Language execution basics

.NET

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

Contents

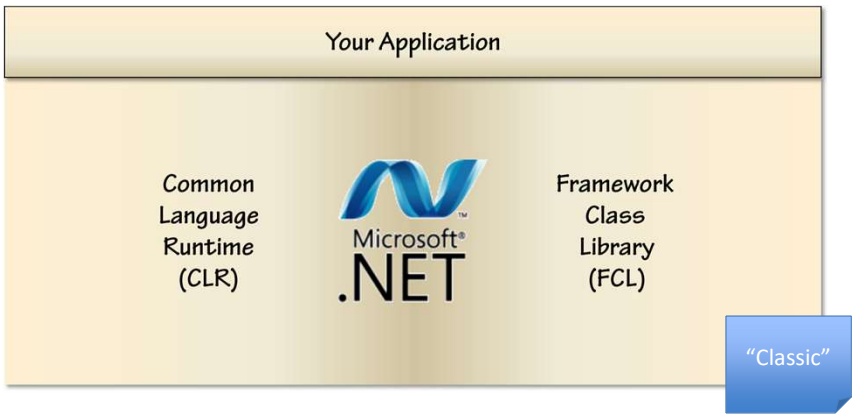
- What is .NET exactly?
  - The Common Language Runtime (CLR)
  - The Framework Class Libraries (FCL)
  - Intermediate Language (IL)
- C# in depth (some examples)
  - C# is strongly typed
  - struct vs class (stack vs heap)
  - Span
- Benchmarking / profiling

PXL IT

This module focuses on the "Virtual Machine" that runs .NET and we start to add some skipped language constructs in C# in order to write more advanced programs. Finally we focus on how to measure performance.

What is .NET?

- A Software framework



Common Language Runtime => Virtual Machine

Framework Class Library => Supporting classes upon which you can build your programs

Types of programs:

- Standalone (desktop)
- Apps
- Web
- Windows Services
- …

Correct version of .NET needs to be installed on the target machine. The shipping version often needs to be updated.

Note: this is the "Classical" framework. We will discuss "the future of .NET" further on.

CLR virtualizes execution and manages it, so you as a developer doesn't have to worry about things like memory management.

.NET languages currently supported: C#, F#, VB, C++ and Python

OS

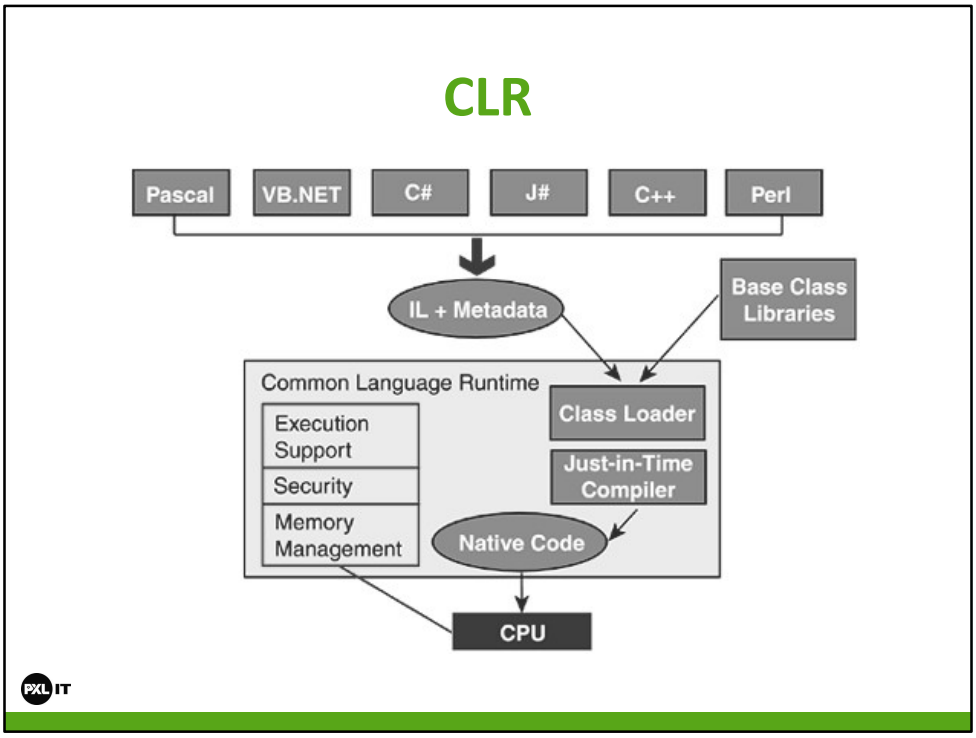
- Windows 95 → Windows 10
- Windows Phone
- Mono: Linux and Mac
- Xamarin: iOS and Android

FCL

- Framework Class Library
  - A library of functionality to build applications
- BCL (Base Class Library)
  - Subset that works everywhere
  - Will be replaced by CoreFx
- Networking, UI, Web, etc

Interesting to follow:

http://blogs.msdn.com/b/dotnet/
https://bcl.codeplex.com/ → https://github.com/dotnet/corefx



The CLR defines a common programming model and a standard type system for cross-platform, multi-language development.

All .NET-aware compilers generate Intermediate Language (IL) instructions and metadata.

The runtime's Just-in-Time (JIT) compiler converts the IL to a machine-specific (native) code when an application actually runs.

Because the CLR is responsible for managing this IL, the code is known as managed code.

Intermediate Language (IL)

- Inspecting IL
  - Virtual Machine Language of the CLR
  - Target language of C#, Visual Basic, etc
  - JIT compiled into native code
- ILDASM
  - Ships with VS
  - IL roundtripping with ILASM (textual language for IL)
- Other tools: .NET Reflector and ILSpy

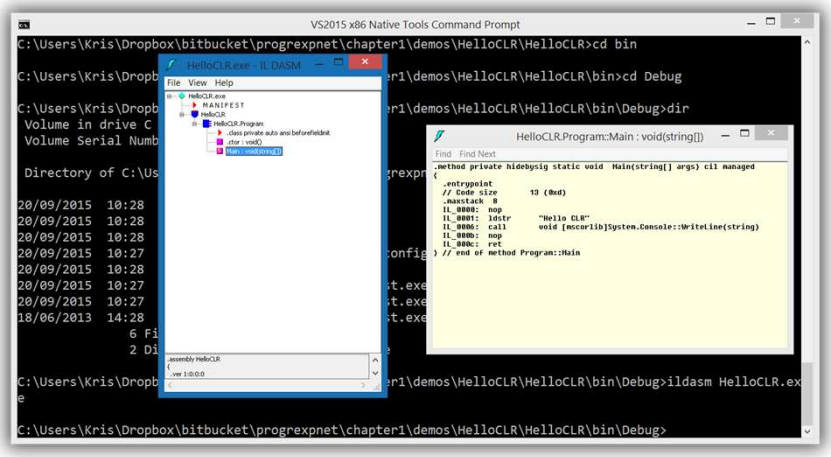
It's interesting to have a basic understanding of the mechanics of IL. There is a basic tool called ILDASM that ships with VS and generates a textual representation of IL. You can actually modify this code and generate IL back (= roundtripping) with ILASM.

Other tools to explore: .NET reflector and ILSpy

https://www.red-gate.com/products/dotnet-development/reflector/ (not free)

http://ilspy.net/ (open source)

Demo ildasm



More info:

https://docs.microsoft.com/en-us/dotnet/framework/tools/ildasm-exe-il-disassembler

https://docs.microsoft.com/en-us/dotnet/framework/tools/developer-command-prompt-for-vs

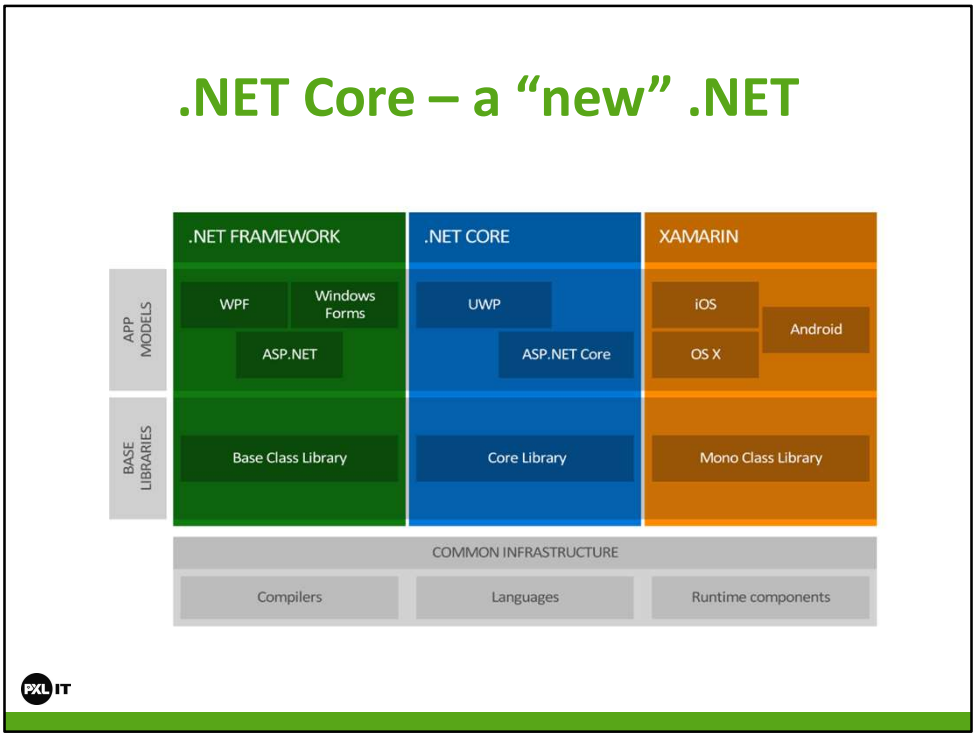
Search for the VS2017 x86 Native Tools Command Prompt, it should be installed together with Visual Studio. This command prompt has several expert tools in its PATH variable, like ildasm.exe

Once opened, go to the Debug folder of your project and execute the command:

ildasm.exe HelloCLR.exe

Ildasm.exe HelloCLR.dll (for .NET Core apps)

Exercise: tryout IlSpy with a Console program that calls a method.



Traditionally, there was the “.NET Framework”:

- You build desktop apps (WPF or Winforms) and web apps (ASP.NET)
- On Windows machines targeting windows machines
- Using the BCL

Then there was Mono, an open source implementation of .NET running on Linux and Mac. From this foundation, Xamarin was build. The “Mono Class Library” is a port of the BCL to the mono runtime. Not everything exists there (like WPF and Winforms).

Now recently “.NET Core” is released, yet again a new beginning:

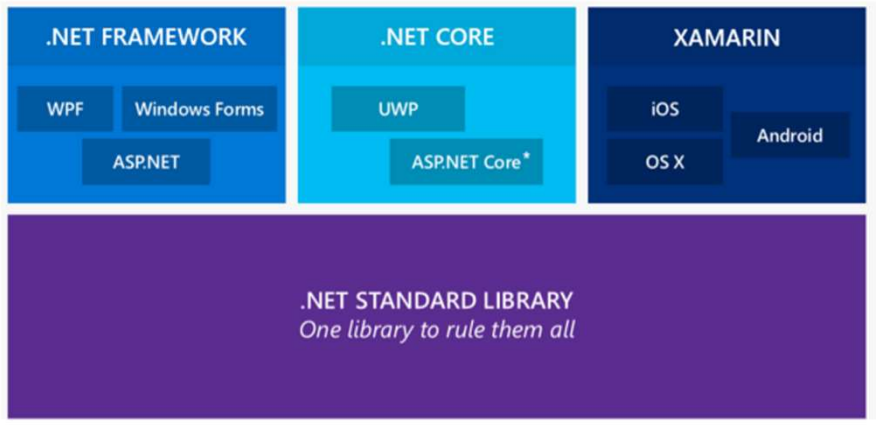
- A .NET runtime running on all major platforms
- Another framework library implementation
- A set of tools (compilers and application host)

Reference (old post):

http://www.telerik.com/blogs/the-new-net-core-1-is-here

.NET Core - .NET Standard

At Build 2016, Scott Hunter presented the following slide:



One library to rule them all, where you don't have to worry about platform incompatibilities.

Reference

https://docs.microsoft.com/en-us/dotnet/standard/net-standard

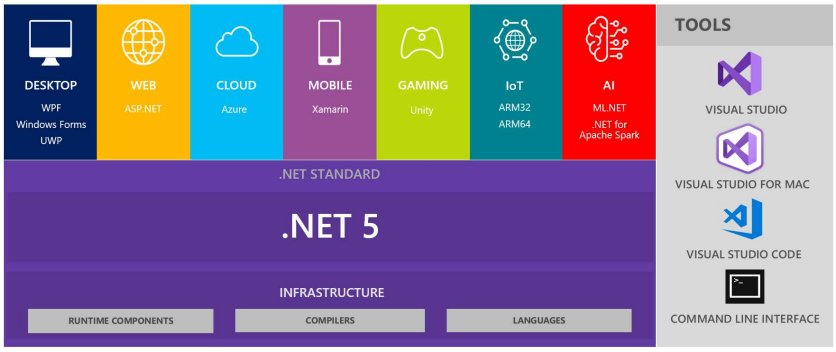
In the above reference, the table shows which runtime platform implements which version of .NET Standard.

Key point to take away: if you use a library in your project, choose a .NET Standard library, not a Portable Class Library. If you are a library author, create .NET Standard libraries as your library will be supported on more platforms.

Demo: .NET Core on linux or Mac

Introducing .NET 5

.NET – A unified platform



https://devblogs.microsoft.com/dotnet/introducing-net-5/

https://devblogs.microsoft.com/dotnet/introducing-net-5/

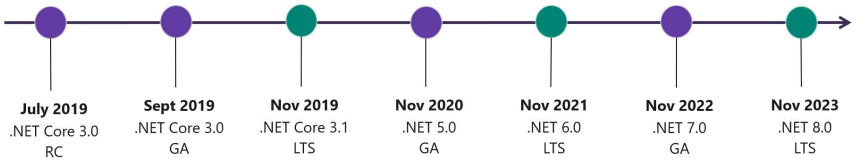
In November 2020, there will be only 1 .NET going forward. You will be able to use it to target Windows, Linux, macOS, iOS, Android, tvOS, watchOS and WebAssembly and more.

.NET 5 = .NET Core vNext

- Produce a single .NET runtime and framework that can be used everywhere and that has uniform runtime behaviors and developer experiences.
- Expand the capabilities of .NET by taking the best of .NET Core, .NET Framework, Xamarin and Mono.
- Build that product out of a single code-base that developers (Microsoft and the community) can work on and expand together and that improves all scenarios.

.NET Schedule

.NET Schedule



- .NET Core 3.0 release in September
- .NET Core 3.1 = Long Term Support (LTS)
- .NET 5.0 release in November 2020
- Major releases every year, LTS for even numbered releases
- Predictable schedule, minor releases if needed

PXL IT

C# is strongly typed

- Every variable should be declared with a type
- Assigning one type to another requires conversion
  - string to int
- Primitive types: int, long, etc.
- Creating a class means defining a new type

Demo: typeconversion

Questions:

- Where are types converted between each other?
- Run this from the console, where is the executable located?
- Create a .NET Core app, how do you run this from the command line?

The var keyword

- Only for local vars
- The type of the variable will (and must) be determined from the initialisation

```
var firstname = Console.ReadLine();

var firstname = "Kris";
```

➔ C# remains strongly typed, the type of the variable is clear at compile-time!

The keyword is only usable for local variables, so not for member variables etc.

So illegal statements are:

var firstname;

var firstname = null;

…

Why is this introduced? → Readability for long classnames or classnames with generics, eg:

ObservableCollection<Employee> list = new ObservableCollection<Employee>();

becomes:

var list = new ObservableCollection<Employee>();

Watch out for readability issues:

```
var a = 12; // int
var b = 123456778990; // long
```

Reference types

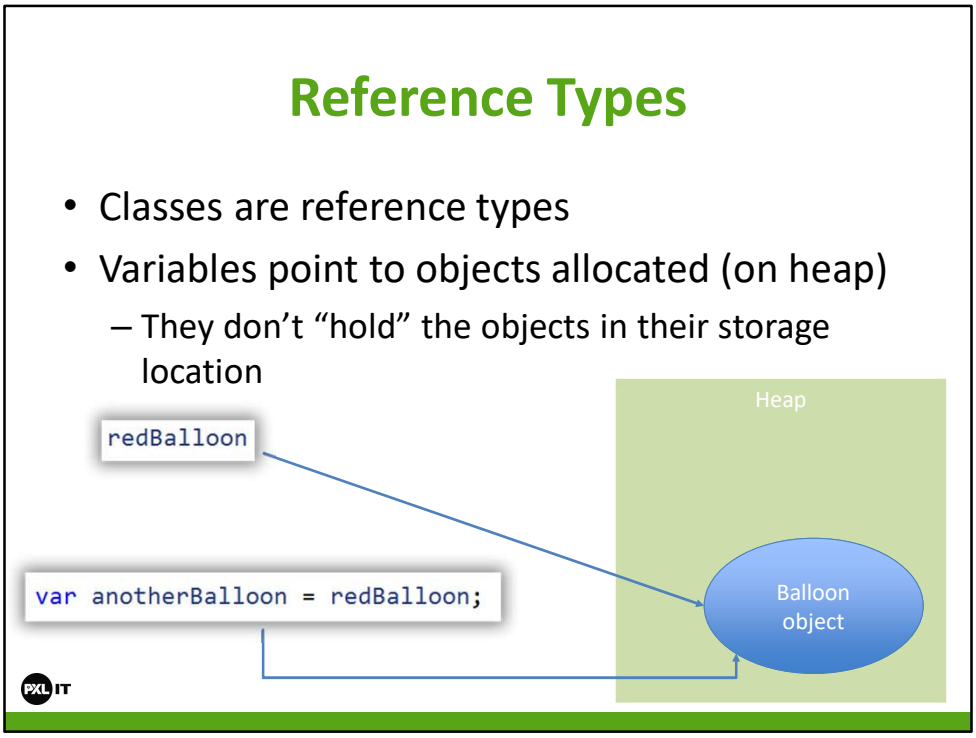
- Denoted by “class”

```
public class Balloon
{
    public double Size { get; set; }
    public string ColorHex { get; set; }
}
```

Reference types

```
[Test]
public void TwoBalloonReferencesSameObject()
{
    var redBalloon = new Balloon()
    {
        ColorHex = "#FF0000",
        Size = 15.0
    };

    var anotherBalloon = redBalloon;

    Assert.AreSame(anotherBalloon, redBalloon);
}
```

Demo: RefsVsValue.BalloonTests



https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types

There are two reference variables (pointers) to the SAME Balloon object.

If you write a class, you create a new reference type.

Note that objects are always allocated on a special storage location called the heap.

Value Types

- Variables hold the value
  - No pointers, no references
- Many built-in primitives are value types
  - Int (System.Int32), DateTime, double, float
- You can make your own with the `struct` keyword
- Value types are fast to instantiate

Reference: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types

Every type is either a reference type, or a value type

A variable of a Value type, holds the actual value.

e.g. int y = 32 ➔ y holds the actual value 32 and not a reference to it.

If you assign a variable to another, you COPY the value

e.g: int x = y → you COPY the value of 32 and assign it to x

Because you don't hold a reference but the actual value, these types are fast to instantiate.

Value Types

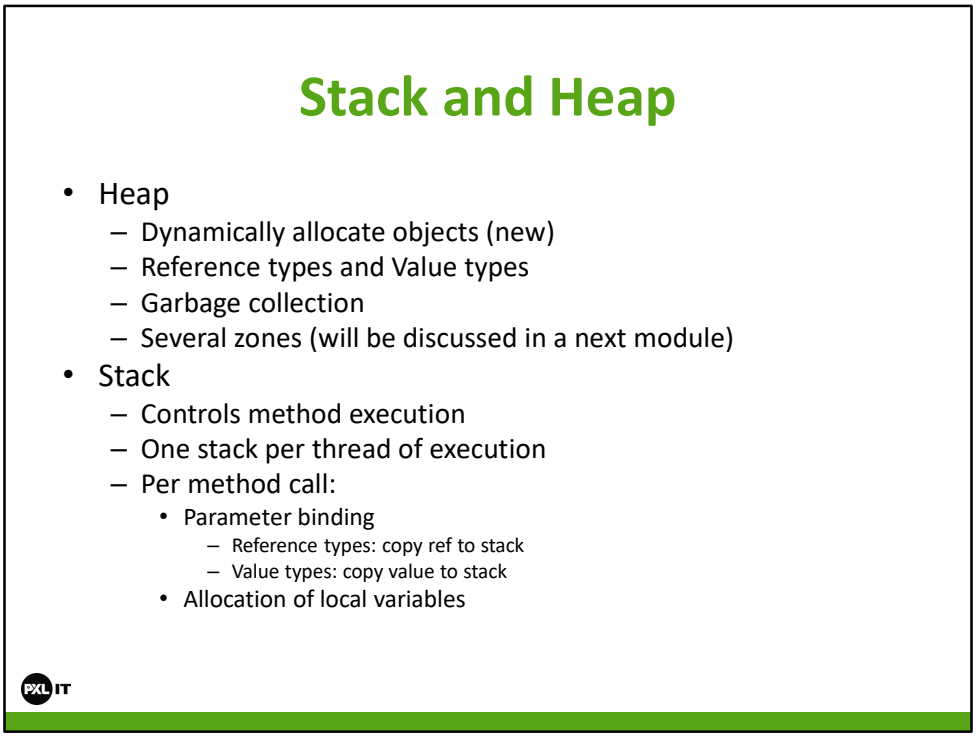
```
[Test]
public void TwoIntsAreNotTheSame()
{
    int x = 32;
    int y = x;
    Assert.AreNotSame(x, y);
}
```

```
[Test]
public void IntIsImmutable()
{
    int x = 32;
    int y = x;

    Assert.AreEqual(x, y);

    x = 33;

    Assert.AreNotEqual(x, y);
}
```

PXL IT

x is assigned to y, but the objects are not the same: explain!

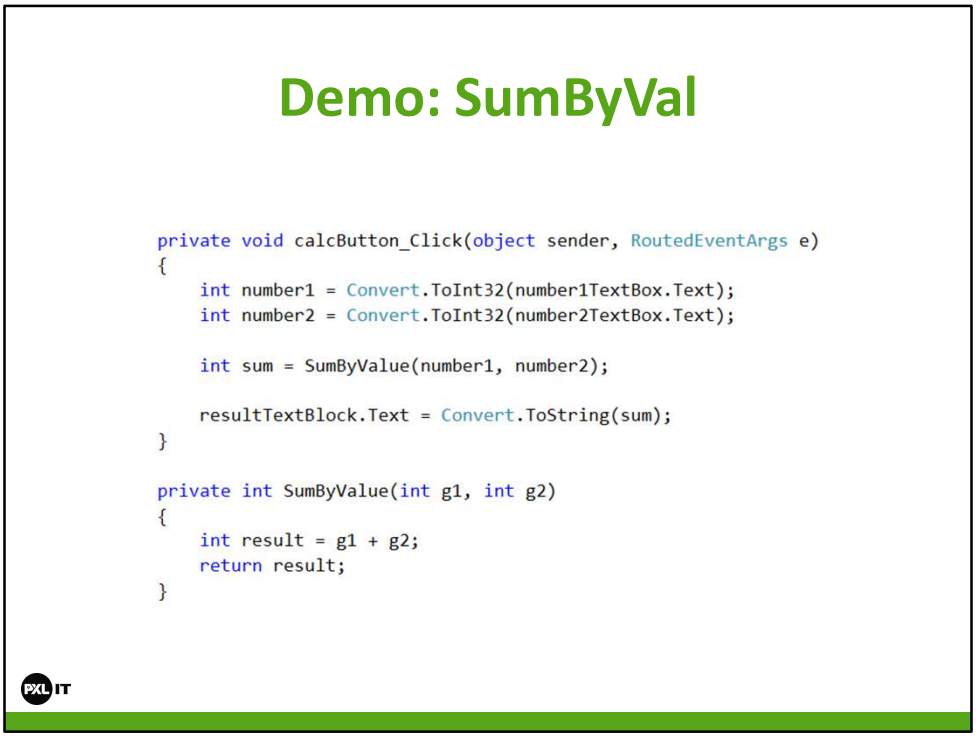
If you change x, y is not changed accordingly, because they are different objects!



When you call a method, a new block of memory is allocated on the Stack. A Stack is a LIFO structure, so the last allocated block is released first (e.g. a stack of cards).

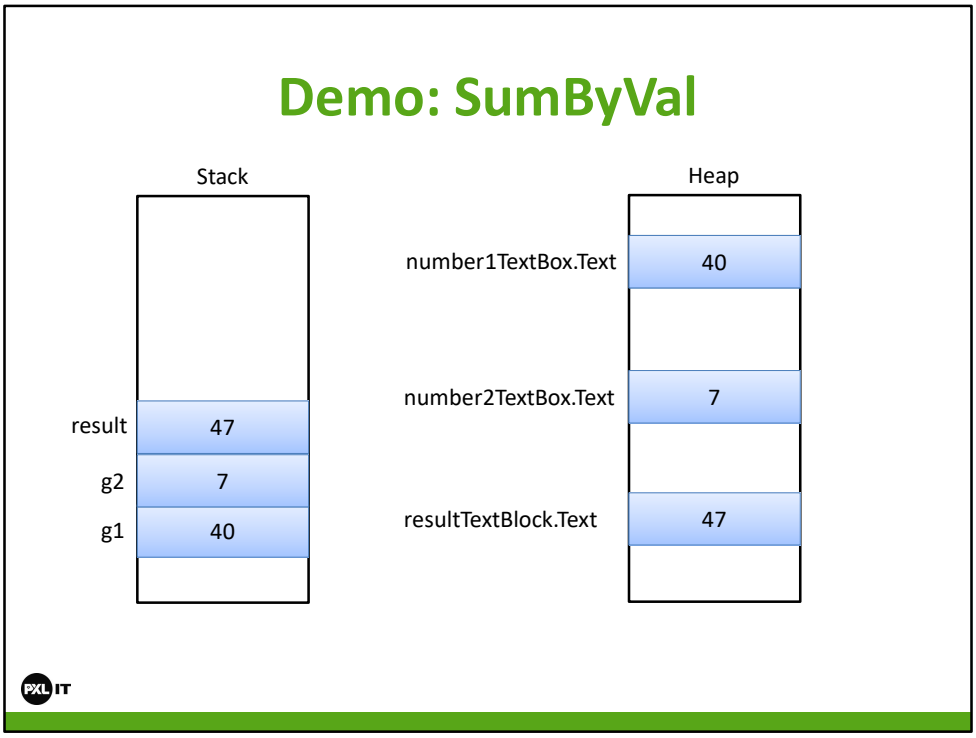
Per block of memory, you will find:

- Actual values of the parameters (parameter binding). Depending on the way this occurs (by value or by reference) you will find different values:
    - Reference types : copy a reference to the object. The actual object resides on the heap
    - Value types: copy the whole object onto the stack. This is a thus a copy.
    - When you use the ref/out keyword, you always copy a reference (pointer) to the object.
- Local variables are also allocated on the stack. However, depending on the type (Value or Reference) you allocate the object itself or a reference to the heap.



Demo: SumByVal

➔ Parameter values are copied on to the Stack



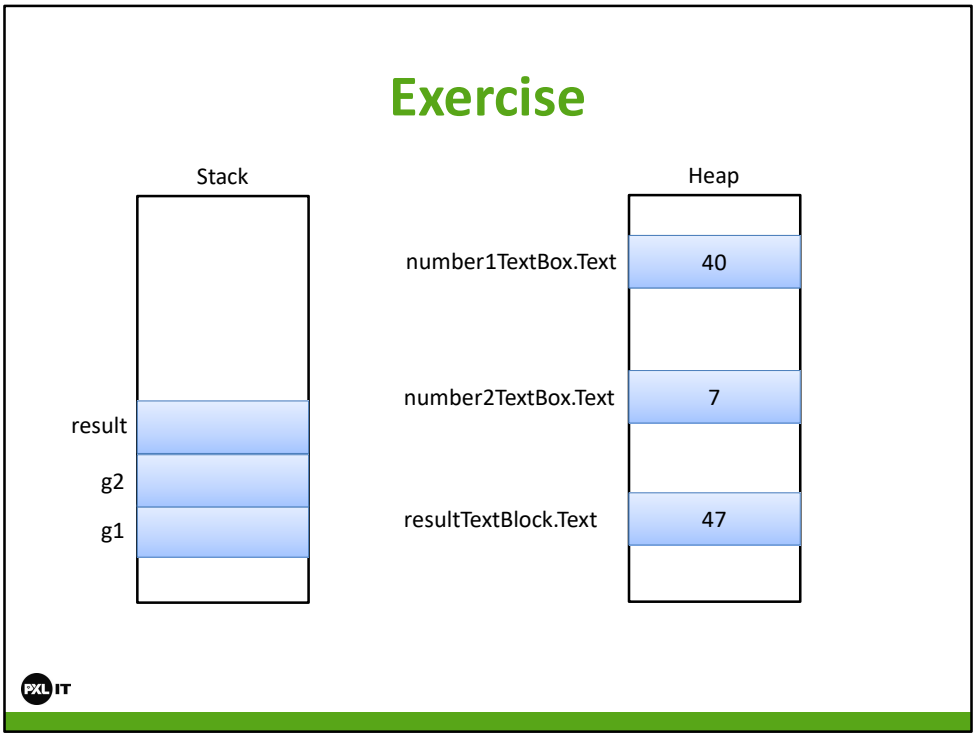
number1TextBox and number2TextBox live on the heap, because they are reference types (class TextBox).

When you execute the method, you copy the int values to the Stack

Demo: SumByRef

```
private void calcButton_Click(object sender, RoutedEventArgs e)
{
    int number1 = Convert.ToInt32(number1TextBox.Text);
    int number2 = Convert.ToInt32(number2TextBox.Text);

    int sum = SumByRef(ref number1, ref number2);

    resultTextBlock.Text = Convert.ToString(sum);
}

private int SumByRef(ref int g1, ref int g2)
{
    int result = g1 + g2;
    return result;
}
```

Now you don’t copy the values onto the stack, but you pass a reference to the original locations on the heap.



number1TextBox and number2TextBox live on the heap, because they are reference types (class TextBox).

When you execute the method, you copy the int values to the Stack

Struct

- Struct definitions create value types
  - Should represent a single value
  - Should be small
  - Are passed by value

```
public struct DateTime
{
    // ...
}
```

When do you create a class and when a struct?

→ Normally always a class, unless you have performance considerations

→ Are passed by value, so copied onto the stack

Example: DateTime, int → String is NO struct!!

More info: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/structs

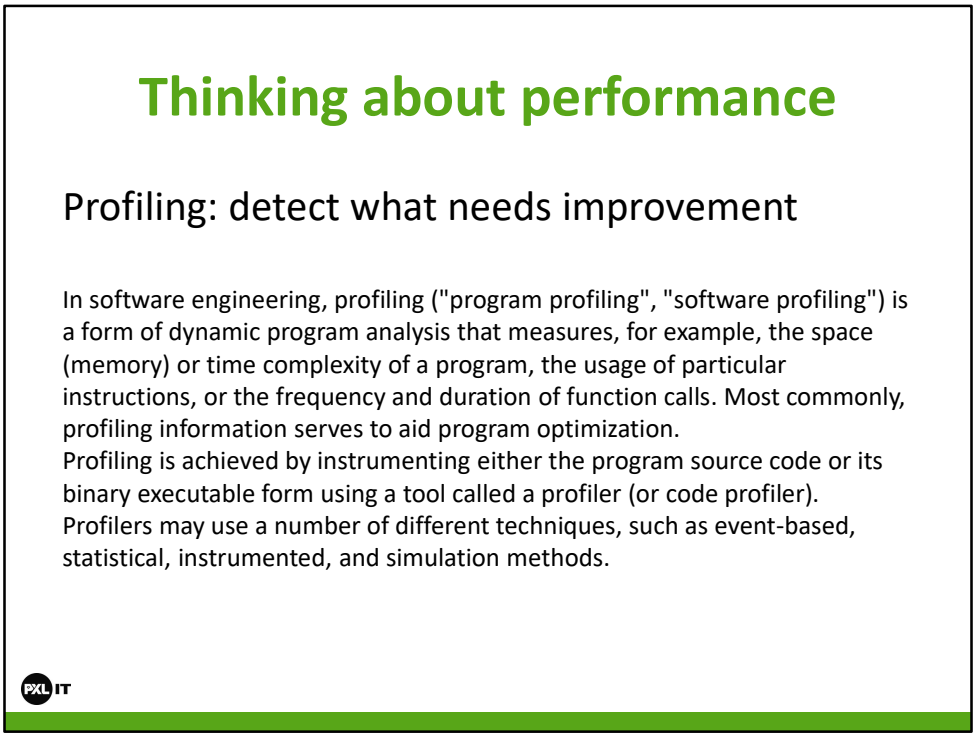
Enum

- An enum creates a value type
  - A set of named constants
  - Underlying data type is int by default

```
public enum PayrollType
{
    Contractor = 1,
    Salaried,
    Executive,
    Hourly
}
```

```
if(employee.Role == PayrollType.Hourly)
{
    // ...
}
```

You could change the data type, but it should be “enumerable” → int, long, etc (no string)



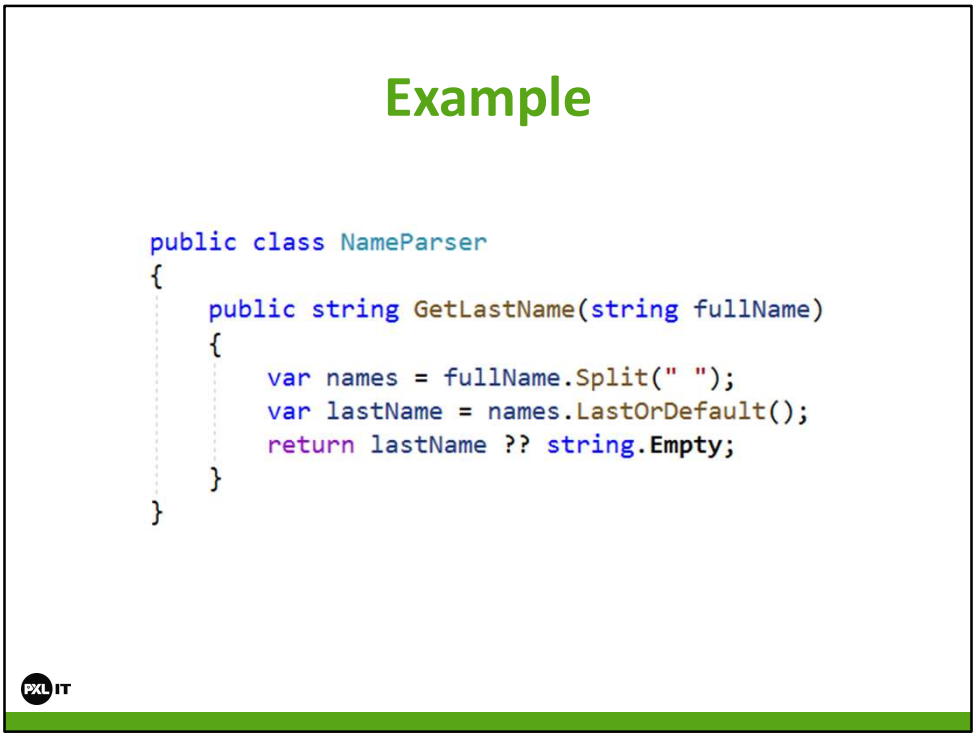
https://en.wikipedia.org/wiki/Profiling_(computer_programming)

You use profilers to detect bottlenecks in your code. Where could you improve? Typically, you profile on two main area's:

1. **Memory allocations**: where in your code occurs the most allocations? Do you free (release) your objects to prevent memory leaks?
2. **CPU**: where in your code is the most time spent by the CPU? These methods are prime candidate for optimisation.

Once you now a method or class that could be improved, you can benchmark it. This means that you measure its performance (execution time, memory consumption) to set a base line. Then you switch the implementation and compare with your original implementation.

Note: benchmarks are always (by definition) run outside the context of the real system in action, so be aware that sometimes results could be not as satisfying as you would like. E.g. : some camera manufacturers optimize only only to get a high DxOMark ranking, but that does not mean your camera will perform good in real life… The same argument can be made for optimizing software.



This is a simple implementation to guess the lastname from a full name. Question: how does it perform? We will figure this out by writing a benchmark.

Attempt 1: Stopwatch

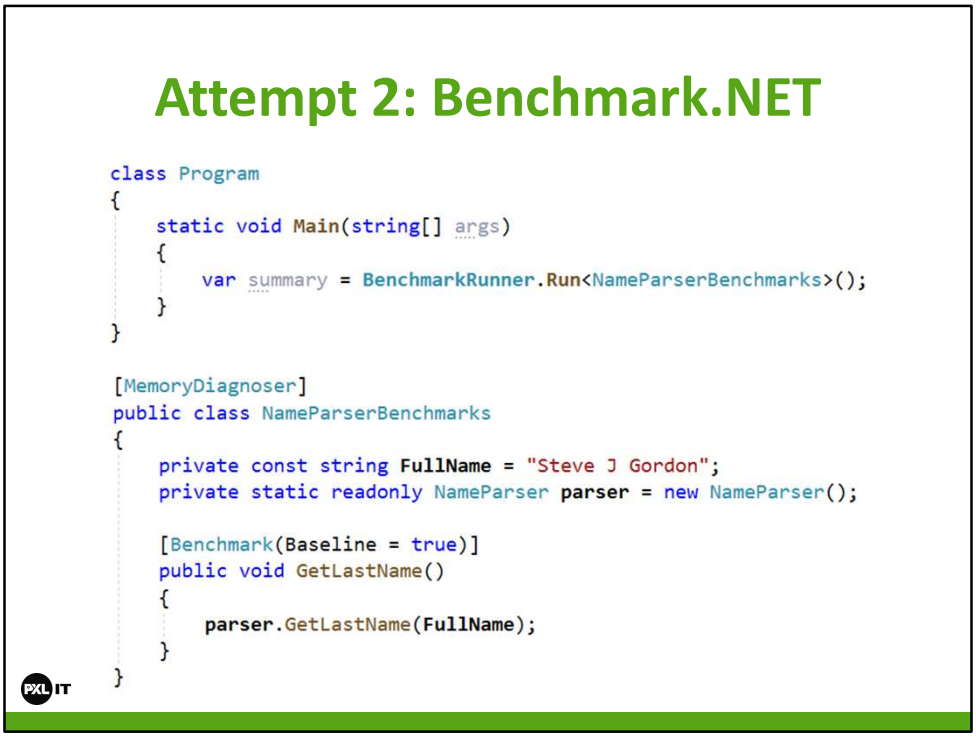
```
class Program
{
    private const string FullName = "Steve J Gordon";
    private static readonly NameParser parser = new NameParser();

    static void Main(string[] args)
    {
        Console.WriteLine("Naive benchmark");

        var stopwatch = new Stopwatch();
        stopwatch.Start();
        parser.GetLastName(FullName);
        stopwatch.Stop();
        Console.WriteLine($"Elapsed time: {stopwatch.ElapsedMilliseconds} ms");
    }
}
```

You should run this from the command line and outside of Visual Studio. Also, use the Release version of your application!

dotnet <<name of dll>>

Problems with this approach

- Not statistically correct
  - You should repeat a lot of times and take average
- Only cold start is measured
- Only elapsed time, no memory consumption
- No output data available (CSV, Json, html, etc)



Demo: Benchmarking

Reference: https://www.stevejgordon.co.uk/introduction-to-benchmarking-csharp-code-with-benchmark-dot-net

Create a new console project and add BenchmarkDotNet NuGet package. Add a reference to the solution under test (this is the same approach as with unit tests).

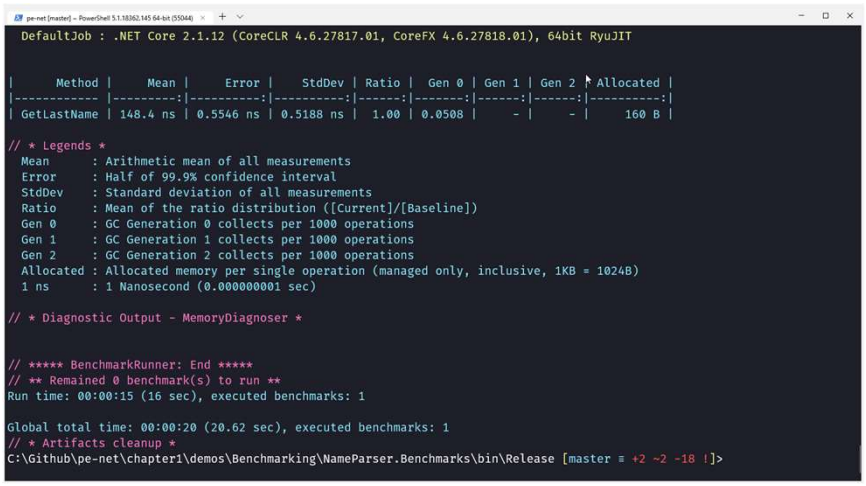
[MemoryDiagnoser]

Collect memory usage

[Benchmark]

Annotates a special kind of “Test”, ie. a benchmark. Baseline = true sets the baseline test to compare with other benchmark methods.

Attempt 2: results



The runner does a warm up fase, then collects data and prints out a nice report.

Note: details in nanoseconds. Difficult to achieve with the Stopwatch class.

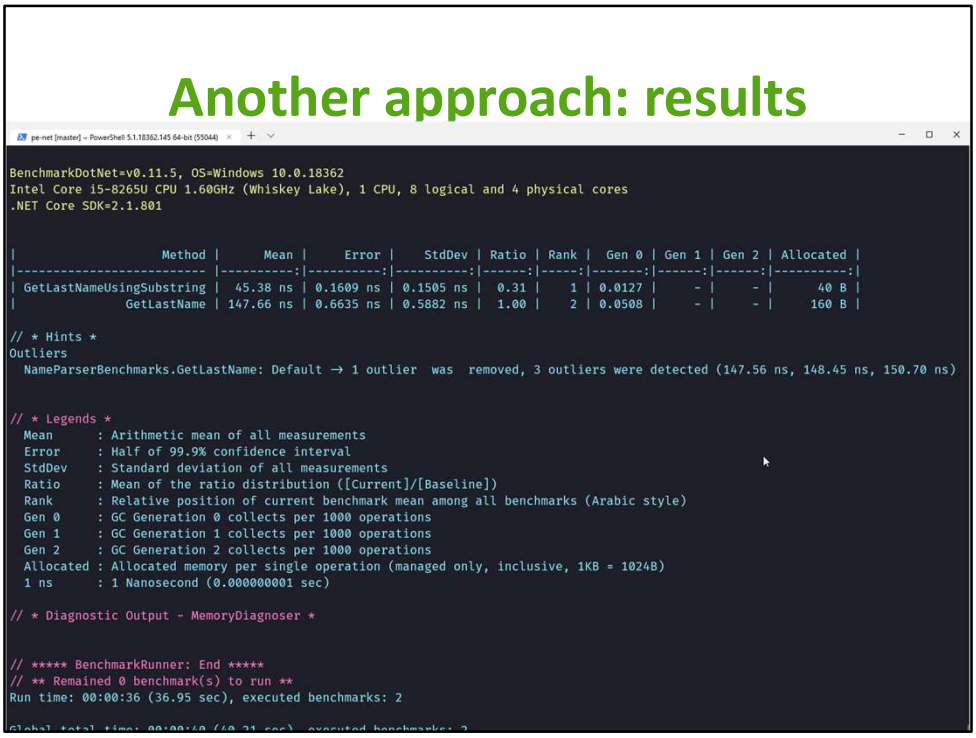
Another approach

```
[RankColumn]
[Orderer(SummaryOrderPolicy.FastestToSlowest)]
[MemoryDiagnoser]
public class NameParserBenchmarks


public string GetLastNameUsingSubstring(string fullName)
{
    var lastSpaceIndex = fullName.LastIndexOf(" ", StringComparison.Ordinal);

    return lastSpaceIndex == -1
        ? string.Empty
        : fullName.Substring(lastSpaceIndex + 1);
}
```

Is this approach faster or slower than the other method?

[RankColumn] and [Ordered…] provide for a sorted output…

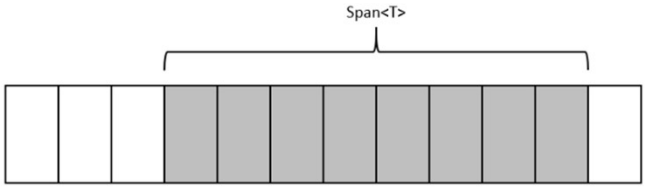
Another approach: results



```
pe-net [master] - PowerShell 5.1.18362.145 64-bit (55044)

BenchmarkDotNet=v0.11.5, OS=Windows 10.0.18362
Intel Core i5-8265U CPU 1.60GHz (Whiskey Lake), 1 CPU, 8 logical and 4 physical cores
.NET Core SDK=2.1.801


|                    Method |      Mean |     Error |    StdDev | Ratio | Rank |  Gen 0 | Gen 1 | Gen 2 | Allocated |
|------------------------- |----------:|----------:|----------:|------:|-----:|-------:|------:|------:|----------:|
| GetLastNameUsingSubstring |  45.38 ns | 0.1609 ns | 0.1505 ns |  0.31 |    1 | 0.0127 |     - |     - |      40 B |
|              GetLastName | 147.66 ns | 0.6635 ns | 0.5882 ns |  1.00 |    2 | 0.0508 |     - |     - |     160 B |

// * Hints *
Outliers
  NameParserBenchmarks.GetLastName: Default → 1 outlier  was  removed, 3 outliers were detected (147.56 ns, 148.45 ns, 150.70 ns)


// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Ratio     : Mean of the ratio distribution ([Current]/[Baseline])
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen 0     : GC Generation 0 collects per 1000 operations
  Gen 1     : GC Generation 1 collects per 1000 operations
  Gen 2     : GC Generation 2 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 ns      : 1 Nanosecond (0.000000001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
// ** Remained 0 benchmark(s) to run **
Run time: 00:00:36 (36.95 sec), executed benchmarks: 2

Global total time: 00:00:40 (40.21 sec), executed benchmarks: 2
```

Span<T>

- Provides type-safe access to a continuous area of memory. This memory can be on the Heap or Stack
- ReadOnlySpan<T> for readonly access
  - Used for immutable types like String
- The Span<T> object itself cannot be used inside a class as a member field

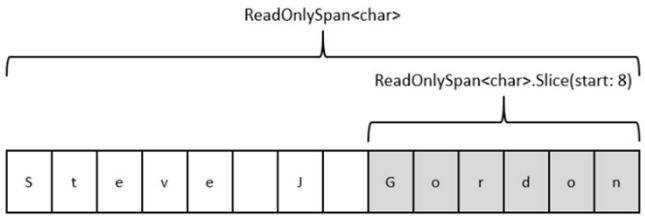


https://www.stevejgordon.co.uk/an-introduction-to-optimising-code-using-span-t

https://docs.microsoft.com/en-us/dotnet/api/system.span-1?view=netstandard-2.1

Version 3

```
public ReadOnlySpan<char> GetLastNameWithSpan(ReadOnlySpan<char> fullName)
{
    var lastSpaceIndex = fullName.LastIndexOf(' ');

    return lastSpaceIndex == -1
        ? ReadOnlySpan<char>.Empty
        : fullName.Slice(lastSpaceIndex + 1);
}
```



Because you use Span, you don't allocate new strings. The Slice methods also accesses the same memory

Final results

```
pe-net [master] – PowerShell 5.1.18362.145 64-bit (55044)
  [Host]     : .NET Core 2.1.12 (CoreCLR 4.6.27817.01, CoreFX 4.6.27818.01), 64bit RyuJIT
  DefaultJob : .NET Core 2.1.12 (CoreCLR 4.6.27817.01, CoreFX 4.6.27818.01), 64bit RyuJIT


|                    Method |      Mean |     Error |    StdDev | Ratio | Rank |  Gen 0 | Gen 1 | Gen 2 | Allocated |
|------------------------- |----------:|----------:|----------:|------:|-----:|-------:|------:|------:|----------:|
|       GetLastNameWithSpan |  17.27 ns | 0.0810 ns | 0.0757 ns |  0.12 |    1 |      - |     - |     - |         - |
| GetLastNameUsingSubstring |  45.41 ns | 0.1385 ns | 0.1296 ns |  0.30 |    2 | 0.0127 |     - |     - |      40 B |
|              GetLastName | 149.63 ns | 0.4468 ns | 0.3960 ns |  1.00 |    3 | 0.0508 |     - |     - |     160 B |

// * Hints *
Outliers
  NameParserBenchmarks.GetLastName: Default → 1 outlier  was  removed (154.89 ns)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Ratio     : Mean of the ratio distribution ([Current]/[Baseline])
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen 0     : GC Generation 0 collects per 1000 operations
  Gen 1     : GC Generation 1 collects per 1000 operations
  Gen 2     : GC Generation 2 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 ns      : 1 Nanosecond (0.000000001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
// ** Remained 0 benchmark(s) to run **
Run time: 00:00:55 (55.41 sec), executed benchmarks: 3

Global total time: 00:00:58 (58.71 sec), executed benchmarks: 3
// * Artifacts cleanup *
```

Span allocates NO memory!

Some interesting tools and refs

- ILDASM / ILASM
- ILSpy
- JetBrains: dotTrace
- BenchmarkDotNet