

Module 08:

"Async, Await, and Task Combinators"



TEKNOLOGISK
INSTITUT

Agenda

- ▶ **Async and Await**
- ▶ Synchronization Context
- ▶ More Task Combinators
- ▶ Concluding Remarks

C# 5.0 await Operator

- ▶ C# 5.0 introduces **await** keyword for methods returning **Task** or **Task<T>**
 - Yields control until awaited task completes
 - Results gets returned
- ▶ Allows you to program just like for synchronous programming...!

```
WebClient client = new WebClient();  
string result = await client.DownloadStringTaskAsync( ... );  
  
Console.WriteLine( result );
```

- ▶ Really complex control flow under the hood is made stunningly simple by compiler

C# 5.0 async Modifier

- ▶ C# 5.0 introduces **async** keyword
 - Marks method or lambda as asynchronous
 - Note: Methods making use of **await** must be marked "**async**"
- ▶ You can now easily define your own asynchronous methods

```
async static void DoStuff()  
{  
    // ...  
  
    string result = await client.DownloadStringTaskAsync( ... );  
  
    // ...  
}
```

- ▶ Can create async methods returning **void**, **Task**, or **Task<T>**

Best Practices for Task Methods

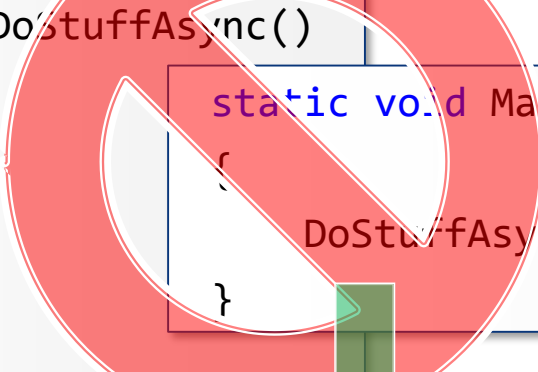
- ▶ Microsoft recommends that the name of methods returning Task or Task<T> should be postfixed with ...Async
 - Regardless of whether it is marked with async modifier...!

```
async Task<string> DoStuffAsync()  
{  
    // ...  
  
    string result = await client.DownloadStringTaskAsync( ... );  
  
    return result;  
}
```

```
Task<string> GetSimpleAsync()  
{  
    return Task.CompletedTask; // <-- We will see this later  
}
```

C# 7.1 Allows Main to be Async

```
static async Task DoStuffAsync()  
{  
    ... await ...  
    ... await ...  
    ... await ...  
}  
  
static void Main(string[] args)  
{  
    DoStuffAsync().GetAwaiter().GetResult();  
}
```



```
static async Task<int> Main( string[] args )  
{  
    ... await ...  
}  
int $GeneratedMain( string[] args )  
{  
    return Main(args).GetAwaiter().GetResult();  
}
```

Exceptions Thrown by Tasks and Awaitable Methods

- ▶ Observe and catch exceptions “as usual” when awaiting tasks

```
try
{
    string data = await client.DownloadStringTaskAsync( ... );
}
catch ( WebException ex ) { ... }
```

- ▶ Note that
 - **Task.WaitXxx()** throws an **AggregateException**
 - **Task.Result** throws an **AggregateException**
 - Awaiting a **Task** throws exceptions “as usual”, however!

Agenda

- ▶ Async and Await
- ▶ **Synchronization Context**
- ▶ More Task Combinators
- ▶ Concluding Remarks

What is a SynchronizationContext?

- ▶ Context handling synchronization of (a)synchronous operations
 - In general a many-to-many relationship with threads

```
public class SynchronizationContext
{
    public virtual void OperationCompleted() { ... }
    public virtual void OperationStarted() { ... }
    public virtual void Post(SendOrPostCallback d, object state)
    {
        // Perform operation asynchronously
    }
    public virtual void Send(SendOrPostCallback d, object state)
    {
        // Perform operation synchronously
    }
}
```

Built-in SynchronizationContexts

▶ **WindowsFormsSynchronizationContext**

- Executes on a specific UI thread
- Executes in the order they were queued.

▶ **DispatcherSynchronizationContext**

- Queues delegates to a specific UI thread with **Normal** priority.
- Executes in the order they were queued
- Installed as current context by **Dispatcher.Run()**

▶ **Default (Thread Pool) SynchronizationContext**

- if a thread's current Synchronization Context is null, then it implicitly has this default Synchronization Context.
- Queues its asynchronous delegates to the Thread Pool but executes its synchronous delegates directly on the calling thread.

Await and SynchronizationContext

- ▶ Await captures the current **Synchronization Context**
 - Essential and very helpful for WPF and WinForms

```
// DispatcherSynchronizationContext here in WPF
```

```
string result = await FactorAsync();  
lblResult.Content = result;
```

```
// Also DispatcherSynchronizationContext here!
```

Not "Thread"!

ConfigureAwait()

- ▶ By default execution continues on the current Synchronization Context after **await**
- ▶ Optionally, this requirement can be manually relaxed by **Task.ConfigureAwait(false)**

```
// DispatcherSynchronizationContext here in WPF

string result = await FactorAsync().ConfigureAwait( false );
lblResult.Content = result;

// Not DispatcherSynchronizationContext here!
```

Agenda

- ▶ Async and Await
- ▶ Synchronization Context
- ▶ **More Task Combinators**
- ▶ Concluding Remarks

Basic Tasks

- ▶ You can form constant tasks synchronously

```
Task Method1Async() => Task.CompletedTask;  
  
Task<DateTime> Method2Async() => Task.FromResult(DateTime.Now);  
  
Task<DateTime> Method3Async(  
    CancellationToken cancellationToken  
) =>  
    Task.FromCanceled<DateTime>(cancellationToken);  
  
Task Method4Async() =>  
    Task.FromException(new NotImplementedException("Oops"));
```

More Task Combinators

- ▶ Combinators also include
 - **Task.WhenAll()** Completes when all tasks have completed
 - **Task.WhenAny()** Completes when any of the tasks completes
 - **Task.Delay()** Completes after a specified time span
 - + more

- ▶ You can also write your own

Task.Delay()

- ▶ **Task.Delay()** completes after a specified time span

```
await Task.Delay(3000);
```

- ▶ The Task-equivalent of **Thread.Sleep()**

Task.WhenAll()

- ▶ **Task.WhenAll()** completes when all tasks have completed

```
Task<string[]> all = Task.WhenAll(  
    FactorAsync(87),  
    FactorAsync(112),  
    FactorAsync(176)  
);  
string[] results = await all;
```

```
Task<string> FactorAsync(int number) { ... }
```

- ▶ There is also an overload for plain Tasks

Task.WhenAny()

- ▶ **Task.WhenAny()** completes when any of the tasks completes
- ▶ Returns the task which is completed

```
List<Task<string>> remaining = new List<Task<string>>
{ ... };

while( remainingTasks.Any() )
{
    Task<string> completedTask = await Task.WhenAny(remaining);
    Console.WriteLine(completedTask.Result);

    remainingTasks.Remove(completedTask);
}
```

TaskCompletionSource<T>

- Any occurrence or computation can be transformed into a **Task<T>** using **TaskCompletionSource<T>**

```
public partial class Form1 : Form
{
    private readonly TaskCompletionSource<DateTime> _tcs =
        new TaskCompletionSource<DateTime>();
    ...
    async private void OnClick(object sender, EventArgs e)
    {
        DateTime dt = await _tcs.Task;
        ...
    }
    private void OnMouseEnter(object sender, EventArgs e)
    {
        _tcs.TrySetResult(DateTime.Now);
    }
}
```

Agenda

- ▶ Async and Await
- ▶ Synchronization Context
- ▶ More Task Combinators
- ▶ **Concluding Remarks**

Three Approaches to Asynchrony

- ▶ Synchronous calls
 - **Xxx()** methods
- ▶ .NET Asynchronous Programming Model (APM) consisting of
 - **BeginXxx()** methods
 - **EndXxx()** methods
- ▶ Event-based Asynchronous Pattern (EAP) consisting of
 - **XxxAsync()** methods
 - **XxxCancelAsync()** methods
 - **XxxCompleted** events
- ▶ Task-based Asynchronous Pattern
 - **XxxAsync()** or **XxxTaskAsync()** methods

Tasks and Asynchronous Programming Model

- ▶ The “traditional” .NET Asynchronous Programming Model consists of
 - **BeginXxx()** methods
 - **EndXxx()** methods
- ▶ Tasks encapsulate this model using **TaskFactory.FromAsync()**

```
HttpWebResponse response =  
    await Task<WebResponse>.Factory.FromAsync(  
        request.BeginGetResponse,  
        request.EndGetResponse,  
        request )  
    as HttpWebResponse;
```

When to Use What?

- ▶ Thread
 - Avoid if possible!
 - Only for "eternal" processing
- ▶ ThreadPool
 - Use for very quick, small, unordered computations
 - Usually callbacks
- ▶ Task
 - Use for "task parallelism": computational independence or I/O-bound work
- ▶ Parallel
 - Use for "data parallelism": processing sets of independent data

Summary

- ▶ Async and Await
- ▶ Synchronization Context
- ▶ More Task Combinators
- ▶ Concluding Remarks



WINCUBATE

Jesper Gulmann Henriksen

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : jgh@wincubate.net

WWW : <http://www.wincubate.net>

Ringgårdsvej 4A

8270 Højbjerg

Denmark