Why did the Java developer wear glasses?

Because he couldn't see sharp.

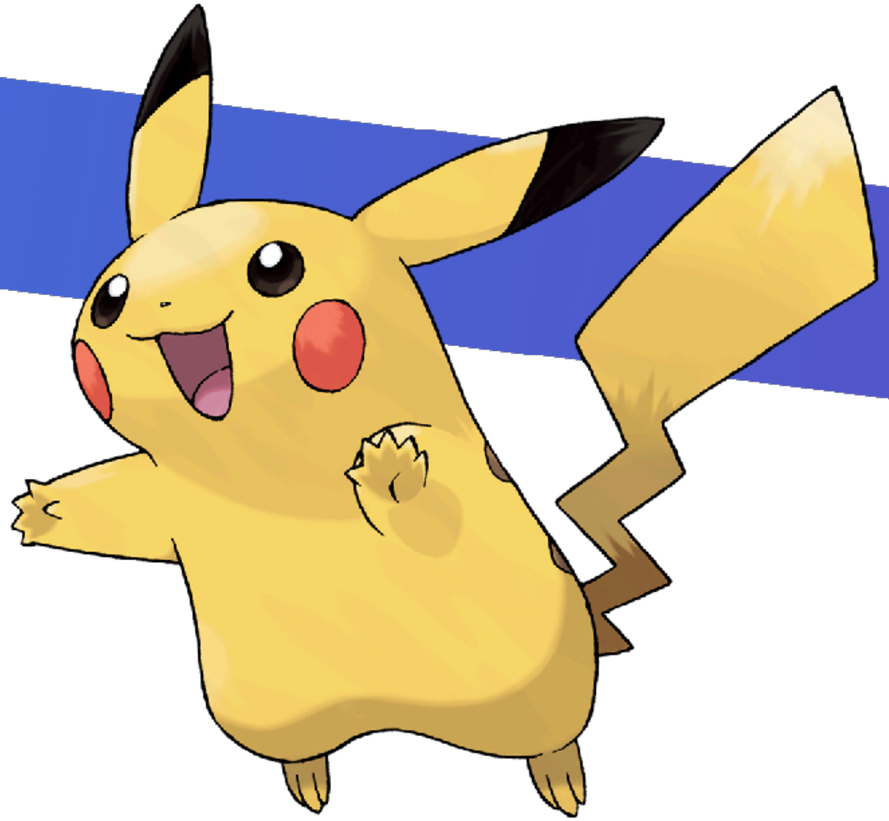
C# Lowering

What is it and why should I care?

Steven Giesel // WeAreDevelopers 2025 // How to misuse SharpLab.io for a whole talk

```
{
  "name": "Steven Giesel",
  "websites": [
    "https://steven-giesel.com",
    "mailto:contact@steven-giesel.com",
    "https://bitspire.ch",
    "mailto:steven.giesel@bitspire.ch"
  ],
  "isMvp": true,
  "github": "linkdotnet",
  "linkedIn": "Steven Giesel",
  "blueSky": "@steven-giesel.com",
  "x": null
}
```

What do those two have in common?



```
foreach (var name in names)
{
    ...
}
```

None of them are known in IL Code

Motivation

"*Understand one level below your normal abstraction layer.*" -Neal Ford

Performance

Motivation

"*Understand one level below your normal abstraction layer.*" -Neal Ford

Performance

Bugs

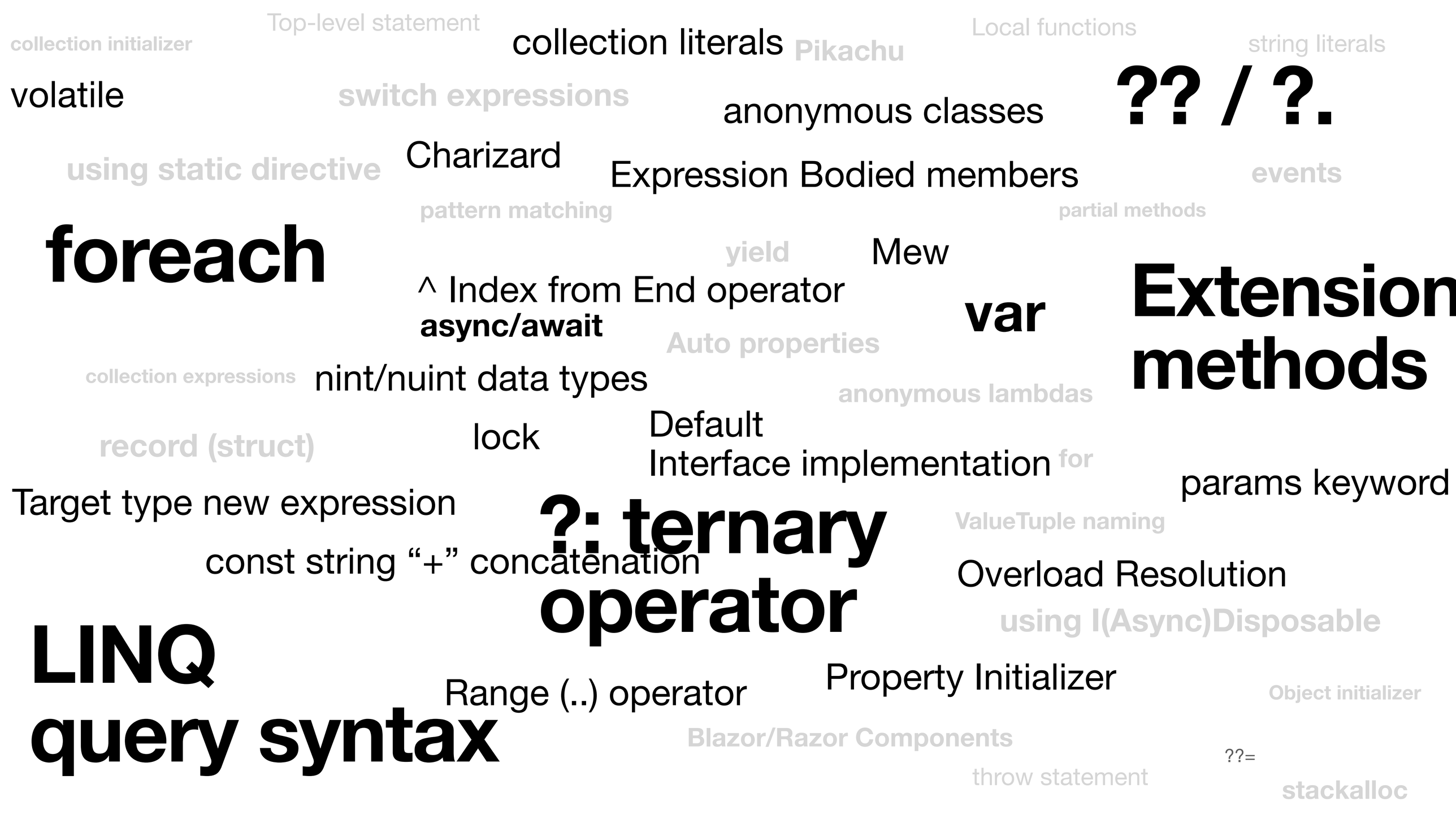
Motivation

"*Understand one level below your normal abstraction layer.*" -Neal Ford

Performance

Bugs

Fundamentals


Top-level statement
collection initializer
collection literals
Pikachu
Local functions
string literals
volatile
switch expressions
anonymous classes
?? / ?.
Charizard
using static directive
Expression Bodied members
events
pattern matching
partial methods
foreach
yield
Mew
^ Index from End operator
async/await
var
Extension methods
Auto properties
collection expressions
nint/nuint data types
anonymous lambdas
record (struct)
lock
Default Interface implementation
for
params keyword
Target type new expression
?: ternary operator
ValueTuple naming
const string “+” concatenation
Overload Resolution
using I(Async)Disposable
LINQ query syntax
Property Initializer
Range (..) operator
Object initializer
Blazor/Razor Components
??=
throw statement
stackalloc

And the most important one:

dynamic

What is “Lowering”?

What is “Lowering”?

Compiling

Translating a language into a language closer to the machine

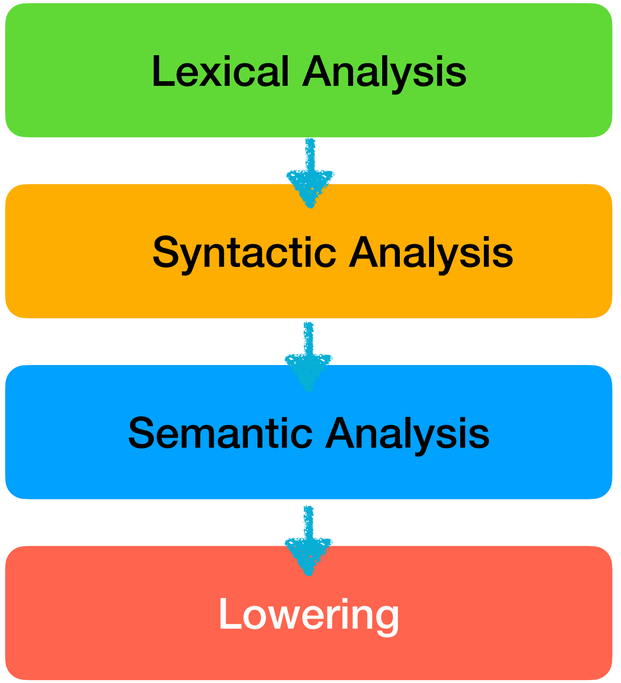
IL
(Intermediate Language)

What is “Lowering”?

Lowering

Compiling

What is “Lowering”?



- Often time referred to as “syntactic sugar”
- Or “compiler magic”
- Lowering is part of the compiler process, when C# code is transformed to IL-Code (or to machine code)

Thanks to NativeAOT

Advantages of “Lowering”

Optimization

Advantages of “Lowering”

Optimization

Simplicity

Advantages of “Lowering”

Optimization

Simplicity

Compatibility
Consistency

Advantages of “Lowering”

Optimization

Simplicity

Compatibility
Consistency

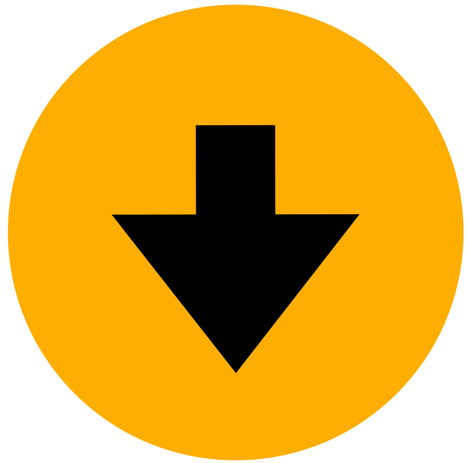
Compiler

What is “Lowering”?

Fly

What is “Lowering”?

Travel fast through air

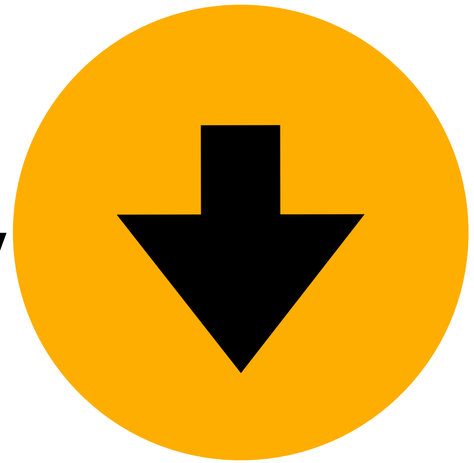


What is “Lowering”?

Horizon

What is “Lowering”?

Border between land and sky




<code/>

Let’s start simple - var

```
var myString = "Hello World";

Console.Write(myString);
```


```
string myString = "Hello World";

Console.Write(myString);
```

- **var** doesn’t exist and will resolved to the concrete type
- We call that: **type interference** (the ability to deduct the type from the context)

What is the output?

```
var entity = new MyEntity();

Console.Write($"{entity.GetCounter}, ");
Console.Write($"{entity.GetCounter}, ");
Console.Write($"{entity.ExprCounter}, ");
Console.Write($"{entity.ExprCounter}");


public class MyEntity
{
  private static int a = 0;
  private static int b = 0;

  public int GetCounter { get; } = ++a;
  public int ExprCounter => ++b;
}
```

What is the output?

```
var entity = new MyEntity();

Console.Write($"{entity.GetCounter}, ");
Console.Write($"{entity.GetCounter}, ");
Console.Write($"{entity.ExprCounter}, ");
Console.Write($"{entity.ExprCounter}");


public class MyEntity
{
  private static int a = 0;
  private static int b = 0;

  public int GetCounter { get; } = ++a;
  public int ExprCounter => ++b;
}
```

A. 1,1,1,1

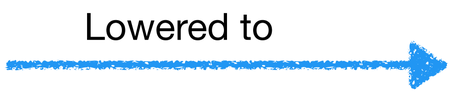
B. 1,2,1,2

C. 1,1,1,2

D. 1,2,1,1

Expression member VS get w/ backing field

```
var entity = new MyEntity();

Console.Write($"{entity.GetCounter}, ");
Console.Write($"{entity.GetCounter}, ");
Console.Write($"{entity.ExprCounter}, ");
Console.Write($"{entity.ExprCounter}");


public class MyEntity
{
    private static int a = 0;
    private static int b = 0;

    public int GetCounter { get; } = ++a;
    public int ExprCounter => ++b;
}
```



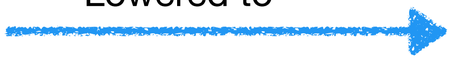

```
public class MyEntity
{
    private static int a;
    private static int b;

    [CompilerGenerated]
    private readonly int <GetCounter>k__BackingField = ++a;

    public int GetCounter
    {
        [CompilerGenerated]
        get
        {
            return <GetCounter>k__BackingField;
        }
    }

    public int ExprCounter
    {
        get
        {
            return ++b;
        }
    }
}
```

- Expression Bodied member getter is called **every time**
- Backing fields are initialized once

foreach Array

```
var array = new[] { 1, 2 };

foreach(var item in array)
    Console.Write(item);
```

Lowered to

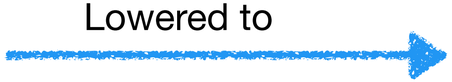


```
int[] array = new int[2];
array[0] = 1;
array[1] = 2;
int[] array2 = array;
int num = 0;
while (num < array2.Length)
{
    int value = array2[num];
    Console.Write(value);
    num++;
}
```

- No collection initializer
- No **foreach** inside lowered code
  - Translated into a **while** loop
  - Also **for** loops are translated into **while** loops

foreach List

```
var list = new List<int> { 1, 2 };

foreach(var item in list)
    Console.Write(item);
```




```
List<int> list = new List<int>();
list.Add(1);
list.Add(2);
List<int>.Enumerator enumerator =
    list.GetEnumerator();
try
{
    while (enumerator.MoveNext())
    {
        Console.Write(enumerator.Current);
    }
}
finally
{
    ((IDisposable)enumerator).Dispose();
}
```

- Still no **foreach**
- But we have “**Enumerators**”
- And a **try-finally** block

using in combination with async/await

```
Task<string> GetContentFromUrlAsync(string url)
{
    // Don't do this! Creating new HttpClients
    // is expensive and has other caveats
    // This is for the sake of demonstration
    using var client = new HttpClient();
    return client.GetStringAsync(url);
}
```

using in combination with async/await

```
Task<string> GetContentFromUrlAsync(string url)
{
    // Don't do this! Creating new HttpClients
    // is expensive and has other caveats
    // This is for the sake of demonstration
    using var client = new HttpClient();
    return client.GetStringAsync(url);
}
```

using in combination with async/await

```
Task<string> GetContentFromUrlAsync(string url)
{
    // Don't do this! Creating new HttpClients
    // is expensive and has other caveats
    // This is for the sake of demonstration
    using var client = new HttpClient();
    return client.GetStringAsync(url);
}
```

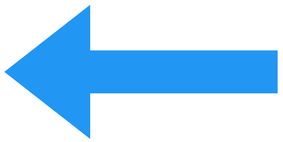
Lowered to

```
HttpClient httpClient = new HttpClient();
try
{
    return httpClient.GetStringAsync(url);
}
finally
{
    if (httpClient != null)
    {
        ((IDisposable)httpClient).Dispose();
    }
}
```

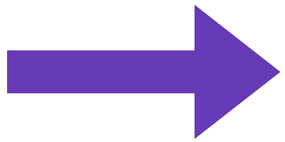
- using guarantees* disposal via finally blocks
- The finally block is called **after** the return statement
- This disposes the HttpClient and therefore the caller is greeted with an ObjectDisposedException

* If you don't pull the plug, get hit by a meteor or SIGKILL via Task-Manager

Yield



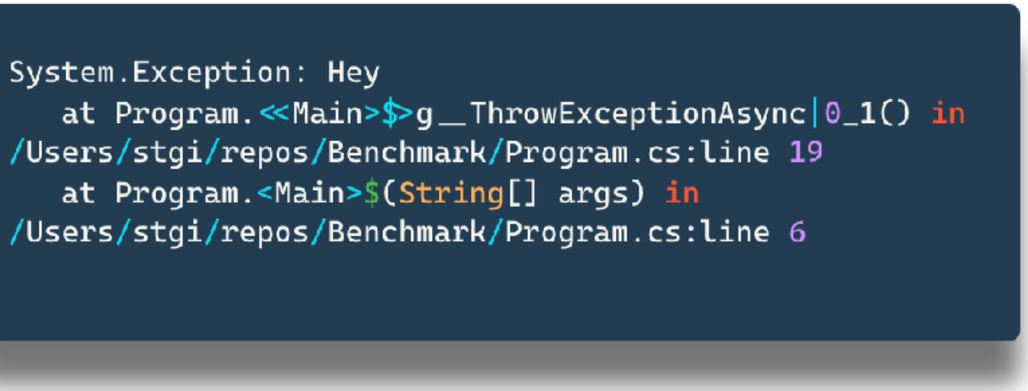
Eliding await



Eliding await

```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```

Output



- The not-**awaited** function (DoWorkWithoutAwaitAsync) ist **not part of the** stack trace

Eliding await

```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```

Lowered to

```
[System.Runtime.CompilerServices.NullableContext(1)]
[CompilerGenerated]
internal static Task <<Main>$>g__DoWorkWithoutAwaitAsync|0_0()
{
    return <<Main>$>g__ThrowExceptionAsync|0_1();
}

[System.Runtime.CompilerServices.NullableContext(1)]
[AsyncStateMachine(typeof(<<<Main>$>g__ThrowExceptionAsync|0_1>d))]
[CompilerGenerated]
internal static Task <<Main>$>g__ThrowExceptionAsync|0_1()
{
    <<<Main>$>g__ThrowExceptionAsync|0_1>d stateMachine
        = default(<<<Main>$>g__ThrowExceptionAsync|0_1>d);

    stateMachine.<>t__builder = AsyncTaskMethodBuilder.Create();
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}
```

- No **async** -> No State-Machine

Eliding await

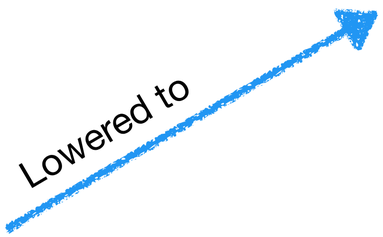
```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```

Lowered to

```
stateMachine
```

Eliding await

```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```




```
try
{
    YieldAwaitable.YieldAwaiter awaiter;
    // Here is some other stuff
    awaiter.GetResult();
    throw new Exception("Hey");
}
catch (Exception exception)
{
    <>1__state = -2;
    <>t__builder.SetException(exception);
}
```

- Exceptions aren’t directly thrown, but rather kept as State on the **Task** object
  👋Hello👋 **async void**
- But where is the caller?

“A stack trace does not tell
you where you came from.

A stack trace tells you
where you are going next.” - *Eric Lippert*

Eliding await

```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```

Eliding await

```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```

Eliding await

```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```

Eliding await

```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```

Eliding await

```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```

Eliding await

```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```

Eliding await

```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```

Eliding await

```
try
{
    await DoWorkWithoutAwaitAsync();
}
catch (Exception e)
{
    Console.WriteLine(e);
}

static Task DoWorkWithoutAwaitAsync()
    => ThrowExceptionAsync();

static async Task ThrowExceptionAsync()
{
    await Task.Yield();
    throw new Exception("Hey");
}
```

Yield

```
static IEnumerable<int> GetTenRandomNumbers()
{
    for (var i = 0; i < 10; i++)
        yield return Random.Shared.Next();
}
```

Yield

```
static IEnumerable<int> GetTenRandomNumbers()
{
    for (var i = 0; i < 10; i++)
        yield return Random.Shared.Next();
}
```

```
static List<int> GetTenRandomNumbersList()
{
    var list = new List<int>();
    for (var i = 0; i < 10; i++)
        list.Add(Random.Shared.Next());
    return list;
}
```

Yield

```
static IEnumerable<int> GetTenRandomNumbers()
{
    for (var i = 0; i < 10; i++)
        yield return Random.Shared.Next();
}
```

Yield

```
class GetTenRandomNumbersEnumerable : IEnumerable<int>, IEnumerator<int>
{
    private int state;
    private int current;
    private int i;

    public int Current => current;
    object IEnumerator.Current => Current;
    public void Dispose() { }

    public IEnumerator<int> GetEnumerator() => this;
    IEnumerator IEnumerable.GetEnumerator() => this;

    public bool MoveNext()
    {
        if (state != 0)
        {
            // We were done once and MoveNext was called again
            if (state != 1)
                return false;

            state = -1;
            i++;
        }
        else
        {
            state = -1;
            i = 0;
        }

        // We are done! (false means we don't move to the next element)
        if (i >= 10)
            return false;

        current = Random.Shared.Next();
        state = 1;
        return true;
    }
}
```

Yield

```
class GetTenRandomNumbersEnumerable : IEnumerable<int>, IEnumerator<int>
{
    private int state;
    private int current;
    private int i;

    public int Current => current;
    object IEnumerator.Current => Current;
    public void Dispose() { }

    public IEnumerator<int> GetEnumerator() => this;
    IEnumerator IEnumerable.GetEnumerator() => this;

    public bool MoveNext()
    {
        if (state != 0)
        {
            // We were done once and MoveNext was called again
            if (state != 1)
                return false;

            state = -1;
            i++;
        }
        else
        {
            state = -1;
            i = 0;
        }

        // We are done! (false means we don't move to the next element)
        if (i >= 10)
            return false;

        current = Random.Shared.Next();
        state = 1;
        return true;
    }
}
```

```
Console.WriteLine("Start Here");
foreach(var   item in   GetTenRandomNumbers())
    Console.WriteLine(item);
Console.WriteLine("End Here");
```

state: 0 1

i: 0 1

current: 36 36

Yield

```
[Benchmark]
public int GetSumOf1000NumbersViaYield()
{
    return Get1000RandomNumbers().Sum();
}

[Benchmark]
public int GetSumOf1000NumbersViaList()
{
    return Get1000RandomNumbersList().Sum();
}
```

```
private static IEnumerable<int> Get1000RandomNumbers()
{
    for (var i = 0; i < 1000; i++)
        yield return i;
}

private static List<int> Get1000RandomNumbersList()
{
    var list = new List<int>(1000);
    for (var i = 0; i < 1000; i++)
        list.Add(i);
    return list;
}
```

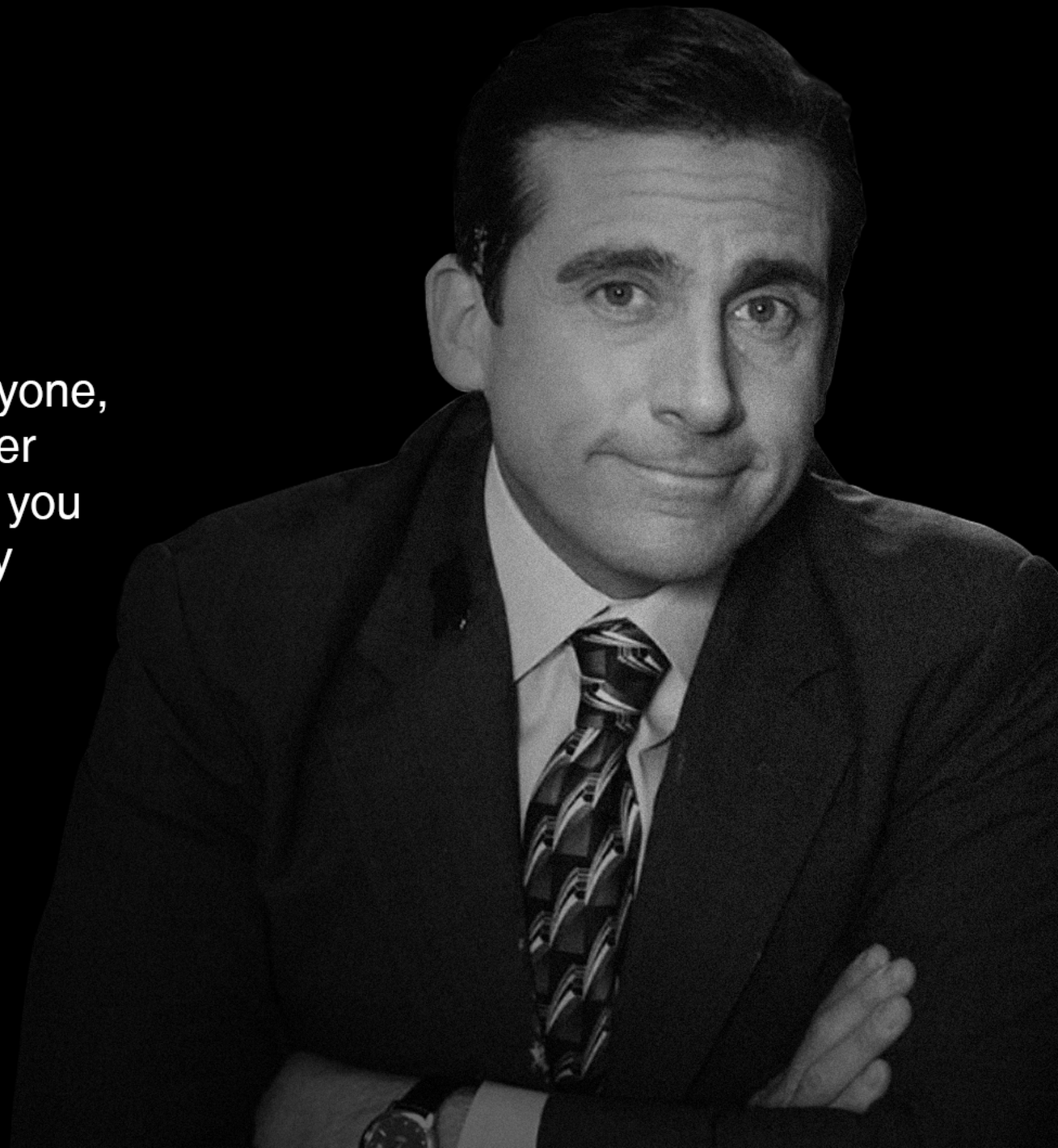
Yield

```
[Benchmark]
public int GetSumOf20NumbersViaYield()
{
    return Get1000RandomNumbers().Take(20).Sum();
}

[Benchmark]
public int GetSumOf20NumbersViaList()
{
    return Get1000RandomNumbersList().Take(20).Sum();
}
```

```
private static IEnumerable<int> Get1000RandomNumbers()
{
    for (var i = 0; i < 1000; i++)
        yield return i;
}

private static List<int> Get1000RandomNumbersList()
{
    var list = new List<int>(1000);
    for (var i = 0; i < 1000; i++)
        list.Add(i);
    return list;
}
```

Don't ever, for any reason, do anything, to anyone, for any reason, ever, no matter what, no matter where, or who, or who you are with, or where you are going, or where you've been, ever, for any reason whatsoever.

- Michael Scott



It’s a wrap!