Fachhochschule Nordwestschweiz
Hochschule für Technik

CONCURRENCY

ASYNCHRONOUS PROGRAMMING

Martin Kropp, Yves Senn
University of Applied Sciences Northwestern Switzerland

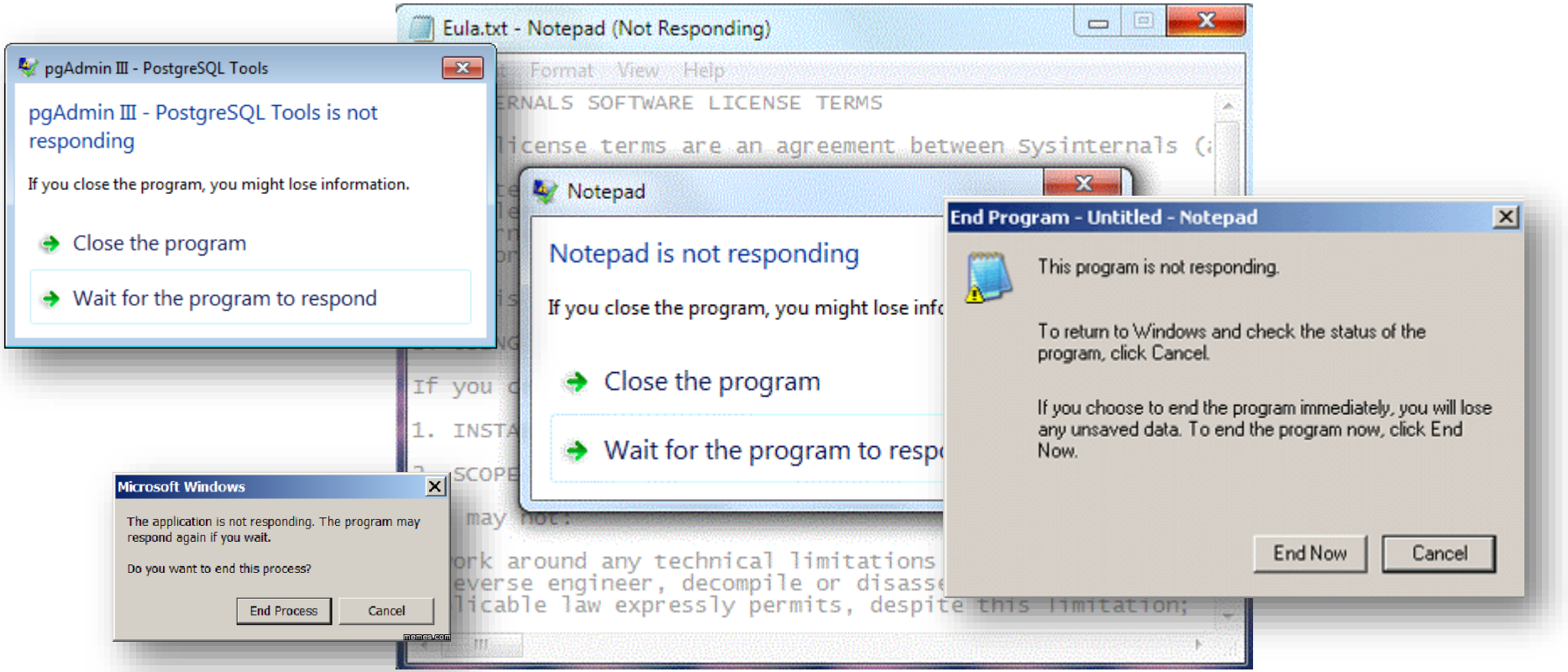
Learning Targets

- You
  - can explain the .NET concurrency and threading model
  - can know and explain how `await`/`async` work internally
  - can apply .NET asynchronous programming concepts for concurrent programming efficiently

Content

- The Concurrency Problem
- Task/Async/Await Concepts
- Task-based Asynchronous Pattern

Problem: Responsive applications



→ Threads?

The Responsiveness Problem

- I/O-bound tasks
  - CPU is mostly just waiting for data (I/O)
  - Examples:
    - Searching for files on a hard disk
    - Downloading files from the internet

- Compute-bound tasks
  - CPU is heavily occupied
  - Examples:
    - Searching for prime numbers
    - Running Machine Learning algorithms

The Responsiveness Problem

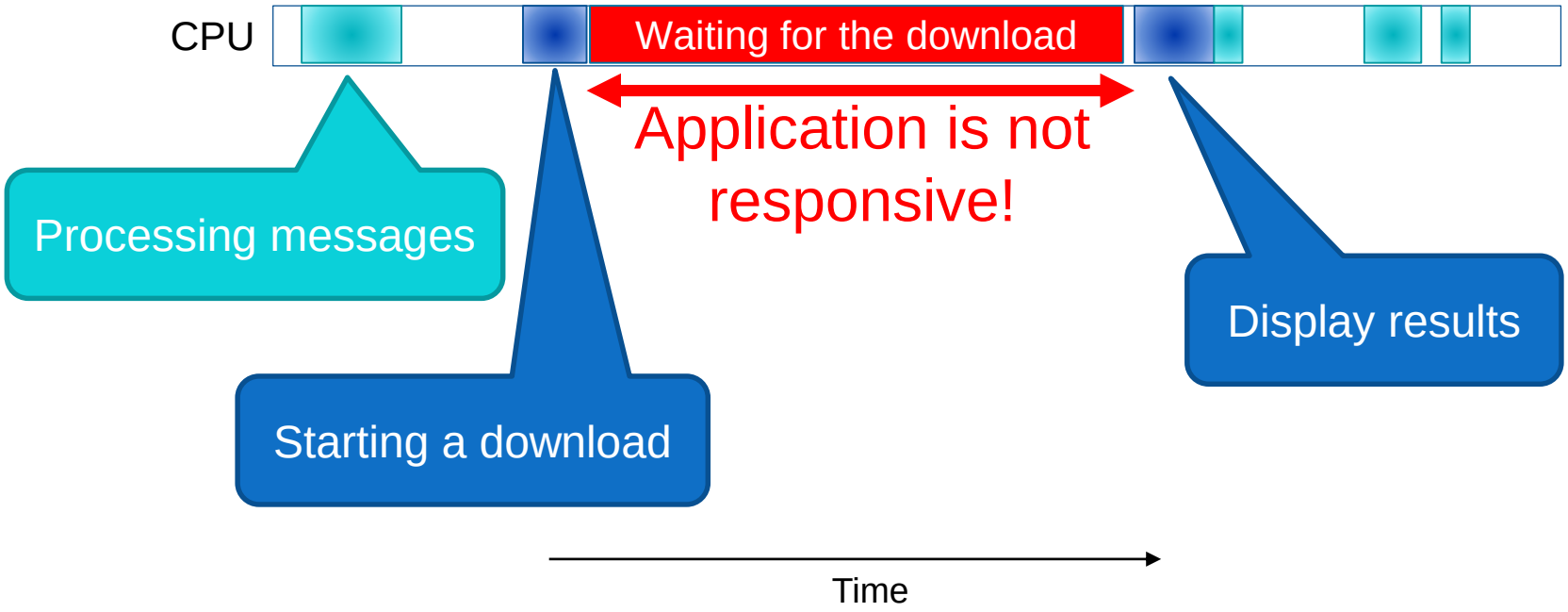
Instead of taking care of the UI, the CPU is occupied with waiting or calculating things:

```
//start download
var html = new WebClient().DownloadString("http://www.fhnw.ch");



//display results
output.Text = html;
```

`DownloadString` is a blocking call
→ Application is not responsive

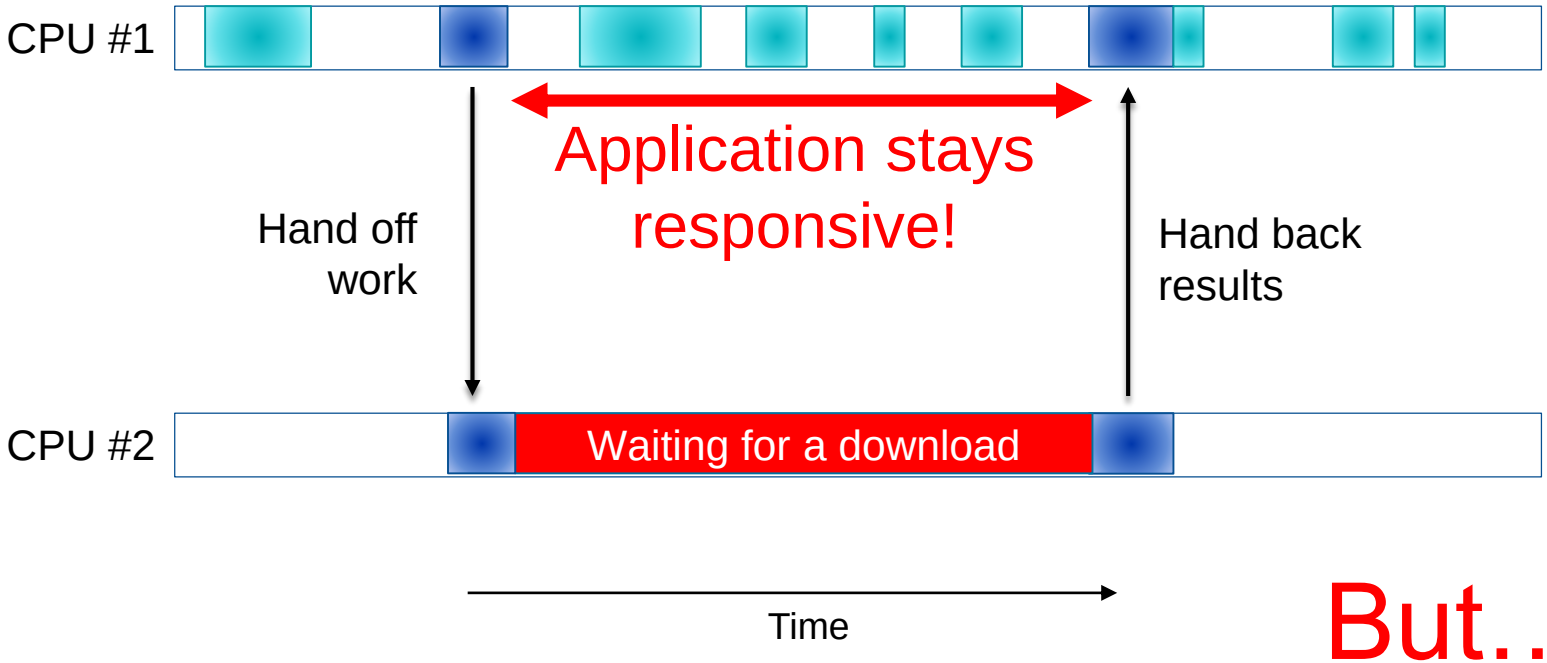
The Responsiveness Problem

Instead of taking care of the UI, the CPU is occupied with waiting or calculating things:



For Better Responsiveness

Free from active waiting:

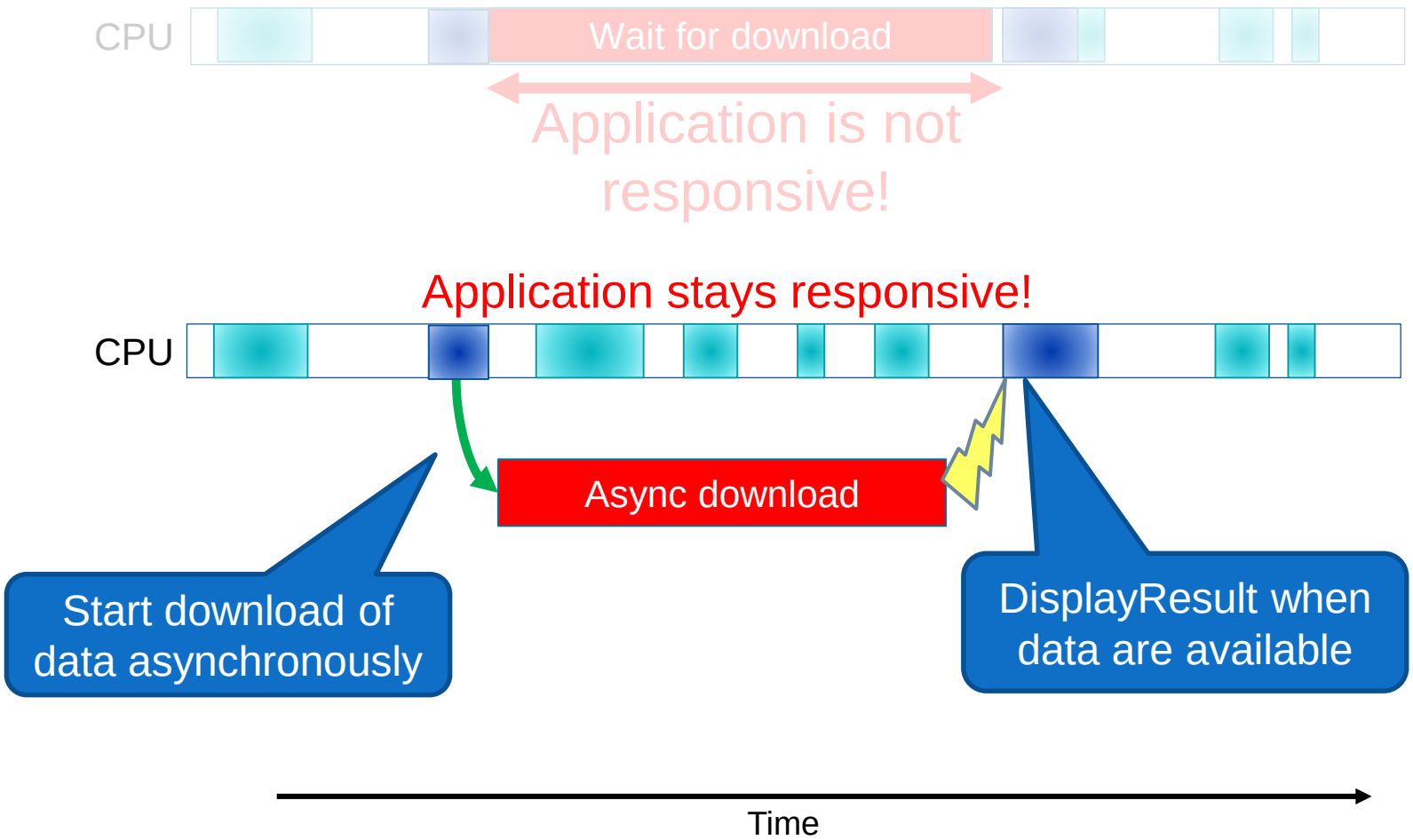


But…

Asynchronous programming

“Performing things nearly simultaneously”

Scenario: Downloading a file



Scenario: Downloading a file in .NET

Synchronous
Blocks application while waiting for result

```
string Download(..);
```

```
void button1_Clicked()
{
  var f = Download("http://foo/bar.txt");
  Process(f);
}
```

Asynchronous
Returns immediately, raises events or uses callbacks (delegates) when results arrive

```
void DownloadAsync(…,
                   .. Callback);
```

```
void button1_Clicked()
{
  DownloadAsync("http://foo/bar.txt",
      (f) => Process(f));
}
```

Download Web Page: Synchronous

```
static void Test_Sync()
{
  var w = new WebClient();



  var html = w.DownloadString(new Uri("http://www.fhnw.ch/"));

  Console.WriteLine(html);
}
```

The synchronous, blocking call

Download Web Page: Asynchronous1

```
static void TestAsync()
{
  var w = new WebClient();

  w.DownloadStringCompleted += (sender,e) =>
                               { Console.WriteLine(e.Result); };

  w.DownloadStringAsync(new Uri("http://www.fhnw.ch/"));

//Console.WriteLine(html);
}
```

Attaching an event-handler

The asynchronous, non-blocking call

Task/async/await

The C# asynchronous concepts to avoid spaghetti callback code

Example 1: Async IO Bound Task

```
static        void Test()
{
  var w = new WebClient();

       string  t = w.DownloadString         (
                        "http://www.fhnw.ch/");
  /// more to do
  DoOtherStuff();

  Console.WriteLine(      t);
}
```

Example 1: Async IO Bound Task

```
static void Test()
{
  var w = new WebClient();

  string t = w.DownloadString(
                "http://www.fhnw.ch/");
  /// more to do
  DoOtherStuff();

  Console.WriteLine(t);
}
```

Step #1: Use Async Method Version

```
static void Test()
{
  var w = new WebClient();

  string t = w.DownloadStringTaskAsync(
                  "http://www.fhnw.ch/");
  /// more to do
  DoOtherStuff();

  Console.WriteLine(t);
}
```

Step #2: Create Task Object

```
static void Test()
{
  var w = new WebClient();

  Task<string> t = w.DownloadStringTaskAsync(
      "http://www.fhnw.ch/");
  /// more to do
  DoOtherStuff();

  Console.WriteLine(t);
}
```

Step #3: await the results

```
static void Test()
{
  var w = new WebClient();

  Task<string> t = w.DownloadStringTaskAsync(
      "http://www.fhnw.ch/");
  /// more to do
  DoOtherStuff();

  Console.WriteLine(await t);
}
```

Step #4: Tell the Compiler

```
static async Task Test()
{
  var w = new WebClient();

  Task<string> t = w.DownloadStringTaskAsync(
      "http://www.fhnw.ch/");
  /// more to do
  DoOtherStuff();

  Console.WriteLine(await t);
}
```

What is `Task/async/await`?

- `Task<int> t;` represents a concurrent operation that will eventually return a value
- `int x = await t;` waits for the result from the task in a **non-blocking** way (i.e. other code runs while waiting).
- `async` is a keyword required to mark methods that use the `await` operator

Example 2: Async CPU-bound Task

```
private static void Main()
{
    DoLongRunningTask();
    DoSomeForegroundWork();
    ...
}

private void DoLo
{
    var su
    Co
}

void ComputeValue(int start, int count)
    // takes a looooooong time
    Enumerable.Range(start, count).Count(n => Enumerable.
        Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0));
}
```

Now let's do it asynchronously

Step #1: Asynchronize Method

"asynchronize" computation using Task class:

```
private Task<int> ComputeValueAsync(int start, int count)
{
    var resultAsTask = Task.Run(() =>
                         ComputeValue(start, count));

    return resultAsTask;
}
```

Step #2: Await the result

`await` the result from the asynchronous call:

```
var sum = await ComputeValueAsync(1, 1000);
Console.WriteLine($"Result (asynchronously): {sum}");
```

Step #3: Mark method

Mark the awaiting method with `async`:

```
public async Task<int> DoLongRunningTask()
{
    var sum = await ComputeValueAsync(1, 1000);
    Console.WriteLine($"Result (asynchronously): {sum}");
    return sum;
}
```

Step #4: Nothing more

- `DoLongRunningTask` is now executed asynchronously
- Note: The calling code does not change

```
private static void Main()
{
    DoLongRunningTask();
    DoSomeForegroundWork();
    ...
}
```

Where to get `Tasks` from

Most blocking API have `async` versions:

- File I/O
- Network access
- Camera access
- Image processing
- *All* Universal Windows Platform (UWP) APIs
- (Waiting for other threads)

Where to get Tasks from

The compiler automatically wraps return values of `async` methods with `Task` objects:

```
async Task<string> SomeMethod()
{
    return "Developers";
}
```

Return value is wrapped with `Task<string>`

Compiler creates a `Task` result for you

```
async Task AnotherMethod()
{
    Console.WriteLine("Developers");
    Task.CompletedTask;
}
```

Beware of `Task.Run`

- Many tasks run asynchronously without additional threads. This is the preferred way.

- Task.Run must *only* be used with CPU-bound activities:

```
Task<double> c = Task.Run(() => ComplexCalculation(42));
```

This will run `ComplexCalculation` on a separate thread, with all associated problems: Synchronization issues, overhead, …

How to get a result of a Task

- ~~`task.Wait();` BLOCKS current thread, → DEADLOCK~~
  `await task;` doesn't block thread (instead it'll execute other code while waiting)

- ~~`Console.Write(task.Result);` BLOCKS current thread, → DEADLOCK~~
  `Console.Write(await task);` doesn't block thread (instead it'll execute other code while waiting)

→ *Always* use `await`
Never use `.Result` or `.Wait()`

Details in http://blog.stephencleary.com/2012/07/dont-block-on-async-code.html

Best practices

```
private async void DoSomethingAsync()
{
    SomeMethodAsync();
    Console.WriteLine("hey");
}
```

```
private async Task DoSomethingAsync()
{
    await SomeMethodAsync();
    Console.WriteLine("hey");
}
```

- Callers must `await` async methods
- Async methods must return `Task`, never *) `void`

*) except Event handlers in ASP.NET, WPF, … or `Main()` method.

Worksheet – Part 1

Behind the scenes

Code C# master (16 Aug 2017)

```
using System;
using System.Threading.Tasks;
public class C {
    async Task<string> SomeMethod()
    {
        return "Developers";
    }
}
```

```
async Task<string>
SomeMethod()
{
    return "Developers";
}
```

Branch master, last commit

2e1b538cd13bc1b64113f96babe02a128b4f823f at 16 Aug 2017 by CyrusNajmabadi
Merge pull request #21531 from CyrusNajmabadi/moveTypeLeadingBlankLines

Results C# Debug

```
using System.Diagnostics;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Security;
using System.Security.Permissions;
using System.Threading.Tasks;

[assembly: AssemblyVersion("0.0.0.0")]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.Default | DebuggableAttribute.DebuggingModes.DisableOptimizat
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: SecurityPermission(SecurityAction.RequestMinimum, SkipVerification = true)]
[module: UnverifiableCode]
public class C
{
    [CompilerGenerated]
    private sealed class <SomeMethod>d__0 : IAsyncStateMachine
    {
        public int <>1__state;

        public AsyncTaskMethodBuilder<string> <>t__builder;

        public C <>4__this;

        void IAsyncStateMachine.MoveNext()
        {
            int num = this.<>1__state;
            string result;
            try
            {
                result = "Developers";
            }
            catch (Exception arg_10_0)
            {
                Exception exception = arg_10_0;
                this.<>1__state = -2;
                this.<>t__builder.SetException(exception);
                return;
            }
            this.<>1__state = -2;
            this.<>t__builder.SetResult(result);
        }

        [DebuggerHidden]
        void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine stateMachine)
        {
        }
    }

    [DebuggerStepThrough, AsyncStateMachine(typeof(C.<SomeMethod>d__0))]
    private Task<string> SomeMethod()
    {
        C.<SomeMethod>d__0 <SomeMethod>d__ = new C.<SomeMethod>d__0();
        <SomeMethod>d__.<>4__this = this;
        <SomeMethod>d__.<>t__builder = AsyncTaskMethodBuilder<string>.Create();
        <SomeMethod>d__.<>1__state = -1;
        AsyncTaskMethodBuilder<string> <>t__builder = <SomeMethod>d__.<>t__builder;
        <>t__builder.Start<C.<SomeMethod>d__0>(ref <SomeMethod>d__);
        return <SomeMethod>d__.<>t__builder.Task;
    }
}
```

Behind the scenes

```
public async Task<int> DoLongRunningTaskAsync()
{
    var sum = await ComputeValueAsync(1, 1000);
    Console.WriteLine($"Result (asynchronously): {sum}",);
    return sum;
}
```

Compiler generates something like

```
public Task<int> DoLongRunningTaskAsync()
{
    var tcs = new TaskCompletionSource<int>();
    ComputeValueAsync(1, 1000).ContinueWith(task =>
    {
        var sum = task.Result;
        tcs.SetResult(sum);
        Console.WriteLine($"Result (asynchronously): {sum}");
    });
    return tcs.Task;
}
```

Behind the scenes

A `TaskCompletionSource` lets you construct new `Tasks`:

1. Create a `TaskCompletionSource`
2. Use/return `TaskCompletionSource.Task`
3. Eventually call one of these methods
   1. `TaskCompletionSource.SetResult(…)`
   2. `TaskCompletionSource.SetException(…)`
   3. `TaskCompletionSource.SetCanceled()`

Exception handling

Exceptions are passed via `Task` object.
Exceptions “appear”, when you *`await`* a `Task`.
Therefore:

- Either handle all exceptions (`try`/`catch`) or return a `Task` object (not `void`)
- Always *`await`* `Tasks`

Task-based Asynchronous Pattern (TAP)

Asynchronous methods should follow the TAP:

1. Return an already running `Task` (hot) or `Task<TResult>`
2. Add an *`Async`* suffix to the method name
3. If needed, accept a cancellation token and/or `IProgress<T>` in overloads
4. Return quickly to the caller
5. Do not use threads unnecessarily

Concurrent tasks

Executing multiple tasks concurrently:

```
async Task<int> Delay1() { await Task.Delay(1000); return 1; }
async Task<int> Delay2() { await Task.Delay(2000); return 2; }
async Task<int> Delay3() { await Task.Delay(3000); return 3; }

var t1 = Delay1();
var t2 = Delay2();
var t3 = Delay3();

// Include handling of results  ↓↓↓ this takes 3s in total ↓↓↓
Console.WriteLine("{0} {1} {2}", await t1, await t2, await t3);

// Or, without handling the results:
await Task.WhenAll(t1, t2, t3);
Console.WriteLine("All done!");
```

Worksheet – Part 2

Applications

- Most tasks → Async
  - Often just waiting for a response
  - Goal: efficiency, simplicity
  - Examples
    - Reading a file
    - Uploading data
    - Searching for files on a hard disk
    - Waiting 5 seconds

- Long-running, compute-bound tasks → Threads/Task.Run

Resources

- I'll Get Back to You: Task, Await, and Asynchronous Methods in C# - Jeremy Clark
  https://vimeo.com/157300741

- Asynchronous Programming with async and await (C#)
  https://msdn.microsoft.com/en-us/library/mt674882.aspx

- Asynchronous Programming Patterns
  https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/