“Why did the developer quit his job? He didn't get arrays.”

C# Lowering

What is it and why should I care?

Steven Giesel // .NET User Group Zurich // How to misuse sharplab.io for a whole talk!

```
$ ~ whoami --full
name: Steven Giesel
website: steven-giesel.com
github: linkdotnet
linkedin: Steven Giesel
twitter: -
working: Zühlke Engineering AG
```

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

Question: What does these two have in common?

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

Question: What does these two have in common?

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

Answer: The C# compiler doesn’t know them when creating CIL code!

Motivation

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

- Understanding better what your C# really does
- Predicting performance (and) implications of your code
- Detect bugs / understand bugs or even better: Prevent them
- You want to understand why some constructs don't "really" exist like:
  foreach, var, lock, using, async / await, yield, Anonymous lambda, records, extension methods, LINQ query syntax, stackalloc, Pikachu, events, is / as operator, ?? / ?. operator, pattern matching, Blazor or Razor components, type interference, anonymous types, switch expression, index ranges, string concat of const strings via "+" operator, ternary operator, local functions, using static directive, ValueTuple, Deconstructor, Range …
- You are geeky like me and want to understand the core of your language

What is “Lowering”?

What is “Lowering”?

Compiling

Translating one language to another language

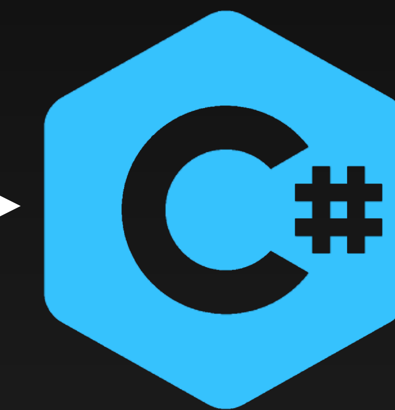
CIL

(Common Intermediate Language)

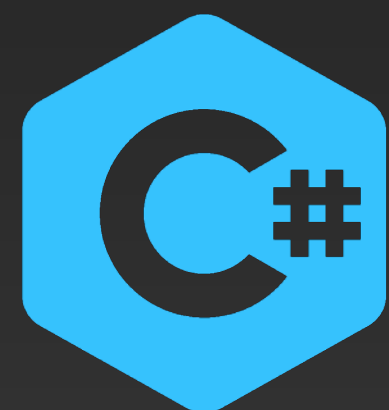
What is "Lowering"?

Lowering

Translating high level features to low level features in the **same** language



Compiling



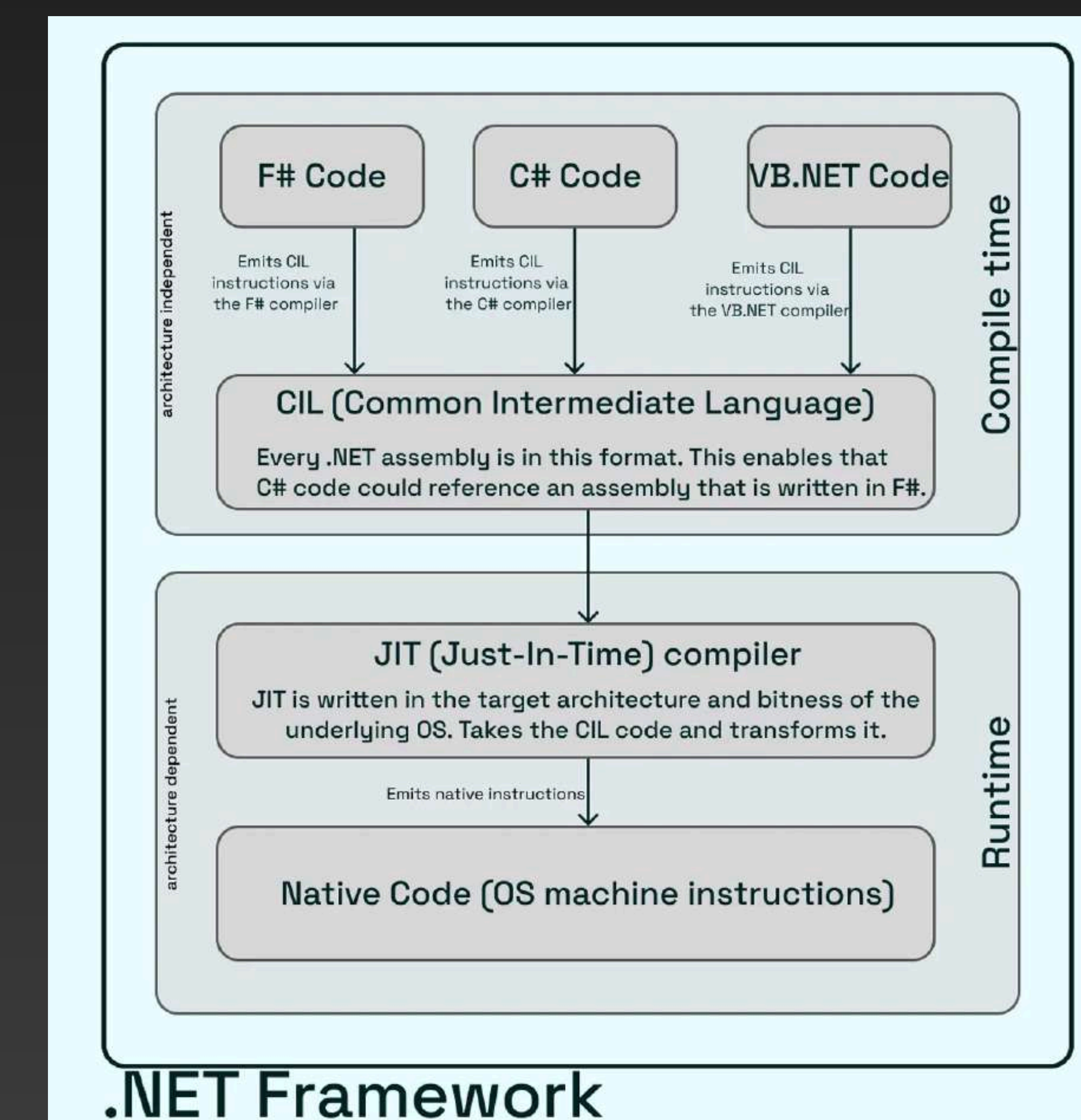
Translating one language to another language

CIL

(Common Intermediate Language)

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 (CIL code)
- Lowering is like synonyms in the same language:
  “a or b or both” instead for “a and/or b”
  “from now on” instead of “henceforth”
- Can bring benefits in terms of optimization



Let's start easy - var

```csharp
var myString = "Hello World";

Console.Write(myString);
```

```csharp
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)

Case Study 2.1: 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 foreach anymore in the lowered code
  - Translated into a while loop
  - Also for loops get usually lowered to a while loop
- Also there is no collection initializer anymore

Case Study 2.2: 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.GetEnumerato
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)
  - IEnumerable is like a basket full of apples
  - IEnumerator goes through one at a time, until you find the perfect apple
- Try-Finally block as Enumerator inherits from Disposable

Foreach without IEnumerable

Let’s code some magic

Foreach without IEnumerable

Foreach without IEnumerable

Range-object is not enumerable ... normally

```
foreach (int number in 2..5)
{
    Console.WriteLine(number);
}
```

Foreach without IEnumerable

Range-object is not enumerable … normally

```
foreach (int number in 2..5)
{
    Console.WriteLine(number);
}

public static class Extensions
{
    public static IEnumerator<int> GetEnumerator(this Range r)
        => Enumerable.Range(r.Start.Value, r.End.Value - r.Start.Value)
            .GetEnumerator();
}
```

C# 3: If IEnumerable isn’t implemented try to grab appropriate GetEnumerator method.
C# 9: Extension GetEnumerator support for foreach loops.

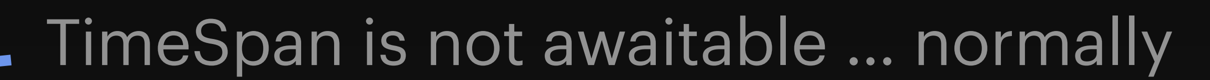
Extra: Await everything

Extra: Await everything

```
await 2.Seconds();
await TimeSpan.FromSeconds(2);
```

TimeSpan is not awaitable ... normally

Extra: Await everything




```
await 2.Seconds();
await TimeSpan.FromSeconds(2);

public static class Extensions
{
    public static TaskAwaiter GetAwaiter(this TimeSpan ts)
        => Task.Delay(ts).GetAwaiter();

    public static TimeSpan Seconds(this int s)
        => TimeSpan.FromSeconds(s);
}
```

Back on track

Case Study 3: using

```
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

Case Study 3: using

```
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 with 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

Case Study 4: is operator

Are these code snippets equal?

```
int Do(Person? p)
{
    if (p is { Age: < 25 })
        return 1;
    return 10;
}
```

```
int Do(Person? p)
{
    if (p.Age < 25)
        return 1;
    return 10;
}
```

Case Study 4: is operator

Are these code snippets equal?

```
int Do(Person? p)
{
    if (p is { Age: < 25 })
        return 1;
    return 10;
}
```

```
p != null && p.Age < 25;
```

```
int Do(Person? p)
{
    if (p.Age < 25)
        return 1;
    return 10;
}
```

Case Study 4: is operator

Are these code snippets equal?

```
int Do(Person? p)
{
    if (p is { Age: < 25 })
        return 1;
    return 10;
}
```

```
int Do(Person? p)
{
    if (p.Age < 25)
        return 1;
    return 10;
}
```

```
p != null && p.Age < 25;
```

- is checks also for null values.
- This is also true if you have nested properties

Bonus: Case Study 5: anonymous functions

What is the output of the following snippet?

```
for (var i = 0; i < 5; i++)
{
    list.Add(() => Console.WriteLine(i));
}

list.ForEach(action => action());
```

Let’s try it out on sharplab.io.

A detailed example can be found here.

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

And of course: You <3