Todd Howard: "If you're running low on memory, you can reboot the original Xbox and the user can't tell. You can throw a screen up. When Morrowind loads sometimes you get a very long load. That's us rebooting the Xbox."

Why did the Java developer wear glasses?

Because he couldn't see sharp.

C# Lowering

What is it and why should I care?

```
$ ~ whoami --full-profile
name: Steven Giesel
website: steven-giesel.com
can_hire: true
linkedin: true
twitter: false
is_mvp: true
```

Question: What does these two have in common?

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

Answer: Neither 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

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

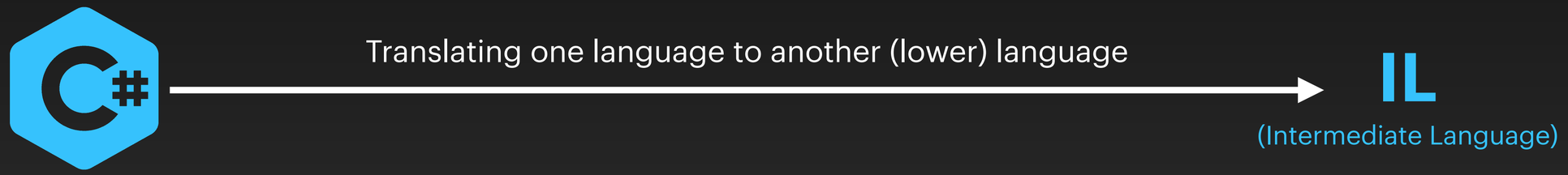
And the most important keyword of all time

dynamic

What is “Lowering”?

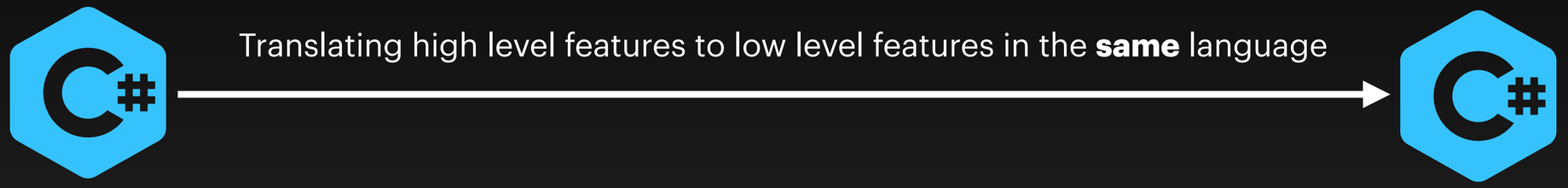
What is “Lowering”?

Compiling

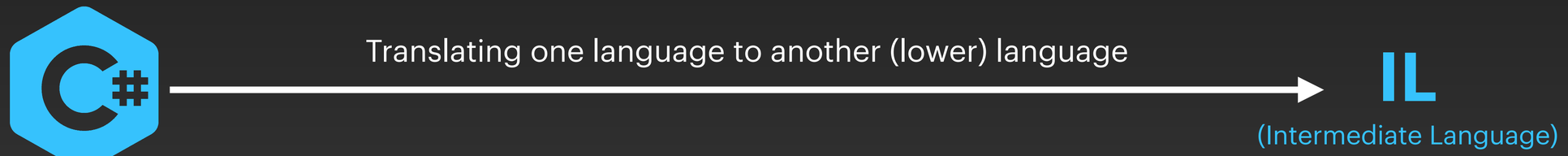


What is “Lowering”?

Lowering



Compiling



What is “Lowering”?

- Another name you know for that is “syntactic sugar”
- Or “compiler magic”
- Lowering is part of the whole process, when you compile your C# code into an assembly (IL code)

Benefits of “Lowering”

Optimization

Benefits of “Lowering”

Optimization

Simplicity

Benefits of “Lowering”

Optimization

Simplicity

Compatibility / Consistency

Benefits of “Lowering”

Optimization

Simplicity

Compatibility / Consistency

Compiler

What is “Lowering”?

Fly

What is “Lowering”?

Move quickly through air

What is “Lowering”?

Horizon

What is “Lowering”?

Line where sky meets land

<code />

Let’s start easy - var

```
var myString = "Hello World";

Console.Write(myString);
```

Gets lowered to →

```
string myString = "Hello World";

Console.Write(myString);
```

- Easy one, var does not exist and gets resolved to its concrete type
- That is called type interference (the ability to deduct the type from the context)

What is the output of the following code snippet?

```
var d = new DotNetDay();

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

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

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

A. 1,1,1,1

B. 1,2,1,2

C. 1,1,1,2

D. 1,2,1,1

What is the output of the following code snippet?

```
var d = new DotNetDay();

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

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

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

```
public class DotNetDay
{
  private static int a = 0;
  private static int b = 0;

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

Gets lowered to →

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

   [CompilerGenerated]
   private readonly int k__BackingField = ++b;

   public int ExprCounter
   {
      get { return ++a; }
   }

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

- Bodied member getter will call the function every time
- With “only” the backing field - we only initialize once

foreach array

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

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

Gets 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++;
}
```

- There is no collection initializer anymore
- There is no foreach anymore in the lowered code
  - Translated into a while loop
  - Also for loops get lowered to a while loop

foreach list

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

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

Gets lowered to

```
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 in sight
- We are using Enumerators with (MoveNext and Current)
- Try-Finally block as Enumerator inherits from Disposable

using and 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);
}
```

- Let’s have a look how using works to understand what might be an issue here

using and 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);
}
```

Gets lowered to →

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

- using guarantees* to dispose via a finally block
- The finally block gets executed after return
- This will dispose the HttpClient and therefore the awaiter of our call will be presented with a nice ObjectDisposedException

* If you don't pull the plug out of your PC, get hit by a meteor or kill it via task manager

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

```
System.Exception: Hey
   at Program.<<Main>$>g__ThrowExceptionAsync|0_1() in
/Users/stgi/repos/Benchmark/Program.cs:line 19
   at Program.<Main>$(String[] args) in
/Users/stgi/repos/Benchmark/Program.cs:line 6
```

- The “not” awaited method (DoWorkWithoutAwaitAsync) is 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");
}
```

gets 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 await -> 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");
}
```

gets 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;
}
```

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");
}
```

gets lowered to

```
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 don’t bubble up - they are stored on the Task object
- But why isn’t the caller part of it?

“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");
}
```

- At the await boundary, we give control back to the caller.
- The caller does not await so we pass control to the next caller (that awaits the call)

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");
}
```

- Now the continuation gets called and throws the exception
- On the stack trace there is no DoWorkWithoutAwaitAsync anymore as the method finished

Thanks to sharplab.io for making my presentation possible ;)

And of course: You <3

Gold

M&S SOFTWARE ENGINEERING

MS MAISON DU SOFTWARE

OFFERING SOLUTIONS SOFTWARE

A<A>EMY

Silver

NOSER ENGINEERING

Digitec Galaxus AG