Turbocharged:

Writing High-Performance C# and .NET Code

@stevejgordon
https://stevejgordon.co.uk

Resources: **http://bit.ly/highperfdotnet**

Microsoft Most Valuable Professional

.NET southeast
https://www.meetup.com/dotnetsoutheast

PLURALSIGHT

madgex

Aspects of Performance

Execution Time

Throughput

Memory Allocations

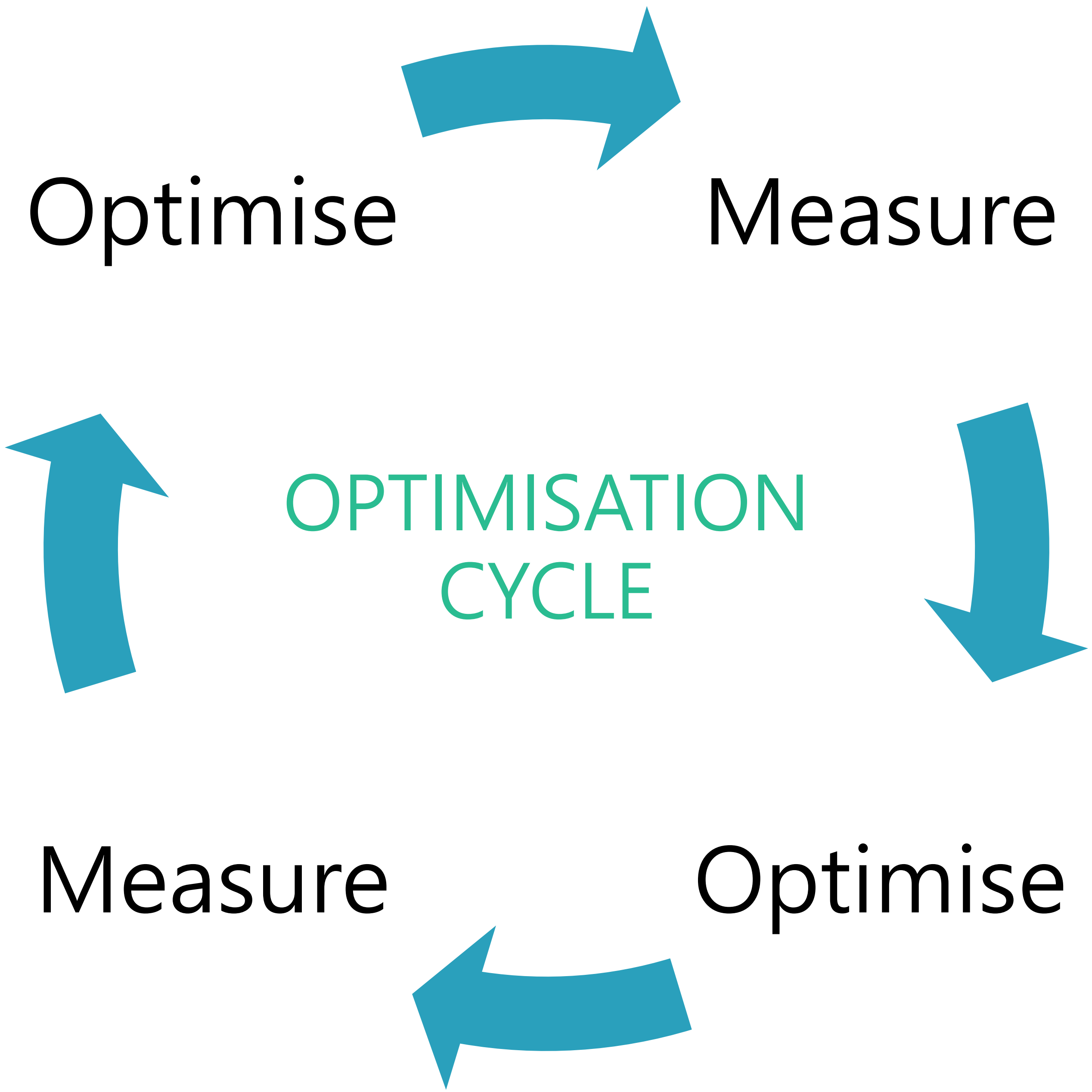
PERFORMANCE

IS

CONTEXTUAL

PERFORMANCE

READABILITY


Optimise
Measure
OPTIMISATION CYCLE
Measure
Optimise

Measuring Application Performance

- Visual Studio Diagnostic Tools (debugging)
- Visual Studio Profiling / PerfView / dotTrace / dotMemory
- ILSpy / JustDecompile / dotPeek
- Production metrics and monitoring

BENCHMARK.NET

https://benchmarkdotnet.org

```csharp
namespace BenchmarkExample
{
    public class Program
    {
        public static void Main(string[] args) =>
            _ = BenchmarkRunner.Run<NameParserBenchmarks>();
    }

    [MemoryDiagnoser]
    public class NameParserBenchmarks
    {
        private const string FullName = "Steve J Gordon";
        private static readonly NameParser Parser = new NameParser();

        [Benchmark]
        public void GetLastName()
        {
            Parser.GetLastName(FullName);
        }
    }
}
```

```csharp
namespace BenchmarkExample
{
    public class Program
    {
        public static void Main(string[] args) =>
            _ = BenchmarkRunner.Run<NameParserBenchmarks>();
    }

    [MemoryDiagnoser]
    public class NameParserBenchmarks
    {
        private const string FullName = "Steve J Gordon";
        private static readonly NameParser Parser = new NameParser();

        [Benchmark]
        public void GetLastName()
        {
            Parser.GetLastName(FullName);
        }
    }
}
```

```
namespace BenchmarkExample
{
    public class Program
    {
        public static void Main(string[] args) =>
            _ = BenchmarkRunner.Run<NameParserBenchmarks>();
    }

    [MemoryDiagnoser]
    public class NameParserBenchmarks
    {
        private const string FullName = "Steve J Gordon";
        private static readonly NameParser Parser = new NameParser();

        [Benchmark]
        public void GetLastName()
        {
            Parser.GetLastName(FullName);
        }
    }
}
```

```csharp
namespace BenchmarkExample
{
    public class Program
    {
        public static void Main(string[] args) =>
            _ = BenchmarkRunner.Run<NameParserBenchmarks>();
    }

    [MemoryDiagnoser]
    public class NameParserBenchmarks
    {
        private const string FullName = "Steve J Gordon";
        private static readonly NameParser Parser = new NameParser();

        [Benchmark]
        public void GetLastName()
        {
            Parser.GetLastName(FullName);
        }
    }
}
```

```
namespace BenchmarkExample
{
    public class Program
    {
        public static void Main(string[] args) =>
            _ = BenchmarkRunner.Run<NameParserBenchmarks>();
    }

    [MemoryDiagnoser]
    public class NameParserBenchmarks
    {
        private const string FullName = "Steve J Gordon";
        private static readonly NameParser Parser = new NameParser();

        [Benchmark]
        public void GetLastName()
        {
            Parser.GetLastName(FullName);
        }
    }
}
```

```
// * Summary *
BenchmarkDotNet=v0.11.5, OS=Windows 10.0.18362
Intel Core i7-6700 CPU 3.40GHz (Skylake), 1 CPU, 8 logical and 4 physical cores
.NET Core SDK=3.0.100
  [Host]     : .NET Core 3.0.0 (CoreCLR 4.700.19.46205, CoreFX 4.700.19.46214), 64bit RyuJIT
  DefaultJob : .NET Core 3.0.0 (CoreCLR 4.700.19.46205, CoreFX 4.700.19.46214), 64bit RyuJIT
```

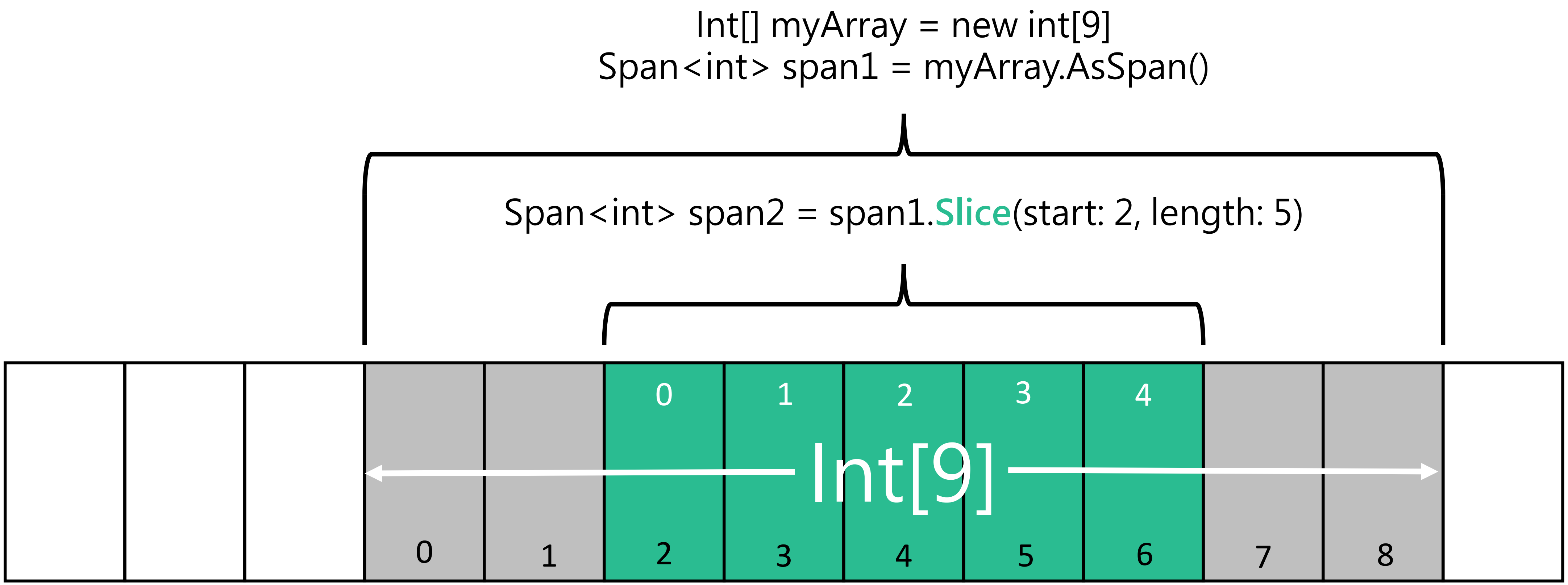
Method	Mean	Error	StdDev	Median	Gen 0	Gen 1	Gen 2	Allocated
GetLastName	163.18 ns	3.1903 ns	4.2590 ns	161.87 ns	0.0379	-	-	160 B

(1 / 0.0379) x 1000 = 26,385.2 operations before Gen 0 collection.

Span<T>

- **System.Memory** package. Built into .NET Core 2.1.
- Provides a read/write 'view' onto a contiguous region of memory
  - Heap (Managed objects) – e.g. Arrays, Strings
  - Stack (via stackalloc)
  - Native/Unmanaged (P/Invoke)
- Index / Iterate to modify the memory within the Span
- Almost no overhead

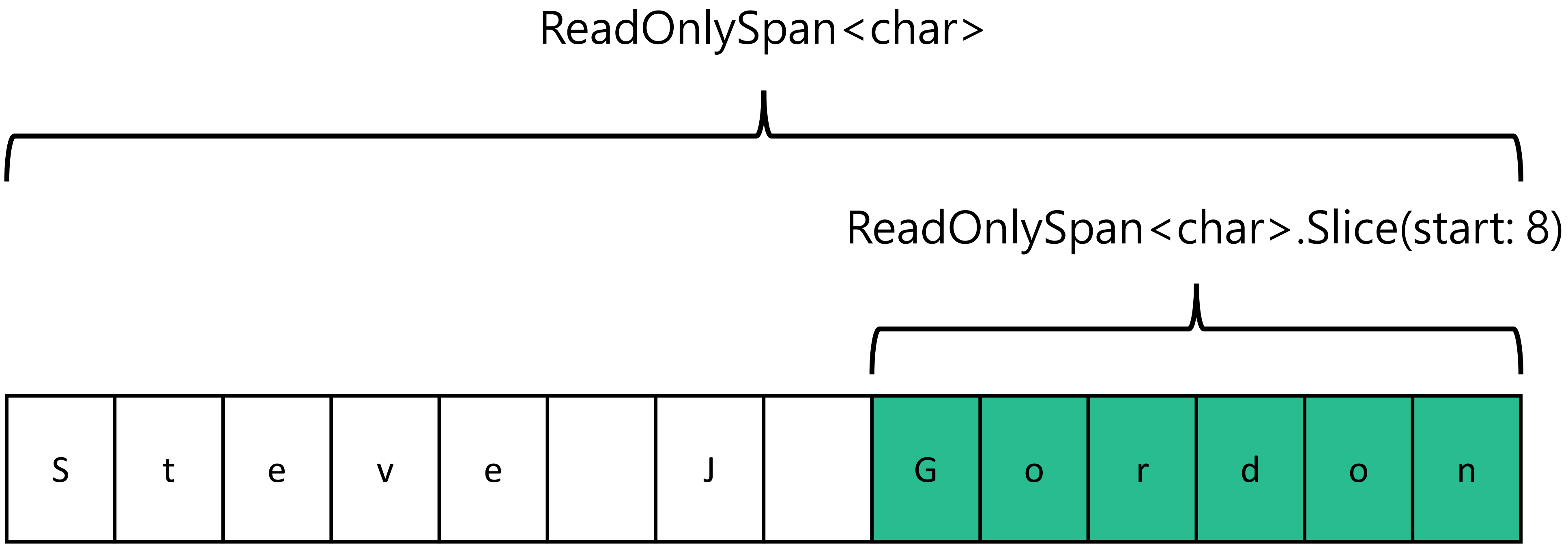
Span<T>.Slice



Slicing a Span is a constant time/cost operation – O(1)

Working with Strings

```
ReadOnlySpan<char> span = "Steve J Gordon".AsSpan();
```



Span<T> Limitations

- It's a **stack only** Value Type - ref struct
- Requires C# >= 7.2 for ref struct feature
- Cannot be boxed
- Cannot be a field in a class or standard (non ref) struct
- Cannot be used as an argument or local variable inside async methods
- Cannot be captured by lambda expressions

Memory<T>

- Similar to Span<T> but can live on the heap
- A readonly struct but not a ref struct
- Slightly slower to slice into Memory<T>
- Call its Span property to get a Span over the same data

```csharp
// CS4012 Parameters or locals of type 'Span<byte>' cannot be declared
// in async methods or lambda expressions.
private async Task SomethingAsync(Span<byte> data)
{
    ... // Would be nice to do something with the Span here

    await Task.Delay(1000);
}
```

```
private async Task SomethingAsync(Memory<byte> data)
{
    ...

    await Task.Delay(1000);
}
```

```csharp
private async Task SomethingAsync(Memory<byte> data)
{
    Memory<byte> dataSliced = data.Slice(0, 100);

    await Task.Delay(1000);
}
```

```
private async Task SomethingAsync(Memory<byte> data)
{
    Memory<byte> dataSliced = data.Slice(0, 100);

    await Task.Delay(1000);
}

private void SomethingNotAsync(Span<byte> data)
{
    // some code
}
```

```
private async Task SomethingAsync(Memory<byte> data)
{
    SomethingNotAsync(data.Span.Slice(1));

    await Task.Delay(1000);
}

private void SomethingNotAsync(Span<byte> data)
{
    // some code
}
```

Putting it into practice – Key Builder

Microservice which:

1. Reads SQS message
2. Deserialise the JSON message
3. Stores a copy of the message to S3 using an object key derived from properties of the message.

S3ObjectKeyGenerator

Object Key Builder Benchmarks

Method	Mean	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Original	1,088.0 ns	1.00	0.1812	-	-	1144 B
SpanBased	449.0 ns	0.41	0.0305	-	-	192 B

~2.5x Faster
~6x Less Allocations

18 million messages:

Reduction of 17GB of allocations daily

Removes approx. 2711 (~2 per minute) Gen 0 collections (562 vs. 3273)

ArrayPool

- Pool of arrays for re-use
- Found in System.Buffers
- ArrayPool<T>.Shared.Rent(int length)
- You are likely to get an array larger than your minimum size
- ArrayPool<T>.Shared.Return(T[] array, bool clearArray = false)
- Warning! By default returned arrays are not cleared in .NET Core <= 2.2

```csharp
public class Processor
{
    public void DoSomeWorkVeryOften()
    {
        var buffer = new byte[1000];  // allocates

        DoSomethingWithBuffer(buffer);
    }

    private void DoSomethingWithBuffer(byte[] buffer)
    {
        // use the array
    }
}
```

```csharp
public class Processor
{
    public void DoSomeWorkVeryOften()
    {
        var arrayPool = ArrayPool<byte>.Shared;
        var buffer = arrayPool.Rent(1000);

        DoSomethingWithBuffer(buffer);
    }

    private void DoSomethingWithBuffer(byte[] buffer)
    {
        // use the array
    }
}
```

```csharp
public class Processor
{
    public void DoSomeWorkVeryOften()
    {
        var arrayPool = ArrayPool<byte>.Shared;
        var buffer = arrayPool.Rent(1000);

        try
        {
            DoSomethingWithBuffer(buffer);
        }
        finally
        {
            arrayPool.Return(buffer);
        }
    }

    private void DoSomethingWithBuffer(byte[] buffer)
    {
        // use the array
    }
}
```

System.IO.Pipelines

- Originally created by ASP.NET team to improve Kestrel rps
- Improves I/O performance scenarios (~2x vs. streams)
- Removes common hard to write, boilerplate code
- Unlike streams, pipelines manages buffers for you from the ArrayPool
- Two ends to a pipe, a **PipeWriter** and a **PipeReader**

Pipelines

PipeWriter : IBufferWriter<byte>

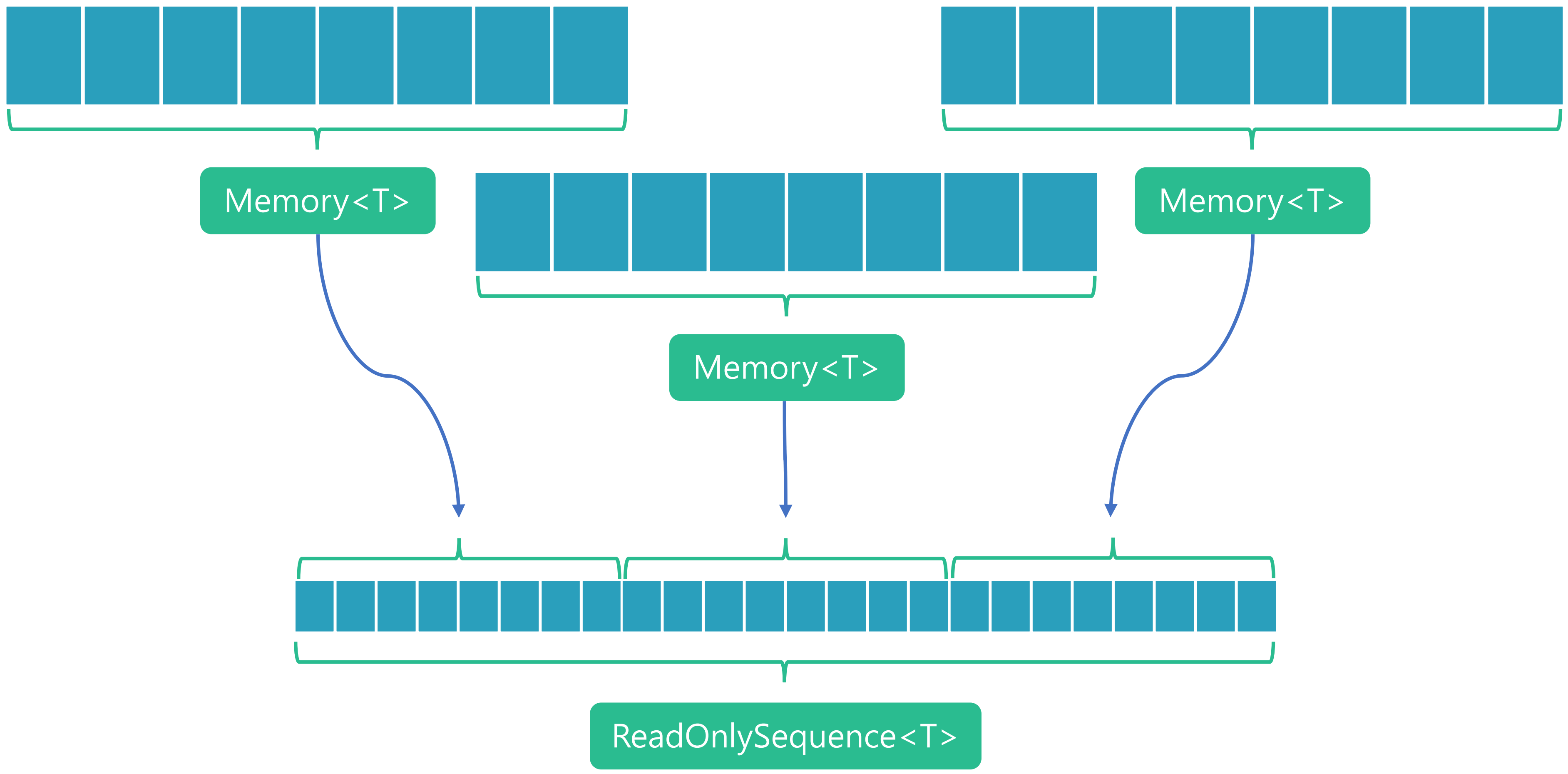
```
Memory<byte> m = pw.GetMemory();
…
pw.Advance(1000)
await pw.FlushAsync()
```

Pipe

PipeReader

```
ReadResult r = await reader.ReadAsync();
ReadOnlySequence<byte> b = r.Buffer;
```

ReadOnlySequence<T>



Putting it into practice – Span<T> Parsing

Microservice which:

1. Retrieves S3 object (TSV file) from AWS
2. Decompresses file
3. Parses TSV to get 3 of 25 columns for each row
4. Indexes data to ElasticSearch

CloudFrontParser

TSV Parsing Optimisation - Results

Processing 75 files of 10,000 rows each

Method	Mean	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Original	8,500.9 ms	1.00	1548000.0	267000.0	109000.0	7205.44 MB
Optimised	957.5 ms	0.11	43000.0	20000.0	2000.0	242.41 MB

~9x Faster

~30x Less Heap Memory Allocated
NOTE: ~203.5Mb are the string allocations for the parsed data

Business Buy-In

- Identify a quick win
- Use a scientific approach to demonstrate gains
- Put gains into a monetary value
- Cost to benefit ratio

Cost Saving Example: Input Processor

For a single microservice handling

18 million messages per day

~50% fewer allocations

~2x message per second throughput

~1 less VM required in the container cluster

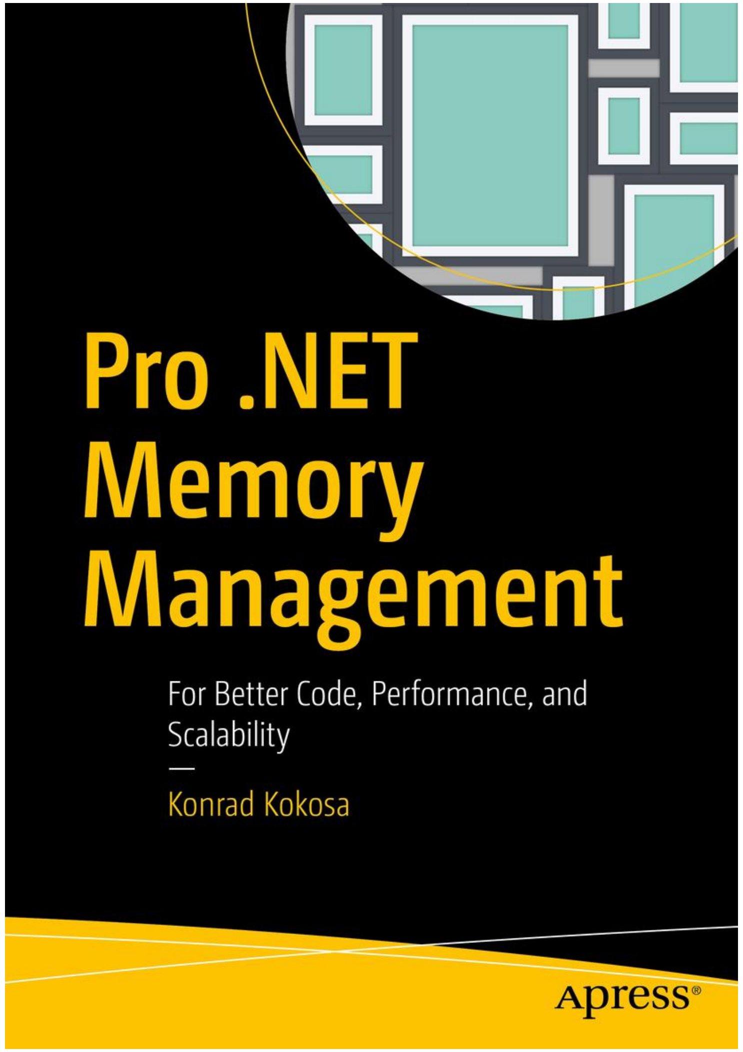
Saving $1,700

Summary

- Everything you’ ve seen if for mostly advanced situations.
- Measure, don't assume!
- Be scientific; make small changes each time and measure again
- Focus on hot paths
- Don't copy memory, slice it! Span<T> is less complex than it may first seem.
- Use ArrayPools where appropriate to reduce array allocations
- Consider Pipelines for I/O scenarios

Pro .NET Memory Management

By Konrad Kokosa



Thanks for listening!

@stevejgordon

www.stevejgordon.co.uk

http://bit.ly/highperfdotnet

Microsoft® Most Valuable Professional

.NET southeast

https://www.meetup.com/dotnetsoutheast

madgex