# Task-based Asynchronous Programming

- 03/30/2017
- 36 minutes to read

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[1] 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[2], 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[3] method provides a convenient way to run any number of arbitrary statements concurrently. Just pass in an Action[4] 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[5] 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`.

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[6].

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

The number of Task[7] instances that are created behind the scenes by Invoke[8] 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[9].

For greater control over task execution or to return a value from the task, you have to work with Task[10] objects more explicitly.

## Creating and running tasks explicitly

A task that does not return a value is represented by the System.Threading.Tasks.Task[11] class. A task that returns a value is represented by the System.Threading.Tasks.Task<TResult>[12] class, which inherits from Task[13]. 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[14] 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[15] 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[16] method to ensure that the task completes execution before the console mode application ends.

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

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

      // Create a task and supply a user delegate by using a lambda expression.
      Task taskA =  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.
Imports System.Threading
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
        Thread.CurrentThread.Name = "Main"

        ' Create a task and supply a user delegate by using a lambda expression.
        Dim taskA = New Task(Sub() 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()
   End Sub
End Module
' The example displays output like the following:
'    Hello from thread 'Main'.
'    Hello from taskA.
```

You can also use the Task.Run[17] methods to create and start a task in one operation. To manage the task, the Run[18] methods use the default task scheduler, regardless of which task scheduler is associated with the current thread. The Run[19] methods are the preferred way to create and start tasks when more control over the creation and scheduling of the task is not needed.

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

```
using System.Threading.Tasks;

public class Example
{
   public static  ()
   {
      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.
Imports System.Threading
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
        Thread.CurrentThread.Name = "Main"

        Dim taskA As Task = Task.Run(Sub() Console.WriteLine("Hello from taskA."))

        ' Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name)
        taskA.Wait()
   End Sub
End Module
' The example displays output like the following:
'    Hello from thread 'Main'.
'    Hello from taskA.
```

You can also use the TaskFactory.StartNew[20] 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 through its AsyncState[21] property, as shown in the following example.

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

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

      // Better: Create and start the task in one operation.
      Task taskA = Task.Factory.StartNew(() => 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.
Imports System.Threading
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Thread.CurrentThread.Name = "Main"

      ' Better: Create and start the task in one operation.
      Dim taskA = Task.Factory.StartNew(Sub() Console.WriteLine("Hello from taskA."))

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

      taskA.Wait()
   End Sub
End Module
' The example displays output like the following:
'       Hello from thread 'Main'.
'       Hello from taskA.
```

Task[22] and Task<TResult>[23] each expose a static Factory[24] property that returns a default instance of TaskFactory[25], 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>[26], they each have a public Task<TResult>.Result[27] property that contains the result of the computation. The tasks run asynchronously and may complete in any order. If the Result[28] property is accessed before the computation finishes, the property blocks the calling thread until the value is available.

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

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

         results =  Double[taskArray.Length];
        Double sum = ;

         ( i = ; i < taskArray.Length; i++) {
            results[i] = taskArray[i].Result;
            Console.Write("{0:N1} {1}", results[i],
```

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

   private static Double DoComputation(Double start)
   {
      Double sum = ;
       ( value = start; value <= start + ; value += )
         sum += value;

      return sum;
   }
}
// The example displays the following output:
//        606.0 + 10,605.0 + 100,495.0 = 111,706.0
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
        Dim taskArray() = { Task(Of Double).Factory.StartNew(Function() DoComputation(1.0)),
                            Task(Of Double).Factory.StartNew(Function() DoComputation(100.0)),
                            Task(Of Double).Factory.StartNew(Function() DoComputation(1000.0)) }

        Dim results(taskArray.Length - 1) As Double
        Dim sum As Double

        For i As Integer = 0 To taskArray.Length - 1
            results(i) = taskArray(i).Result
            Console.Write("{0:N1} {1}", results(i),
                              If(i = taskArray.Length - 1, "= ", "+ "))
            sum += results(i)
        Next
        Console.WriteLine("{0:N1}", sum)
   End Sub

   Private Function DoComputation(start As Double) As Double
      Dim sum As Double
      For value As Double = start To start + 10 Step .1
         sum += value
      Next
      Return sum
   End Function
End Module
' 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[29].

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.

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

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

public class Example
{
   public static  ()
   {
      // Create the task object by using an Action(Of Object) to pass in the loop
      // counter. This produces an unexpected result.
      Task[] taskArray =  Task[];
       ( i = ; i < taskArray.Length; i++) {
         taskArray[i] = Task.Factory.StartNew( (Object obj) => {
                                                 data =  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.
Imports System.Threading
Imports System.Threading.Tasks

Class CustomData
   Public CreationTime As Long
   Public Name As Integer
   Public ThreadNum As Integer
End Class

Module Example
   Public Sub Main()
      ' Create the task object by using an Action(Of Object) to pass in the loop
      ' counter. This produces an unexpected result.
```

```
      Dim taskArray(9) As Task
      For i As Integer = 0 To taskArray.Length - 1
         taskArray(i) = Task.Factory.StartNew(Sub(obj As Object)
                                                 Dim data As New CustomData With {.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)
                                              End Sub,
                                              i )
      Next
      Task.WaitAll(taskArray)
   End Sub
End Module
' 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.

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

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

public class Example
{
   public static  ()
   {
      // 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 =  Task[];
       ( i = ; i < taskArray.Length; i++) {
         taskArray[i] = Task.Factory.StartNew( (Object obj ) => {
                                                  CustomData data = obj  CustomData;
                                                   (data == )
                                                     return;
```

```
                                                data.ThreadNum = Thread.CurrentThread.ManagedThreadId;
                                                Console.WriteLine("Task #{0} created at {1} on thread #{2}.",
                                                                  data.Name, data.CreationTime, data.ThreadNum);
                                            },
                                             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.
Imports System.Threading
Imports System.Threading.Tasks

Class CustomData
   Public CreationTime As Long
   Public Name As Integer
   Public ThreadNum As Integer
End Class

Module Example
   Public Sub 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.
      Dim taskArray(9) As Task
      For i As Integer = 0 To taskArray.Length - 1
         taskArray(i) = Task.Factory.StartNew(Sub(obj As Object)
                                                 Dim data As CustomData = TryCast(obj, CustomData)
                                                 If data Is Nothing Then Return

                                                 data.ThreadNum = Thread.CurrentThread.ManagedThreadId
                                                 Console.WriteLine("Task #{0} created at {1} on thread #{2}.",
                                                                   data.Name, data.CreationTime, data.ThreadNum)
                                              End Sub,
                                              New CustomData With {.Name = i, .CreationTime = DateTime.Now.Ticks} )
      Next
      Task.WaitAll(taskArray)
   End Sub
End Module
' 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[30] property. The following example is a variation on the previous example. It uses the AsyncState[31] property to display information about the `CustomData` objects passed to the lambda expression.

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

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

public class Example
{
   public static  ()
   {
      Task[] taskArray =  Task[];
       ( i = ; i < taskArray.Length; i++) {
         taskArray[i] = Task.Factory.StartNew( (Object obj ) => {
                                                 CustomData data = obj  CustomData;
                                                  (data == )
                                                    return;

                                                 data.ThreadNum = Thread.CurrentThread.ManagedThreadId;
                                               },
                                                CustomData() {Name = i, CreationTime = DateTime.Now.Ticks} );
      }
      Task.WaitAll(taskArray);
      foreach ( task  taskArray) {
          data = task.AsyncState  CustomData;
          (data != )
            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.
Imports System.Threading
Imports System.Threading.Tasks

Class CustomData
   Public CreationTime As Long
   Public Name As Integer
   Public ThreadNum As Integer
End Class

Module Example
   Public Sub Main()
      Dim taskArray(9) As Task
      For i As Integer = 0 To taskArray.Length - 1
         taskArray(i) = Task.Factory.StartNew(Sub(obj As Object)
                                                 Dim data As CustomData = TryCast(obj, CustomData)
                                                 If data Is Nothing Then Return

                                                 data.ThreadNum = Thread.CurrentThread.ManagedThreadId
                                              End Sub,
                                              New CustomData With {.Name = i, .CreationTime = DateTime.Now.Ticks} )
      Next
      Task.WaitAll(taskArray)

      For Each task In taskArray
         Dim data = TryCast(task.AsyncState, CustomData)
         If data IsNot Nothing Then
            Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                              data.Name, data.CreationTime, data.ThreadNum)
         End If
      Next
   End Sub
End Module
' The example displays output like the following:
'    Task #0 created at 635116451245250515, ran on thread #3, RanToCompletion
'    Task #1 created at 635116451245270515, ran on thread #4, RanToCompletion
'    Task #2 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #3 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #4 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #5 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #6 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #7 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #8 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #9 created at 635116451245270515, ran on thread #3, RanToCompletion
```

## 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[32] 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[33] and Using the Parallel Stacks Window[34].

## Task Creation Options

Most APIs that create tasks provide overloads that accept a TaskCreationOptions[35] 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[36] 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 Attached and Detached Child Tasks[37]. |
| DenyChildAttach | Specifies that if an inner task specifies the `AttachedToParent` option, that task will not become an attached child task. |
| HideScheduler | Specifies that the task scheduler for tasks created by calling methods like TaskFactory.StartNew[38] or Task<TResult>.ContinueWith[39] 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.

```
 task3 =  Task(() => MyLongRunningMethod(),
                   TaskCreationOptions.LongRunning | TaskCreationOptions.PreferFairness);
task3.Start();

Dim task3 = New Task(Sub() MyLongRunningMethod(),
                        TaskCreationOptions.LongRunning Or TaskCreationOptions.PreferFairness)
task3.Start()
```

## Tasks, threads, and culture

Each thread has an associated culture and UI culture, which is defined by the Thread.CurrentCulture[40] and Thread.CurrentUICulture[41] 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[42] and CultureInfo.DefaultThreadCurrentUICulture[43] 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[44] 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[45] 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 `formatDelegate` 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.

```
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  ()
   {
       decimal[] values = { 163025412.32m, 18905365.59m };
       string formatString = ;
       Func<String> formatDelegate = () => { string output = String.Format("Formatting using the {0} culture on thread {1}.\n",
                                                                            CultureInfo.CurrentCulture.Name,
                                                                            Thread.CurrentThread.ManagedThreadId);
                                             foreach ( value  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);
        (CultureInfo.CurrentCulture.Name == "fr-FR")
          Thread.CurrentThread.CurrentCulture =  CultureInfo("en-US");

          Thread.CurrentThread.CurrentCulture =  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:");
        t1 = Task.Run(formatDelegate);
       Console.WriteLine(t1.Result);

       Console.WriteLine("Executing a task synchronously:");
        t2 =  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 €
Imports System.Globalization
Imports System.Runtime.Versioning
Imports System.Threading
Imports System.Threading.Tasks

<Assembly:TargetFramework(".NETFramework,Version=v4.6")>

Module Example
   Public Sub Main()
       Dim values() As Decimal = { 163025412.32d, 18905365.59d }
       Dim formatString As String = "C2"
       Dim formatDelegate As Func(Of String) = Function()
                                               Dim output As String = String.Format("Formatting using the {0} culture on thread {1}.",
                                                                                    CultureInfo.CurrentCulture.Name,
                                                                                    Thread.CurrentThread.ManagedThreadId)
                                               output += Environment.NewLine
                                               For Each value In values
                                                  output += String.Format("{0}   ", value.ToString(formatString))
                                               Next
```

```
                                                output += Environment.NewLine
                                                Return output
                                            End Function

      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" Then
         Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US")
      Else
         Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR")
      End If
      Console.WriteLine("Changed the current culture to {0}.",
                        CultureInfo.CurrentCulture.Name)
      Console.WriteLine()

      ' 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:")
      Dim t1 = Task.Run(formatDelegate)
      Console.WriteLine(t1.Result)

      Console.WriteLine("Executing a task synchronously:")
      Dim t2 = New Task(Of String)(formatDelegate)
      t2.RunSynchronously()
      Console.WriteLine(t2.Result)
   End Sub
End Module
' 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 Imports the fr-FR culture on thread 1.
'          163 025 412,32 €   18 905 365,59 €
'
'          Executing a task asynchronously:
'          Formatting Imports the fr-FR culture on thread 3.
'          163 025 412,32 €   18 905 365,59 €
'
'          Executing a task synchronously:
'          Formatting Imports 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[46] 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[47] 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[48] topic.

## Creating task continuations

The Task.ContinueWith[49] and Task<TResult>.ContinueWith[50] 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[51] 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[52] 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>[53] object returned by the `processData` lambda expression is accessible to the `displayData` task through the `processData` task's Task<TResult>.Result[54] 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[55] property.

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

public class Example
{
   public static  ()
   {
       getData = Task.Factory.StartNew(() => {
                                           Random rnd =  Random();
                                           [] values =  [];
                                            ( ctr = ; ctr <= values.GetUpperBound(); ctr++)
                                              values[ctr] = rnd.Next();

                                           return values;
                                       } );
       processData = getData.ContinueWith((x) => {
                                               n = x.Result.Length;
```

```
                                            sum = ;
                                           double mean;

                                            ( ctr = ; ctr <= x.Result.GetUpperBound(); ctr++)
                                              sum += x.Result[ctr];

                                           mean = sum / (double) n;
                                           return Tuple.Create(n, sum, mean);
                                         } );
       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
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim getData = Task.Factory.StartNew(Function()
                                          Dim rnd As New Random()
                                          Dim values(99) As Integer
                                          For ctr = 0 To values.GetUpperBound(0)
                                             values(ctr) = rnd.Next()
                                          Next
                                          Return values
                                        End Function)
      Dim processData = getData.ContinueWith(Function(x)
                                             Dim n As Integer = x.Result.Length
                                             Dim sum As Long
                                             Dim mean As Double

                                             For ctr = 0 To x.Result.GetUpperBound(0)
                                                sum += x.Result(ctr)
                                             Next
                                             mean = sum / n
                                             Return Tuple.Create(n, sum, mean)
                                          End Function)
      Dim displayData = processData.ContinueWith(Function(x)
                                              Return String.Format("N={0:N0}, Total = {1:N0}, Mean = {2:N2}",
                                                                   x.Result.Item1, x.Result.Item2,
                                                                   x.Result.Item3)
                                           End Function)
      Console.WriteLine(displayData.Result)
   End Sub
End Module
' The example displays output like the following:
'   N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82
```

Because Task.ContinueWith[56] is an instance method, you can chain method calls together instead of instantiating a Task<TResult>[57] 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[58] method. Note that the Task<TResult>[59] object returned by the chain of method calls is the final continuation task.

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

public class Example
{
   public static  ()
   {
       displayData = Task.Factory.StartNew(() => {
                                              Random rnd =  Random();
                                              [] values =  [];
                                               ( ctr = ; ctr <= values.GetUpperBound(); ctr++)
                                                 values[ctr] = rnd.Next();

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

                                        ( ctr = ; ctr <= x.Result.GetUpperBound(); 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
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim displayData = Task.Factory.StartNew(Function()
                                              Dim rnd As New Random()
                                              Dim values(99) As Integer
                                              For ctr = 0 To values.GetUpperBound(0)
                                                 values(ctr) = rnd.Next()
                                              Next
                                              Return values
                                           End Function). _
                        ContinueWith(Function(x)
                                       Dim n As Integer = x.Result.Length
                                       Dim sum As Long
                                       Dim mean As Double
```

```
                                      For ctr = 0 To x.Result.GetUpperBound(0)
                                         sum += x.Result(ctr)
                                      Next
                                      mean = sum / n
                                      Return Tuple.Create(n, sum, mean)
                                   End Function). _
                        ContinueWith(Function(x)
                                      Return String.Format("N={0:N0}, Total = {1:N0}, Mean = {2:N2}",
                                                           x.Result.Item1, x.Result.Item2,
                                                           x.Result.Item3)
                                   End Function)
      Console.WriteLine(displayData.Result)
   End Sub
End Module
' The example displays output like the following:
'   N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82
```

The ContinueWhenAll[60] and ContinueWhenAny[61] methods enable you to continue from multiple tasks.

For more information, see Chaining Tasks by Using Continuation Tasks[62].

## 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.

```
 outer = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Outer task beginning.");

     child = Task.Factory.StartNew(() =>
    {
        Thread.SpinWait(5000000);
        Console.WriteLine("Detached task completed.");
    });

});

outer.Wait();
Console.WriteLine("Outer task completed.");
// The example displays the following output:
//    Outer task beginning.
//    Outer task completed.
//    Detached task completed.
Dim outer = Task.Factory.StartNew(Sub()
                                      Console.WriteLine("Outer task beginning.")
                                      Dim child = Task.Factory.StartNew(Sub()
                                                                            Thread.SpinWait(5000000)
                                                                            Console.WriteLine("Detached task completed.")
                                                                        End Sub)
                                  End Sub)
outer.Wait()
Console.WriteLine("Outer task completed.")
```

```
' The example displays the following output:
'     Outer task beginning.
'     Outer task completed.
'    Detached child 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[63] method to wait for the parent task to finish, it does not have to explicitly wait for the attached child tasks to complete.

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

public class Example
{
   public static  ()
   {
       parent = Task.Factory.StartNew(() => {
                      Console.WriteLine("Parent task beginning.");
                       ( ctr = ; ctr < ; ctr++) {
                          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.
Imports System.Threading
Imports System.Threading.Tasks
```

```
Module Example
   Public Sub Main()
      Dim parent = Task.Factory.StartNew(Sub()
                                            Console.WriteLine("Parent task beginning.")
                                            For ctr As Integer = 0 To 9
                                               Dim taskNo As Integer = ctr
                                               Task.Factory.StartNew(Sub(x)
                                                                        Thread.SpinWait(5000000)
                                                                        Console.WriteLine("Attached child #{0} completed.",
                                                                                          x)
                                                                     End Sub,
                                                                     taskNo, TaskCreationOptions.AttachedToParent)
                                            Next
                                         End Sub)
      parent.Wait()
      Console.WriteLine("Parent task completed.")
   End Sub
End Module
' 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[64].

## Waiting for tasks to finish

The System.Threading.Tasks.Task[65] and System.Threading.Tasks.Task<TResult>[66] types provide several overloads of the Task.Wait[67] and `System.Threading.Tasks.Task.Wait` methods that enable you to wait for a task to finish. In addition, overloads of the static Task.WaitAll[68] and Task.WaitAny[69] 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.

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

```
//Block until all tasks complete.
Task.WaitAll(tasks);

// Continue on this thread...
Dim tasks() =
{
    Task.Factory.StartNew(Sub() MethodA()),
    Task.Factory.StartNew(Sub() MethodB()),
    Task.Factory.StartNew(Sub() MethodC())
}

' Block until all tasks complete.
Task.WaitAll(tasks)

' Continue on this thread...
```

For an example that shows exception handling, see Exception Handling[70].

Some overloads let you specify a time-out, and others take an additional CancellationToken[71] 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[72] returns immediately if the task has already completed. Any exceptions raised by a task will be thrown by a Task.Wait[73] method, even if the Task.Wait[74] method was called after the task completed.

## Composing tasks

The Task[75] and Task<TResult>[76] 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[77], WhenAny[78], Delay[79], and FromResult[80] methods.

### Task.WhenAll

The Task.WhenAll[81] method asynchronously waits for multiple Task[82] or Task<TResult>[83] 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[84] and Task<TResult>[85] objects to complete from one method call.

### Task.WhenAny

The Task.WhenAny[86] method asynchronously waits for one of multiple Task[87] or Task<TResult>[88] objects to finish. As in the Task.WhenAll[89] method, this method provides overloaded versions that enable you to wait for non-uniform sets of tasks. The WhenAny[90] 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[91] 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[92] 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[93] method to extend the previous scenario by limiting the number of concurrent operations.
- Expired operations. You can use the WhenAny[94] 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[95] method. The Delay[96] method is described in the following section.

### Task.Delay

The Task.Delay[97] method produces a Task[98] 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[99] method, you can create a Task<TResult>[100] object that holds a pre-computed result. This method is useful when you perform an asynchronous operation that returns a Task<TResult>[101] object, and the result of that Task<TResult>[102] object is already computed. For an example that uses FromResult[103] to retrieve the results of asynchronous download operations that are held in a cache, see How to: Create Pre-Computed Tasks[104].

## Handling exceptions in tasks

When a task throws one or more exceptions, the exceptions are wrapped in an AggregateException[105] 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[106] 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[107] method
- The WaitAll[108] method
- The WaitAny[109] method
- The Result[110] property

The joining thread can also handle exceptions by accessing the Exception[111] 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[112].

## Canceling tasks

The `Task` class supports cooperative cancellation and is fully integrated with the System.Threading.CancellationTokenSource[113] and System.Threading.CancellationToken[114] classes, which were introduced in the .NET Framework 4. Many of the constructors in the System.Threading.Tasks.Task[115] class take a CancellationToken[116] object as an input parameter. Many of the StartNew[117] and Run[118] overloads also include a CancellationToken[119] parameter.

You can create the token, and issue the cancellation request at some later time, by using the CancellationTokenSource[120] class. Pass the token to the Task[121] 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[122] and How to: Cancel a Task and Its Children[123].

## The TaskFactory class

The TaskFactory[124] class provides static methods that encapsulate some common patterns for creating and starting tasks and continuation tasks.

- The most common pattern is StartNew[125], which creates and starts a task in one statement.
- When you create continuation tasks from multiple antecedents, use the ContinueWhenAll[126] method or ContinueWhenAny[127] method or their equivalents in the Task<TResult>[128] class. For more information, see Chaining Tasks by Using Continuation Tasks[129].
- To encapsulate Asynchronous Programming Model `BeginX` and `EndX` methods in a Task[130] or Task<TResult>[131] instance, use the FromAsync[132] methods. For more information, see TPL and Traditional .NET Framework Asynchronous Programming[133].

The default TaskFactory[134] can be accessed as a static property on the Task[135] class or Task<TResult>[136] class. You can also instantiate a TaskFactory[137] directly and specify various options that include a CancellationToken[138], a TaskCreationOptions[139] option, a TaskContinuationOptions[140] option, or a TaskScheduler[141]. Whatever options are specified when you create the task factory will be applied to

all tasks that it creates, unless the Task[142] is created by using the TaskCreationOptions[143] 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[144] 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[145] methods. If that is not the case, you can use the TaskCompletionSource<TResult>[146] object to wrap the operation in a task and thereby gain some of the benefits of Task[147] programmability, for example, support for exception propagation and continuations. For more information, see TaskCompletionSource<TResult>[148].

## 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[149]. 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[150].

## 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[151] namespace, and several new synchronization types, for example, System.Threading.Semaphore[152] and System.Threading.ManualResetEventSlim[153], 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[154] and System.Threading.SpinLock[155], provide functionality that was not available in earlier releases. For more information, see Data Structures for Parallel Programming[156].

## Custom task types

We recommend that you do not inherit from System.Threading.Tasks.Task[157] or System.Threading.Tasks.Task<TResult>[158]. Instead, we recommend that you use the AsyncState[159] property to associate additional data or state with a Task[160] or Task<TResult>[161] object. You can also use extension methods to extend the functionality of the Task[162] and Task<TResult>[163] classes. For more information about extension methods, see Extension Methods[164] and Extension Methods[165].

If you must inherit from Task[166] or Task<TResult>[167], you cannot use Run[168], Run[169], or the System.Threading.Tasks.TaskFactory[170], System.Threading.Tasks.TaskFactory<TResult>[171], or System.Threading.Tasks.TaskCompletionSource<TResult>[172] classes to create instances of your custom task type because these mechanisms create only Task[173] and Task<TResult>[174] objects. In addition, you cannot use the task continuation mechanisms that are provided by Task[175], Task<TResult>[176], TaskFactory[177], and TaskFactory<TResult>[178] to create instances of your custom task type because these mechanisms also create only Task[179] and Task<TResult>[180] objects.

## Related topics

| TITLE | DESCRIPTION |
| --- | --- |
| Chaining Tasks by Using Continuation Tasks[181] | Describes how continuations work. |
| Attached and Detached Child Tasks[182] | Describes the difference between attached and detached child tasks. |
| Task Cancellation[183] | Describes the cancellation support that is built into the Task[184] object. |
| Exception Handling[185] | Describes how exceptions on concurrent threads are handled. |
| How to: Use Parallel.Invoke to Execute Parallel Operations[186] | Describes how to use Invoke[187]. |
| How to: Return a Value from a Task[188] | Describes how to return values from tasks. |

| TITLE | DESCRIPTION |
| --- | --- |
| How to: Cancel a Task and Its Children[189] | Describes how to cancel tasks. |
| How to: Create Pre-Computed Tasks[190] | Describes how to use the Task.FromResult[191] method to retrieve the results of asynchronous download operations that are held in a cache. |
| How to: Traverse a Binary Tree with Parallel Tasks[192] | Describes how to use tasks to traverse a binary tree. |
| How to: Unwrap a Nested Task[193] | Demonstrates how to use the Unwrap[194] extension method. |
| Data Parallelism[195] | Describes how to use For[196] and ForEach[197] to create parallel loops over data. |
| Parallel Programming[198] | Top level node for .NET Framework parallel programming. |

## See Also

Parallel Programming[199]

Samples for Parallel Programming with the .NET Framework[200]

## Feedback

We'd love to hear your thoughts. Choose the type you'd like to provide:

Product feedback [201]

Our new feedback system is built on GitHub Issues. Read about this change in our blog post[202].

Title Leave a comment
Loading feedback...

View on GitHub [203]

Links

1. https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadpool
2. https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadpool
3. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.invoke
4. https://docs.microsoft.com/en-us/dotnet/api/system.action
5. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.invoke
6. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/lambda-expressions-in-plinq-and-tpl
7. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
8. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.invoke
9. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-use-parallel-invoke-to-execute-parallel-operations
10. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
11. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
12. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
13. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
14. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.status
15. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus
16. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.wait

17. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.run
18. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.run
19. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.run
20. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.startnew
21. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.asyncstate
22. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
23. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
24. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.factory
25. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory
26. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
27. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1.result
28. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1.result
29. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-return-a-value-from-a-task
30. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.asyncstate
31. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.asyncstate
32. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.id
33. https://docs.microsoft.com/en-us/visualstudio/debugger/using-the-tasks-window
34. https://docs.microsoft.com/en-us/visualstudio/debugger/using-the-parallel-stacks-window
35. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcreationoptions
36. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcreationoptions
37. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/attached-and-detached-child-tasks
38. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.startnew
39. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1.continuewith
40. https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.currentculture
41. https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.currentuiculture
42. https://docs.microsoft.com/en-us/dotnet/api/system.globalization.cultureinfo.defaultthreadcurrentculture
43. https://docs.microsoft.com/en-us/dotnet/api/system.globalization.cultureinfo.defaultthreadcurrentuiculture
44. https://docs.microsoft.com/en-us/dotnet/api/system.runtime.versioning.targetframeworkattribute
45. https://docs.microsoft.com/en-us/dotnet/api/system.runtime.versioning.targetframeworkattribute
46. https://docs.microsoft.com/en-us/dotnet/api/system.runtime.versioning.targetframeworkattribute
47. https://docs.microsoft.com/en-us/dotnet/api/system.runtime.versioning.targetframeworkattribute
48. https://docs.microsoft.com/en-us/dotnet/api/system.globalization.cultureinfo
49. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.continuewith
50. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1.continuewith
51. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1.result
52. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1.result
53. https://docs.microsoft.com/en-us/dotnet/api/system.tuple-3
54. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1.result
55. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1.result
56. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.continuewith
57. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
58. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.continuewith
59. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
60. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.continuewhenall
61. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.continuewhenany
62. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/chaining-tasks-by-using-continuation-tasks

63. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.wait
64. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/attached-and-detached-child-tasks
65. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
66. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
67. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.wait
68. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.waitall
69. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.waitany
70. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/exception-handling-task-parallel-library
71. https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken
72. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.wait
73. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.wait
74. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.wait
75. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
76. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
77. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenall
78. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenany
79. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.delay
80. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.fromresult
81. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenall
82. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
83. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
84. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
85. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
86. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenany
87. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
88. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
89. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenall
90. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenany
91. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenany
92. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenany
93. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenany
94. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.whenany
95. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.delay
96. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.delay
97. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.delay
98. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
99. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.fromresult
100. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
101. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
102. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
103. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.fromresult
104. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-create-pre-computed-tasks
105. https://docs.microsoft.com/en-us/dotnet/api/system.aggregateexception
106. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1.result
107. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.wait
108. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.waitall

109. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.waitany
110. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1.result
111. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.exception
112. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/exception-handling-task-parallel-library
113. https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtokensource
114. https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken
115. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
116. https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken
117. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.startnew
118. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.run
119. https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken
120. https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtokensource
121. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
122. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-cancellation
123. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-cancel-a-task-and-its-children
124. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory
125. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.startnew
126. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.continuewhenall
127. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.continuewhenany
128. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
129. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/chaining-tasks-by-using-continuation-tasks
130. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
131. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
132. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.fromasync
133. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/tpl-and-traditional-async-programming
134. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory
135. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
136. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
137. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory
138. https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken
139. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcreationoptions
140. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcontinuationoptions
141. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskscheduler
142. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
143. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcreationoptions
144. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
145. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.fromasync
146. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcompletionsource-1
147. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
148. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcompletionsource-1
149. https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadpool
150. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskscheduler
151. https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent
152. https://docs.microsoft.com/en-us/dotnet/api/system.threading.semaphore
153. https://docs.microsoft.com/en-us/dotnet/api/system.threading.manualreseteventslim
154. https://docs.microsoft.com/en-us/dotnet/api/system.threading.barrier

155. https://docs.microsoft.com/en-us/dotnet/api/system.threading.spinlock
156. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/data-structures-for-parallel-programming
157. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
158. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
159. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.asyncstate
160. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
161. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
162. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
163. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
164. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods
165. https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/procedures/extension-methods
166. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
167. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
168. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.run
169. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.run
170. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory
171. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory-1
172. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcompletionsource-1
173. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
174. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
175. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
176. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
177. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory
178. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory-1
179. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
180. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1
181. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/chaining-tasks-by-using-continuation-tasks
182. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/attached-and-detached-child-tasks
183. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-cancellation
184. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task
185. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/exception-handling-task-parallel-library
186. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-use-parallel-invoke-to-execute-parallel-operations
187. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.invoke
188. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-return-a-value-from-a-task
189. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-cancel-a-task-and-its-children
190. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-create-pre-computed-tasks
191. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.fromresult
192. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-traverse-a-binary-tree-with-parallel-tasks
193. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-unwrap-a-nested-task
194. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskextensions.unwrap
195. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/data-parallelism-task-parallel-library
196. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.for
197. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.foreach
198. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/index
199. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/index
200. https://code.msdn.microsoft.com/Samples-for-Parallel-b4b76364

201. https://developercommunity.visualstudio.com/spaces/61/index.html

202. https://docs.microsoft.com/teamblog/a-new-feedback-system-is-coming-to-docs

203. https://github.com/dotnet/docs/issues?utf8=%E2%9C%93&q=%227d23f296-9a5e-ca0a-08c3-f718d0a90d94%22&in=body