable collection classes in the [System.Collections.Concurrent](#) namespace, and several new synchronization types, for example, [System.Threading.Semaphore](#) and [System.Threading.ManualResetEventSlim](#), which are more efficient than their predecessors for specific kinds of workloads. Other new types in the .NET Framework 4, for example, [System.Threading.Barrier](#) and [System.Threading.SpinLock](#), provide functionality that was not available in earlier releases. For more information, see [Data Structures for Parallel Programming](#).

# Custom task types

We recommend that you do not inherit from [System.Threading.Tasks.Task](#) or [System.Threading.Tasks.Task<TResult>](#). Instead, we recommend that you use the [AsyncState](#) property to associate additional data or state with a [Task](#) or [Task<TResult>](#) object. You can also use extension methods to extend the functionality of the [Task](#) and [Task<TResult>](#) classes. For more information about extension methods, see [Extension Methods](#) and [Extension Methods](#).

If you must inherit from [Task](#) or [Task<TResult>](#), you cannot use [Run](#), or the [System.Threading.Tasks.TaskFactory](#), [System.Threading.Tasks.TaskFactory<TResult>](#), or [System.Threading.Tasks.TaskCompletionSource<TResult>](#) classes to create instances of your custom task type because these mechanisms create only [Task](#) and [Task<TResult>](#) objects. In addition, you cannot use the task continuation mechanisms that are provided by [Task](#), [Task<TResult>](#), [TaskFactory](#), and [TaskFactory<TResult>](#) to create instances of your custom task type because these mechanisms also create only [Task](#) and [Task<TResult>](#) objects.

## Related topics

Title	Description
<a href="#">Chaining Tasks by Using Continuation Tasks</a>	Describes how continuations work.
<a href="#">Attached and Detached Child Tasks</a>	Describes the difference between attached and detached child tasks.
<a href="#">Task Cancellation</a>	Describes the cancellation support that is built into the <a href="#">Task</a> object.
<a href="#">Exception Handling</a>	Describes how exceptions on concurrent threads are handled.
<a href="#">How to: Use Parallel.Invoke to Execute Parallel Operations</a>	Describes how to use <a href="#">Invoke</a> .
<a href="#">How to: Return a Value from a Task</a>	Describes how to return values from tasks.



Title	Description
<a href="#">How to: Cancel a Task and Its Children</a>	Describes how to cancel tasks.
<a href="#">How to: Create Pre-Computed Tasks</a>	Describes how to use the <a href="#">Task.FromResult</a> method to retrieve the results of asynchronous download operations that are held in a cache.
<a href="#">How to: Traverse a Binary Tree with Parallel Tasks</a>	Describes how to use tasks to traverse a binary tree.
<a href="#">How to: Unwrap a Nested Task</a>	Demonstrates how to use the <a href="#">Unwrap</a> extension method.
<a href="#">Data Parallelism</a>	Describes how to use <a href="#">For</a> and <a href="#">ForEach</a> to create parallel loops over data.
<a href="#">Parallel Programming</a>	Top level node for .NET Framework parallel programming.

## See also

- [Parallel Programming](#)
- [Samples for Parallel Programming with the .NET Framework](#)

Is this page helpful?

 Yes  No