A webinar series brought to you by CODE Magazine

CODE PRESENTS

Improving String Handling Performance in C#

A webinar *and* CODE Magazine article presented by Joydip Kanjilal

Ask CODE Magazine authors questions about the topics they write about!

Joydip Kanjilal
CODE Magazine Author
MVP (2007-2012)

Kicking Things Off

Jim Duffy

- Director of Business Development
  CODE Magazine & Consulting
- Former Developer – Drawn to the Dark Side: Now Responsible for Marketing & Business Development
- jduffy@codemag.com / My Bio
- International Author and Speaker
- Former Microsoft RD (Regional Director) 9 years
- Former 11-time Microsoft Most Valuable Professional (MVP)
- Twitter: @jmduffy

Vue Plug-ins, Entity Framework, REST, Ajax
CODE
Understanding Docker
Tapping into the EF Core Pipeline
More Vue Plug-Ins
Deploying ExpressJS Applications

Entity Framework, Web Workers, Vue 3, ExpressJS, ASP.NET Core
CODE
Amplify the cloud
The Wonderful World of Web Workers
Discovering Data with Azure Data Explorer
The Magic of Morning Pages

About CODE Consulting

"Helping People Build Better Software"

- Microsoft Certified Partner
- Custom Software Development, Training, Mentoring,...
- Web, Cloud, Mobile, Desktop, Serverless, Databases,...
- User Interface and Interaction Design
- Project Rescue, App Modernization (VB, VFP, Access, etc.)
- Development Team Staff Augmentation

CODE
ASP.NET Core, NestJS, SwiftUI, AWS
Stay at
ASP.NET Core, Vue.js, Amazon Web Services, Power Query
Transform Your ASP.NET Core API into AWS Lambda Functions
Exploring What's New in Vue 3
Building WPF Apps with Custom Scripts
Working with Managed Identity

Your Ticket to Free Consulting

- One hour on us. Really. Schedule a call today. Slots are limited.
- No strings. No commitment. No credit card required.
- Just help from our team of experienced software developers.
- Got questions? Stuck on an issue? Platform and/or architecture decisions to make? We can help!

Contact us at:
info@codemag.com or
jduffy@codemag.com

CODE
CONSULTING

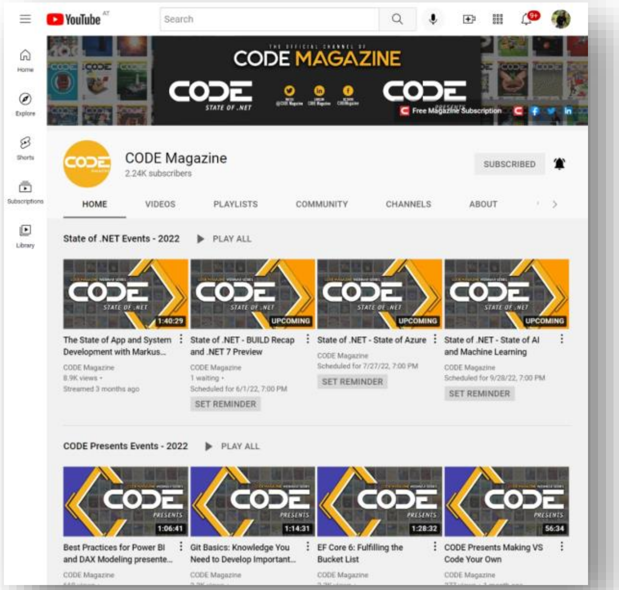
CODE Is Hiring!

We Need Developers!

- We have **current** openings for:
  - Data Engineer
  - Python Developer
  - Senior C# Developer
  - .NET Desktop Developer
  - REACT & JavaScript Developers
- Details here: https://codemag.com/Jobs

Join Us On Social Media

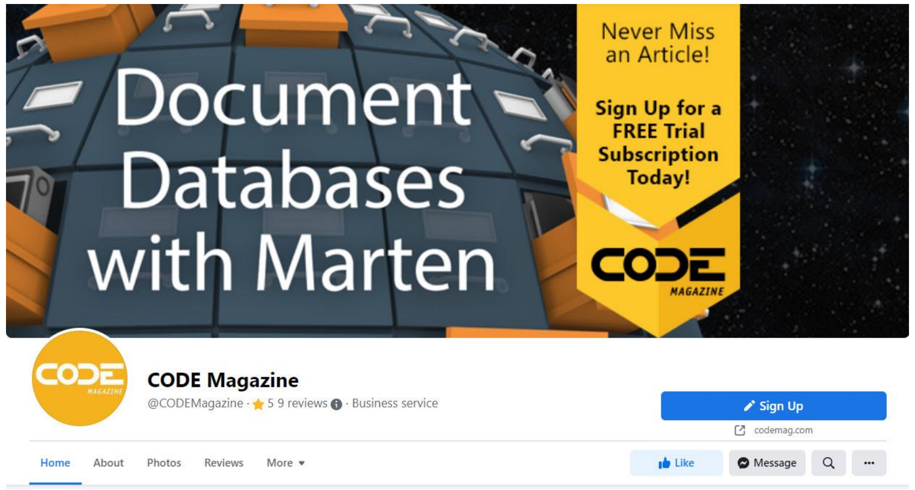
- YouTube channel: https://tinyurl.com/CODEYTC
- Twitter: https://twitter.com/CODEmagazine
- Facebook: www.facebook.com/CODEMagazine/
- LinkedIn: https://www.linkedin.com/company/code-magazine/



The State of App and System Development with Markus...
State of .NET - BUILD Recap and .NET 7 Preview
State of .NET - State of Azure
State of .NET - State of AI and Machine Learning
Best Practices for Power BI and DAX Modeling presente...
Git Basics: Knowledge You Need to Develop Important...
EF Core 6: Fulfilling the Bucket List
CODE Presents Making VS Code Your Own

CODE Magazine - In Print & Digital Formats!
6,692 Tweets

CODE Magazine - In Print & Digital Formats!
@CODEmagazine
Follow us for in-depth articles for software developers and their managers.



Document Databases with Marten
Never Miss an Article!
Sign Up for a FREE Trial Subscription Today!
CODE MAGAZINE

CODE Magazine
@CODEMagazine · 5 9 reviews · Business service

Free Subscription to CODE Magazine!

- The leading software development magazine written by expert developers for developers.
- All registered attendees will automatically receive a free digital subscription to CODE Magazine – no need to do anything, it'll happen auto-magically.
- Upgrade to Print and receive our .NET 7 CODE Focus issue FREE! (while supplies last)

- Please share this free subscription link: https://bit.ly/3NTrS6F

MAUI, EF 7, Minimal APIs, Blazor, C# 11
CODE FOCUS
.NET 7
What's new in the latest and fastest release yet?
C# Goes to 11
Sailing to MAUI
.NET 7 Performance Deep Dive

C# 10, .NET MAUI, Blazor, Visual Studio 2022
CODE FOCUS
.NET 6.0
Explore What's New in ASP.NET Core
Performance Gains in EF CORE
Building Power Apps with .NET 6

CODE Mobile App

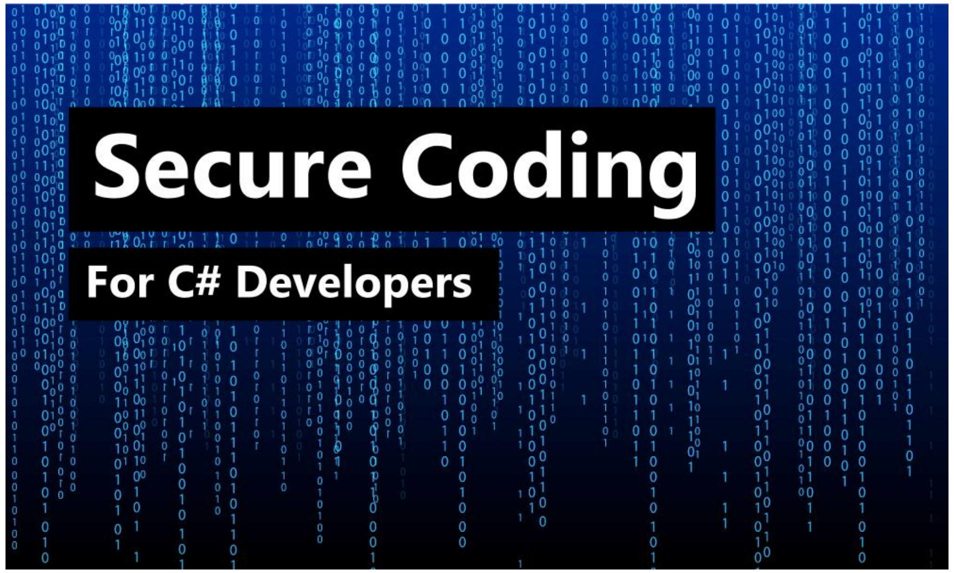
- Check out the new CODE Magazine Mobile application!
- Available for iOS & Android

Download on the App Store

GET IT ON Google Play

New CODE Training Course

- 3-Day
- Online
- Hands-on
- Learn to develop robust and secure C# applications
- January 24-26 https://bit.ly/3G3WMr1
- March 7-9 https://bit.ly/3Txcvlf
- April 18-20 https://bit.ly/3WRXugZ
- June 6-8 https://bit.ly/3ObqWe5
- https://www.codemag.com/training



Build Cross Platform Apps With Photino

Photino

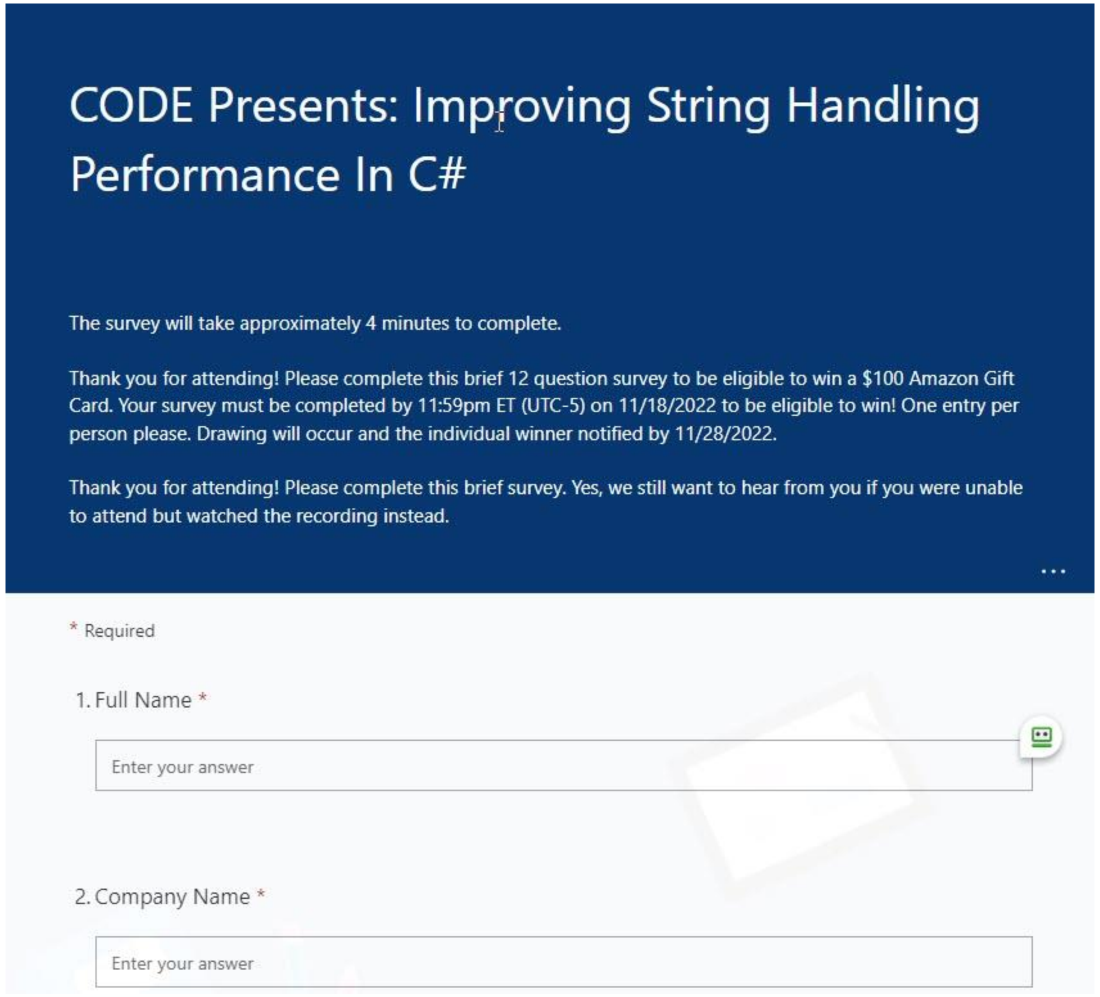
- Build native, cross-platform desktop apps that are lighter than light.
- Lightweight **open-source** framework for building native, cross-platform desktop applications with Web UI technology.
- Photino is maintained by the CODE Magazine team with the help of the open-source community.
- tryphotino.io
- Github.com/tryphotino

Event Survey – Win $100!

- **Starting at 1:15pm ET** Complete this very short 12 question survey for a chance at a $100 Amazon Gift Card!

https://bit.ly/3UI4jQV

- Survey must be completed by **11:59pm ET** on **Friday 11/18/2022** to be eligible!
- Completed survey is required to be eligible.



CODE Presents: Improving String Handling Performance In C#

The survey will take approximately 4 minutes to complete.

Thank you for attending! Please complete this brief 12 question survey to be eligible to win a $100 Amazon Gift Card. Your survey must be completed by 11:59pm ET (UTC-5) on 11/18/2022 to be eligible to win! One entry per person please. Drawing will occur and the individual winner notified by 11/28/2022.

Thank you for attending! Please complete this brief survey. Yes, we still want to hear from you if you were unable to attend but watched the recording instead.

* Required

1. Full Name *

Enter your answer

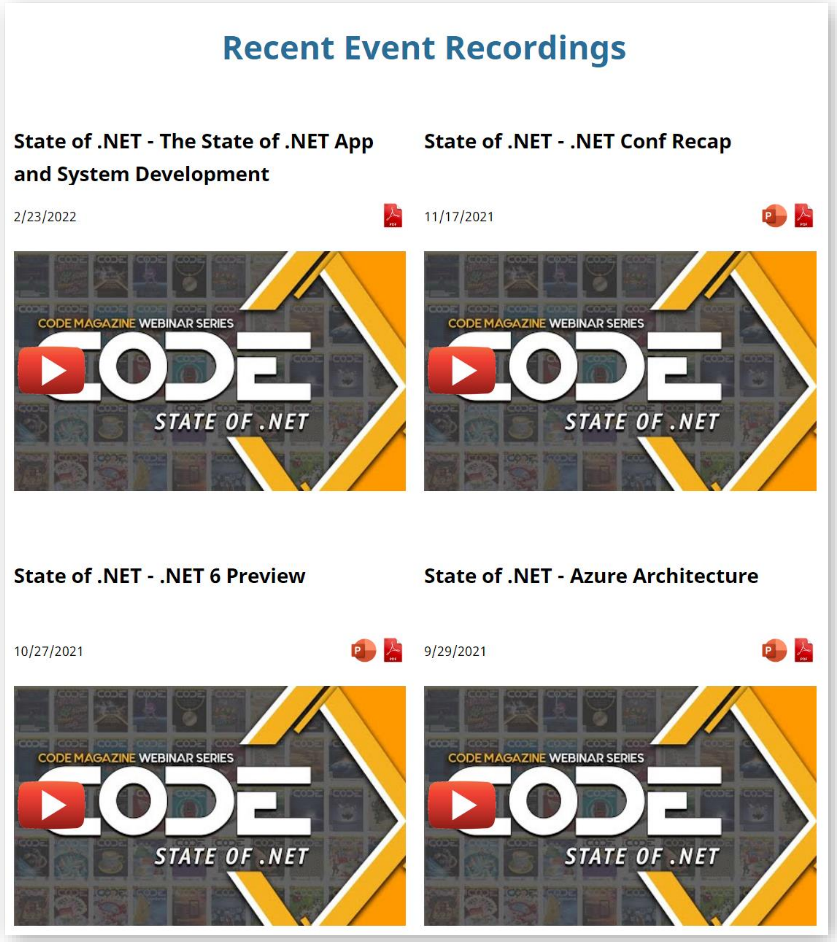
2. Company Name *

Enter your answer

THIS SLIDE WILL BE REPEATED AT THE END AND SURVEY LINK REPEATED IN THE CHAT WINDOW!

Recordings & Slide Decks

- State of .NET Webinar Series
  - https://www.codemag.com/StateOfDotNet
- CODE Presents Webinar Series
  - https://www.codemag.com/CODEPresents


Recent Event Recordings

State of .NET - The State of .NET App and System Development
2/23/2022

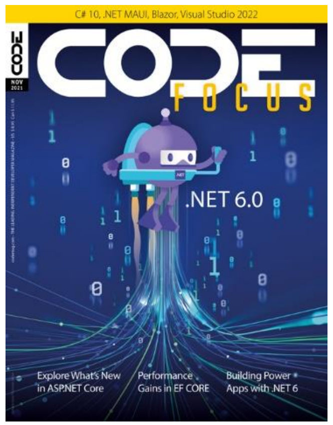
State of .NET - .NET Conf Recap
11/17/2021

State of .NET - .NET 6 Preview
10/27/2021

State of .NET - Azure Architecture
9/29/2021

About the Presenter

- **Joydip Kanjilal**
  - Microsoft MVP
  - CODE Author
  - Author, Speaker
  - 25+ Years Software Development Experience



Improving String Performance in .NET

Joydip Kanjilal

(Microsoft Most Valuable Professional (2007 – 2012), Author & Speaker)

Agenda

- Understanding the Common Language Runtime
  - Just-in-Time Compiler
  - Common Type System
  - Garbage Collection
  - Generations
- How are Strings Represented in the Memory?
- Performance Challenges in String Handling in C#
- Improving String Performance in C#
- Benchmarking String Performance
- Best Practices

The Common Language Runtime (CLR)

The Common Language Runtime (CLR) is a runtime engine that is responsible for executing your programs. It is language-agnostic, meaning it can execute code written in any supported .NET languages. This is made possible by using just-in-time (JIT) compilation, which converts IL code into native machine code at runtime.

The CLR provides several services, such as the following:

- ✓ Common Type System
- ✓ Common Intermediate Language
- ✓ Memory Management & Garbage Collection
- ✓ Just-in-time Compilation
- ✓ Thread Management
- ✓ Exception Handling
- ✓ Security

The Common Type System (CTS)

The Common Type System (CTS) is a standard that specifies how .NET types are actually stored in the managed memory. The CTS permits the coexistence of several languages targeted at the CLR. The CTS is adept at defining how types in the managed execution environment of the CLR are declared, used, and managed.

The CTS provides support for the following five categories of types:

- ✓ Classes
- ✓ Structures
- ✓ Enumerations
- ✓ Interfaces
- ✓ Delegates

The Common Intermediate Language (CIL)

Originally known as Microsoft Intermediate Language (MSIL), Common Intermediate Language (CIL) is a platform-independent language that is used to write source code for .NET applications. It is a collection of platform-independent instructions created by the compiler from the source code.

The Just-in-Time (JIT) compiler converts the CIL into machine code at runtime. In other words, the JIT compiler turns CIL into machine code specific to the target environment, i.e., the environment in which the application must run.

Memory Management & Garbage Collection

When a new process is launched, the CLR reserves a block of memory known as the managed heap. The managed heap holds a reference to the location where the next item in a heap will be allocated, which is initially set at the base address. All reference types are allocated on the managed heap, and the CLR will allocate address space from the heap automatically as long as enough addresses are in the address space.

The garbage collector in .NET takes care of how your application releases or frees up memory. The common language runtime creates a new object by taking memory from the managed heap. As long as the address space is in the managed heap, the runtime will keep providing new object space.

A garbage collection is eventually required to free up space in the managed memory. Based on the allocations made, the optimizing engine of the garbage collector figures out when is the appropriate time to do a collection. A garbage collector checks the managed heap for objects no longer used by the application and releases their occupied memory during a collection.

Conditions for Garbage Collection

A Garbage Collection cycle is triggered when any of the following is true:

- The system is low on available physical memory, i.e., the available physical memory is insufficient. Usually, the operating system detects low memory and sends a notification accordingly.
- On the managed heap, allocated objects use more memory than is acceptable. This threshold is continually updated while the process is running.
- The GC.Collect() method is invoked explicitly in the application. You seldom need to invoke this method since the garbage collector runs continuously in the background. However, you might need to invoke this method in specific scenarios.

Generations

The managed heap is split into three generations, 0, 1, and 2, allowing short-lived objects to be treated independently from long-lived objects to optimize performance. Generation 0, or the first generation, is used to store short-lived objects. If an object has a longer life expectancy, it is moved into one of the higher generations, i.e., generation 1 or 2.

When we collect a generation, we collect all the objects in that generation and its younger generations. Due to the fact that a generation 2 garbage collection reclaims objects across all generations, i.e., the entire managed heap, it is also referred to as a full garbage collection.

Small Object Heap (SOH) & Large Object Heap (LOH)

The GC creates two heap segments (the small object heap or the SOH and the large object heap or the LOH) when the CLR is loaded in the memory. The small object heap can store small objects generally less than 85KB in size.

Large object heaps can hold large objects typically larger than 85KB. The SOH is a section of the managed heap reserved for objects with a size of fewer than 85,000 bytes. The LOH is a section of the managed heap that may contain objects greater than 85KB in size.

Large objects are costly in terms of performance because:

- The allocation cost is high
- The LOH is collected with the rest of the heap

Although the GC doesn't compact the large object heap for you, you can compact the LOH programmatically:

```
GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

What are Strings in .NET?

A string is defined as a sequence of characters and represented using the BSTR rule. It is a reference type, is immutable, can contain null characters and overloads the == operator.

At a quick glance, a string in .NET has the following characteristics:

- Is a Reference type
- Is Immutable
- Is Thread-safe
- Can contain null values

Performance Challenges of Strings in .NET

A string is an immutable type which means that once you have created a string object, you cannot change it in any way. When you change or modify a string object, a new instance is created. There are several ways to overcome this performance issue.

This session talks about how strings are represented in the CLR memory, the performance challenges you would often encounter when working with strings and how to fix them in C#.

Benchmarking Performance using BenchmarkDotNet in .NET 7

Demo

Benchmarking performance using BenchmarkDotNet in .NET 7

- Create a Console Application Project in Visual Studio 2022 Preview
- Install the BenchmarkDotNet NuGet Package
  - ✓ Install-Package BenchmarkDotNet
- Execute the Benchmark Methods
- Compare & Analyze Performance

Benchmarking Performance: Getting Started

Create a class named StringPerformanceBenchmarks in a file having the same name with a ".cs" extension and write the following code in there:

```
[Orderer(BenchmarkDotNet.Order.SummaryOrderPolicy.SlowestToFastest)]
[RankColumn]
[MemoryDiagnoser]
public class StringPerformanceBenchmarks
{
    const int COUNTER = 100;
    const string TEXT = "This is a sample string for testing purposes only.";
    //Write your benchmark methods here
}
```

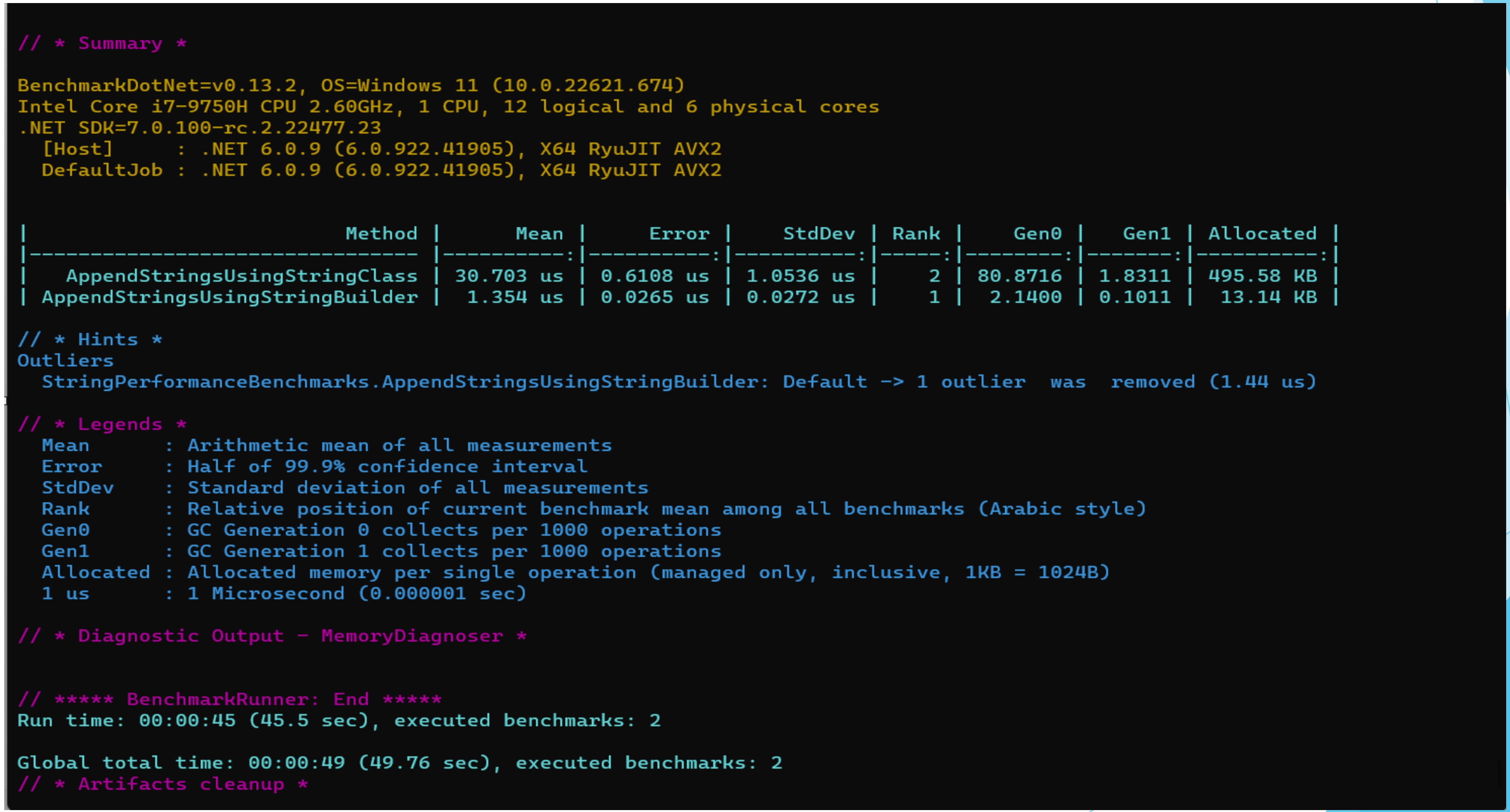
String vs StringBuilder

```
[Benchmark]
public void AppendStringsUsingStringClass()
{
    string str = string.Empty;
    for (int i = 0; i < COUNTER; i++)
    {
        str += TEXT;
    }
}
```

String vs StringBuilder

```
[Benchmark]
public void AppendStringsUsingStringBuilder()
{
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < COUNTER; i++)
    {
        stringBuilder = stringBuilder.Append(TEXT);
    }
}
```

Benchmarking String vs StringBuilder Performance



```
// * Summary *

BenchmarkDotNet=v0.13.2, OS=Windows 11 (10.0.22621.674)
Intel Core i7-9750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK=7.0.100-rc.2.22477.23
  [Host]     : .NET 6.0.9 (6.0.922.41905), X64 RyuJIT AVX2
  DefaultJob : .NET 6.0.9 (6.0.922.41905), X64 RyuJIT AVX2


|                          Method |      Mean |     Error |    StdDev | Rank |    Gen0 |   Gen1 | Allocated |
|-------------------------------- |----------:|----------:|----------:|-----:|--------:|-------:|----------:|
|   AppendStringsUsingStringClass | 30.703 us | 0.6108 us | 1.0536 us |    2 | 80.8716 | 1.8311 | 495.58 KB |
| AppendStringsUsingStringBuilder |  1.354 us | 0.0265 us | 0.0272 us |    1 |  2.1400 | 0.1011 |  13.14 KB |

// * Hints *
Outliers
  StringPerformanceBenchmarks.AppendStringsUsingStringBuilder: Default -> 1 outlier  was  removed (1.44 us)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen0      : GC Generation 0 collects per 1000 operations
  Gen1      : GC Generation 1 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 us      : 1 Microsecond (0.000001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
Run time: 00:00:45 (45.5 sec), executed benchmarks: 2

Global total time: 00:00:49 (49.76 sec), executed benchmarks: 2
// * Artifacts cleanup *
```

Create StringBuilder Using StringBuilderPool

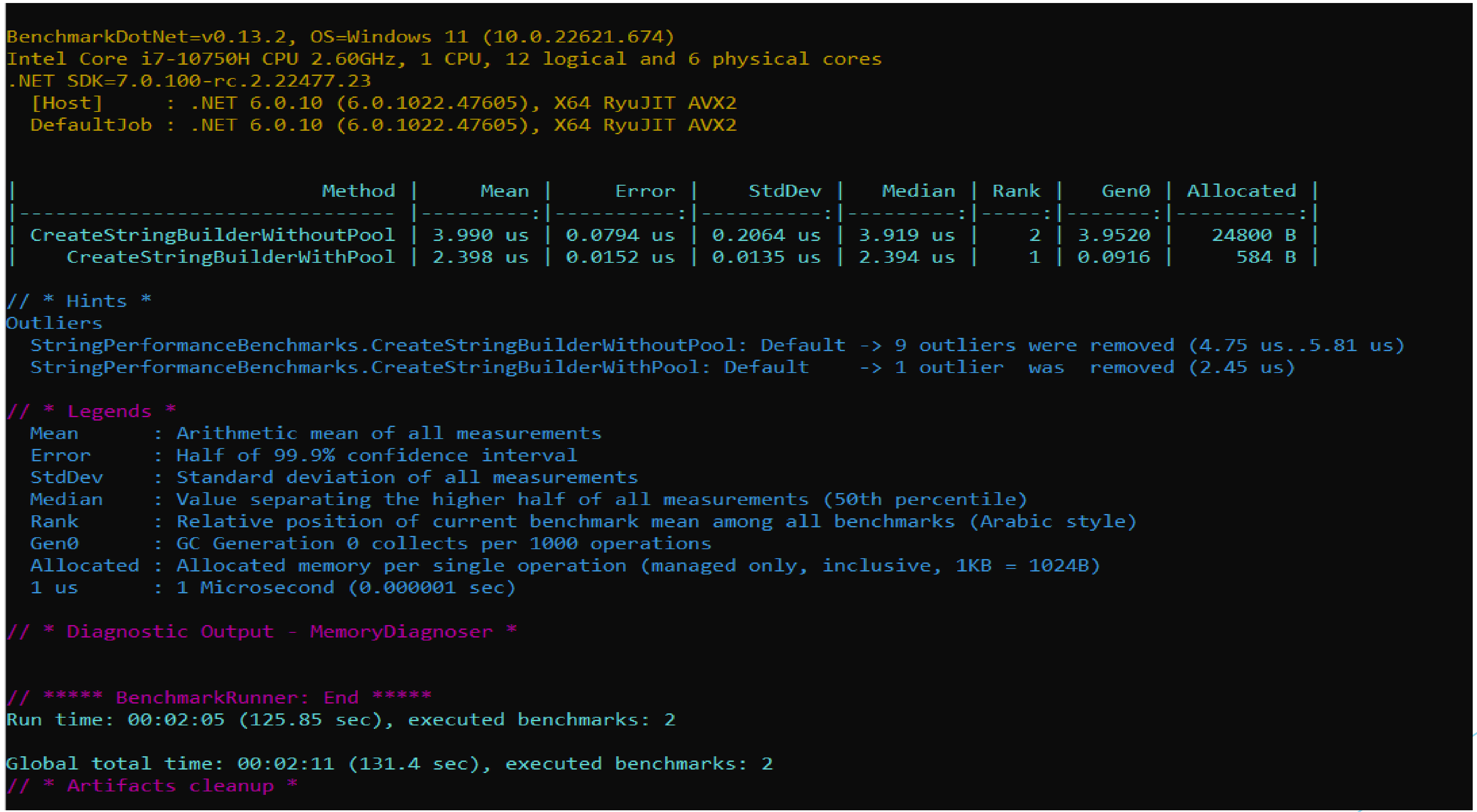
```
[Benchmark]
public void CreateStringBuilderWithPool()
{
var stringBuilderPool = new DefaultObjectPoolProvider().
CreateStringBuilderPool();

for (var i = 0; i < COUNTER; i++) {
       var stringBuilder = stringBuilderPool.Get();
         stringBuilder.Append(TEXT);
         stringBuilderPool.Return(stringBuilder);
   }
}
```

Create StringBuilder Without StringBuilderPool

```
[Benchmark]
public void CreateStringBuilderWithoutPool()
{
  for (int i = 0; i < COUNTER; i++)
  {
    var stringBuilder = new StringBuilder();
     stringBuilder.Append(TEXT);
   }
}
```

Benchmarking Performance: Create StringBuilder With & Without Pool



```
BenchmarkDotNet=v0.13.2, OS=Windows 11 (10.0.22621.674)
Intel Core i7-10750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK=7.0.100-rc.2.22477.23
  [Host]     : .NET 6.0.10 (6.0.1022.47605), X64 RyuJIT AVX2
  DefaultJob : .NET 6.0.10 (6.0.1022.47605), X64 RyuJIT AVX2


|                          Method |     Mean |      Error |     StdDev |   Median | Rank |   Gen0 | Allocated |
|-------------------------------- |---------:|-----------:|-----------:|---------:|-----:|-------:|----------:|
| CreateStringBuilderWithoutPool | 3.990 us | 0.0794 us | 0.2064 us | 3.919 us |    2 | 3.9520 |   24800 B |
|    CreateStringBuilderWithPool | 2.398 us | 0.0152 us | 0.0135 us | 2.394 us |    1 | 0.0916 |     584 B |

// * Hints *
Outliers
  StringPerformanceBenchmarks.CreateStringBuilderWithoutPool: Default -> 9 outliers were removed (4.75 us..5.81 us)
  StringPerformanceBenchmarks.CreateStringBuilderWithPool: Default    -> 1 outlier  was  removed (2.45 us)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Median    : Value separating the higher half of all measurements (50th percentile)
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen0      : GC Generation 0 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 us      : 1 Microsecond (0.000001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
Run time: 00:02:05 (125.85 sec), executed benchmarks: 2

Global total time: 00:02:11 (131.4 sec), executed benchmarks: 2
// * Artifacts cleanup *
```

Accessing Contiguous Memory: Span<T> and Memory<T>

Types of Memory Supported in .NET

- Stack memory
- Managed memory
- Unmanaged memory

The Span<T> and Memory<T> structs give low-level interfaces to any contiguous managed or unmanaged memory block regardless of whether that memory is managed, is unmanaged, or is on the stack. They represent a type-safe access to a contiguous block of memory.

Their primary goal is to encourage micro-optimization and to design low-allocation code that lowers managed memory allocations, thereby relieving the garbage collector. They also enable you to slice or deal with a piece of an array, string, or memory block without having to duplicate the whole chunk of memory.

Accessing Contiguous Memory: Span<T> and Memory<T>

Span<T> is an allocation-free value type with almost zero overhead. It provides a type-safe way to work with a contiguous block of memory such as: Arrays, Strings, and Unmanaged memory buffers.

The following are the characteristics of Span<T> at a quick glance:

- ✓ It is a Value type
- ✓ It has Low or zero overhead
- ✓ It is High performant
- ✓ It provides memory and type safety

```
public readonly ref struct Span<T>
{
    private readonly ref T _pointer;
    private readonly int _length;
}
```

Accessing Contiguous Memory: Span<T> and Memory<T>

```
//Convert an Array to a Span
int[] intArray = new[] { 1, 2, 3, 4, 5 };
Span<int> intSpan = intArray;
var mySpanArray = intArray.AsSpan();

//Convert a List to a Span
List<int> intList = new() { 1, 2, 3, 4, 5 };
var myListSpan = CollectionsMarshal.AsSpan(intList);
```

Extract Strings: String.Substring vs StringBuilder.Append vs Span<T>

```
[Benchmark]
public void ExtractStringUsingSubstring()
{
    for (int i = 0; i < COUNTER; i++)
    {
      _= TEXT.Substring(0, 10);
    }
}
```

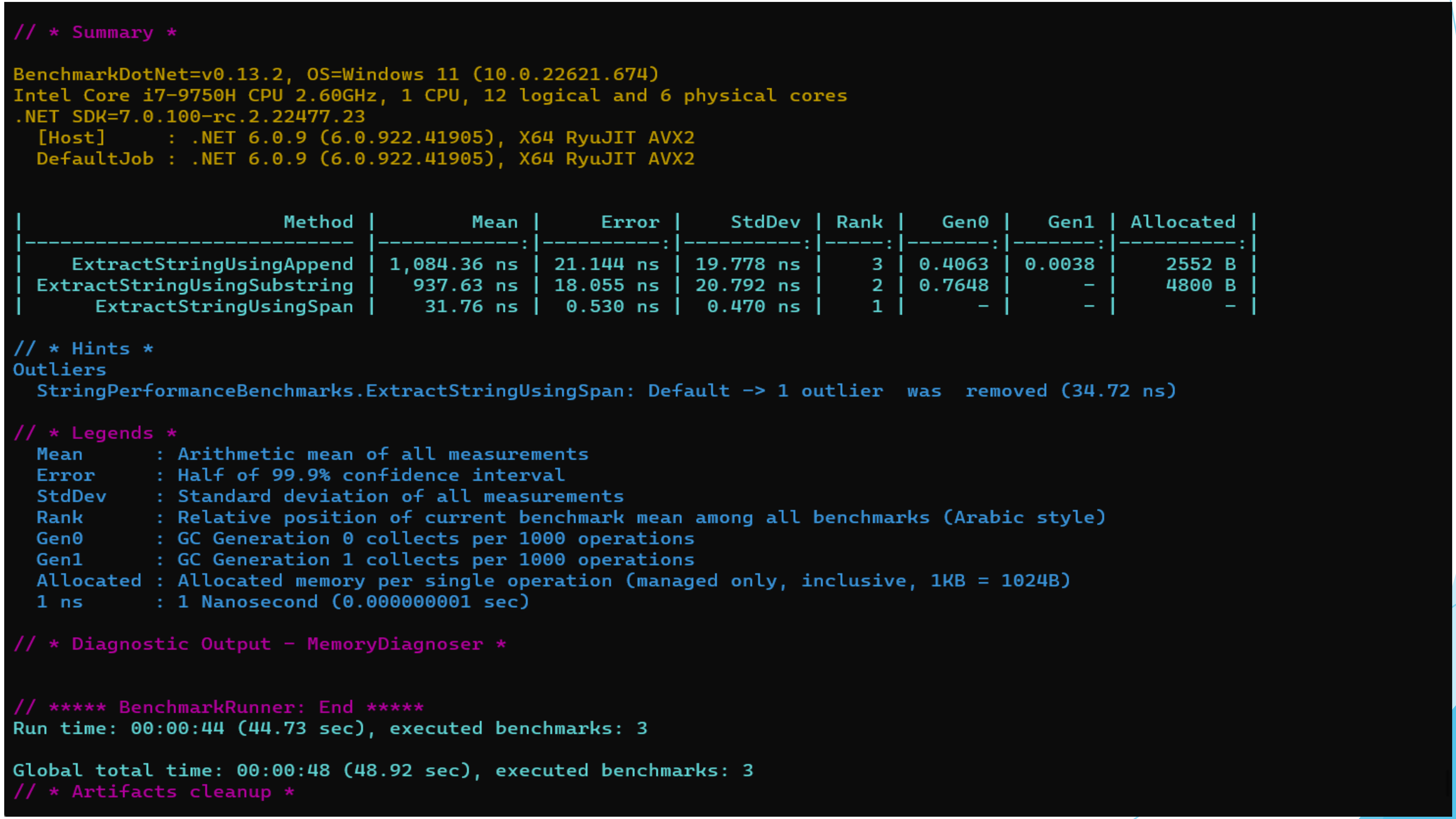
Extract Strings: String.Substring vs StringBuilder.Append vs Span<T>

```
[Benchmark]
public void ExtractStringUsingAppend()
{
   StringBuilder stringBuilder = new StringBuilder();
   for (int i = 0; i < COUNTER; i++) {
       _= stringBuilder.Append(TEXT, 0, 10);
   }
}
```

Extract Strings: String.Substring vs StringBuilder.Append vs Span<T>

```
[Benchmark]
public void ExtractStringUsingSpan()
{
   for (int i = 0; i < COUNTER; i++)
   {
      var data = TEXT.AsSpan().Slice(0, 10);
   }
}
```

Benchmarking Performance: String.Substring vs StringBuilder.Append vs Span<T>



```
// * Summary *

BenchmarkDotNet=v0.13.2, OS=Windows 11 (10.0.22621.674)
Intel Core i7-9750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK=7.0.100-rc.2.22477.23
  [Host]     : .NET 6.0.9 (6.0.922.41905), X64 RyuJIT AVX2
  DefaultJob : .NET 6.0.9 (6.0.922.41905), X64 RyuJIT AVX2


|                      Method |        Mean |      Error |     StdDev | Rank |   Gen0 |   Gen1 | Allocated |
|---------------------------- |------------:|-----------:|-----------:|-----:|-------:|-------:|----------:|
|   ExtractStringUsingAppend | 1,084.36 ns | 21.144 ns | 19.778 ns |    3 | 0.4063 | 0.0038 |    2552 B |
| ExtractStringUsingSubstring |   937.63 ns | 18.055 ns | 20.792 ns |    2 | 0.7648 |      - |    4800 B |
|      ExtractStringUsingSpan |    31.76 ns |  0.530 ns |  0.470 ns |    1 |      - |      - |         - |

// * Hints *
Outliers
  StringPerformanceBenchmarks.ExtractStringUsingSpan: Default -> 1 outlier  was  removed (34.72 ns)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen0      : GC Generation 0 collects per 1000 operations
  Gen1      : GC Generation 1 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 *****
Run time: 00:00:44 (44.73 sec), executed benchmarks: 3

Global total time: 00:00:48 (48.92 sec), executed benchmarks: 3
// * Artifacts cleanup *
```

What is String Interning?

String interning involves using a hash table to store a single copy of each string in the string intern pool. The key is a string hash, and the value is a pointer to the real String object.

Thus, interning guarantees that only one instance of that string is allocated memory, even if the string appears many number of times. When comparing strings, you only need to perform a reference comparison if they are interned.

The Intern method in C# is used to obtain a reference to specified string object. This method looks into the "intern pool" for a string that is identical to the supplied string. If such a string exists, its intern pool reference is returned.

Although string interning is an exciting feature, it is only used sparingly in practice. Misusing it can undoubtedly adversely affect our application's performance. We should only consider it if we plan to create many strings with the same content.

Characteristics of String Interning

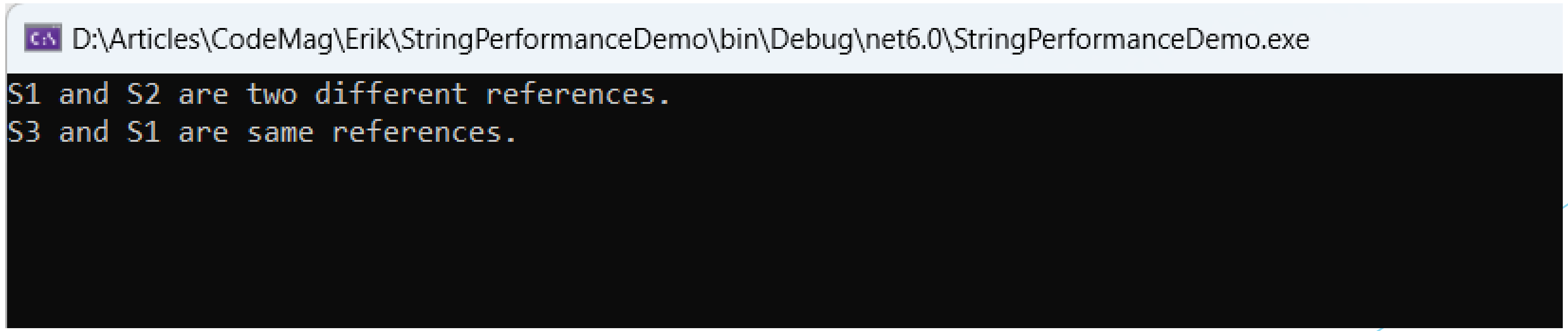
- The memory location of two identical interned strings will be the same.
- Until your application is terminated, internal string memory is not released.
- In order to intern a string, a hash must be computed and the string must be searched in a dictionary, which both consume CPU resources.
- Since the dictionary of interned strings is serialized, several threads interning strings simultaneously will block each other.
- Interning a string requires computing a hash and searching a dictionary, both consuming CPU cycles.
- When you use the same string several times in your application, it will only keep one copy of the string in memory.
- This will reduce the amount of memory you need for your application.
- If you have a lot of interned strings, string interning can be slow, as threads will block each other as they access the interned string dictionary.

Code Example Illustrating String Interning

```
string s1 = "Demonstrating String Interning";
string s2 = new StringBuilder().Append("Demonstrating ").Append("String Interning").ToString();
string s3 = String.Intern(s2);

if ((Object)s2 != (Object)s1)
    Console.WriteLine("S1 and S2 are two different references.");

if ((Object)s3 == (Object)s1)
    Console.WriteLine("S3 and S1 are same references.");
```


D:\Articles\CodeMag\Erik\StringPerformanceDemo\bin\Debug\net6.0\StringPerformanceDemo.exe

```
S1 and S2 are two different references.
S3 and S1 are same references.
```

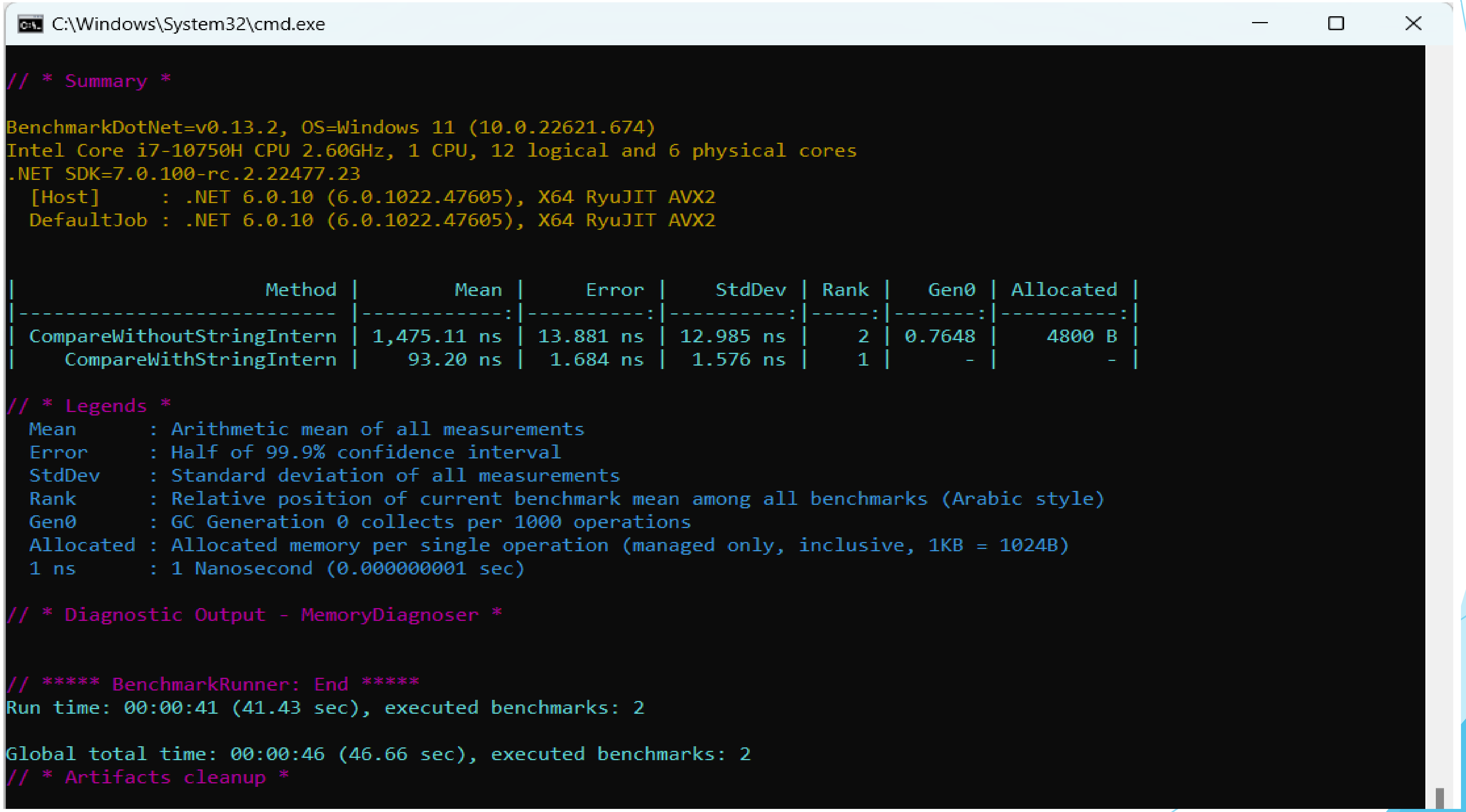
Compare Strings (Using Interning)

```
[Benchmark]
public void CompareWithStringIntern ()
{
  string str = string.Intern("HelloWorld");
  bool isEqual = false;
  for (int i = 0; i < COUNTER; i++)
  {
     var s1 = str; // Uses the interned string
     var s2 = str; // Uses the interned string
     isEqual = (str == s2);
  }
}
```

Compare Strings (Not Using Intering)

```
[Benchmark]
public void CompareWithoutStringIntern ()
{
  string str = "HelloWorld";
  bool isEqual = false;
  for (int i = 0; i < COUNTER; i++)
  {
    var s1 = "World";
    var s2 = "Hello" + s1;
    isEqual = (str == s2);
  }
}
```

Benchmarking String Compare Performance (Using Interned & Non-Interned Strings)



```
// * Summary *

BenchmarkDotNet=v0.13.2, OS=Windows 11 (10.0.22621.674)
Intel Core i7-10750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK=7.0.100-rc.2.22477.23
  [Host]     : .NET 6.0.10 (6.0.1022.47605), X64 RyuJIT AVX2
  DefaultJob : .NET 6.0.10 (6.0.1022.47605), X64 RyuJIT AVX2


|                      Method |        Mean |     Error |    StdDev | Rank |   Gen0 | Allocated |
|---------------------------- |------------:|----------:|----------:|-----:|-------:|----------:|
| CompareWithoutStringIntern | 1,475.11 ns | 13.881 ns | 12.985 ns |    2 | 0.7648 |    4800 B |
|    CompareWithStringIntern |    93.20 ns |  1.684 ns |  1.576 ns |    1 |      - |         - |

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen0      : GC Generation 0 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 *****
Run time: 00:00:41 (41.43 sec), executed benchmarks: 2

Global total time: 00:00:46 (46.66 sec), executed benchmarks: 2
// * Artifacts cleanup *
```

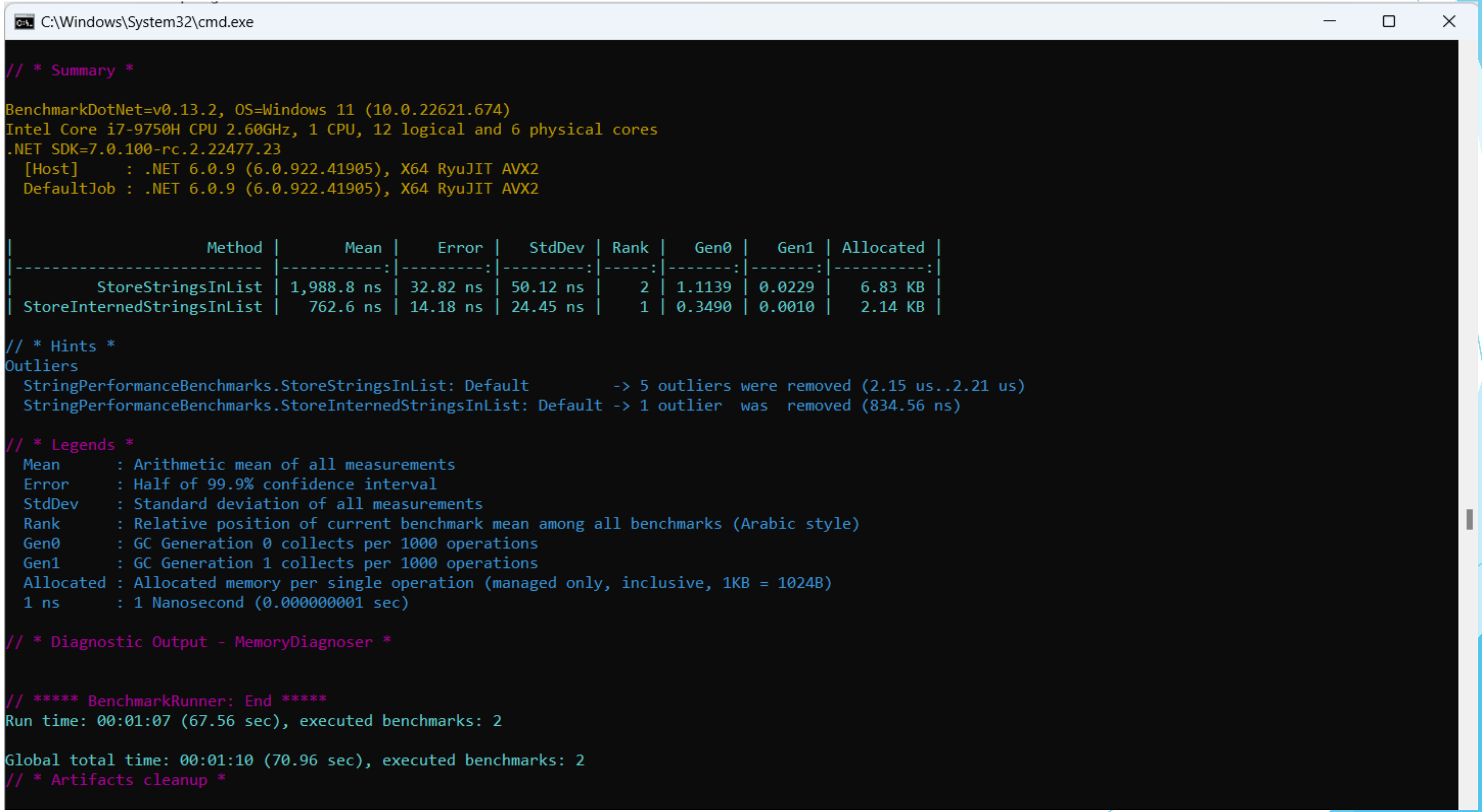
Store Interned Strings in a List

```
[Benchmark]
public void StoreInternedStringsInList()
{
   string text = "Hello World";
   string temp = null;  var list = new List<string>();
   for (int i = 0; i < COUNTER; i++)
   {
      if (temp == null)
        temp = string.Intern(text);
      else
          list.Add(temp);
   }
}
```

Store Non-Interned Strings in a List

```
[Benchmark]
public void StoreStringsInList()
{
  string text = "Hello";
  var list = new List<string>();
  for (int i = 0; i < COUNTER; i++)
  {
      list.Add(text + " World");
  }
}
```

Benchmarking String Interning Performance (Using Interned & Non-Interned Strings)



```
// * Summary *

BenchmarkDotNet=v0.13.2, OS=Windows 11 (10.0.22621.674)
Intel Core i7-9750H CPU 2.60GHz, 1 CPU, 12 logical and 6 physical cores
.NET SDK=7.0.100-rc.2.22477.23
  [Host]     : .NET 6.0.9 (6.0.922.41905), X64 RyuJIT AVX2
  DefaultJob : .NET 6.0.9 (6.0.922.41905), X64 RyuJIT AVX2


|                      Method |       Mean |     Error |    StdDev | Rank |   Gen0 |   Gen1 | Allocated |
|---------------------------- |-----------:|----------:|----------:|-----:|-------:|-------:|----------:|
|          StoreStringsInList | 1,988.8 ns | 32.82 ns | 50.12 ns |    2 | 1.1139 | 0.0229 |    6.83 KB |
| StoreInternedStringsInList |   762.6 ns | 14.18 ns | 24.45 ns |    1 | 0.3490 | 0.0010 |    2.14 KB |

// * Hints *
Outliers
  StringPerformanceBenchmarks.StoreStringsInList: Default         -> 5 outliers were removed (2.15 us..2.21 us)
  StringPerformanceBenchmarks.StoreInternedStringsInList: Default -> 1 outlier  was  removed (834.56 ns)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen0      : GC Generation 0 collects per 1000 operations
  Gen1      : GC Generation 1 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 *****
Run time: 00:01:07 (67.56 sec), executed benchmarks: 2

Global total time: 00:01:10 (70.96 sec), executed benchmarks: 2
// * Artifacts cleanup *
```

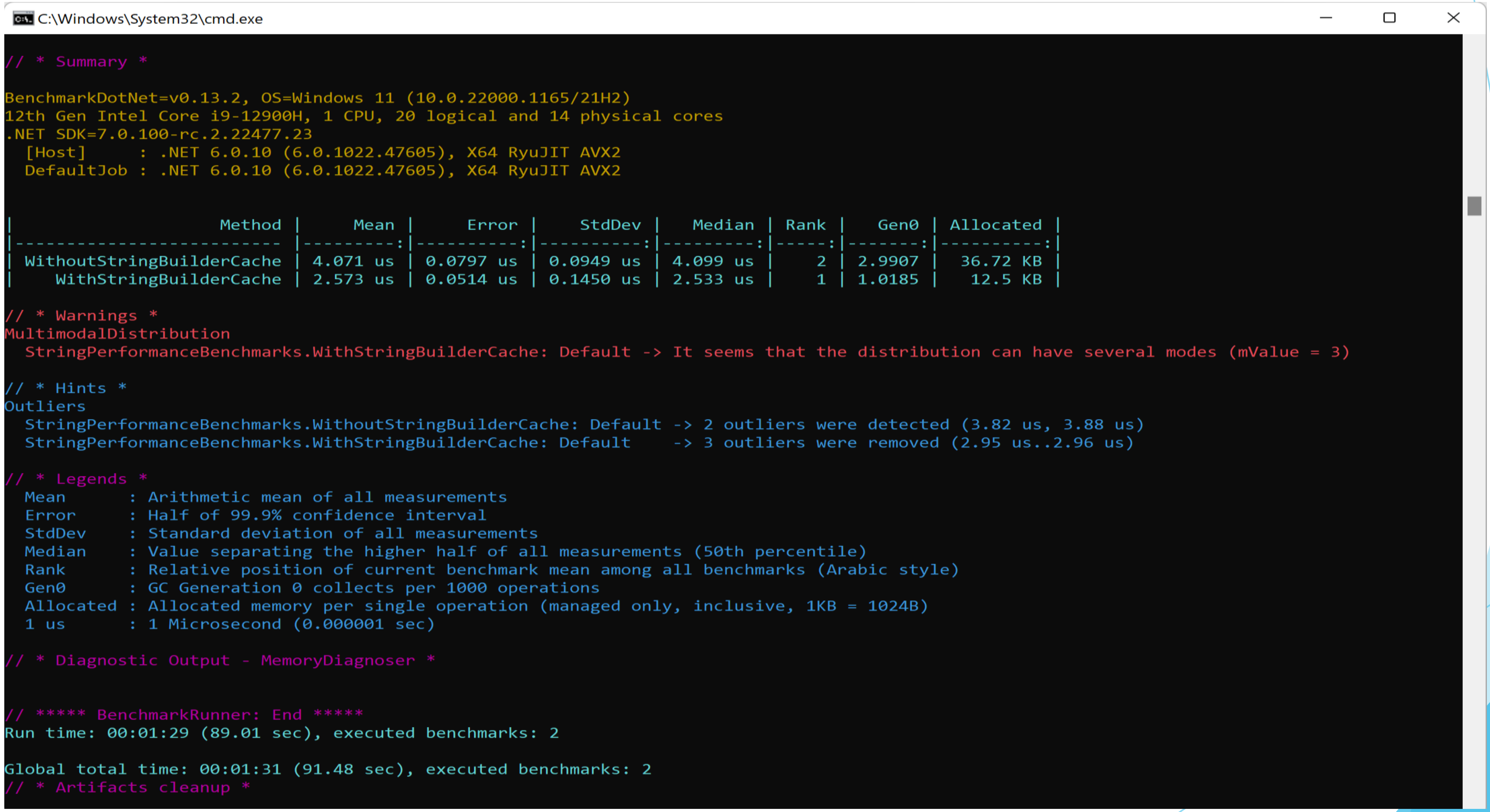
Leverage StringBuilderCache: Reuse StringBuilder Instances

```
[Benchmark]
public void WithoutStringBuilderCache()
{
  for (int i = 0; i < COUNTER; i++)
  {
     var stringBuilder = new StringBuilder();
        stringBuilder.Append(TEXT);
  }
}
```

Leverage StringBuilderCache: Reuse StringBuilder Instances

```
 [Benchmark]
public void WithStringBuilderCache()
{
   for (int i = 0; i < COUNTER; i++)
   {
       var stringBuilder = StringBuilderCache.Acquire();
       stringBuilder.Append(TEXT);
       _ = StringBuilderCache.GetStringAndRelease(stringBuilder);
    }
}
```

Benchmarking Performance: Using & Not Using StringBuilderCache


C:\Windows\System32\cmd.exe

```
// * Summary *

BenchmarkDotNet=v0.13.2, OS=Windows 11 (10.0.22000.1165/21H2)
12th Gen Intel Core i9-12900H, 1 CPU, 20 logical and 14 physical cores
.NET SDK=7.0.100-rc.2.22477.23
  [Host]     : .NET 6.0.10 (6.0.1022.47605), X64 RyuJIT AVX2
  DefaultJob : .NET 6.0.10 (6.0.1022.47605), X64 RyuJIT AVX2


|                     Method |     Mean |     Error |    StdDev |   Median | Rank |   Gen0 | Allocated |
|--------------------------- |---------:|----------:|----------:|---------:|-----:|-------:|----------:|
| WithoutStringBuilderCache | 4.071 us | 0.0797 us | 0.0949 us | 4.099 us |    2 | 2.9907 |  36.72 KB |
|    WithStringBuilderCache | 2.573 us | 0.0514 us | 0.1450 us | 2.533 us |    1 | 1.0185 |   12.5 KB |

// * Warnings *
MultimodalDistribution
  StringPerformanceBenchmarks.WithStringBuilderCache: Default -> It seems that the distribution can have several modes (mValue = 3)

// * Hints *
Outliers
  StringPerformanceBenchmarks.WithoutStringBuilderCache: Default -> 2 outliers were detected (3.82 us, 3.88 us)
  StringPerformanceBenchmarks.WithStringBuilderCache: Default    -> 3 outliers were removed (2.95 us..2.96 us)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Median    : Value separating the higher half of all measurements (50th percentile)
  Rank      : Relative position of current benchmark mean among all benchmarks (Arabic style)
  Gen0      : GC Generation 0 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 us      : 1 Microsecond (0.000001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
Run time: 00:01:29 (89.01 sec), executed benchmarks: 2

Global total time: 00:01:31 (91.48 sec), executed benchmarks: 2
// * Artifacts cleanup *
```

How Are Strings Represented in Memory?

While strings in C/C++ are represented in the memory based on the PWSZ (Pointer to Wide-character String, Zero-terminated) rule, in C#, strings are stored in the memory based on the BSTR (Basic string or binary string) rule. Note that both PWSZ and BSTR, strings are null-terminated at the end of the string. Each character in a string consumes 2 bytes and is in the UTF-16 coding, i.e., two-byte characters of a string in the UTF-16 format.

Generally, a string instance comprises the following:

- An 8-byte object header, which consists of a 4-byte SyncBlock and a 4-byte type descriptor.
- The length of the string represented as an int32 field. (Used prior to .NET 4)
- A total count of the number of characters in the character buffer represented by an int32 field.
- The first character in a string represented as System.Char.
- A character buffer containing the remaining characters in the string terminated by a null character at the end.

How Are Strings Represented in Memory?

So, the length of a string on a x86 system is given by 14 + length x 2 bytes in a x86 system and 26 + length x 2 bytes in a x64 system. Hence, on x86 systems the size of an empty string object is 14 bytes while on x64 systems it is 26 bytes.

Note that prior to .NET 4.0, a string object had an additional field m_arrayLength to represent capacity. This field has been discarded from the subsequent versions of .NET. In a x86 system, the length of a string is 14 + length x 2 bytes, while in a x64 system, it is 26 + length x 2 bytes.

Best Practices

- Use StringBuilder for appending strings multiple times
- Use Span<T> for allocation free access to contiguous memory
- Use StringBuilderPool to reduce allocations
- Use StringBuilderCache to reuse StringBuilder instances
- Use String Interning to reduce allocation
- To extract a string from another, use Span<T>

Suggested Readings

- Writing High-Performance Code Using Span<T> and Memory<T> in C# (Code Magazine)
- Writing High-Performance .NET Code by Ben Watson
- https://github.com/adamsitnik/awesome-dot-net-performance

Thanks!

E-Mail: joydipkanjilal@yahoo.com
Website: https://joydipkanjilal.com/
LinkedIn: https://in.linkedin.com/in/joydipkanjilal
Twitter: https://twitter.com/joydipkanjilal
Blog: https://www.infoworld.com/author/Joydip-Kanjilal/
Github: https://github.com/joydipkanjilal

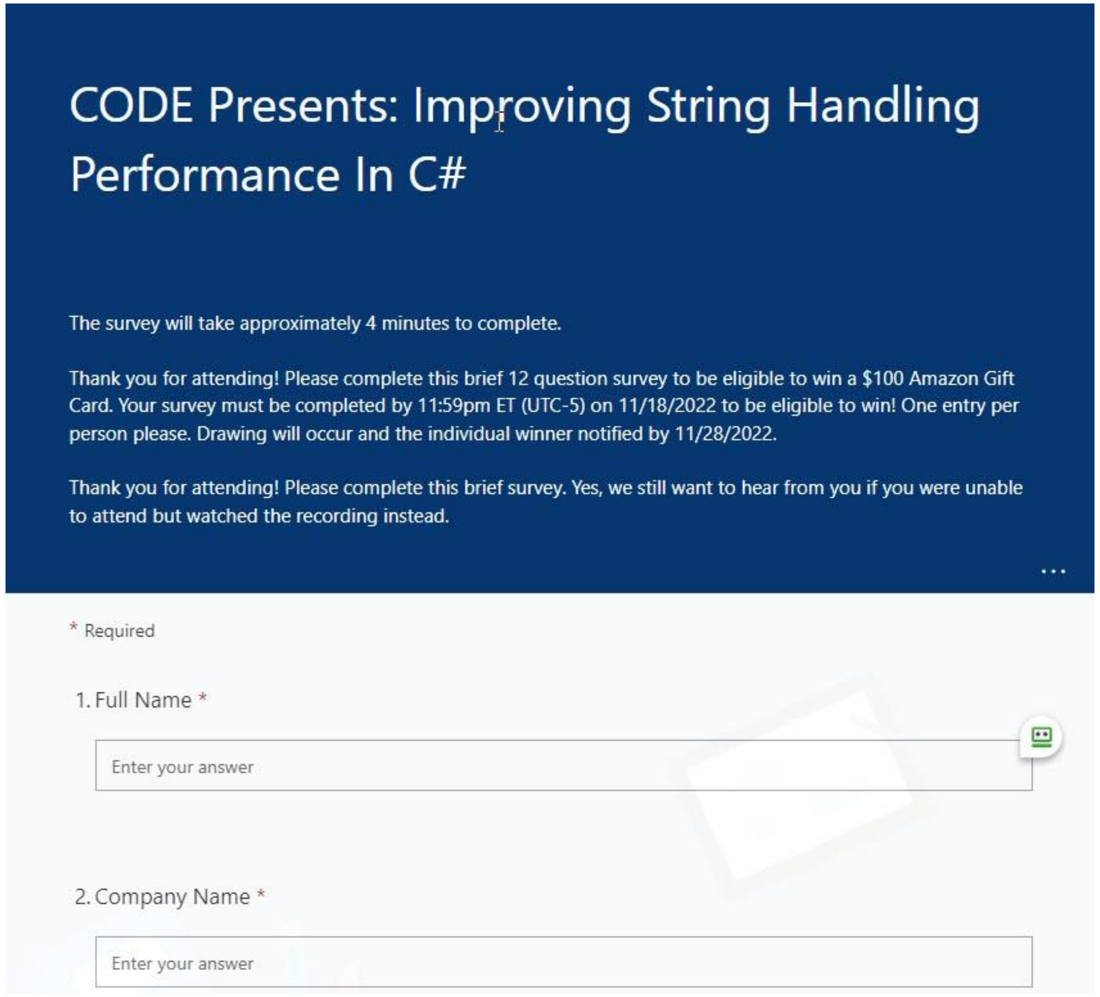
Other Announcements

Event Survey – Win $100!

- **Starting at 1:15pm ET** Complete this very short 12 question survey for a chance at a $100 Amazon Gift Card!

https://bit.ly/3UI4jQV

- Survey must be completed by **11:59pm ET** on **Friday 11/18/2022** to be eligible!
- Completed survey is required to be eligible.



CODE Presents: Improving String Handling Performance In C#

The survey will take approximately 4 minutes to complete.

Thank you for attending! Please complete this brief 12 question survey to be eligible to win a $100 Amazon Gift Card. Your survey must be completed by 11:59pm ET (UTC-5) on 11/18/2022 to be eligible to win! One entry per person please. Drawing will occur and the individual winner notified by 11/28/2022.

Thank you for attending! Please complete this brief survey. Yes, we still want to hear from you if you were unable to attend but watched the recording instead.

* Required

1. Full Name *

Enter your answer

2. Company Name *

Enter your answer

Free Subscription to CODE Magazine!

- The leading software development magazine written by expert developers for developers.
- All registered attendees will automatically receive a free digital subscription to CODE Magazine – no need to do anything, it'll happen auto-magically.
- Upgrade to Print and receive our .NET 7 CODE Focus issue FREE! (while supplies last)

- Please share this free subscription link: https://bit.ly/3NTrS6F

MAUI, EF 7, Minimal APIs, Blazor, C# 11
CODE FOCUS
.NET 7
What's new in the latest and fastest release yet?
C# Goes to 11
Sailing to MAUI
.NET 7 Performance Deep Dive

C# 10, .NET MAUI, Blazor, Visual Studio 2022
CODE FOCUS
.NET 6.0
Explore What's New in ASP.NET Core
Performance Gains in EF CORE
Building Power Apps with .NET 6

CODE Is Hiring!

We Need Developers!

- We have **current** openings for:
  - Data Engineer
  - Python Developer
  - Senior C# Developer
  - .NET Desktop Developer
  - REACT & JavaScript Developers
- Details here: https://codemag.com/Jobs

Your Ticket to Free Consulting

- One hour on us. Really. Schedule a call today. Slots are limited.
- No strings. No commitment. No credit card required.
- Just help from our team of experienced software developers.
- Got questions? Stuck on an issue? Platform and/or architecture decisions to make? We can help!

Contact us at:
info@codemag.com or
jduffy@codemag.com

CODE CONSULTING

CODE Mobile App

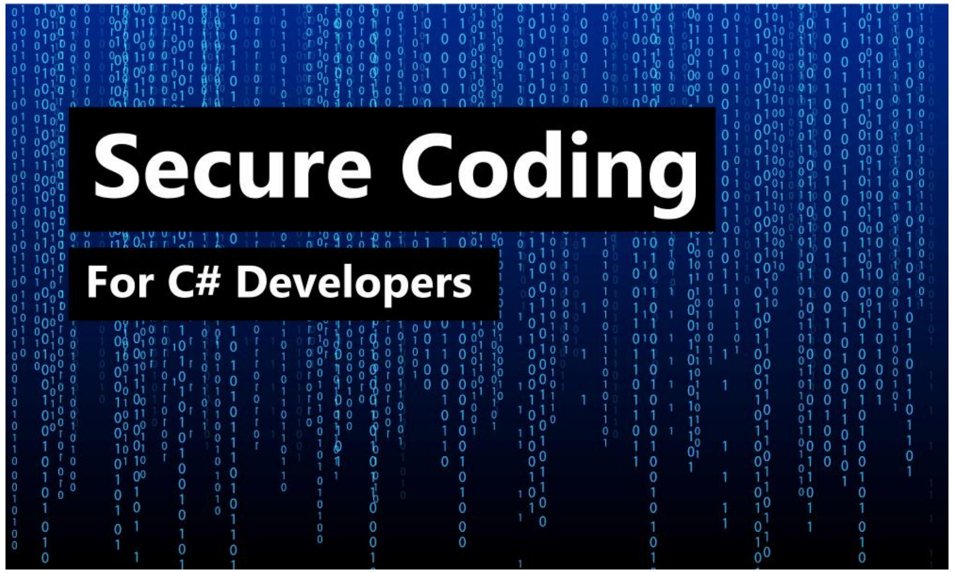
- Check out the new CODE Magazine Mobile application!
- Available for iOS & Android

Download on the App Store

GET IT ON Google Play

New CODE Training Course

- 3-Day
- Online
- Hands-on
- Learn to develop robust and secure C# applications
- January 24-26 https://bit.ly/3G3WMr1
- March 7-9 https://bit.ly/3Txcvlf
- April 18-20 https://bit.ly/3WRXugZ
- June 6-8 https://bit.ly/3ObqWe5
- https://www.codemag.com/training



Q&A

Contact us with questions!

CODE/EPS Contact:

www.codemag.com
info@codemag.com
facebook.com/codemag
twitter.com/codemagazine

Contact:

jduffy@codemag.com