# High-Performance C#

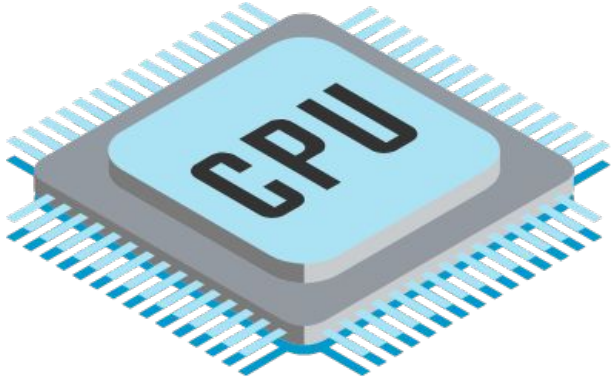
by Stefan Panko

VISMA

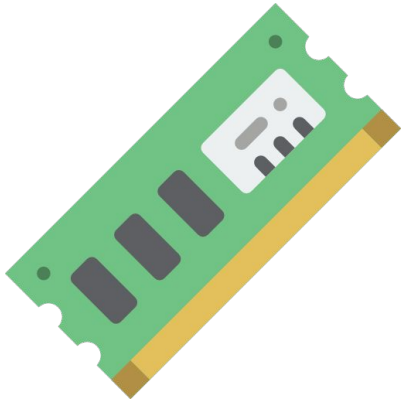
# Aspects of Performance

**Resources:**

**Execution Time - CPU**




**Memory Allocations - RAM**



VISMA

# Types

## Reference types

class

- object
- string

interface, array, delegate, record, dynamic

## Value types

bool, byte, sbyte, char, decimal, double, float, int, uint, nint, nuint, long, ulong, short, ushort, struct, enum, tuples

# Memory allocation

| Feature | Stack | Heap |
|---|---|---|
| **Allocation** | Automatically managed by C# runtime. | Requires garbage collector for memory management. |
| **Speed** | Fast and efficient in allocating and deallocating memory. | Slower in allocating and deallocating memory. |
| **Purpose** | Storing temporary data such as function call parameters, local variables, and function return addresses. | Storing longer-lived data such as objects and their data members. |
| **Data Structures** | A stack is ideal for small or temporary data structures. | Heap is suitable for large and dynamic data structures. |
| **Accessibility** | Limited to the function or thread that created the data. | Multiple Functions or Threads can access it. |

# Memory allocation

- **Stack**
  - Value type declared as variable in a method
  - Value type declared as parameter in method
  - Value type declared as a member of a struct allocated on stack
  - Ref structs
- **Heap**
  - Value type declared as a member of a class
  - Value type declared as a member of a struct allocated on heap

# Memory allocation

Heap/Stack

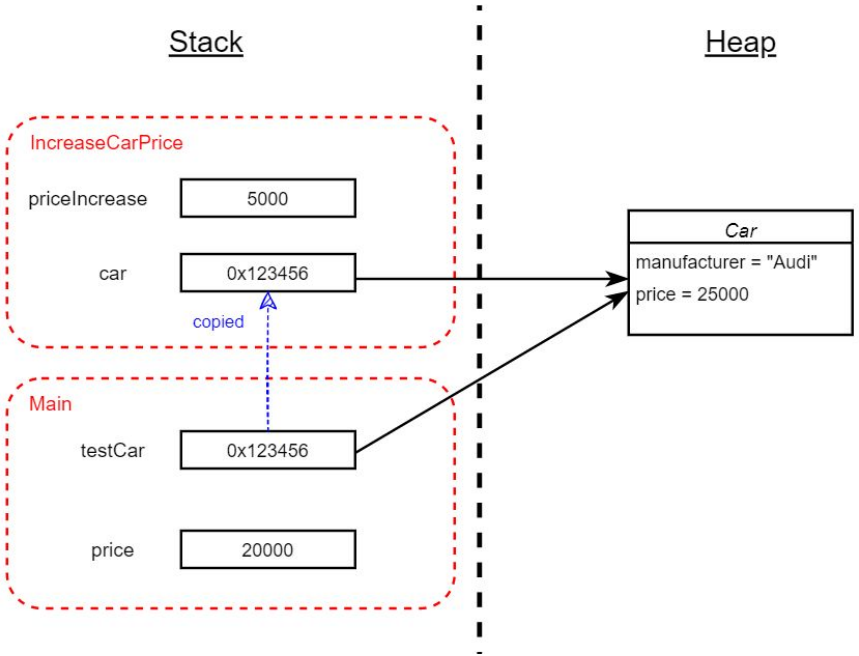
```
public class MyClass
{
    private int _field1;
    private string _field2;

    public void Method(int par1, string[] par2)
    {
        int price = 20000;
        Car testCar = new Car("Audi", price);
    }
}
```

```
public struct MyStruct
{
    private int _field1;
    private string _field2;

    public void Method(int par1, string[] par2)
    {
        int price = 20000;
        Car testCar = new Car("Audi", price);
    }
}

public ref struct MyRefStruct
{
}
```

VISMA

# Memory allocation

```
class Program
{
    static void Main(string[] args)
    {
        int price = 20000;
        Car testCar = new Car("Audi", price);
        InreaseCarPrice(testCar, 5000);
    }

    static void IncreaseCarPrice(Car car,
        int priceIncrease)
    {
        car.price += priceIncrease;
    }
}
```



# Optimization Cycle

# Tools

- **Optimization**
  - **Visual Studio Performance Profiler**
  - PerfView
  - Jetbrains dotTrace/dotMemory
  - sharplab.io
- **Measurement**
  - **BenchmarkDotNet** (Microbenchmarking)
  - Microsoft.Crank (APIs)
  - Grafana k6 (APIs)

# Strings

- Immutable
  - Thread Safety
  - Security
  - Optimizations
  - Caching and Interning
  - Predictable Behavior
- Reference type allocated on heap
- Operations like concatenation causing new string allocations

**We should try to avoid**

- Creating new string by methods like **Substring**, **Remove**, **Replace**, **Trim**, **ToLower**, **ToUpper** etc. as much as possible, especially in **loops**.
- Creating new strings instead of reusing static or constant strings

# Strings

**To save memory, we can**

- Use
  - string **constants**
  - **static readonly** string variables
  - StringBuilder
  - string.Concat()
  - string.Create()
- Don’t use string where it’s not needed, for example single character strings instead of **char**
- Never concatenate strings in a loop using the **+ (add)** operator or **interpolation**

# Strings - Concatenation

Creating new string by concatenating allocates a lot of memory, especially in loops.

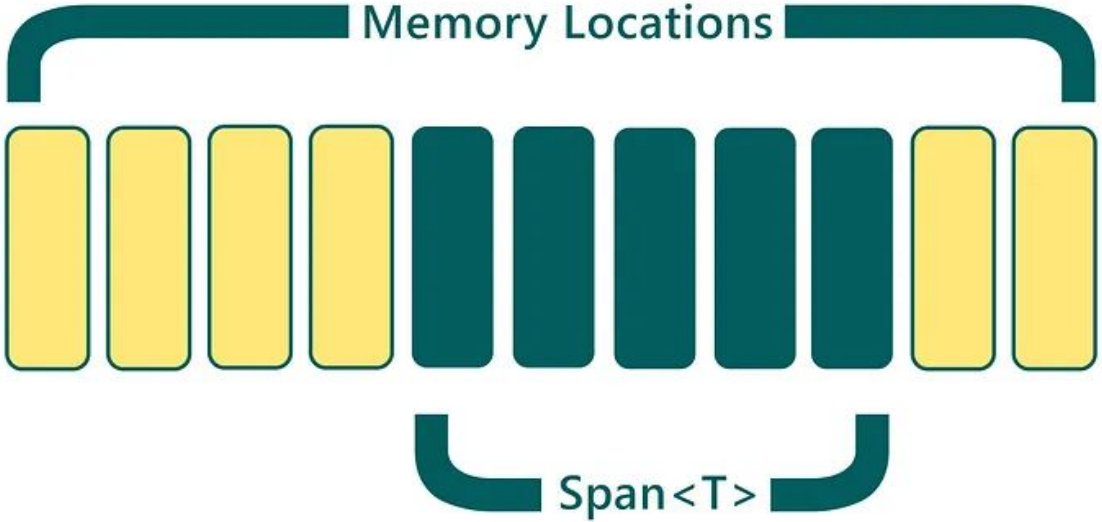
- For simply concatenation a **fixed collection** of strings, the most efficient way is to use **string.Concat()** method.
- For building and optionally format string in a loop use **StringBuilder.**
- For creating a string from parts of an existing string, cast the string to **ReadOnlySpan<char>**, extract slices and create using **string.Create()** method. **Span** is a **ref struct**, can be allocated only on **stack**.

# Strings - Concatenation

```
string[] strings = ["Hello", " ", "world", "!"];

string helloWorld = string.Concat(strings);
// Or
StringBuilder sb = new();

foreach (string str in strings)
{
    sb.Append(str);
}

helloWorld = sb.ToString();
```

# Span<T> struct

**System.Span<T>** is a new value type at the heart of .NET. It enables the representation of contiguous regions of arbitrary memory, regardless of whether that memory is associated with a managed object, is provided by native code via interop, or is on the stack. And it does so while still providing safe access with performance characteristics like that of arrays.




VISMA

# Strings - Concatenation

```
string hello = "This is a Hello string";
string world = "This is a World! string";

ReadOnlySpan<char> helloSpan = hello.AsSpan().Slice(10, 6);
ReadOnlySpan<char> worldSpan = world.AsSpan().Slice(10, 6);
char[] buffer = new char[helloSpan.Length + worldSpan.Length];
Span<char> span = new(buffer);
helloSpan.CopyTo(span.Slice(0, helloSpan.Length));
worldSpan.CopyTo(span.Slice(helloSpan.Length, worldSpan.Length));

string helloWorld = span.ToString();
```

# Strings - Concatenation Benchmark

| Method | N | Mean | Error | StdDev | Ratio | RatioSD | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|
| StringConcatArray | 3 | 570.0 ns | 225.9 ns | 149.44 ns | 1.00 | 0.00 | 464 B | 1.00 |
| StringConcatEnumerable | 3 | 1,444.4 ns | 518.7 ns | 308.67 ns | 2.50 | 0.34 | 504 B | 1.09 |
| StringInterpolation | 3 | 1,400.0 ns | 533.3 ns | 352.77 ns | 2.56 | 0.77 | 272 B | 0.59 |
| Addition | 3 | 1,200.0 ns | 482.7 ns | 287.23 ns | 2.11 | 0.57 | 560 B | 1.21 |
| StringBuilder | 3 | 1,620.0 ns | 517.9 ns | 342.54 ns | 2.95 | 0.72 | 712 B | 1.53 |
| StringJoin | 3 | 1,066.7 ns | 489.9 ns | 291.55 ns | 1.86 | 0.47 | 464 B | 1.00 |
| StringCreateFromEnumerable | 3 | 2,420.0 ns | 971.5 ns | 642.56 ns | 4.56 | 1.87 | 544 B | 1.17 |
| StringCreateFromArray | 3 | 2,140.0 ns | 985.5 ns | 651.84 ns | 4.00 | 1.56 | 496 B | 1.07 |
| | | | | | | | | |
| StringConcatArray | 10 | 720.0 ns | 408.2 ns | 269.98 ns | 1.00 | 0.00 | 568 B | 1.00 |
| StringConcatEnumerable | 10 | 1,980.0 ns | 401.9 ns | 265.83 ns | 3.07 | 1.15 | 608 B | 1.07 |
| StringInterpolation | 10 | 1,550.0 ns | 474.1 ns | 313.58 ns | 2.39 | 0.87 | 1424 B | 2.51 |
| Addition | 10 | 1,480.0 ns | 477.0 ns | 315.52 ns | 2.26 | 0.72 | 1424 B | 2.51 |
| StringBuilder | 10 | 1,900.0 ns | 719.8 ns | 476.10 ns | 2.96 | 1.31 | 1152 B | 2.03 |
| StringJoin | 10 | 2,977.8 ns | 953.5 ns | 567.40 ns | 4.45 | 1.77 | 568 B | 1.00 |
| StringCreateFromEnumerable | 10 | 2,062.5 ns | 259.3 ns | 135.62 ns | 3.02 | 1.26 | 648 B | 1.14 |
| StringCreateFromArray | 10 | 1,860.0 ns | 856.6 ns | 566.57 ns | 2.70 | 0.57 | 264 B | 0.46 |
| | | | | | | | | |
| StringConcatArray | 100 | 980.0 ns | 160.2 ns | 105.93 ns | 1.00 | 0.00 | 2008 B | 1.00 |
| StringConcatEnumerable | 100 | 3,380.0 ns | 1,381.6 ns | 913.84 ns | 3.44 | 0.77 | 2048 B | 1.02 |
| StringInterpolation | 100 | 7,325.0 ns | 1,521.9 ns | 795.97 ns | 7.41 | 1.25 | 82064 B | 40.87 |
| Addition | 100 | 7,712.5 ns | 1,993.4 ns | 1,042.58 ns | 7.74 | 0.90 | 82064 B | 40.87 |
| StringBuilder | 100 | 4,500.0 ns | 645.4 ns | 384.06 ns | 4.64 | 0.77 | 4600 B | 2.29 |
| StringJoin | 100 | 1,450.0 ns | 102.2 ns | 53.45 ns | 1.47 | 0.17 | 2008 B | 1.00 |
| StringCreateFromEnumerable | 100 | 7,320.0 ns | 808.8 ns | 535.00 ns | 7.52 | 0.71 | 2088 B | 1.04 |
| StringCreateFromArray | 100 | 5,160.0 ns | 657.8 ns | 435.12 ns | 5.32 | 0.74 | 2040 B | 1.02 |

VISMA

# Strings - Split

Creating new strings by splitting allocates additional memory. Use this method only if you need the results, don't use it only for comparison.

```
string csv = "col0;col1;col2;col3";
// Don't:
string parts = csv.Split(';');
bool val0 = parts[0] == "col0";
// Do:
int idx = csv.IndexOf(';');
bool val0 = csv.AsSpan(0, idx) == "col0";
```

# Strings - Split Benchmark

| Method | N | Mean | Error | StdDev | Ratio | Gen0 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|
| StringSplit | 1 | 20.747 ns | 0.4190 ns | 0.2494 ns | 1.00 | 0.0083 | 104 B | 1.00 |
| Substring | 1 | 7.112 ns | 0.1159 ns | 0.0690 ns | 0.34 | 0.0051 | 64 B | 0.62 |
| SpanSlice | 1 | 1.293 ns | 0.0281 ns | 0.0167 ns | 0.06 | - | - | 0.00 |
| | | | | | | | | |
| StringSplit | 100 | 1,959.945 ns | 31.1836 ns | 18.5568 ns | 1.00 | 0.8278 | 10400 B | 1.00 |
| Substring | 100 | 700.486 ns | 9.4027 ns | 5.5954 ns | 0.36 | 0.5093 | 6400 B | 0.62 |
| SpanSlice | 100 | 146.399 ns | 2.1296 ns | 1.4086 ns | 0.07 | - | - | 0.00 |

# Strings - Comparison

Case insensitive string comparison is quite common practice. In order to make it effective, we should avoid changing the case of both strings, since it creates 2 new strings.

```
// Don't:
bool eqCi = left.ToLower() == right.ToLower();
bool eqCi = string.Equals(left.ToUpper(), right.ToUpper());

// Do:
bool eqCi = string.Equals(left, right, StringComparison.OrdinalIgnoreCase);
```

# Strings - Comparison Benchmark

| Method | N | Mean | Error | StdDev | Median | Ratio | RatioSD | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|---|
| ToLowerEqualityOperator | 1 | 1,380.0 ns | 656.3 ns | 434.10 ns | 1,250.0 ns | 1.00 | 0.00 | 424 B | 1.00 |
| ToLowerEquals | 1 | 1,680.0 ns | 546.5 ns | 361.48 ns | 1,800.0 ns | 1.32 | 0.47 | 424 B | 1.00 |
| EqualsOrdinalIgnoreCase | 1 | 220.0 ns | 171.6 ns | 113.53 ns | 200.0 ns | 0.17 | 0.11 | 400 B | 0.94 |
| ToLowerEqualityOperator | 10 | 1,440.0 ns | 813.2 ns | 537.90 ns | 1,150.0 ns | 1.00 | 0.00 | 448 B | 1.00 |
| ToLowerEquals | 10 | 1,530.0 ns | 552.3 ns | 365.30 ns | 1,550.0 ns | 1.17 | 0.43 | 448 B | 1.00 |
| EqualsOrdinalIgnoreCase | 10 | 340.0 ns | 238.5 ns | 157.76 ns | 400.0 ns | 0.24 | 0.10 | 400 B | 0.89 |
| ToLowerEqualityOperator | 50 | 1,211.1 ns | 596.8 ns | 355.12 ns | 1,100.0 ns | 1.00 | 0.00 | 528 B | 1.00 |
| ToLowerEquals | 50 | 1,630.0 ns | 519.1 ns | 343.35 ns | 1,650.0 ns | 1.45 | 0.38 | 528 B | 1.00 |
| EqualsOrdinalIgnoreCase | 50 | 433.3 ns | 145.5 ns | 86.60 ns | 500.0 ns | 0.38 | 0.12 | 400 B | 0.76 |

# Collection Iteration

- Loop is one of the most important features in C#
- Most common ways to loop through a list or a array:
  - **for**
  - **while**
  - **foreach**
- Additional ways:
  - **Linq**
  - **Parallel Foreach**
  - **Span**

# Collection Iteration - Low Level - for

```
// C#

for (int i = 0; i < list.Count; i++)
{


}
```

```
// Low-Level C#

int num = 0;

while (num < list.Count)
{
    num++;
}
```

# Collection Iteration - Low Level - foreach

```
// C#

foreach (int item in list)
{

}
```

```
// Low-Level C#

try
{
    while (enumerator.MoveNext())
    {
        int current = enumerator.Current;
    }
}
finally
{
    ((IDisposable)enumerator).Dispose();
}
```

VISMA

# Collection Iteration - Other

```
list.ForEach(item => { _ = item; });

Parallel.ForEach(list, item => { _ = item; });

// Span
Span<int> listSpan = CollectionsMarshal.AsSpan(list);

for (int i = 0; i < listSpan.Length; i++)
{
    _ = listSpan[i];
}
```

# Collection Iteration - for Span

```
Span<int> listSpan = CollectionsMarshal.AsSpan(list);
ref int firstItem = ref MemoryMarshal.GetReference(listSpan);

for (int i = 0; i < listSpan.Length; i++)
{
    _ = Unsafe.Add(ref firstItem, i);
}
```

# Collection Iteration - while Span

```
Span<int> listSpan = CollectionsMarshal.AsSpan(list);
ref int firstItem = ref MemoryMarshal.GetReference(listSpan);
ref int lastItem = ref Unsafe.Add(ref firstItem, listSpan.Length);

while (Unsafe.IsAddressLessThan(ref firstItem, ref lastItem))
{
    _ = firstItem;
    firstItem = ref Unsafe.Add(ref firstItem, 1);
}
```

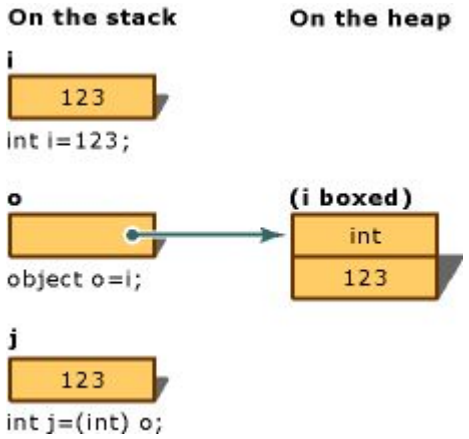
# Collection Iteration - Benchmark

| Method | N | Mean | Error | StdDev | Ratio | RatioSD | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|
| ForStatement | 100 | 1,011.1 ns | 177.13 ns | 105.41 ns | 1.00 | 0.00 | 400 B | 1.00 |
| WhileStatement | 100 | 870.0 ns | 160.16 ns | 105.93 ns | 0.89 | 0.12 | 400 B | 1.00 |
| ForEachStatement | 100 | 812.5 ns | 122.53 ns | 64.09 ns | 0.79 | 0.11 | 400 B | 1.00 |
| ForEachMethod | 100 | 175.0 ns | 88.51 ns | 46.29 ns | 0.17 | 0.04 | 400 B | 1.00 |
| ParallelForEach | 100 | 15,312.5 ns | 3,922.96 ns | 2,051.78 ns | 14.88 | 2.56 | 3448 B | 8.62 |
| ForListSpan | 100 | 944.4 ns | 357.57 ns | 212.79 ns | 0.95 | 0.26 | 64 B | 0.16 |
| ForListRef | 100 | 900.0 ns | 174.57 ns | 115.47 ns | 0.89 | 0.19 | 400 B | 1.00 |
| WhileListRef | 100 | 775.0 ns | 88.51 ns | 46.29 ns | 0.75 | 0.06 | 64 B | 0.16 |
| ForStatement | 10000 | 11,880.0 ns | 2,953.87 ns | 1,953.80 ns | 1.00 | 0.00 | 400 B | 1.00 |
| WhileStatement | 10000 | 11,790.0 ns | 2,888.36 ns | 1,910.47 ns | 0.99 | 0.03 | 400 B | 1.00 |
| ForEachStatement | 10000 | 12,733.3 ns | 3,195.04 ns | 1,901.32 ns | 1.06 | 0.14 | 400 B | 1.00 |
| ForEachMethod | 10000 | 12,290.0 ns | 710.73 ns | 470.11 ns | 1.06 | 0.18 | 400 B | 1.00 |
| ParallelForEach | 10000 | 315,470.0 ns | 188,485.50 ns | 124,671.49 ns | 26.87 | 10.73 | 4032 B | 10.08 |
| ForListSpan | 10000 | 9,840.0 ns | 2,096.35 ns | 1,386.60 ns | 0.83 | 0.05 | 400 B | 1.00 |
| ForListRef | 10000 | 9,700.0 ns | 2,651.38 ns | 1,753.73 ns | 0.81 | 0.03 | 400 B | 1.00 |
| WhileListRef | 10000 | 10,520.0 ns | 3,981.42 ns | 2,633.46 ns | 0.88 | 0.09 | 400 B | 1.00 |
| ForStatement | 1000000 | 309,388.9 ns | 16,682.17 ns | 9,927.29 ns | 1.00 | 0.00 | 400 B | 1.00 |
| WhileStatement | 1000000 | 299,287.5 ns | 4,624.38 ns | 2,418.64 ns | 0.96 | 0.03 | 112 B | 0.28 |
| ForEachStatement | 1000000 | 301,260.0 ns | 8,137.29 ns | 5,382.32 ns | 0.98 | 0.03 | 400 B | 1.00 |
| ForEachMethod | 1000000 | 1,257,022.2 ns | 45,101.90 ns | 26,839.42 ns | 4.07 | 0.20 | 400 B | 1.00 |
| ParallelForEach | 1000000 | 1,662,777.8 ns | 157,627.88 ns | 93,801.82 ns | 5.38 | 0.39 | 5680 B | 14.20 |
| ForListSpan | 1000000 | 203,130.0 ns | 6,182.02 ns | 4,089.02 ns | 0.65 | 0.02 | 400 B | 1.00 |
| ForListRef | 1000000 | 204,300.0 ns | 12,786.50 ns | 8,457.48 ns | 0.66 | 0.04 | 400 B | 1.00 |
| WhileListRef | 1000000 | 252,630.0 ns | 125,895.23 ns | 83,271.90 ns | 0.84 | 0.29 | 400 B | 1.00 |

VISMA

# Boxing

- **Boxing** and **unboxing** are computationally expensive processes
- When a value type is boxed, a new object must be allocated and constructed
- **structs** when casted to the **interface** that imeplements are boxed, since interfaces are reference types




```
int i = 123;       // a value type
object o = i;      // boxing
int j = (int)o;    // unboxing
```

VISMA

# Boxing

**How to avoid the boxing of structs that implement interfaces, when passed as arguments to methods**

```
public interface IFoo { ... }

public struct Foo : IFoo { ... }

public void Bar(IFoo barParameter) { ... }
```

**struct** is a **value type** it will get **boxed**. Boxing the struct costs performance as an operation, but even worse we have memory allocation on the heap, which means that eventually the **garbage collector** will have to collect it.

# Boxing

**Changing the definition of our method to use generics**

```
public void Bar<T>(T barParameter) where T : IFoo { ... }
```

Even if the **barParameter** stays a value type when it is a struct, if inside our method we use methods of the object class, like **ToString** or **Equals** the **barParameter** will have to get boxed, because it inherits those methods from the **object** class. We have to make sure that our struct (in our case the **Foo** struct) overrides those methods. That way, boxing won't occur. In case that we use the **Equals** method for comparison and our **Bar** method is going to be used by others, a good practice is to require that the **T** type also implements the **IEquatable<T>** interface.

```
public void Bar<T>(T barParameter) where T : IFoo, IEquatable<T> { ... }
```

# Boxing

**Adding the in keyword**

**barParameter** is a value type when it is a struct, if our struct is big (above **16 bytes**) we will have performance problems. We tried to gain performance by avoiding boxing, but we will lose in performance because of the copying of the value types. We just have to pass our value type by **reference** by changing how we define our method, one more time.

```
public void Bar<T>(in T barParameter) where T : IFoo, IEquatable<T> { ... }
```

- To make sure that the parameter's value stays the same the compiler make a **defensive copy** of the parameter every time a method/property is used.
- If the struct is **readonly** then the **compiler removes the defensive copy** the same way as it does for **readonly fields**.
- You should never pass a non-readonly struct as **in** parameter. It almost always will make the performance worse.

# Boxing

## Conclusion

- The change from
  ```
  public void Bar(IFoo barParameter) { ... }
  ```
  to
  ```
  public void Bar<T>(in T barParameter) where T : IFoo { ... }
  ```
  or
  ```
  public void Bar<T>(in T barParameter) where T : IFoo, IEquatable<T> { ... }
  ```
  is a good performance improvement when we have structs.
- **readonly structs** are very useful from the design and the performance points of view.
- If the size of a readonly struct is **bigger than IntPtr.Size** you should pass it as an **in-parameter** for performance reasons.
- You should **never** use a **non-readonly struct** as the **in** parameters because it may negatively affect performance and could lead to an obscure behavior if the **struct is mutable**.

# Boxing - Benchmark

| Method | N | Mean | Error | StdDev | Median | Ratio | RatioSD | Gen0 | Allocated |
|---|---|---|---|---|---|---|---|---|---|
| DirectAssignment | 1 | 0.0167 ns | 0.0161 ns | 0.0151 ns | 0.0185 ns | ? | ? | - | - |
| BoxedAssignment | 1 | 1.7041 ns | 0.0241 ns | 0.0188 ns | 1.7031 ns | ? | ? | 0.0019 | 24 B |
| InterfaceAssignment | 1 | 0.0008 ns | 0.0024 ns | 0.0023 ns | 0.0000 ns | ? | ? | - | - |
| DirectAssignment | 100 | 25.4246 ns | 0.0870 ns | 0.0771 ns | 25.4407 ns | 1.00 | 0.00 | - | - |
| BoxedAssignment | 100 | 173.8024 ns | 3.0313 ns | 6.9039 ns | 172.0512 ns | 6.84 | 0.27 | 0.1912 | 2400 B |
| InterfaceAssignment | 100 | 26.0632 ns | 0.2145 ns | 0.1791 ns | 26.1204 ns | 1.03 | 0.01 | - | - |
| DirectAssignment | 10000 | 1,999.1732 ns | 16.3677 ns | 12.7788 ns | 2,002.3108 ns | 1.00 | 0.01 | - | - |
| BoxedAssignment | 10000 | 15,039.0604 ns | 294.8220 ns | 372.8554 ns | 14,959.2682 ns | 7.52 | 0.19 | 19.1193 | 240000 B |
| InterfaceAssignment | 10000 | 1,967.4176 ns | 12.9494 ns | 10.8133 ns | 1,967.4011 ns | 0.98 | 0.01 | - | - |

# Regular expressions

Regular expressions are very useful, performs pattern matching rapidly and efficiently. However, in some cases, it can be slow, or even stop responding. Factors affecting the performance are:

- **Instantiation**
- **Compiled/Interpreted**
- **Timeout**

# Regular expressions - Instantiation

Creating a new instance of **Regex** is quite costly. Better alternatives are:

- **Static variable**

```
private static readonly Regex Regex = new(pattern);
```

- **Static method**

```
Regex.IsMatch(pattern);
```

  The pattern is cached, the default cache size is 15.
  Could be increased for better performance:

```
Regex.CacheSize += 100;
```

- **Source generated Regex (.NET 7)**

```
[GeneratedRegex(pattern)]
private static partial Regex Regex();
```

# Regular expressions - Compiled

- Regular expressions are **interpreted** by default. When a method is called, the operation code is converted to **MSIL** and executed by the **JIT compiler**. **Regularly interpreted** phrases reduce **startup time** at the expense of slower **run time**.
- When the **Compiled** flag is used, when a regular expression object is prototyped, the regular expression engine converts the regular expression into a set of intermediate operating codes, then converts to **MSIL**. When a method is called, the compiler executes **JIT MSIL**. Unlike **regular interpreted** expressions, **regular compiled** expressions increase **startup time** but execute individual pattern matching methods **faster**. Source generated Regex is compiled by default.

```
private static readonly Regex Regex = new(pattern, RegexOptions.Compiled);
```

VISMA

# Regular expressions - Timeout

- Regular expressions could be time-consuming
- Can often rely on excessive **backtracking**, which impacts its performance significantly
- A malicious user can provide input causing a **Denial-of-Service** attack

```
// Individual Regex object:
private static readonly Regex Regex = new(
     pattern,
     RegexOptions.Compiled,
     TimeSpan.FromMilliseconds(100));

// Globally:
AppDomain.CurrentDomain.SetData(
     "REGEX_DEFAULT_MATCH_TIMEOUT",
     TimeSpan.FromMilliseconds(100));
```

VISMA

# Regular expressions - Benchmark

| Method | N | Mean | Error | StdDev | Ratio | RatioSD | Gen0 | Gen1 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|---|---|
| NewInstanceInterpreted | 1 | 949.66 ns | 10.608 ns | 7.016 ns | 1.00 | 0.00 | 0.2365 | 0.0019 | 2968 B | 1.00 |
| NewInstanceCompiled | 1 | 1,139,621.52 ns | 13,741.572 ns | 9,089.199 ns | 1,200.09 | 13.15 | - | - | 15250 B | 5.14 |
| StaticVariableInterpreted | 1 | 52.03 ns | 0.401 ns | 0.265 ns | 0.05 | 0.00 | - | - | - | 0.00 |
| StaticVariableCompiled | 1 | 29.72 ns | 0.736 ns | 0.385 ns | 0.03 | 0.00 | - | - | - | 0.00 |
| StaticMethodInterpreted | 1 | 55.21 ns | 0.648 ns | 0.429 ns | 0.06 | 0.00 | - | - | - | 0.00 |
| StaticMethodCompiled | 1 | 29.59 ns | 0.201 ns | 0.120 ns | 0.03 | 0.00 | - | - | - | 0.00 |
| SourceGenerated | 1 | 25.03 ns | 0.488 ns | 0.323 ns | 0.03 | 0.00 | - | - | - | 0.00 |
| | | | | | | | | | | |
| NewInstanceInterpreted | 100 | 97,444.14 ns | 3,428.218 ns | 2,267.554 ns | 1.00 | 0.00 | 23.5596 | 0.1221 | 296800 B | 1.00 |
| NewInstanceCompiled | 100 | 116,861,950.00 ns | 1,288,408.324 ns | 852,202.333 ns | 1,199.77 | 25.35 | - | - | 1524880 B | 5.14 |
| StaticVariableInterpreted | 100 | 5,359.60 ns | 65.069 ns | 43.039 ns | 0.06 | 0.00 | - | - | - | 0.00 |
| StaticVariableCompiled | 100 | 3,002.00 ns | 57.983 ns | 38.352 ns | 0.03 | 0.00 | - | - | - | 0.00 |
| StaticMethodInterpreted | 100 | 5,560.59 ns | 71.776 ns | 42.713 ns | 0.06 | 0.00 | - | - | - | 0.00 |
| StaticMethodCompiled | 100 | 3,108.49 ns | 37.068 ns | 24.518 ns | 0.03 | 0.00 | - | - | - | 0.00 |
| SourceGenerated | 100 | 2,506.75 ns | 47.327 ns | 31.304 ns | 0.03 | 0.00 | - | - | - | 0.00 |

VISMA

# Reflection

C# Reflection ecosystem primarily consists of **System.Type** class and the classes/interfaces from **System.Reflection** namespace. It should be used carefully since it can access types, fields, properties, methods etc. not accessible at compile time. Reflection has also some performance impact as well.

# Reflection - Dynamic instantiation

Ways to create an object instance using reflection:

- Standard Reflection using Invoke
- Activator.CreateInstance
- Compiled expressions
- Reflection.Emit

# Reflection - Dynamic instantiation

```
// Invoke

ConstructorInfo ctor = typeof(MyType)
    .GetConstructor(System.Type.EmptyTypes);
object myInstance = ctor.Invoke(null);

// Activator.CreateInstance

object myInstance = Activator.CreateInstance(typeof(MyType));
```

# Reflection - Dynamic instantiation

```
// Compiled expressions

NewExpression constructorExpression = Expression.New(typeof(MyType));
Expression<Func<object>> lambdaExpression = Expression
    .Lambda<Func<object>>(constructorExpression);
Func<object> createMyTypeFunc = lambdaExpression.Compile();
object myInstance = createMyTypeFunc();
```

# Reflection - Dynamic instantiation

```
// Emit


DynamicMethod createMyTypeMethod = new("DynamicMethodData",
typeof(MyType), null, typeof(Benchmark).Module, false);
ILGenerator il = createMyTypeMethod.GetILGenerator();
il.Emit(OpCodes.Newobj, typeof(MyType).GetConstructors()[0]);
il.Emit(OpCodes.Ret);
Func<object> dynamicMethodActivator =
(Func<object>)createMyTypeMethod.CreateDelegate(typeof(Func<object>));
object myInstance = dynamicMethodActivator();
```

VISMA

# Reflection - Dynamic instantiation - Benchmark

| Method | N | Mean | Error | StdDev | Median | Ratio | RatioSD | Gen0 | Allocated |
|---|---|---|---|---|---|---|---|---|---|
| New | 1 | 0.0128 ns | 0.0150 ns | 0.0079 ns | 0.0166 ns | ? | ? | - | - |
| ActivatorCreateInstance | 1 | 4.4437 ns | 0.2400 ns | 0.1587 ns | 4.3897 ns | ? | ? | 0.0025 | 32 B |
| ConstructorInfo | 1 | 5.4410 ns | 0.5105 ns | 0.3376 ns | 5.4847 ns | ? | ? | 0.0025 | 32 B |
| ObjectActivatorDelegate | 1 | 1.8625 ns | 0.1224 ns | 0.0728 ns | 1.8496 ns | ? | ? | 0.0025 | 32 B |
| ReflectionEmit | 1 | 2.6693 ns | 0.3477 ns | 0.2069 ns | 2.5833 ns | ? | ? | 0.0025 | 32 B |
| New | 100 | 26.0068 ns | 0.2859 ns | 0.1891 ns | 26.0043 ns | 1.00 | 0.00 | - | - |
| ActivatorCreateInstance | 100 | 398.7737 ns | 23.6254 ns | 15.6268 ns | 394.8362 ns | 15.33 | 0.60 | 0.2546 | 3200 B |
| ConstructorInfo | 100 | 498.3280 ns | 14.9231 ns | 8.8805 ns | 497.7489 ns | 19.17 | 0.45 | 0.2546 | 3200 B |
| ObjectActivatorDelegate | 100 | 213.7059 ns | 11.0639 ns | 7.3181 ns | 214.6135 ns | 8.22 | 0.28 | 0.2549 | 3200 B |
| ReflectionEmit | 100 | 291.4132 ns | 23.9418 ns | 14.2474 ns | 288.8131 ns | 11.21 | 0.51 | 0.2546 | 3200 B |
| New | 10000 | 2,001.2704 ns | 20.0624 ns | 13.2701 ns | 2,004.1809 ns | 1.00 | 0.00 | - | - |
| ActivatorCreateInstance | 10000 | 47,563.3356 ns | 6,646.5850 ns | 4,396.3045 ns | 46,085.7239 ns | 23.77 | 2.26 | 25.4517 | 320000 B |
| ConstructorInfo | 10000 | 54,243.7283 ns | 5,701.2659 ns | 3,392.7319 ns | 52,929.9866 ns | 27.11 | 1.71 | 25.4517 | 320000 B |
| ObjectActivatorDelegate | 10000 | 21,043.0437 ns | 780.1899 ns | 464.2785 ns | 21,142.1082 ns | 10.52 | 0.29 | 25.4822 | 320000 B |
| ReflectionEmit | 10000 | 30,429.2664 ns | 1,653.0854 ns | 983.7246 ns | 30,569.4885 ns | 15.21 | 0.45 | 25.4822 | 320000 B |

VISMA

# Reflection - Dynamic method invocation

```
MyClass obj = new();

// MethodInfo
typeof(MyClass)
    .GetMethod(nameof(MyClass.Add))
    .Invoke(obj, [1]);
```

# Reflection - Dynamic method invocation

```
private delegate int AddDelegate(int value);
MyClass obj = new();

// Typed delegate
AddDelegate addDelegate = (AddDelegate)Delegate
    .CreateDelegate(typeof(AddDelegate), _obj,
nameof(MyClass.Add));
addDelegate.Invoke(1);

// Delegate
addDelegate.DynamicInvoke(1);
```

# Reflection - Dynamic method invocation - Benchmark

| Method | N | Mean | Error | StdDev | Median | Ratio | RatioSD | Gen0 | Allocated |
|---|---|---|---|---|---|---|---|---|---|
| DirectInvoke | 1 | 0.0011 ns | 0.0030 ns | 0.0020 ns | 0.0000 ns | ? | ? | - | - |
| MethodInfoInvoke | 1 | 13.7295 ns | 0.2968 ns | 0.1766 ns | 13.7303 ns | ? | ? | 0.0063 | 80 B |
| DelegateInvoke | 1 | 0.0250 ns | 0.0430 ns | 0.0284 ns | 0.0136 ns | ? | ? | - | - |
| DelegateDynamicInvoke | 1 | 30.7716 ns | 1.0172 ns | 0.6728 ns | 30.4518 ns | ? | ? | 0.0063 | 80 B |
| | | | | | | | | | |
| DirectInvoke | 100 | 31.8769 ns | 0.4070 ns | 0.2422 ns | 31.8812 ns | 1.00 | 0.00 | - | - |
| MethodInfoInvoke | 100 | 1,471.7631 ns | 35.6650 ns | 18.6535 ns | 1,471.3875 ns | 46.16 | 0.69 | 0.6371 | 8000 B |
| DelegateInvoke | 100 | 44.3655 ns | 0.6716 ns | 0.3996 ns | 44.1856 ns | 1.39 | 0.01 | - | - |
| DelegateDynamicInvoke | 100 | 3,471.3875 ns | 332.0072 ns | 219.6022 ns | 3,587.5906 ns | 110.21 | 5.72 | 0.6371 | 8000 B |
| | | | | | | | | | |
| DirectInvoke | 10000 | 2,308.1608 ns | 6.8915 ns | 3.6044 ns | 2,308.8995 ns | 1.00 | 0.00 | - | - |
| MethodInfoInvoke | 10000 | 143,226.0444 ns | 11,329.1992 ns | 6,741.8247 ns | 142,433.4717 ns | 61.82 | 3.01 | 63.7207 | 800000 B |
| DelegateInvoke | 10000 | 3,633.6291 ns | 59.8751 ns | 35.6307 ns | 3,621.9547 ns | 1.57 | 0.02 | - | - |
| DelegateDynamicInvoke | 10000 | 311,574.0723 ns | 6,568.8470 ns | 4,344.8856 ns | 312,198.0225 ns | 135.17 | 1.82 | 63.4766 | 800001 B |

# Reflection - Dynamic member access

```
private sealed class MyClass
{
    public readonly int _publicField = 69;
    private readonly int _privateField = 69;
}

MyClass obj = new();
FieldInfo fieldInfo = typeof(MyClass).GetField(
    "_privateField",
    BindingFlags.Instance | BindingFlags.NonPublic);

int value = fieldInfo.GetValue(obj);
```

# Reflection - Dynamic member access

```
// .NET 8+
[UnsafeAccessor(UnsafeAccessorKind.Field, Name = "_privateField")]
extern static ref int GetInstanceField(MyClass @this);

int value = GetInstanceField(obj);
```

# Reflection - Dynamic member access

| Method | N | Mean | Error | StdDev | Median | Ratio | RatioSD | Gen0 | Allocated |
|---|---|---|---|---|---|---|---|---|---|
| DirectAccess | 1 | 0.0010 ns | 0.0034 ns | 0.0022 ns | 0.0000 ns | ? | ? | - | - |
| DynamicAccess | 1 | 27.7639 ns | 0.4577 ns | 0.3027 ns | 27.7231 ns | ? | ? | 0.0019 | 24 B |
| CachedDynamicAccess | 1 | 20.3646 ns | 0.5212 ns | 0.3447 ns | 20.3673 ns | ? | ? | 0.0019 | 24 B |
| UnsafeAccessor | 1 | 0.0028 ns | 0.0046 ns | 0.0030 ns | 0.0017 ns | ? | ? | - | - |
| DirectAccess | 100 | 25.4315 ns | 0.1310 ns | 0.0685 ns | 25.4461 ns | 1.00 | 0.00 | - | - |
| DynamicAccess | 100 | 2,672.5078 ns | 30.0680 ns | 19.8881 ns | 2,666.1329 ns | 104.93 | 0.76 | 0.1907 | 2400 B |
| CachedDynamicAccess | 100 | 1,989.3222 ns | 29.7202 ns | 19.6581 ns | 1,987.9147 ns | 78.07 | 0.73 | 0.1907 | 2400 B |
| UnsafeAccessor | 100 | 27.1051 ns | 0.0646 ns | 0.0384 ns | 27.1075 ns | 1.07 | 0.00 | - | - |
| DirectAccess | 10000 | 1,971.9254 ns | 31.2252 ns | 20.6535 ns | 1,969.8759 ns | 1.00 | 0.00 | - | - |
| DynamicAccess | 10000 | 274,917.2510 ns | 5,397.4818 ns | 3,570.1000 ns | 275,118.7988 ns | 139.43 | 2.35 | 19.0430 | 240000 B |
| CachedDynamicAccess | 10000 | 205,163.4985 ns | 5,453.0558 ns | 3,606.8588 ns | 205,584.4849 ns | 104.05 | 1.72 | 19.0430 | 240000 B |
| UnsafeAccessor | 10000 | 2,060.3037 ns | 99.7547 ns | 59.3624 ns | 2,072.7692 ns | 1.05 | 0.04 | - | - |

# Logging

Logging is important but it should be done the correct way, otherwise it could significantly degrade the performance.

- **(Re)use message templates**
  Messages are parsed and cached. Threatening parameters as string in the message will degrade performance and consume cache memory.
- **Check event level**
  Level checking is extremely cheap and the overhead of calling disabled logger methods very low.
- Use **LoggerMessage**
- Use **Compile-time logging source generation**

# Logging - Message templates

The message template should be a constant string to avoid unnecessary caching or allocation.

```
// Don't:
_logger.Information("The time is " + DateTime.Now);
_logger.Information($"The time is {DateTime.Now}");

// Do:
public const string MessageTemplate = "The time is {Now}";
_logger.Information("The time is {Now}", DateTime.Now);
_logger.Information(MessageTemplate, DateTime.Now);
```

# Logging - Event levels

The log level is set in the app configuration and should be checked in order the avoid unnecessary logging and use of resources.

```
if (_logger.IsEnabled(LogEventLevel.Debug))
{
    _logger.Debug("Someone is stuck debugging...");
}
```

# Logging - LoggerMessage

The **LoggerMessage** class exposes functionality to create cacheable **delegates** that require **fewer object allocations** and **reduced computational overhead** compared to logger extension methods, such as **LogInformation** and **LogDebug**.

- Logger extension methods require **"boxing"** (converting) value types, such as int, into object. The **LoggerMessage** pattern avoids boxing by using static **Action fields** and extension methods with strongly typed parameters.
- Logger extension methods must parse the message template (named format string) every time (if not cached) a log message is written. **LoggerMessage** only requires parsing a template once when the message is defined.

# Logging - LoggerMessage

```
public static class LoggerExtensions
{
    private static readonly Action<ILogger, Exception> _failedToProcessWorkItem
        = LoggerMessage.Define(
            LogLevel.Critical,
            new EventId(13, nameof(FailedToProcessWorkItem)),
            "Epic failure processing item!");

    public static void FailedToProcessWorkItem(this ILogger logger, Exception ex)
    {
        _failedToProcessWorkItem(logger, ex);
    }
}
```

# Logging - Source generation

**NET 6** introduces the **LoggerMessageAttribute** type. The auto-generated source code relies on the **ILogger** interface in conjunction with **LoggerMessage.Define** functionality.

- Logging methods must be **partial** and return **void**.
- Logging method names must not start with an underscore.
- Parameter names of logging methods must not start with an underscore.
- Logging methods may not be defined in a nested type.
- Logging methods cannot be generic.
- If a logging method is static, the **ILogger** instance is required as a parameter.
- Checks for **log level** by default.

# Logging - Source generation

```
public static partial class Log
{
    [LoggerMessage(
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{HostName}`")]
    public static partial void CouldNotOpenSocket(ILogger logger, string hostName);

    // Or as an extension method
    [LoggerMessage(
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{HostName}`")]
    public static partial void CouldNotOpenSocket(this ILogger logger, string hostName);
}
```

# Logging - Source generation

```
public partial class InstanceLoggingExample(ILogger logger)
{
    private readonly ILogger _logger = logger;

    [LoggerMessage(
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{HostName}`")]
    private partial void CouldNotOpenSocket(string hostName);
}
```

# Logging - Benchmark

| Method | Mean | Error | StdDev | Ratio | RatioSD | Gen0 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|
| Basic | 27.9133 ns | 0.7055 ns | 0.4667 ns | 1.00 | 0.02 | 0.0044 | 56 B | 1.00 |
| ConstantMessageTemplate | 26.4735 ns | 0.8573 ns | 0.5671 ns | 0.95 | 0.02 | 0.0044 | 56 B | 1.00 |
| DynamicMessageTemplate | 85.0843 ns | 1.1440 ns | 0.6808 ns | 3.05 | 0.05 | 0.0069 | 88 B | 1.57 |
| LogLevelCheck | 0.4085 ns | 0.0381 ns | 0.0252 ns | 0.01 | 0.00 | - | - | 0.00 |
| LoggerMessageLog | 1.0315 ns | 0.0339 ns | 0.0225 ns | 0.04 | 0.00 | - | - | 0.00 |
| SourceGeneratedLog | 0.9488 ns | 0.0262 ns | 0.0137 ns | 0.03 | 0.00 | - | - | 0.00 |

# Sealed classes

- By default, **classes** are not **sealed**
- It has performance implications (.NET 6+)
- When a class is **sealed** the JIT can apply optimizations and slightly improve the performance of the application

**Virtual** methods on a **sealed** type are more likely to be **devirtualized** by the runtime. If the runtime can see that a given instance on which a virtual call is being made is actually **sealed**, then it knows for certain what the actual target of the call will be, and it can invoke that target directly rather than doing a virtual dispatch operation. Better yet, once the call is **devirtualized**, it might be **inlineable**, and then if it's **inlined**, all the previously discussed benefits around optimizing the caller+callee combined kick in.

# Sealed classes

```
public class BaseType
{
    public virtual int Method() => 1 + 2;
}
// Don't:
public class NonSealedType : BaseType
{
    public override int Method() => 6 + 9;
}
// Do:
public sealed class SealedType : BaseType
{
    public override int Method() => 6 + 9;
}
```

# Sealed classes

| Method | Mean | Error | StdDev | Median | Ratio | RatioSD | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|
| NonSealed | 0.2033 ns | 0.0154 ns | 0.0091 ns | 0.2000 ns | 1.00 | 0.00 | - | NA |
| Sealed | 0.0046 ns | 0.0113 ns | 0.0075 ns | 0.0000 ns | 0.03 | 0.04 | - | NA |

# Honorable mentions - ArrayPool<T>

Provides a resource pool that enables reusing instances of type **T[]**. Rent and return buffers, can improve performance in situations where arrays are created and destroyed frequently, resulting in significant memory pressure on the garbage collector.

# Honorable mentions - ObjectPool<T>

Supports keeping a group of objects in memory for reuse rather than allowing the objects to be garbage collected. Apps might want to use the object pool if the objects that are being managed are:

- Expensive to allocate/initialize.
- Represent a limited resource.
- Used predictably and frequently.

Example: **StringBuilder** pool

Object pooling doesn't always improve performance:

- Unless the initialization cost of an object is high, it's usually slower to get the object from the pool.
- Objects managed by the pool aren't de-allocated until the pool is de-allocated.

# Honorable mentions - stackalloc

A stackalloc expression allocates a block of memory on the stack. A stack-allocated memory block created during the method execution is automatically discarded when that method returns. A stack allocated memory block isn't subject to garbage collection and doesn't have to be pinned with a fixed statement.

```
const int MaxStackLimit = 1024;

Span<byte> buffer = inputLength <= MaxStackLimit ?
    stackalloc byte[MaxStackLimit] :
    new byte[inputLength];
```

# Honorable mentions - dynamic type

## Don’t!

```
dynamic y = (dynamic)6 + 9;
// Low level C#
[CompilerGenerated]
private static class <>o__0
{
    public static CallSite<Func<CallSite, object, int, object>> <>p__0;
}
if (<>o__0.<>p__0 == null)
{
    Type typeFromHandle = typeof(C);
    CSharpArgumentInfo[] array = new CSharpArgumentInfo[2];
    array[0] = CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null);
    array[1] = CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.UseCompileTimeType | CSharpArgumentInfoFlags.Constant, null);
    <>o__0.<>p__0 = CallSite<Func<CallSite, object, int,
object>>.Create(Microsoft.CSharp.RuntimeBinder.Binder.BinaryOperation(CSharpBinderFlags.None, ExpressionType.Add, typeFromHandle,
array));
}
object obj = <>o__0.<>p__0.Target(<>o__0.<>p__0, 6, 9);
```

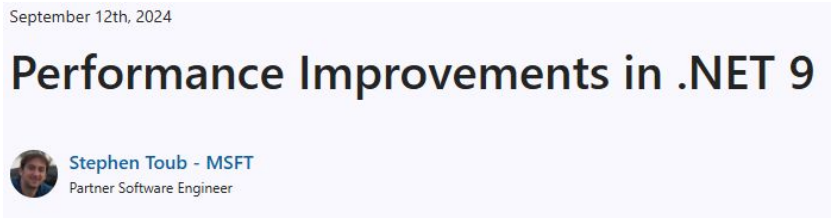
VISMA

# Honorable mentions - dynamic - Benchmark

| Method | N | Mean | Error | StdDev | Median | Ratio | RatioSD | Gen0 | Allocated |
|---|---|---|---|---|---|---|---|---|---|
| DirectAccess | 1 | 0.0018 ns | 0.0052 ns | 0.0031 ns | 0.0000 ns | ? | ? | - | - |
| DynamicAccess | 1 | 5.9123 ns | 0.1744 ns | 0.0912 ns | 5.8949 ns | ? | ? | 0.0019 | 24 B |
| DirectAccess | 100 | 25.9838 ns | 0.6660 ns | 0.3963 ns | 26.0792 ns | 1.00 | 0.00 | - | - |
| DynamicAccess | 100 | 595.7484 ns | 7.4037 ns | 3.8723 ns | 597.1856 ns | 22.96 | 0.43 | 0.1907 | 2400 B |
| DirectAccess | 10000 | 1,974.9794 ns | 27.4212 ns | 18.1374 ns | 1,975.5033 ns | 1.00 | 0.00 | - | - |
| DynamicAccess | 10000 | 60,659.3369 ns | 1,047.5335 ns | 547.8803 ns | 60,763.3972 ns | 30.69 | 0.45 | 19.1040 | 240000 B |

# Honorable mentions - .NET version

Each .NET version usually comes with some performance improvements described in a regular blog: https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-9/. For example .NET 9 comes with some LINQ improvements.




| Method | Runtime | Mean | Ratio | Gen0 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|
| Any | .NET 6.0 | 5,052.8 ns | 1.00 | - | 40 B | 1.00 |
| Any | .NET 7.0 | 5,962.0 ns | 1.18 | - | 40 B | 1.00 |
| Any | .NET 8.0 | 749.2 ns | 0.15 | 0.0029 | 40 B | 1.00 |
| Any | .NET 9.0 | 209.9 ns | 0.04 | - | - | 0.00 |
| All | .NET 6.0 | 5,241.5 ns | 1.00 | - | 40 B | 1.00 |
| All | .NET 7.0 | 5,981.8 ns | 1.14 | - | 40 B | 1.00 |
| All | .NET 8.0 | 741.5 ns | 0.14 | 0.0029 | 40 B | 1.00 |
| All | .NET 9.0 | 213.1 ns | 0.04 | - | - | 0.00 |
| Count | .NET 6.0 | 5,000.4 ns | 1.00 | - | 40 B | 1.00 |
| Count | .NET 7.0 | 5,985.7 ns | 1.20 | - | 40 B | 1.00 |
| Count | .NET 8.0 | 753.7 ns | 0.15 | 0.0029 | 40 B | 1.00 |
| Count | .NET 9.0 | 216.0 ns | 0.04 | - | - | 0.00 |
| First | .NET 6.0 | 2,410.5 ns | 1.00 | - | 40 B | 1.00 |
| First | .NET 7.0 | 3,011.9 ns | 1.25 | - | 40 B | 1.00 |
| First | .NET 8.0 | 414.8 ns | 0.17 | 0.0029 | 40 B | 1.00 |
| First | .NET 9.0 | 110.3 ns | 0.05 | - | - | 0.00 |
| Single | .NET 6.0 | 5,006.4 ns | 1.00 | - | 40 B | 1.00 |
| Single | .NET 7.0 | 6,041.5 ns | 1.21 | - | 40 B | 1.00 |
| Single | .NET 8.0 | 742.2 ns | 0.15 | 0.0029 | 40 B | 1.00 |
| Single | .NET 9.0 | 214.1 ns | 0.04 | - | - | 0.00 |

VISMA

# Summary

**Don’t:**

- Create unnecessary/unused object instances
- Allocate on heap if it’s not necessary
- Use inefficient/expensive methods for simple tasks
- Use reflection if there’s better alternative
- Create too many logs and check the log level
- Use dynamic type

# Summary

**Do:**

- Reuse object instances if it's possible
- Use pools
- Try to avoid boxing
- Use the right collection type
- Add sealed modifier for classes if it's possible
- Use Span<T>
- Use the latest .NET version, since there are performance improvements in each version