Applying Asynchronous Programming in C#

GETTING STARTED WITH ASYNCHRONOUS PROGRAMMING IN C# USING ASYNC AND AWAIT

Filip Ekberg
PRINCIPAL CONSULTANT & CEO

@fekberg fekberg.com

Works in Any .NET Application

WPF, WinForms, Xamarin

Console

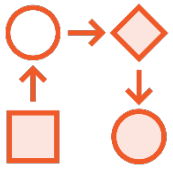
ASP.NET

Asynchronous Programming in .NET

Threading
(Low-level)

Background worker
(Event-based asynchronous pattern)

Task Parallel Library



Async and await

Synchronous vs Asynchrinous

Synchronous

```
private void Search_Click(...)
{
    var client = new WebClient();
    var content =
            client.DownloadString(URL);


}
```

Asynchronous

```
private async void Search_Click(...)
{
    var client = new HttpClient();

    var response = await
            client.GetAsync(URL);

    var content = await response.
            Content.ReadAsStringAsync();
}
```

Asynchronous Web Request

```
private async void Search_Click(...)
{
    var client = new HttpClient();

    var response = await
            client.GetAsync(URL);

    var content = await response.
            Content.ReadAsStringAsync();
}
```

Asynchronous Web Request

```
private async void Search_Click(...)
{
    var client = new HttpClient();

    var response = await
            client.GetAsync(URL);

    var content = await response.
            Content.ReadAsStringAsync();
}
```

Asynchronous Web Request

```
private async void Search_Click(...)
{
    var client = new HttpClient();

    var response = await
            client.GetAsync(URL);

    var content = await response.
            Content.ReadAsStringAsync();
}
```

An **asynchronous operation** occurs in parallel and **relieves the calling thread** of the work

Setting up the Exercise Files

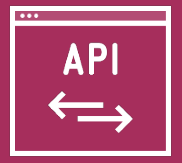
Ask questions on the discussion board

Introducing Async and Await in C#

Suited for I/O Operations

Disk

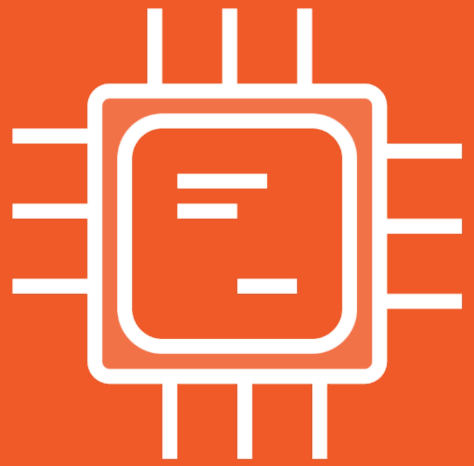
Memory



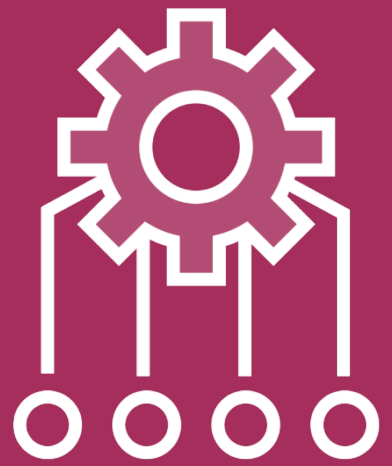
Web/API

Database

When to Use Parallel Programming



CPU bound operations



Independent chunks of data

An **asynchronous** operation **occurs in parallel**

Task Parallel Library

```
await Task.Run(() => {
    // I'm an asynchronous operation that is awaited
});


Parallel.Invoke(
    () => { /* Parallel Thread 1 */ },
    () => { /* Parallel Thread 2 */ },
    () => { /* Parallel Thread 3 */ },
    () => { /* Parallel Thread 4 */ },
);
```

Calling **Result** or **Wait()**
may cause a **deadlock**

Using **async** and **await** in **ASP.NET** means the **web server** can **handle other requests**

Obtaining the Result

```
Task<string> asynchronousOperation = GetStringAsync();

string result = await asynchronousOperation;
```

Using async and await

```
private async void Search_Click(...)
{
    var store = new DataStore();

    var responseTask = store.GetStockPrices("MSFT");

    var data = await responseTask;

    // Code below will run
    // when responseTask has completed

    Stocks.ItemsSource = data;
}
```

Always use **async** and
await together

Understanding a Continuation

Using async and await

```
private async void Search_Click(...)
{
    var store = new DataStore();

    var responseTask = store.GetStockPrices("MSFT");

    var data = await responseTask;

    // Code below will run
    // when responseTask has completed

    Stocks.ItemsSource = data;
}
```

The Await Keyword

Gives you a potential result

Validates the success of the operation

Continuation is back on calling thread

The **await** keyword introduces a **continuation**, **allowing** you to **get back to** the **original context** (thread)

Asynchronous Web Request

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

Continuation executed when GetAsync completes

```
var content =  await response.Content.ReadAsStringAsync();
```

Continuation executed when ReadAsStringAsync completes

```
var data = JsonConvert.DeserializeObject(...)
```

Creating Your Own Asynchronous Method

Implementing GetStocks()

Option 1:

Retrieve, process and return the stock data

Option 2:

Retrieve and process the stock data, then update the UI

Implementing GetStocks()

Option 1:

Retrieve, process and return the stock data

Option 2:

Retrieve and process the stock data, then update the UI

Only use **async void** for
event handlers

Handling an Exception

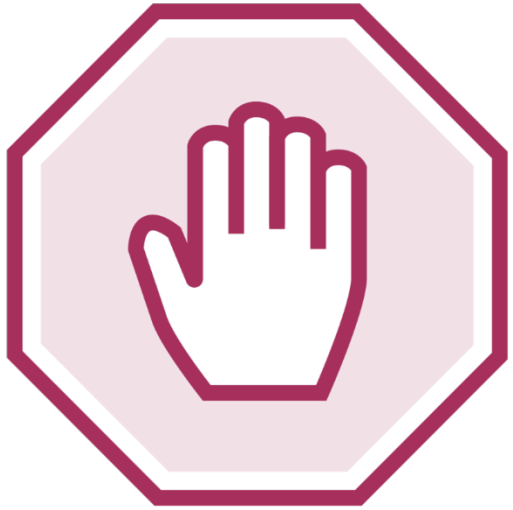
Introducing **asynchronous principles** can **improve** the **user experience**

Exceptions occurring
in an **async void** method
cannot be caught

Always use **await**
to **validate** your
asynchronous operations

Key Takeaways

Always **await** asynchronous operations



Avoid using **async void**

Best Practices

Using async & await

```
async Task Download()
{
    var client = new HttpClient();

    var response = await client.GetAsync(URL);

    var content = await response.
            Content.ReadAsStringAsync();
}
```

Using async & await

Task<HttpResponseMessage>

```
async Task Download()
{
    var client = new HttpClient();

    var response = await client.GetAsync(URL);

}
```

HttpResponseMessage

Using async & await

```
async Task Download()
{
    var client = new HttpClient();

    var response = await client.GetAsync(URL);

}
```

Validates the Task<HttpResponseMessage>
any exceptions will be re-thrown

Using async & await

```
async Task Download()
{
    var client = new HttpClient();

    var response = await client.GetAsync(URL);

    var content = await response.
                Content.ReadAsStringAsync();
}
```

Avoid using async void

```
async Task Good()
{
    throw new Exception("Find me on the Task");
}


async void Bad()
{
    throw new Exception("No one can catch me");
}
```

Unable to await

```
async Task Good()
{
    Bad(); // Can't await...


   // No way to run this line in a continuation
}

async void Bad()
{
    throw new Exception("No one can catch me");
}
```

Don’t call
Result or **Wait()**

Deadlocking

```
private async void Search_Click(...)
{
    GetStocks().Wait();
}

private async Task GetStocks()
{
    ...
}
```

Causes a deadlock!

Using the result after await

```
private async void Search_Click(...)
{
    var store = new DataStore();

    var responseTask = store.GetStockPrices("MSFT");

    await responseTask;

    // In the continuation you may use Result
    var data = responseTask.Result;
}
```

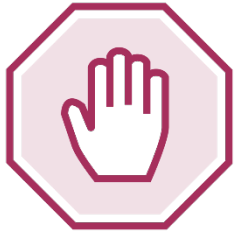
Best Practices

Always use async and await together

Always return a Task from an asynchronous method

Always await an asynchronous method to validate the operation

Use async and await all the way up the chain

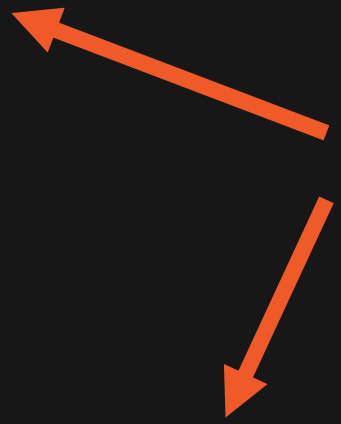


Never use async void unless it's an event handler or delegate

Never block an asynchronous operation by calling Result or Wait()

Different types of continuations

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



Very different continuations!

```
client.GetAsync(URL).ContinueWith((response) => {

});
```

Using the Task Parallel Library for Asynchronous Programming

Filip Ekberg
PRINCIPAL CONSULTANT & CEO

@fekberg fekberg.com

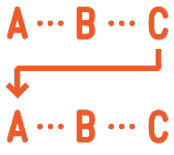
Introducing the Task

```
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

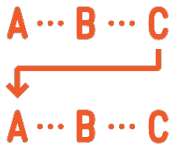
Task from the Task Parallel Library

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);
```

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

});
```

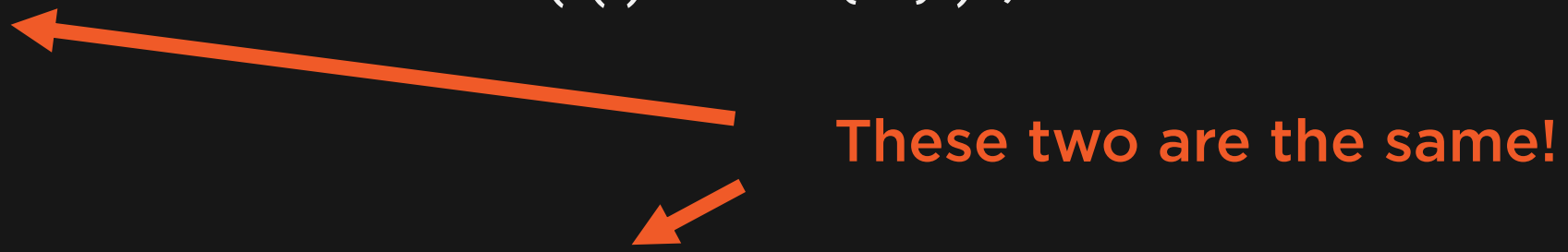
Introduce a Continuation

```
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(() => { });

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.

Cancellation Token Source

```
CancellationTokenSource cancellationTokenSource;
```

Cancellation Token Source

```
CancellationTokenSource cancellationTokenSource;
```

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

Signals to a Cancellation Token
that it should cancel

Cancellation Token Source

```
CancellationTokenSource cancellationTokenSource;

cancellationTokenSource.Cancel();

cancellationTokenSource.CancelAfter(5000);
```

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 **operaiton**

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

cancellationTokenSource.Cancel();


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

Cancellation Token and ContinueWith

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


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

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

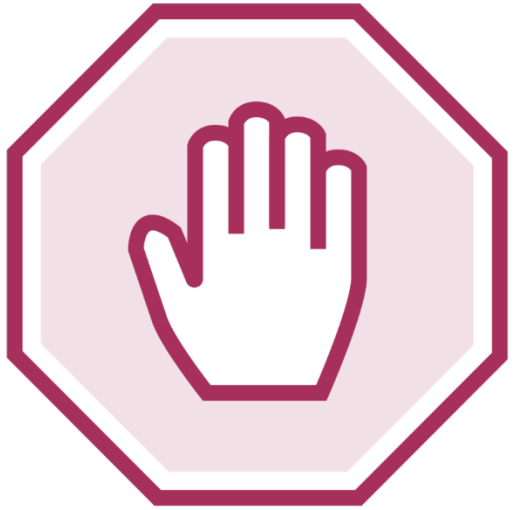
```
var task = Task.Run(() => {
    return ... ;
});

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 versioon



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

Exploring Useful Methods in the Task Parallel Library

Filip Ekberg
PRINCIPAL CONSULTANT & CEO

@fekberg fekberg.com

Overview

How to know if all the tasks in a collection all have been completed

How to run a continuation when at least one task in a collection has completed

Starting multiple tasks and process the result as it arrives

Creating a task with a precomputed result

Learn more about the execution context and controlling the continuation

Knowing When All or Any Task Completes

Task.WhenAll

```
var task1 Task.Run(() => { return “1”; });
var task2 Task.Run(() => { return “2”; });

var tasks = new [] { task1, task2 };

string[] result = await Task.WhenAll(tasks);
```

Precomputed Results of a Task

Task.CompletedTask

```
public override Task Run()
{
    return Task.CompletedTask;
}


await Run(); // Completes immediately
```

Adding **async** and **await** **when you don't** need to introduce **unnecessary complexity**

Task.FromResult

```
public Task<IEnumerable<StockPrice>> Get(...)
{
    var stocks = new List<StockPrice>
    {
        new StockPrice { ... },
        new StockPrice { ... },
        ...
    };

    var task = Task.FromResult(stocks);

    return task;
}
```

Process Tasks as They Complete

Don’t use List<T>
for **parallel** operations it is
not thread-safe

ConcurrentBag<T>

```
var bag = new ConcurrentBag<StockPrice>();
```

Generic & thread-safe!

Execution Context and Controlling the Continuation

ConfigureAwait

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

await task.ConfigureAwait(false);
```

ConfigureAwait

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

await task.ConfigureAwait(false);
```

Configures how the
continuation will be executed

ConfigureAwait(false) could **slightly improve performance** as it **doesn't** have to **switch context**

Don’t rely on the captured context

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

await task.ConfigureAwait(false);

// No code below should require the original context
```

Demo

Demo: ConfigureAwait in ASP.NET

ConfigureAwait(false) in ASP.NET 4.x

Will continue executing the continuation using the current tasks thread

Thread static variables
from the **original context**
won't be available!

ConfigureAwait in ASP.NET Core

ASP.NET Core doesn't use a **synchronization context** which means it will not capture the context like traditional ASP.NET.

Thus, making **ConfigureAwait(false)** useless.

Library developer?

Always use ConfigureAwait(false)

Use ConfigureAwait in libraries

```
public async Task MyLibraryMethod()
{
    var task = ...;

    var result = await task.ConfigureAwait(false);

    // Won't go back to the original thread
    // when handling the result
}
```

Summary

How to best use the Task Parallel Library

Configure the continuation

Start multiple asynchronous operations that execute in parallel

Use Task.WhenAll and Task.WhenAny

Construct a pre-computed result with Task.FromResult

When a pre-computed result is necessary

Processing Tasks as they complete

Using the ConcurrentBag<T>

Controlling the continuation with ConfigureAwait

You're now **ready** to **learn** about the **advance topics**!

Async & Await Advanced Topics and Best Practices

Filip Ekberg
PRINCIPAL CONSULTANT & CEO

@fekberg fekberg.com

Asynchronous Streams & Disposables

Asynchronous streams
requires at least
C# 8.0 and **.NET Core 3.0**

IAsyncEnumerable<T>

"Exposes an **enumerator** that **provides asynchronous iteration** over values of a specified type."

- Microsoft Docs

Asynchronous Stream

Producing a Stream

```
public async
       IAsyncEnumerable<string> Get()
{
    await Task.Delay(2000);

    yield return “Hello”;

    await Task.Delay(2000);

    yield return “World”;
}
```

Consuming a Stream

```
await foreach(var word in Get())
{

}
```

Consuming an Asynchronous Stream

```
await foreach(var price in GetAllStockPrices())
{
    // Consume the price as soon as it's returned
    // by the stream

    // Each item is retrieved asynchronously!
}
```

Allowing you to
asynchronously
retrieve each **item**

Producing an Asynchronous Stream

```
public async IAsyncEnumerable<StockPrice> GetAllStockPrices()
{
    using var stream = new StreamReader(...);

    await stream.ReadLineAsync();

    string line;

    while((line = await stream.ReadLineAsync()) != null)
    {
        yield return StockPrice.FromCSV(line);
    }
}
```

Producing an Asynchronous Stream

```
public async IAsyncEnumerable<StockPrice> GetAllStockPrices()
{
    using var stream = new StreamReader(...);

    await stream.ReadLineAsync();

    string line;

    while((line = await stream.ReadLineAsync()) != null)
    {
        yield return StockPrice.FromCSV(line);
    }
}
```

Producing an Asynchronous Stream

```
public async IAsyncEnumerable<StockPrice> GetAllStockPrices()
{
    using var stream = new StreamReader(...);

    await stream.ReadLineAsync();

    string line;

    while((line = await stream.ReadLineAsync()) != null)
    {
        yield return StockPrice.FromCSV(line);
    }
}
```

The object is returned to the foreach loop as soon as it's parsed!

Asynchronous Disposable

Clean up resources asynchronously by implementing the interface IAsyncDisposable

Asynchronous Disposable

```
public class Service : IAsyncDisposable
{
    public async ValueTask DisposeAsync()
    {
        await Task.Delay(500);
    }
}
```

Asynchronous Disposable

```
public class Service : IAsyncDisposable
{
    public async ValueTask DisposeAsync()
    {
        await Task.Delay(500);
    }
}

public class Consumer
{
    public async Task Run()
    {
        await using var service = new Service();



    }
}
```

Asynchronous Disposable

```
public class Service : IAsyncDisposable
{
    public async ValueTask DisposeAsync()
    {
        await Task.Delay(500);
    }
}

public class Consumer
{
    public async Task Run()
    {
        await using var service = new Service();

        // Use service

        // service is asynchronously disposed at the end of the method
    }
}
```

The Implications of Async and Await

The State Machine

Keeping track of tasks

Executes the continuation

Provides the continuation with a result

Handles context switching

Report errors

Don't **underestimate** the **code generated** by the **compiler**!

Demo

Demo: Reducing the amount of state machines

Reduce the Amount of State Machines

With the same method signatures

Generates a lot of state machines

```
public async Task<string> Run()
{
    return await Compute();
}

public async Task<string> Compute()
{
    return await Load();
}

public async Task<string> Load()
{
    return await Task.Run(() => ...);
}
```

No state machines

```
public Task<string> Run()
{
    return Compute();
}

public Task<string> Compute()
{
    return Load();
}

public Task<string> Load()
{
    return Task.Run(() => ...);
}
```

Next: Deadlocking

Deadlocking

A **deadlock** may **occur** if two **threads depend** on **each other** and **one** is **blocked**

The **state machine** runs on
the **same thread** (UI)
that **we are blocking**!

Deadlock

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

    Dispatcher.Invoke(() => { });

});

task.Wait();
```

Needs to invoke the UI thread before the task can be marked as completed

Blocks the UI thread

The **state machine** runs on the calling thread

Deadlock

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

task.Wait();
```

Don’t use Wait()!

Asynchronous Streams & Disposables in C# 8.0

Streams

```
async IAsyncEnumerable<StockPrice> Get()
{

   while(...)
   {
       yield return item;
   }
}
```

Disposables

```
class Service : IAsyncDisposable
{
    public async ValueTask DisposeAsync()
    {
        await Task.Delay(500);
    }
}

class Consumer
{
    public async Task Run()
    {
        await using var service
                          = new Service();
    }
}
```

Asynchronous Programming Deep Dive

Filip Ekberg
PRINCIPAL CONSULTANT & CEO

@fekberg fekberg.com

Report on the Progress of a Task

Out of the box the **Task**
does not automatically
report the **progress**

What Determines the Progress?

Is it how much of the data that has been loaded?

Is it how much of the data that is completely processed?

Progress<T>

“Provides an **IProgress<T>** that invokes **callbacks** for **each reported progress** value.”

- Microsoft Docs

Progress<T>

```
var progress = new Progress<string>();

progress.ProgressChanged = (_, string progressValue) => {

    // Use the “progressValue” here!

};
```

Progress reporting can be **complex** and **diffucult** but it's **made easier** with **IProgress<T>**

There is **no way** for a **task**
to **automatically figure** out
its **own progress**

We have to **introduce**
something like
Progress<T>

Using Task Completion Source

How Would You Use This with Async & Await?

Event-based asynchronous pattern

Manually queue work on the thread pool

Event-based Asynchronous Pattern

```
var worker = new BackgroundWorker();

worker.DoWork += (sender, e) => {
    // Runs work on a different thread
};

worker.RunWorkerCompleted += (sender, e) => {
    // Event triggered when work is done
};
```

Manually Queue Work on the Thread Pool

```
ThreadPool.QueueUserWorkItem(_ => {
    // Run work on a different thread
});
```

TaskCompletionSource<T>

"**Represents** the producer side of a **Task<T>** unbound to a delegate, **providing access** to the consumer side **through the Task property**."

- Microsoft Docs

Task Completion Source

```
var tcs = new TaskCompletionSource<string>();

Task<string> task = tcs.Task;
```

Use **TaskCompletionSource** to create **awaitables** out **of legacy code** that **don't use** the **TPL**

Working with Attached and Detached Tasks

Nested / Child Tasks

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

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

});
```

These are child tasks

Task.Factory.StartNew Overloads

```
StartNew(Action)
StartNew(Action, CancellationToken)
StartNew(Action, TaskCreationOptions)
StartNew(Action, CancellationToken, TaskCreationOptions, TaskScheduler)

StartNew(Action<Object>, Object)
StartNew(Action<Object>, Object, CancellationToken)
StartNew(Action<Object>, Object, TaskCreationOptions)
StartNew(Action<Object>, Object, CancellationToken,
                                 TaskCreationOptions, TaskScheduler)
```

+ 8 more

Task.Factory.StartNew Overloads

```
StartNew(Action)
StartNew(Action, CancellationToken)
StartNew(Action, TaskCreationOptions)
StartNew(Action, CancellationToken, TaskCreationOptions, TaskScheduler)

StartNew(Action<Object>, Object)
StartNew(Action<Object>, Object, CancellationToken)
StartNew(Action<Object>, Object, TaskCreationOptions)
StartNew(Action<Object>, Object, CancellationToken,
                                 TaskCreationOptions, TaskScheduler)


+ 8 more
```

Task.Factory.StartNew Overloads

```
StartNew(Action)
StartNew(Action, CancellationToken)
StartNew(Action, TaskCreationOptions)
StartNew(Action, CancellationToken, TaskCreationOptions, TaskScheduler)

StartNew(Action<Object>, Object)
StartNew(Action<Object>, Object, CancellationToken)
StartNew(Action<Object>, Object, TaskCreationOptions)
StartNew(Action<Object>, Object, CancellationToken,
                                 TaskCreationOptions, TaskScheduler)


+ 8 more
```

Task.Factory.StartNew Overloads

```
StartNew(Action)
StartNew(Action, CancellationToken)
StartNew(Action, TaskCreationOptions)
StartNew(Action, CancellationToken, TaskCreationOptions, TaskScheduler)

StartNew(Action<Object>, Object)
StartNew(Action<Object>, Object, CancellationToken)
StartNew(Action<Object>, Object, TaskCreationOptions)
StartNew(Action<Object>, Object, CancellationToken,
                                 TaskCreationOptions, TaskScheduler)


+ 8 more
```

Task.Factory.StartNew Overloads

```
StartNew(Action)
StartNew(Action, CancellationToken)
StartNew(Action, TaskCreationOptions)
StartNew(Action, CancellationToken, TaskCreationOptions, TaskScheduler)

StartNew(Action<Object>, Object)
StartNew(Action<Object>, Object, CancellationToken)
StartNew(Action<Object>, Object, TaskCreationOptions)
StartNew(Action<Object>, Object, CancellationToken,
                                 TaskCreationOptions, TaskScheduler)


+ 8 more
```

Using **Task.Run** is in most situations the **best option**

AttachedToParent

“Specifies that a **task** is **attached** to a **parent** in the task hierarchy. **By default**, a **child task** (that is, an inner task created by an outer task) **executes independently** of its parent.

You can use the **AttachedToParent** option so that the **parent** and **child tasks** are **synchronized**.

Note that if a **parent task** is **configured** with the **DenyChildAttach** option, the **AttachedToParent** option in the child task **has no effect**, and the child task will execute as a detached child task.”

- Microsoft Docs

If a **parent task** is **configured** with the **DenyChildAttach** option

AttachedToParent option in the child task **has no effect**

Task.Run Automatically Unwraps!

```
Task<string> task = Task.Run(async () => {
    await Task.Delay(1000);

    return "Pluralsight";
});


Task<Task<string>> taskFromFactory = Task.Factory.StartNew(async () => {
    await Task.Delay(1000);

    return "Pluralsight";
});

Task<string> unwrappedTask = taskFromFactory.Unwrap();
```

Passing a Value to Task.Factory.StartNew

```
IEnumerable<StockPrice> stocks = ...

Task.Factory.StartNew((state) => {

    // Cast the state to the correct type
    var items = state as IEnumerable<StockPrice>

}, stocks);
```

Using “stocks” directly in the anonymous method would introduce a clojure

Passing a Value to Task.Factory.StartNew

```
IEnumerable<StockPrice> stocks = ...

Task.Factory.StartNew((state) => {

    // Cast the state to the correct type
    var items = state as IEnumerable<StockPrice>

}, stocks);
```

You can pass a reference to the object which will be used by the asynchronous operation

Passing a Value to Task.Factory.StartNew

```
IEnumerable<StockPrice> stocks = ...

Task.Factory.StartNew((state) => {

    // Cast the state to the correct type
    var items = state as IEnumerable<StockPrice>

}, stocks);
```

Task.Run

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

Internally uses the factory with these default values

```
Task.Factory.StartNew(
    () => {},
    CancellationToken.None,
    TaskCreationOptions.DenyChildAttach,
    TaskScheduler.Default
);
```