

Using the Task Parallel Library for Asynchronous Programming



Filip Ekberg

Principal Consultant & CEO

@fekberg fekberg.com



Introducing the Task

```
Task.Run(() => {  
    // Heavy operation to run somewhere else  
});
```

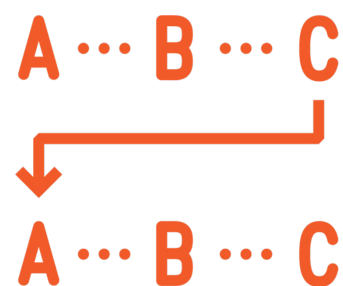
Using Tasks without **async** & **await**



Obtain the result



Capture exceptions



Running continuations depending on success or failure



Cancelling an asynchronous operation



Read File Content Asynchronously

```
using var stream =  
    new StreamReader(File.OpenRead("file")));  
  
var fileContent = await stream.ReadToEndAsync();
```

Using the Task

```
var response = await client.GetAsync(URL);
```

Result of
the operation



Awaits the Task



Returns a Task

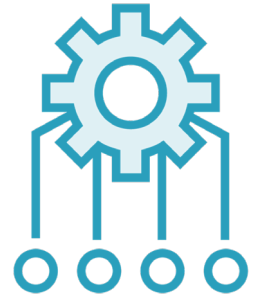


Task from the Task Parallel Library

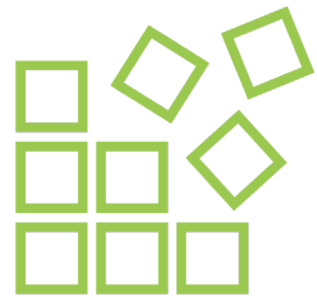
Represents a single asynchronous operation



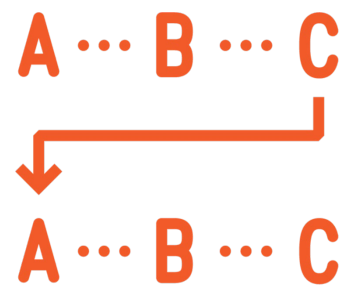
Functionality Provided by the **Task**



Execute work on a different thread



Get the result from the asynchronous operation



Subscribe to when the operation is done by introducing a continuation



It can tell you if there was an exception



Introducing the Task

```
Task.Run(() => { /* Heavy operation */ });
```

```
Task.Run(SomeMetodMethod);
```


Introducing the Task

Queue this anonymous method
on the thread pool for execution



```
Task.Run(() => { /* Heavy operation */ });
```

```
Task.Run(SomeMetodMethod);
```



Queue this method
on the thread pool for execution

Generic vs Non-Generic Task.Run

```
Task<T> task = Task.Run<T>(() => {  
    return new T();  
});
```

```
Task task = Task.Run(() => { });
```

Generic vs Non-Generic Task.Run

```
Task<T> task = Task.Run<T>(() => {  
    return new T();  
});
```

**An asynchronous operation
that returns a value**



```
Task task = Task.Run(() => { });
```

Generic vs Non-Generic Task.Run

Don't need to explicitly
use Task.Run<T>()



```
Task<T> task = Task.Run(() => {  
    return new T();  
});
```

```
Task task = Task.Run(() => { });
```

Avoid queuing
heavy work back
on the **UI thread**



Obtaining the Result of a Task



Introduce a Continuation

```
var task = Task.Run(() => { });
```

```
var continuationTask =  
    task.ContinueWith((theTaskThatCompleted) => {  
  
        // This is the continuation  
        // which will run when "task" has finished  
  
    });
```

Introduce a Continuation

```
var task = Task.Run(() => { });
```

```
var continuationTask =  
    task.ContinueWith((theTaskThatCompleted) => {  
  
});
```



This continuation will **NOT**
execute on the original thread

Introduce a Continuation

```
var task = Task.Run(() => { });
```

These two are the same!



```
task.ContinueWith((theTaskThatCompleted) => {
```

```
    // This is the continuation
```

```
});
```

Multiple Continuations

```
var task = Task.Run(() => { });
```

```
task.ContinueWith((t) => { /* Continuation 1 */ });  
task.ContinueWith((t) => { /* Continuation 2 */ });  
task.ContinueWith((t) => { /* Continuation 3 */ });  
task.ContinueWith((t) => { /* Continuation 4 */ });  
task.ContinueWith((t) => { /* Continuation 5 */ });
```

async & await is a much **more**
readable and **maintainable**
approach



Continuation Differences

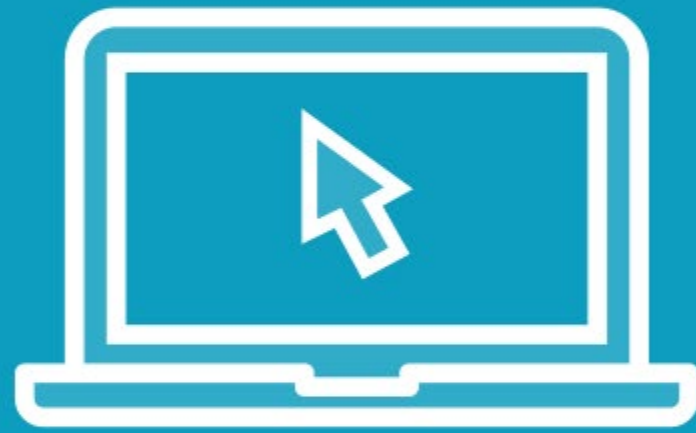
```
task.ContinueWith(_ => {  
    // This continuation executes asynchronously  
    // on a different thread  
});  
  
await task;  
// This continuation executes on the original context
```



async & **await** may be
unnecessary in **certain**
situations



Demo



Demo: Nested asynchronous operations



Asynchronous Anonymous Methods

```
// Thread 1
```

```
Task.Run(async () => {  
    // Thread 2  
  
    await Task.Run(() => {  
        // Thread 3  
    });  
});
```

```
    // Thread 2  
});
```

```
// Thread 1
```

Asynchronous anonymous
methods are **NOT** the same
as **async void**



Next: Handling Task Success and Failure



Handling Task Success and Failure



Continuing After an Exception

```
var loadLinesTask = Task.Run(() => {  
    throw new FileNotFoundException();  
});  
  
loadLinesTask.ContinueWith((completedTask) => {  
    // Running this may be unnecessary  
    // if you expect completedTask.Result!  
});
```

ContinueWith **executes** when
the **Task completes** no
matter if it's **successful**,
faulted or **cancelled**



The Continuation Did Not Fail

```
Task.Run(() => {  
    throw new FileNotFoundException();  
})  
.ContinueWith((completedTask) => {  
  
})  
.ContinueWith((completedContinuationTask) => {  
  
})
```

 **Faulted with attached exception!**

 **Not faulted!**

OnlyOnRanToCompletion

Task has no exceptions

Task was not cancelled



await it will **not throw** an
aggregate exception



Always Validate Your Tasks



You can use **async** & **await**



You can chain a continuation
using **ContinueWith**



TaskContinuationOptions

Specifies the behavior for a task that is created by using the **ContinueWith**



Continuing After an Exception

```
var loadLinesTask = Task.Run(() => {  
    throw new FileNotFoundException();  
});
```

```
loadLinesTask.ContinueWith((completedTask) => {  
    // will always run  
});
```

```
loadLinesTask.ContinueWith((completedTask) => {  
    // will not run if completedTask is faulted  
}, TaskContinuationOptions.OnlyOnRanToCompletion);
```

Always validate your
asynchronous operations



Handling Exceptions

```
try
{
    await task;
}
catch(Exception ex)
{
    // log ex.Message
}

task.ContinueWith((t) => {
    // log ex.InnerException.Message
}, TaskContinuationOptions.OnlyOnFaulted);
```

Next: Cancellation and Stopping a Task



Cancellation and Stopping a Task



Don't force a user to **wait** for
a **result** they **know** is
incorrect.

Allow them to cancel!



CancellationTokenSource

Signals to a **CancellationToken** that it should be canceled.

Always call **Dispose()** on the cancellation token source after the asynchronous operation completed



Cancellation Token Source

```
CancellationTokenSource cancellationTokenSource;
```

Cancellation Token Source

```
CancellationTokenSource cancellationTokenSource;
```

```
cancellationTokenSource.Cancel();  
cancellationTokenSource.Dispose();
```



**Signals to a Cancellation Token
that it should cancel**

Cancellation Token Source

```
CancellationTokenSource cancellationTokenSource;
```

```
cancellationTokenSource.Cancel();
```

```
cancellationTokenSource.CancelAfter(5000);
```

```
...
```

```
cancellationTokenSource.Dispose();
```



Schedules a cancellation that occurs after 5 seconds

Cancellation Token

```
CancellationTokenSource cancellationTokenSource;  
CancellationToken token = cancellationTokenSource.Token;
```

```
Task.Run(() => {}, token);
```

Cancellation Token

```
CancellationTokenSource cancellationTokenSource;  
CancellationToken token = cancellationTokenSource.Token;
```

```
Task.Run(() => {}, token);
```

```
Task.Run(() => {  
    if(token.IsCancellationRequested) {}  
});
```

Calling **Cancel** will **not**
automatically terminate the
asynchronous **operation**



Cancellation

```
CancellationTokenSource cancellationTokenSource;  
CancellationToken token = cancellationTokenSource.Token;
```

```
cancellationTokenSource.Cancel();
```

```
Task.Run(() => {}, token);
```

**Will not start if Cancellation
Token is marked as Cancelled**



Cancellation Token and ContinueWith

```
CancellationTokenSource cancellationTokenSource;  
CancellationToken token = cancellationTokenSource.Token;
```

```
var task = Task.Run(() => {}, token);  
task.ContinueWith((t) => {}, token);
```


Task Status

```
var token = new CancellationToken(canceled: true);

var task = Task.Run(() => "Won't even start", token);

task.ContinueWith((completingTask) => {
    // completingTask.Status = Canceled
});

task.ContinueWith((completingTask2) => {
    // completingTask2.Status = Canceled
});
```

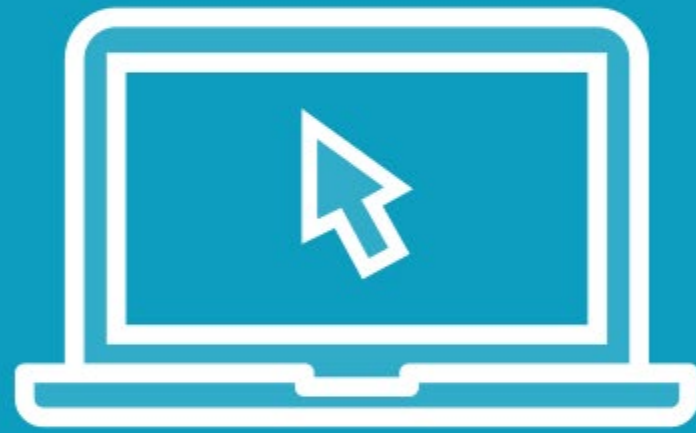
Task Status

```
var token = new CancellationToken(canceled: true);

var task = Task.Run(() => "Won't even start", token);

task.ContinueWith((completingTask) => {
    // completingTask.Status = Canceled
})
.ContinueWith((continuationTask) => {
    // continuationTask.Status = RanToCompletion
});
```

Demo



Example: Cancellation with HttpClient



Every **library** could handle
cancellations differently



Task Parallel Library

```
async Task Process(CancellationToken token)
{
    var task = Task.Run(() => {

        // Perform an expensive operation

        return ... ;

    }, token);

    var result = await task;

    // Use the result of the operation

}
```



ContinueWith

```
var task = Task.Run(() => {  
    return ... ;  
});  
  
task.ContinueWith((completedTask) => {  
    // Continue..  
});
```



ContinueWith

```
var task = Task.Run(() => {
```

```
    return ... ;
```

```
});
```

```
task.ContinueWith((completedTask) => {
```

```
    // Continue..
```

```
});
```

**Asynchronous operation
executed on a different thread**



Cross-Thread Communication

```
var task = Task.Run(() => {  
    return ... ;  
});  
  
task.ContinueWith((completedTask) => {  
    Dispatcher.Invoke(() => { /* Run me on the UI */ });  
});
```



Be careful!

What happens if the method
you point to forces itself onto
the UI/calling thread?



Introducing Asynchronous Methods



Implement two versions of the method if you need both an asynchronous and synchronous version



Do not wrap the synchronous method in a Task.Run just to make the code asynchronous. Copy the code to the asynchronous method and implement it properly



Task Continuation Options

```
var task = Task.Run(() => {  
    throw new FileNotFoundException();  
});  
  
task.ContinueWith((completedTask) => {  
    // will not run if completedTask is faulted  
}, TaskContinuationOptions.OnlyOnRanToCompletion);
```



Summary



Introducing a Task with Task.Run to run work on a different thread

Obtaining the result and exceptions in the continuation of a Task

Configure the continuation to only run on success, failure or a cancellation

How to combine async and await with your own asynchronous operations

Understand the difference between await and ContinueWith



Next: Exploring Useful Methods in the Task Parallel Library

