

Concurrent Programming - Tasks

Paweł Paduch



Politechnika Świętokrzyska

14 stycznia 2021

Plan of the lecture

- 1 Introduction
 - Plan
 - Bibliography
 - Basic concepts
 - Patterns of asynchronous programming
 - Comparison
- 2 TAP
 - Naming and Return Types
 - Initializing
 - Exceptions
 - Execution
- 3 TAP
 - Statuses
 - Cancelation
 - Progress
 - Progress - Implementation
- 4 Examples
 - Task starting
 - Async and Await
 - Waiting and continuation
 - Conditional continuations
 - Task Schedulers

Bibliography

-  Paterny programowania asynchronicznego -
<https://docs.microsoft.com/dotnet/standard/asynchronous-programming-patterns/?view=netframework-4.7.2>
-  TAP -
<https://docs.microsoft.com/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap?view=netframework-4.7.2>

Task

- The *Task* class represents a single operation that does not return a value and usually executes asynchronously.
- For an operation that returns a value, use the class *Task<TResult>*
- Task is the central class that represents TAP (Task-based Asynchronous Pattern)
- Tasks usually execute on the thread pool asynchronously, so you can use *Status* properties such as *IsCanceled*, *IsCompleted* or *IsFaulted*
- To define the task to be performed by *task*, the *Lambda* notation is usually used

Asynchronous Programming Model (APM)

Asynchronous Programming Model (APM) pattern, also called AsyncResult pattern

- It is an older model uses an interface to a IAsyncResult asynchronous behavior
- Synchronous operations require Begin and End methods, such as BeginWrite and EndWrite, to implement asynchronous operations.
- Is no longer recommended for a new application designs.

Event-based Asynchronous Pattern (EAP)

Event-based Asynchronous Pattern (EAP)

- was introduced to the .Net 2.0 framework
- event-based model
- provides asynchronous behavior
- requires:
 - methods with the *Async* suffix
 - one or more events (*event*)
 - delegates
 - event handlers
 - Types derived from *EventArgs*
- currently is not a recommended pattern

Task-based Asynchronous Pattern (TAP)

Task-based Asynchronous Pattern (TAP)

- Introduced in .net framework 4
- Based on *Task* and *Task <TResult>* in the *System.Threading.Tasks* namespace
- Uses a single method to represent the initiation and completion of an asynchronous operation
- Recommended as a pattern for asynchronous operations
- in C# two new keywords *async* and *await*

Comparison APM

For comparison, consider an asynchronous method that reads a certain amount of data from a certain offset to a given buffer. In the case of APM, two methods would be issued

Listing 1: APM example

```
1 public class MyClass
2 {
3     public IAsyncResult BeginRead(
4         byte [] buffer, int offset, int count,
5         AsyncCallback callback, object state);
6     public int EndRead(IAsyncResult asyncResult);
7 }
```


Comparison EAP

EAP would have to issue the following set of types and variables

Listing 2: EAP example

```
1 public class MyClass
2 {
3     public void ReadAsync(byte [] buffer, int offset, int count);
4     public event ReadCompletedEventHandler ReadCompleted;
5 }
```

Comparison TAP

In the case of TAP suffice one method:

Listing 3: TAP example

```
1 public class MyClass
2 {
3     public int Read(byte [] buffer, int offset, int count);
4 }
```

Naming and types

Asynchronous methods in *TAP* contain the suffix *Async* after the operation name and return types of type *await* (*awaitable*) such as:

- *Task*,
- *Task<TResult>*,
- *ValueTask*,
- *ValueTask<TResult>*

For example, an asynchronous *Get* method that returns *Task<string>* can be named *GetAsync*

Naming and types

- If we add asynchronous *TAP* methods to a class that already contains *EAP* methods with the *Async* suffix, we should use the *TaskAsync* suffix
- If an asynchronous method starts an operation but does not return the above types *awaitable*, its name should start with *Begin* or *Start* or some other name suggesting that it will not return *awaitable*
- The *TAP* method returns either *System.Threading.Tasks.Task* or *System.Threading.Tasks.Task<TResult>* depending on whether the corresponding synchronous method returns *void* or type *TResult*.

Parameters

- The parameters of the TAP method should match those of its synchronous counterpart and should be provided in the same order.
- The *out* and *ref* parameters are excluded from this rule and should not be used.
- Data returned by *out* or *ref* should be returned as part of the *TResult* returned by *Task<TResult>*
- When returning multiple values, we should use a collection or a more elaborate structure.
- Consider adding the *CancellationToken* parameter, even if the synchronous counterpart of the TAP method does not offer such a parameter.

Exceptions in the naming

Methods that are purely for creating, manipulating, or combining tasks (where asynchronous intentions of a method are clear in the name of the method or in the name of the type to which the method belongs), do not need to follow this naming pattern; Such methods are often called combinators. For example *WaitAll*, *WaitAny*

Initiating an asynchronous operation

The TAP-based asynchronous method can perform a small part of the task synchronously, for example, check arguments and initiate an asynchronous operation before returning the resulting task. However, the synchronous part should be kept to a minimum for two reasons.

- if an async method is called from a UI (UI) thread, this can freeze it.
- when we want to start multiple asynchronous methods then each synchronous part delays the invocation of the next method.

In some cases, the initiation time of an asynchronous operation may exceed the operation itself performed synchronously, then we should not use async.

Exceptions in the async method

- The async method should throw an exception only in response to a usage error.
- For all other errors, exceptions that occur when an asynchronous method is running should be assigned to the returned task, even if the asynchronous method happens to complete synchronously before the task is returned.
- Typically, a task contains at most one exception. However, if the task represents multiple operations (for example, *WhenAll*), multiple exceptions may be associated with a single task.

Target environment

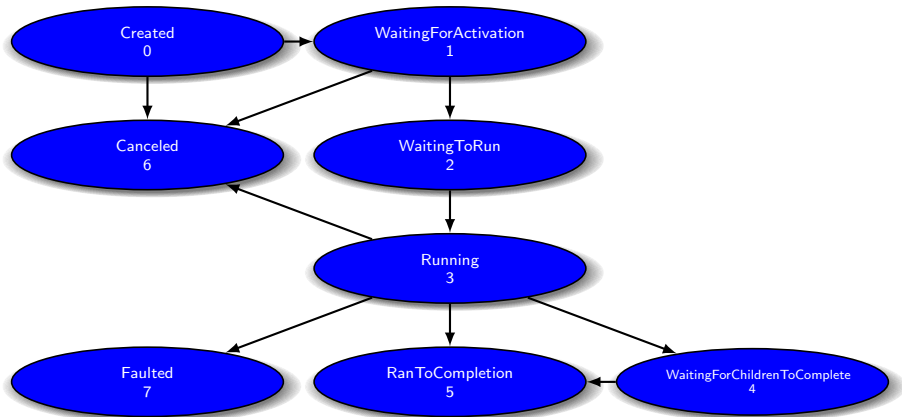
After implementing the TAP method, you can determine where the asynchronous execution occurs.

- execution on a thread pool
- using asynchronous I / O (without binding to a thread for most of the operation).
- run on a specific thread (e.g. UI thread).
- or use any number of potential contexts
- The TAP method doesn't even have to do anything, it just needs to return the task representing the occurrence of some condition elsewhere in the system. For example, with the information that there is data in the queue.

Calling program

- The program calling the asynchronous method can either block while waiting for the resulting task, or call additional continuation code after the asynchronous operation completes.
- The author of the continuation code decides where this code is to be executed. It can be created explicitly using methods of the *Task* class, eg *ContinueWith* or implicitly, eg *await*.

Statuses



Created

- This is a so-called 'cold' task, without activation.
- Task status right after creation by constructor *Task*
- Transition to another state only by calling *Start* or *RunSynchronously* on the task instance.
- If the TAP method internally creates a task using the *Task* constructor, it must activate it before returning its instance.
- Consumers of the TAP method can assume that the returned job is active and should no longer call *Start* to activate it, as this will result in an *InvalidOperationException* exception.

WaitingForActivation

- The status of the task right after it was created by methods such as *ContinueWith*, *ContinueWhenAll*, *ContinueWhenAny* and *FromAsync*
- The task is not scheduled yet and will not be until the tasks they are waiting for are finished.
- The job will be activated and scheduled internally by the .NET infrastructure

WaitingToRun

- A job that is scheduled and waiting to run
- Initial state for tasks created by *TaskFactory.StartNew*. At least until it returns from the *StartNew* function. But it may happen that they immediately return with the status *Running* or even *RanToCompletion*

Running

- Task in progress

WaitingForChildrenToComplete

- The task has completed but is waiting for child tasks to complete

RanToCompletion

- One of the 3 final states.
- The task successfully finished, without exceptions or cancellation.

Canceled

- One of the 3 final states.
- The task in this state is ended with *Cancel*

Faulted

- One of the 3 final states.
- The task enters this state when it completes an unhandled exception
- or one of the child tasks ends with a *Faulted* state.

Cancellation

- In TAP, cancellation is optional for both the implementation of async methods and their consumers.
- If the operation allows cancellation, it issues the overloaded asynchronous method that accepts the *CancellationToken* instance.
- With standard nomenclature, this parameter is called *cancellationToken*.

Listing 4: example

```
1 public Task ReadAsync(byte [] buffer, int offset, int count,  
2                       CancellationToken cancellationToken)
```

Cancellation

- The asynchronous operation checks the cancellation token and can honor it and cancel the operations.
- If this causes an early exit, the TAP method returns the job that ends in the *Canceled* state.
- There is no result available and no exception is thrown.
- The *Canceled* state is considered final. (completed) *IsCompleted = true*, including the states *Faulted* and *RanToCompletion*.
- When a task is canceled in the *Canceled* state, any continuations registered in the task are scheduled or executed unless the *NotOnCanceled* continuation option has been specified to cancel it.

Cancelation

- Any code asynchronously waiting for a canceled task through functions continues to run, but receives the *OperationCanceledException* exception or its derivatives.
- Synchronously blocked code waiting for a task using methods such as *Wait* and *WaitAll* also still works with exception.
- If a cancellation token requested cancellation before calling a TAP method that accepts the token, it should return the canceled task.
- However, if cancellation is required during an asynchronous activity, the asynchronous operation does not need to accept the cancellation request.

Cancellation

How to clearly define what is cancellable and what is not?

- For async methods that want to issue a cancellation, it's best not to issue an overload without a cancellation token.
- The caller of this method, who does not want to cancel it, will have the option to specify *None* instead of the token.
- However, when we want to prevent cancellation, we do not create an overload that accepts a cancellation token.
- This helps to indicate to the caller whether the target method is actually cancellable.

Cancellation

- Returned task should be completed in the state *Canceled* only when the operation will result in the cancellation request.
- If cancellation is requested but the result or exception is still thrown, the task should end in the status *RanToCompletion* or *Faulted*.

Progress

Some asynchronous operations use delivering progress notifications; they are typically used to update the user interface with information about the progress of the asynchronous operation. Provide a progress interface when an async method is called. As with cancellation, TAP implementations should provide the *IProgress<T>* parameter only if the API supports progress notifications.

Progress

The progress interface supports various progress implementations, determined by the code using it. For example:

- the using (consumer) code can only take care of the latest updates or cache all items.
- you can attach an action that handles the event of each update
- you can control whether the call is directed to a specific thread.

All these options can be achieved by using a different implementation of the interface, customized to the specific needs of the consumer.

Progress - example

If the *ReadAsync* method would be able to report indirect progress as the number of bytes read, the callback may be an interface *IProgress<T>*:

Listing 5: example

```
1 public Task ReadAsync(byte[] buffer, int offset, int count,  
2     IProgress<long> progress)
```

Progress - example

If the *FindFilesAsync* method returns a list of all files that match the specified search pattern, a progress callback can provide an estimate of the percentage of work completed as well as the current set of partial scores. It can do this either with a tuple...:

Listing 6: example

```
1 public Task<ReadOnlyCollection<FileInfo>> FindFilesAsync(  
2     string pattern,  
3     IProgress<Tuple<double,  
4     ReadOnlyCollection<List<FileInfo>>>> progress)  
5
```

Progress - example

...or an API-specific data type:

Listing 7: example

```
1 public Task<ReadOnlyCollection<FileInfo>> FindFilesAsync(  
2     string pattern,  
3     IProgress<FindFilesProgressInfo> progress)  
4
```

In the latter case, the special data type is usually supplemented with *ProgressInfo*.

Progress

If the TAP implementations provide overloads, which accept the progress parameter is they must allow the *progress* argument to be *NULL*, In this case, the consumer of the method is not interested in reporting progress and we should not report it. You have to check in the handling methods if *progress! = NULL*

Progress

TAP implementations should report progress to the *Progress<T>* object synchronously, which will enable the asynchronous method to quickly provide data on progress and allow consumers to determine how and where to best handle the obtained information on progress. For example, a progress instance might choose to make callbacks. For example, handle the reporting event.

Progress - Implementation

The .NET Framework 4.5 provides a single implementation of *IProgress<T>*: *Progress<T>*. The *Progress<T>* class is declared as follows:

Listing 8: example

```
1 public class Progress<T> : IProgress<T>
2 {
3     public Progress();
4     public Progress(Action<T> handler);
5     protected virtual void OnReport(T value);
6     public event EventHandler<T> ProgressChanged;
7 }
8
```


Progress

- The *Progress*<*T*> instance provides a *ProgressChanged* event that is called whenever an asynchronous operation reports a progress update.
- The *ProgressChanged* event is called on the *SynchronizationContext* object that was captured when the *Progress*<*T*> instance was created.
- If no synchronization context was available, the default context is used and points to the thread pool.
- You can attach *handlers* to this event. For convenience, one of them can be specified in the *Progress*<*T*> constructor
- Progress updates are triggered asynchronously to avoid delays.

Overloads

If your TAP implementation uses the optional *CancellationToken* parameters and the optional *IProgress<T>*, it can potentially require up to four overloads:

Listing 9: example

```
1 public Task MethodNameAsync(...);  
2 public Task MethodNameAsync(..., CancellationToken cancellationToken);  
3 public Task MethodNameAsync(..., IProgress<T> progress);  
4 public Task MethodNameAsync(...,  
5     CancellationToken cancellationToken, IProgress<T> progress);  
6
```

However, many implementations do not support *CancellationToken* and *IProgress<T>* and the first version will suffice.

Task Start

We create an instance of the Task class and then start the task.
The action is described by the lambda expression.

Listing 10: example of new Task

```
1  var task = new Task(() =>
2      {
3          Console.WriteLine("First task is working...");
4          Thread.Sleep(1000);
5          Console.WriteLine("First task finished");
6      });
7  task.Start();
8  task.Wait();
9
```

After the *Start* method is called, the main thread goes on, to wait for the task to finish, call *Wait()*

Task Run

We use the static *Run* method. It creates *Task* running, so you don't need to call *Start* ()

Listing 11: example of Task.Run

```
1 task = Task.Run(() => {  
2     Console.WriteLine("Task Run is working");  
3     Thread.Sleep(1000);  
4     Console.WriteLine("Task Run finished");  
5 });  
6 task.Wait();  
7
```

Task Factory

We use a factory and the static method *StartNew*. It creates a running *Task*, so you don't need to call *Start()*

Listing 12: example of Task.Factory.Run

```
1 task = Task.Factory.StartNew(() => {  
2     Console.WriteLine("Task from Factory is working");  
3     Thread.Sleep(1000);  
4     Console.WriteLine("Task from Factory finished");  
5 });  
6 task.Wait();  
7
```

Many tasks

We create multiple tasks that perform the same action. CurrentId is assigned when it is referenced, not at startup.

Listing 13: example of Mutli Task start

```
1 Action a = () =>
2     {
3         Console.WriteLine($"Task no {Task.CurrentId} has started");
4         Thread.SpinWait(new Random().Next(300000000));
5         Console.WriteLine($"Task no {Task.CurrentId} finished");
6     };
7 List<Task> listOfTasks = new List<Task>();
8 for (int i = 0; i < 10; i++)
9     {
10        listOfTasks.Add(new Task(a));
11    }
12 listOfTasks.ForEach(t => t.Start());
13 listOfTasks.ForEach(t => t.Wait());
14
```

Parameter passing

Passing information to the task

Listing 14: Example of passing parameter

```
1 Task printSthTask = new Task((o) =>
2     {
3         for (int i = 0; i < 5; i++)
4         {
5             Thread.Sleep(200);
6             Console.WriteLine((string)o + " " + i);
7         }
8     }, "Bla ble blu...");
9 printSthTask.Start();
10 printSthTask.Wait();
```

Returning values

Returning value from the task

Listing 15: Example of returning value

```
1 Task<int> intTask = Task.Run(() =>
2     {
3         Thread.Sleep(1000);
4         return 5;
5     });
6 Console.WriteLine("Waiting for value from task");
7 var result = intTask.Result;
8 Console.WriteLine($"We received value: {result}");
9
```

On line 7. We wait for the task to end, only then can we read the result

Casual work

Usual synchronous method making processor a little bit busy.

Listing 16: An example of a synchronous method

```
1 public void OrdinaryWork(int howMuch)
2     {
3         Console.WriteLine($"Worker {Name} start ordinary work...");
4         for (int i = 1; i <= howMuch; i++)
5             {
6                 Thread.SpinWait(100000000);
7                 Console.WriteLine($"Worker {Name} has elaborated {i} items");
8             }
9         Console.WriteLine($"Worker {Name} finished ordinary work.");
10    }
```

Listing 17: An example of calling a method

```
1 Worker worker1 = new Worker("Mietek");
2 Console.WriteLine("We are delegating a synchronous task to worker");
3 worker1.OrdinaryWork(10);
4 Console.WriteLine("The synchronous task was delegated and done");
```

Task without controll

This method will create a task, but it will be like “run and forget”.

Listing 18: An example of a method creating a task

```
1 public void WorkTask(int howMuch)
2     { Console.WriteLine($"Worker {Name} start task working....");
3       Task t = new Task(() =>
4         { for (int i = 1; i <= howMuch; i++)
5           { Thread.SpinWait(100000000);
6             Console.WriteLine($"Worker {Name} has elaborated {i} items");
7           }
8         Console.WriteLine($"Worker {Name} finished task work");
9       });
10    t.Start();    }
```

Listing 19: An example of calling a method

```
1 Worker worker2 = new Worker("Zenon");
2     Console.WriteLine("We order to the employee work and he creates a Task");
3     worker2.WorkTask(10);
4     Console.WriteLine("Work for Zenon was commissioned and he began to perform. ←
    Unfortunately, here our supervision ends");
```

The task we know about

The method that returns the task, according to the TAP policy, the returned task must already be running. This is an asynchronous method, so it might be called xxxAsync.

Listing 20: An example of a method that creates and returns a task

```
1 public Task WorkAsync(int howMuch)
2     {
3         Console.WriteLine($"Worker {Name} begin async work...");
4         Task t = new Task(() =>
5             {
6                 for (int i = 1; i <= howMuch; i++)
7                 {
8                     Thread.SpinWait(100000000);
9                     Console.WriteLine($"Worker {Name} has elaborated {i} items");
10                }
11                Console.WriteLine($"Worker {Name} finished async work");
12            });
13        t.Start(); //according to the TAP model, the task returned by the asynchronous
14        //method should be started
15        return t;
16    }
```

The task we know about

A call to a method that returns a task. Since it is awaitable, you can wait for it to finish.

Listing 21: An example of calling a method that creates and returns a task

```
1 Worker worker3 = new Worker("Janusz");  
2     Console.WriteLine("We order to the employee work and he creates a Task");  
3     var januszTask = worker3.WorkAsync(10);  
4     Console.WriteLine("Work for Janusz was commissioned and he began to perform. ↔  
5     Janusz returned the task so we can wait for it");  
6     januszTask.Wait();
```

Waiting with await

A method call returning a task. This time waiting with *await*.

Listing 22: An example of calling a method that creates and returns a task

```
1 Worker worker4 = new Worker("Grazyna");  
2 Console.WriteLine("We commission the employee to work and he creates a Task");  
3 var grazynaTask = worker4.WorkAsync(10);  
4 Console.WriteLine("The work for Grazyna was commissioned and she began to perform.\↔  
    nTGrazyna returned task, so we can wait for it with await");  
5 await grazynaTask;
```

When we use the word `await`, the method we use `await` must be `async`.

Async Task Main

How to use async methods in Main. You need to create an asynchronous main

Listing 23: An example of the Main method with async

```
1 public static void Main(string[] args)
2     {
3         Task.Run(async () =>
4             {
5                 Console.WriteLine("In Main before await");
6                 await MainAsync(args); //you need to create an asynchronous Main
7                 Console.WriteLine("In Main after await");
8             }).GetAwaiter().GetResult();//and wait for it to finish
9         Console.WriteLine("We can finish here");
10        Console.ReadLine();
11    }
12 public static async Task MainAsync(string[] args)
13     ...
14
```

Subcontractor async await

An example of a method using async await.

Listing 24: An example of an asynchronous method that uses await

```
1 public async Task WorkOrderedAsync(int howMuch)
2     {
3         Console.WriteLine($"Worker {Name} has ordered job...");
4         await PracaAsync(howMuch);
5         Console.WriteLine($"Worker {Name} received ordered job (used await)");
6     }
7
```

Subcontractor async await

An example of calling a method using async await.

Listing 25: Invoking an async method that uses await

```
1 Worker worker5 = new Worker("Stefan");  
2 Console.WriteLine("We are ordering the job to the worker Stefan and he's ordering the job ↵  
   farther");  
3 var stefanTask = worker5.WorkOrderedAsync(10);  
4 Console.WriteLine("Work for Stefan was commissioned and he commissioned it further.\nStefan↵  
   returned the task so we can wait for it with await");  
5 await stefanTask;
```


We are waiting for tasks

An example of waiting for all tasks

Listing 26: An example of waiting for tasks (SimpleTasks)

```
1      var tasks = new Task[3];
2      for (var i = 0; i < 3; i++)
3      {
4          tasks[i] = (Task.Run(() =>
5              {
6                  Console.WriteLine("One of 3 is waiting");
7                  Thread.Sleep(i * 500);
8                  Console.WriteLine("One of 3 has finished");
9              }));
10     }
11     Console.WriteLine("Waiting for all 3");
12     Task.WaitAll(tasks);
13     Console.WriteLine("We got all 3");
14     var resultTasks = Task.WhenAll(tasks); //Returns the task with everyone
15     // var resultTasks = Task.WhenAll(tasks).Result; //for that to be the case, they have to
16     return something
```

Wrong passing parameter

Wrong passing data to the task.

Listing 27: Example of wrong parameter passing

```
1  var tasks2 = new Task<int>[3];
2  for (var i = 0; i < 3; i++)
3      {
4          tasks2[i] = (Task<int>.Run(() =>
5              {
6                  Thread.Sleep(i * 500);
7                  return i * 2; //Don't do that!!!
8              }));
9      }
10  var resultTasks2 = Task.WhenAll(tasks2);
11  foreach (var item in resultTasks2.Result)
12      {
13      Console.WriteLine($"Task count and return wrong number: {item}");
14      }
15
```

We get the same value because we're using the *i* variable, which ends up being 3

Correct passing parameter

Data passed to the task via the parameter.

Listing 28: An example of a correct parameter passing

```
1  var tasks3 = new Task<int>[3];
2      for (var i = 0; i < 3; i++)
3      {
4          tasks3[i] = (new Task<int>((o) =>
5              {
6                  Thread.Sleep((int)o * 500);
7                  return (int)o * 2;
8              }, i));
9      }
10     tasks3.ToList<Task>().ForEach((t) => t.Start());
11     var resultTasks3 = Task.WhenAll(tasks3);
12     resultTasks3.Result.ToList().ForEach((i) => Console.WriteLine($"Task count and ←
13     return: {i}"));
```

We get the next values because *i* is passed by the parameter.

ContinueWith

An example of a simple continuation

Listing 29: An example of a simple continuation

```
1 Task firstTask = Task.Factory.StartNew(() =>
2 {
3     Console.WriteLine("First task has started");
4     Thread.Sleep(1000);
5     Console.WriteLine("First task has finished");
6 }
7 );
8 Task secondTask = firstTask.ContinueWith(ant =>
9 {
10    Console.WriteLine("Second task has started");
11    Thread.Sleep(1000);
12    Console.WriteLine("Second task has finished");
13 }
14 );
15 await secondTask;
```

ContinueWith and parameter

An example of a task that continues its calculations after its predecessor

Listing 30: An example of the continuation of calculations

```
1 var intTask = Task.Run(() =>
2     {
3         Thread.Sleep(1000);
4         return 12;
5     });
6 var continueTask = intTask.ContinueWith((x) => { Thread.Sleep(1000); return x.Result / 2; });
7 Console.WriteLine($"We have got result = {intTask.Result}");
8 Console.WriteLine($"We have got result2 = {continueTask.Result}");
```

intTask is a task that returns some number (12), that result is passed to the next *continueTask* task.

ContinueWith and parameter, shorter notation

The previous example can be written to a single command using dotted notation.

Listing 31: An example of the continuation of calculations v2

```
1 int result = Task.Run(() =>
2     {
3         Thread.Sleep(1000);
4         return 12;
5     }).ContinueWith((x) => { Thread.Sleep(1000); return x.Result / 2; })
6     .Result;
7
```

Continuation with the exception

Listing 32: Continuation with the exception

```
1 Task<int> task1ex = Task.Factory.StartNew<int>(() => { throw new Exception("The first one ←  
    threw an exception"); });  
2 //Task<int> task2ex = task1ex.ContinueWith<int>(ant => Console.WriteLine("Exception: " +  
    ant.Exception.Message); return 0; );  
3 //A safe pattern states that we should forward an exception to the place where we expect the  
    result  
4 Task<int> task2ex = task1ex.ContinueWith<int>(ant => { if (ant.Exception != null) throw ant.←  
    Exception; return 0; });  
5 try  
6 {  
7     Console.WriteLine($"Result of task2ex: {task2ex.Result}");  
8 }  
9 catch (AggregateException ae)  
10 {  
11     Console.WriteLine("AggregateException has message: " + ae.Message);  
12     foreach (var ex in ae.InnerExceptions)  
13     {  
14         Console.WriteLine("I caught exception: " + ex.Message);  
15         Console.WriteLine("Inner exception:" + ex.InnerException.Message);  
16     }  
17 }
```

Different paths for exceptions

Listing 33: Example of different paths

```
1 Task<int> task1r = Task.Factory.StartNew<int>(() => {
2     //uncomment to simulate error
3     //throw new Exception("The first trew an exception");
4     return 1;
5 });
6 Task<int> taskErr = task1r.ContinueWith<int>(ant => { Console.WriteLine("Exception: " + ant.↵
7     Exception.Message); return 1; },TaskContinuationOptions.OnlyOnFaulted);
8 Task<int> taskNotErr = task1r.ContinueWith<int>(ant => { Console.WriteLine("There was no ↵
9     error"); return ant.Result+2; }, TaskContinuationOptions.NotOnFaulted);
10 try
11 {
12     if (taskErr != null) await taskErr;
13     if (taskNotErr != null) await taskNotErr;
14 }
```


Different paths for exceptions cont.

If the task start condition is not met, then such task is Canceled

Listing 34: Example of different paths

```
1 catch (AggregateException ae)
2 {
3     Console.WriteLine("AggregateException has message: " + ae.Message);
4     foreach (var ex in ae.InnerExceptions)
5     {
6         Console.WriteLine("I caught exception: " + ex.Message);
7         Console.WriteLine("Inner exception:" + ex.InnerException.Message);
8     }
9 }
10 catch (TaskCanceledException) //if the task start condition is not met, such task is Canceled
11 {
12     Console.WriteLine("The alternate task has been canceled");
13 }
```

Continuations and child tasks

An example of collecting exceptions from child process

Listing 35: Descendants throw exceptions

```
1 TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
2 Task TaskParent = Task.Factory.StartNew(() =>
3 {
4     Task.Factory.StartNew(() => { throw new Exception("my error 1"); }, atp);
5     Task.Factory.StartNew(() => { throw new Exception("my error 2"); }, atp);
6     Task.Factory.StartNew(() => { throw new Exception("my error 3"); }, atp);
7 })
8 .ContinueWith(p => {
9     Console.WriteLine("As a parent, I caught something like that: " + p.Exception);
10         throw p.Exception;
11 },
12         TaskContinuationOptions.OnlyOnFaulted);
13 try
14 {
15     await TaskParent;
16 }
```

Continuations and child tasks

Listing 36: Collecting exceptions

```
1 catch (AggregateException ae)
2 {
3     Console.WriteLine("AggregateException has message: " + ae.Message);
4     foreach (var ex in ae.InnerExceptions)
5     {
6         Console.WriteLine("I caught exception: " + ex.Message);
7         Console.WriteLine("Inner exception:" + ex.InnerException.Message);
8     }
9 }
10 catch (Exception ex)
11 {
12     Console.WriteLine("We have a general exception: ", ex.Message);
13 }
```

Conditional continuations

By default, a continuation is scheduled unconditionally — whether the antecedent completes, throws an exception, or is canceled. You can alter this behavior via a set of (combinable) flags included within the *System.Threading.TaskContinuationOptions* enum. The three core flags that control conditional continuation are:

- `NotOnRanToCompletion` = `0x10000`,
- `NotOnFaulted` = `0x20000`,
- `NotOnCanceled` = `0x40000`,

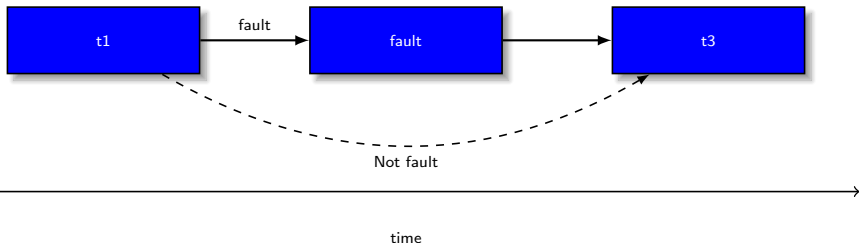
Conditional continuations

These flags are subtractive in the sense that the more you apply, the less likely the continuation is to execute. For convenience, there are also the following precombined values:

- `OnlyOnRanToCompletion` = `NotOnFaulted` | `NotOnCanceled`,
- `OnlyOnFaulted` = `NotOnRanToCompletion` | `NotOnCanceled`,
- `OnlyOnCanceled` = `NotOnRanToCompletion` | `NotOnFaulted`

Conditional continuations

The task *fault* runs only if there is an error in the task *t1*. The task *t3* runs unconditionally after *fault* or *t1*

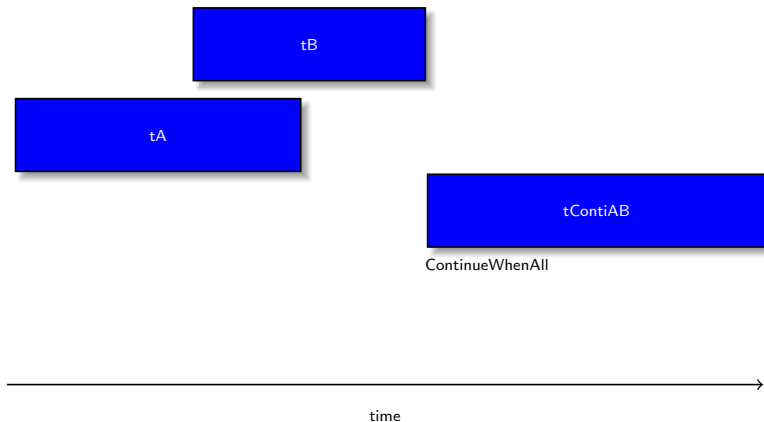


Conditional continuations

Listing 37: An example of conditional continuations

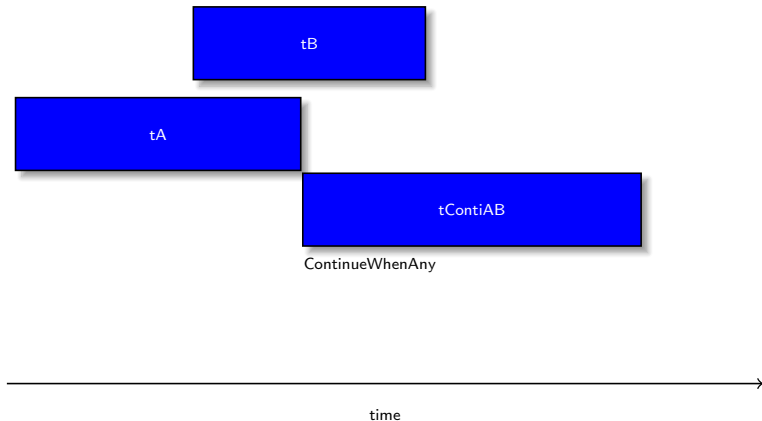
```
1 Task t1 = Task.Factory.StartNew(() =>
2     {
3         Console.WriteLine("t1 works but it will stop in a while");
4         //uncomment to simulate error
5         //throw new Exception("An error in t1");
6     }
7 );
8
9 Task fault = t1.ContinueWith(ant => Console.WriteLine("Task run in case of fault of t1"),
10     TaskContinuationOptions.OnlyOnFaulted);
11
12 Task t3 = fault.ContinueWith(ant => Console.WriteLine("t3 run after task fault"));
13 await t3;
```

A continuation with many predecessors



ContinueWhenAll

A continuation with many predecessors



A conditional continuation - WhenAll

Listing 38: An example of conditional continuation - WhenAll

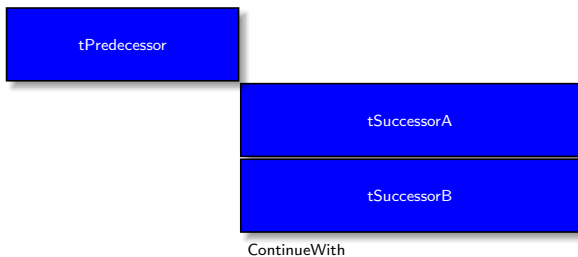
```
1 Task tA = Task.Factory.StartNew(() => Console.Write("A"));
2     Task tB = Task.Factory.StartNew(() => Console.Write("B"));
3     Task tContiAB = Task.Factory.ContinueWhenAll(new Task[] { tA, tB }, tasks =>
4     { Console.WriteLine("\nContinuation after A and B"); }
5     );
6     await tContiAB;
```

Continuation with multiple predecessors and values

Listing 39: n example of conditional continuation - WhenAll and values

```
1 Task<int> tRetA = Task.Factory.StartNew(() => { Console.WriteLine("A returns 11"); return 11; }
  });
2 Task<int> tRetB = Task.Factory.StartNew(() => { Console.WriteLine("B returns 22"); return 22; }
  });
3 Task<int> tRetAB = Task.Factory.ContinueWhenAll(new Task<int>[] { tRetA, tRetB }, tasks =>
4 { Console.WriteLine($"Continuation after task A returning {tasks[0].Result} i and task B
  returning {tasks[1].Result}");
5   return tasks.Sum(t => t.Result); }
6 );
7 Console.WriteLine("Sum from task A and B: " + tRetAB.Result);
```

continuation with many successors



Continuation with many successors

Listing 40: An example of continuation by many successors

```
1 Task tPredecessor = Task.Factory.StartNew(() => Console.WriteLine("Predecessor"));
2 Task tSuccessorA = tPredecessor.ContinueWith(ant =>
3 {
4     Console.WriteLine("\nContinuation A after predecessor");
5 });
6 Task tSuccessorB = tPredecessor.ContinueWith(ant =>
7 {
8     Console.WriteLine("\nContinuation B after predecessor");
9 });
10 await tSuccessorA;
11 await tSuccessorB;
```

Task Scheduler

- The scheduler assigns tasks to threads
- All tasks are associated with the scheduler, which is represented by the abstract *TaskScheduler* class
- The framework provides two implementations
 - Default scheduler that works with the CLR thread pool
 - Synchronization context scheduler, designed primarily to help with the WPF and WinForms threading model where UI elements are only accessible from the thread that created it.

Task Scheduler

- For example, we have a function that returns data for a long time, e.g. calling a webservice or a calculation method.
- After receiving the data, we want to display that data in one of the controls.
- A continuation task of the retrieval task will do.
- The continuation task will have the indicated scheduler context obtained from the window where the given control is embedded
- This way it can be safely updated.

Task Scheduler and UI

Listing 41: Example of Scheduler and UI

```
1 TaskScheduler _uiScheduler;
2 public MainWindow()
3 {
4     InitializeComponent();
5     progress = new Progress<int>(percent =>
6     {
7         progressBar.Value = percent;
8     });
9     _uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
10 }
11 private string SomeMethodLongReturningData()
12 {
13     Thread.Sleep(5000); return "We have it...";
14 }
15 }
```


Task Scheduler and UI

Listing 42: Example of Scheduler and UI

```
1 TaskScheduler _uiScheduler;
2 public MainWindow()
3 {
4     InitializeComponent();
5     _uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
6 }
7 private string SomeMethodLongReturningData()
8 {
9     Thread.Sleep(5000); return "We have it...";
10 }
11
12 private void GoBT_Click(object sender, RoutedEventArgs e)
13 {
14     var client = new HttpClient();
15     Task.Factory.StartNew<string>(JakasMetodaCoDlugoZwracaDane)
16         .ContinueWith(ant =>
17             {
18                 contentView.AppendText(ant.Result);
19             }, _uiScheduler);
20 }
```

Questions

?

THE END

Thank You.