# C# 8 Feature Cheat Sheet

## Default interface methods

Allows you to add new functionality to your interfaces of your libraries and ensure the backward compatibility with code written for older versions of those interfaces.

```
interface IWriteLine
{
 public void WriteLine()
 {
   Console.WriteLine("Wow C# 8!");
 }
}
```

## Nullable reference types

Emits a compiler warning or error if a variable that must not be null is assigned to null.

```
string? nullableString = null;
// WARNING: may be null! Take care!
Console.WriteLine(nullableString.Length)
```

## Pattern matching enhancements

Provides the ability to deconstruct matched objects, and giving you access to parts of their data structures. C# offers a rich set of patterns that can be used for matching:

- Switch expressions
- Property patterns
- Tuple patterns
- Positional patterns

```
static bool Positive(Point p) => p switch
{
    (0, 0) => true,
    (var x, var y) when x > 0 && y > 0 => true,
    _ => false
};
```

## Asynchronous streams

Allows to have enumerators that support async operations.

```
await foreach (var x in enumerable)
{
   Console.WriteLine(x);
}
```

## Using declarations

Enhances the 'using' operator to use with Patterns and make it more natural.

```
using var repository = new Repository();
Console.WriteLine(repository.First());
// repository is disposed here!
```

## Enhancement of interpolated verbatim strings

Allows `@$""` as a verbatim interpolated string,

```
var file = @$"c:\temp\{filename}";//C# 8
```

## Null-coalescing assignment

Simplifies a common coding pattern where a variable is assigned a value if it is null.
It is common to see the code of the form:

```
if (variable == null)
{
  variable = expression;  // C#  1..7
}

variable ??= expression; // C# 8
```

## Static local functions

Allows you to add the 'static' modifier to the local functions.

```
int AddFiveAndSeven()
{
 int y = 1; int x = 2;
 return Add(x, y);

 static int Add(int o, int t) => o + t;
}
```

## Indices and ranges

Allows you to use more natural syntax for specifying subranges in an array or a collection.

```
int[] a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Index: Used to obtain the collection from the beginning or from the end.

```
 // Number 4 from end of the collection
Index i2 = ^4;
 Console.WriteLine($"{a[i2]}"); // "6"
```

Range: Access a sub-collection(slice) from a collection.

```
var slice = a[i1..i2]; // { 3, 4, 5 }
```

## Unmanaged constructed types

Allows you to take a pointer to unmanaged constructed types, such as ValueTuple<int, int>, as long as all the elements of the generic type are unmanaged.

```
struct Foo<T> where T : unmanaged
{
}

public unsafe void Test()
{
 var foo = new Foo<int>();
 var bar = &foo;    // C# 8
}
```

## Readonly-Member

Allows you to apply the readonly modifier to any member of a struct.

```
public struct XValue
{s
  private int X { get; set; }
  public readonly int IncreaseX()
  {
   // This will not compile: C# 8
   X = X + 1;

    var newX = X + 1; // OK
    return newX;
  }
}
```

## Stackalloc in nested expressions

The result of a stackalloc expression is of the System.Span<T> or System.ReadOnlySpan<T> type.

```
Span<int> set = stackalloc[] { 1, 2, 3,
4, 5, 6 };
var subSet = set.Slice(3, 2);
foreach (var n in subSet)
  Console.WriteLine(n);  // Output: 4  5
```

## Disposable ref structs

Allows you to use the 'using' pattern with ref struct or readonly ref struct'.

```
 // Pattern-based using for ref struct
ref struct Test {
  public void Dispose() {}
}

using var test = new Test();
// test is disposed here!
```

## About me


Bassam Alugili
CSharpCorner
www.bassam.ml


***25.10.2019***

Powerful Feature