

# Async & Await Advanced Topics

---



**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.”



# Asynchronous Stream

## Producing a Stream

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

    await stream.ReadLineAsync();

    while(await stream.ReadLineAsync() is string line)
    {
        yield return StockPrice.FromCSV(line);
    }
}
```





# Producing an Asynchronous Stream

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

    await stream.ReadLineAsync();

    while(await stream.ReadLineAsync() is string line)
    {
        yield return StockPrice.FromCSV(line);
    }
}
```



# Producing an Asynchronous Stream

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

    await stream.ReadLineAsync();

    while(await stream.ReadLineAsync() is string line)
    {
        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 `IDisposable`



# 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**

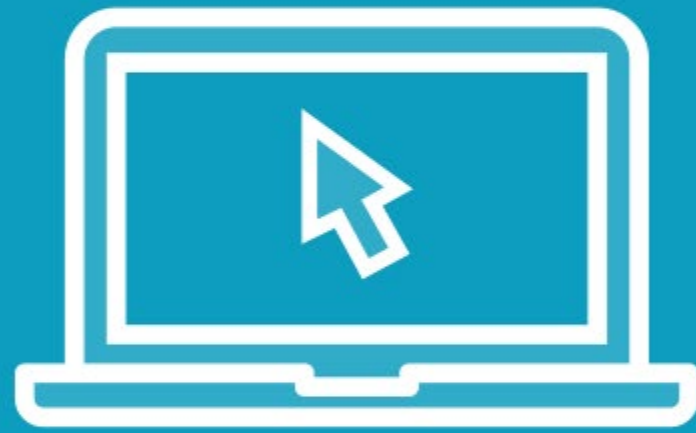




Don't **underestimate**  
the **code generated**  
by the **compiler!**



# 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

## Streams

```
async IEnumerable<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();
    }
}
```