

April, 2020



# Introduction to C# 8



**Author**

Bassam Alugili



C#Corner

This is for you, Mom, Khadja, and my dad Sabah and to my son Rami.

## Contents

Introduction to C# 8 .....	<b>Error! Bookmark not defined.</b>
Introduction .....	5
Nullable reference types.....	6
Enable Nullable Reference Types.....	6
Examples .....	7
Summary .....	8
Default Interface Methods.....	8
Modifiers in Interfaces .....	10
Diamond Problem .....	12
Using “This” Keyword.....	15
The ILogger Example .....	16
The Player Game .....	17
Limitations.....	18
Summary .....	20
Introduction to Asynchronous Streams .....	21
Pull Programming Model vs. Push Programming Model.....	22
Motivation and Background .....	23
Asynchronous pull with Client/Server .....	29
Async Streams.....	32
Syntax.....	33
Cancellation .....	34
Summary .....	35
Indices and Ranges.....	36
Index.....	36
Range .....	36
Summary .....	40
Pattern Matching .....	41
Switch expressions.....	43
Tuple Pattern .....	43
Positional Pattern.....	44
Property Pattern .....	44

Recursive Patterns .....	45
More about Recursive Patterns .....	46
Summary .....	52
Using declarations.....	53
Enhancement of interpolated verbatim strings.....	53
Null-coalescing assignment.....	53
Unmanaged constructed types.....	53
Static local functions .....	54
Readonly-Member .....	54
Stackalloc in nested expressions.....	55
Disposable ref structs.....	55



This icon shows the positive influence of the feature.



This icon shows the negative influence of the feature.



This icon shows the .NET community opinion about the feature.



This icon explains which technology is used behind the scenes by the feature.



This icon means that the feature has a bug, or it is still not done.

## Introduction

C# 8 is moving aggressively into the innovation world to keep C# growing. Because of this strategy, Microsoft left many developers in a confused state. Currently, the developers have a very controversial opinion on the C# 8 new features. In this book, you can have an overview of the new features of C# 8 language. I have described the important features separately and demonstrate them with examples; the other less essential features I have handled them briefly. Besides, I have written about the pros and cons of each important feature. It is also worth to mention that the main features concertation in C# 7 was to add safe, efficient code to C#, and in C# 8, we have seen more big language features and the preparing for the Records, which planned to be released with C# 9. After reading this book, you will have a greater understanding of C# 8, and hopefully, you will be better prepared to use C# 8, and for the new C# challenges you will meet in the future.

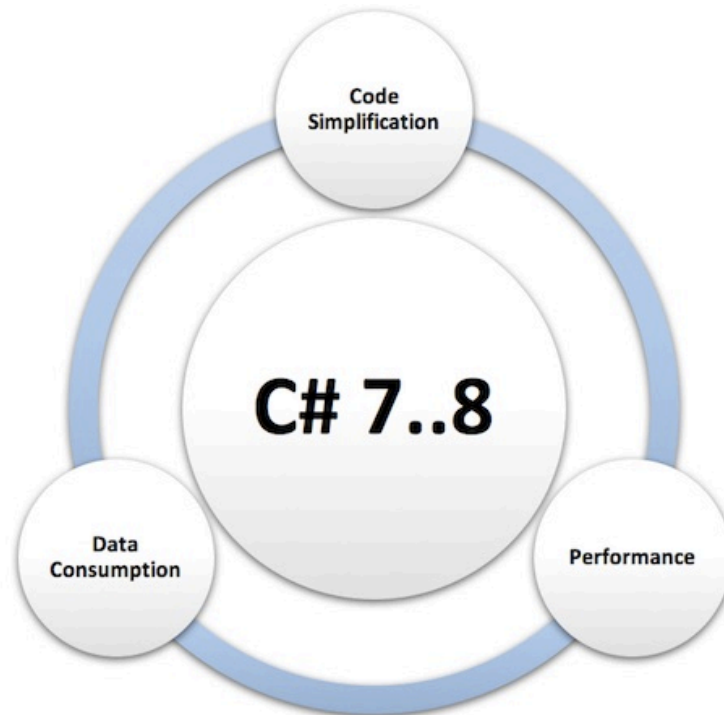


Figure -1- C# 7 focused on data consumption, code simplification, and performance and C# 8 more focus on data consumption

## Nullable reference types

C# 8.0 introduces nullable reference types. This feature is another way to specify that a given parameter, variable, or return value can be null or not. In C# 8, the compiler emits a warning or error if a variable that must not be null is assigned to null. Those warnings can help you to find and fix most of your null exception bugs before they blow up at runtime.

Enable Nullable Reference Types

*Enable Nullable annotations in the project*

You have to edit the project .csproj file and adding

```
<Nullable>enable</Nullable>
```



```
1 <Project Sdk="Microsoft.NET.Sdk">
2
3   <PropertyGroup>
4     <OutputType>Exe</OutputType>
5     <TargetFramework>netcoreapp3.1</TargetFramework>
6     <Nullable>enable</Nullable>
7   </PropertyGroup>
8
9 </Project>
10
```

Figure -2- shows you a demo project where the Nullable Reference Types feature is enabled

*Enable Nullable annotations in a file or part of the code*

You can put **#nullable enable** directive where you want to enable the functionality and **#nullable disable** directive, where you want to disable the functionality.

If you put **#nullable enable** on the file head, that should allow the nullability check for the whole file, as shown below in the image.

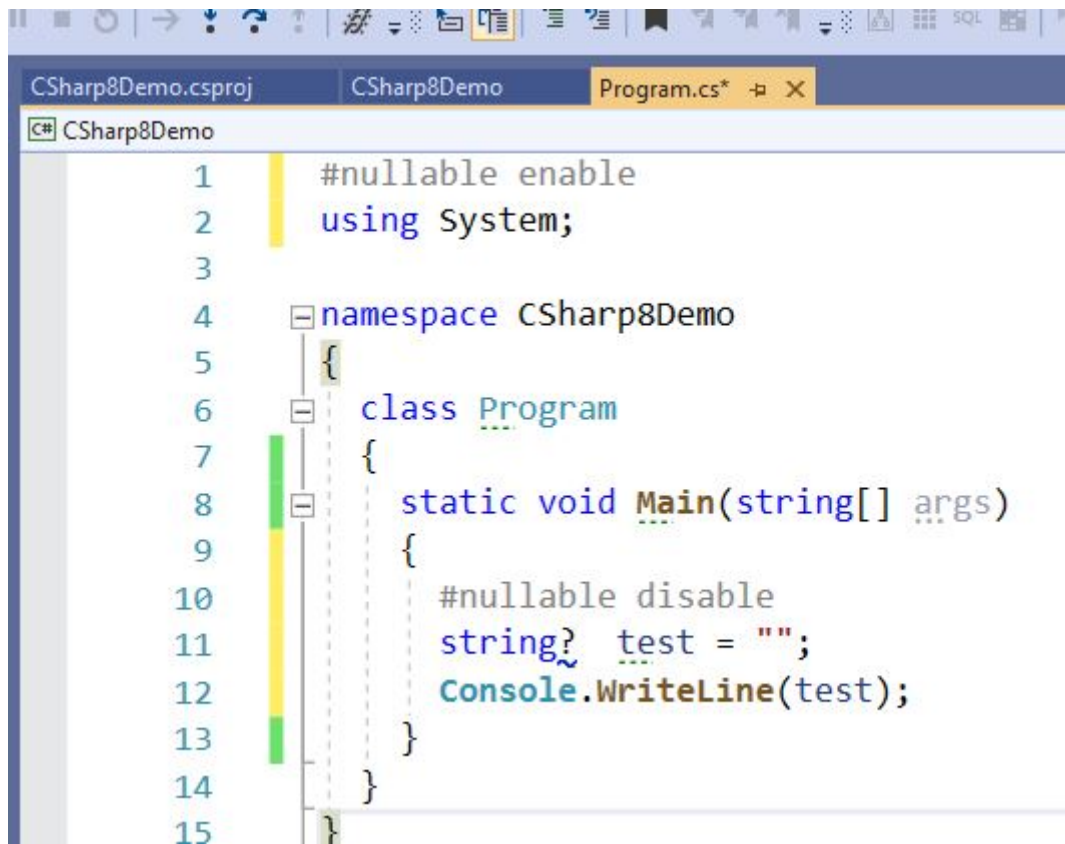


Figure -3- Enabling Nullable Reference Types partially

Finally, you can restore the default setting as below:

```
#nullable restore
```

## Examples

### Example 1

```
string ? nullableString = null; // Is Ok, nullableString it can be null and
it is null.

Console.WriteLine(nullableString.Length); // WARNING: nullableString is null!
Take care!
```



## Example 2

```
#nullable enable
class Person
{
    public string Name { get; set; } // Warning normalString is null!
    public string? NullableName { get; set; }

    // Enable the below code then the warning above will be disappeared
    //public Person(string name)
    //{
    //    Name = name;
    //}
}
```

The first property *Name* is a reference type, and it is null for this reason the compiler warning you.

The Second property is *NullableName* is a nullable reference type that why the compiler is not warning because the *NullableName* can be null, you have defined as nullable.

## Summary



Nullable Reference Types help you to eliminate the *NullReferenceException* (the Billion Dollar Mistake, ALGOL60, Tony). Moreover, it helps you to solve the problems in code like a pyramid of doom.



However, in complicated scenarios, this feature may bring some confusion in regards to reference types and using '?' character. For more information, please read Jon Skeet blog's at:

<https://codeblog.jonskeet.uk/2019/02/10/nullableattribute-and-c-8/>

## Default Interface Methods

Default methods can be provided to an interface without affecting implementing classes as it includes an implementation. Moreover, an implementing class can override the default implementation provided by the interface.


The main benefit that default methods allow you to add new functionality to the interfaces of your libraries and ensure the backward compatibility with code written for older versions of those interfaces.

The C# syntax for an interface in the .NET compiler is extended to accept the new keywords in the interfaces, which are listed below. For example, you can write a private method in the interface, and the code still compiles and work.

Allowed in the interface:

- A body for a method or indexer, property, or event accessor
- Private, protected, internal, public, virtual, abstract, sealed, static, extern
- Static fields
- Static methods, properties, indexers, and events.
- Explicit access modifiers with default access is public

Not allowed:

- Instance state, instance fields, instance auto-properties
-  override keyword is currently not possible, but this might be changed in C# 9

Example

Consider this simple example that illustrates how this feature works.

```
interface IDefaultInterfaceMethod
{
    public void DefaultMethod()
    {
        Console.WriteLine("I am a default method in the interface!");
    }
}
class AnyClass : IDefaultInterfaceMethod
{
}
class Program
{
    static void Main()
    {
        IDefaultInterfaceMethod anyClass = new AnyClass();
        anyClass.DefaultMethod();
    }
}
```

Output:

> I am a default method in the interface!

If you look at the code above, you can see that the interface has a default method, and the implementer class contains neither any knowledge of this default method nor the implementation of that interface method.

Change *IDefaultInterfaceMethod* to *AnyClass*, as shown below:

```
AnyClass anyClass = new AnyClass();
anyClass.DefaultMethod();
```

The above code will produce the compile-time error: *AnyClass* does not contain any member of the Default Interface Method.

That is proof that the inherited class does not know anything about the default method.

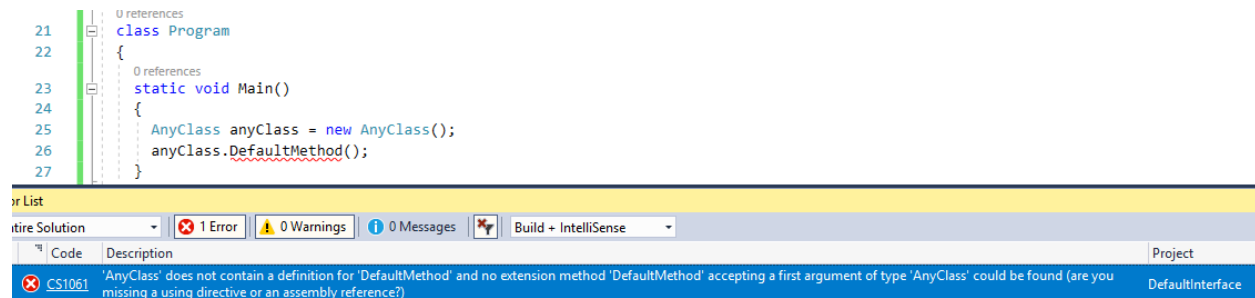


Figure -4- Compile-Error in the Main, because the *AnyClass* does not have any information about the Default Interface Method

### Modifiers in Interfaces

The C# syntax for an interface is extended to accept the following keywords: **protected**, **internal**, **public**, and **virtual**. By default, the default interface methods are **virtual** unless the sealed or private modifier is used. Similarly, **abstract** is the default on interface members without bodies.

### Example

```
// ----- Virtual and Abstract-----

interface IDefaultInterfaceMethod
{
    // By default, this method is virtual. The "virtual" keyword is unnecessary!
    virtual void DefaultMethod()
    {
        Console.WriteLine("I am a default method in the interface!");
    }

    // By default, this method will be abstract, and the abstract keyword can
    be here used
    abstract void Sum();
}

interface IOverrideDefaultInterfaceMethod : IDefaultInterfaceMethod
{
    void IDefaultInterfaceMethod.DefaultMethod()
    {
        Console.WriteLine("I am an overridden default method!");
    }
}

class AnyClass : IDefaultInterfaceMethod, IOverrideDefaultInterfaceMethod
{
    public void Sum()
    {
    }
}
```

```

    }
}

class Program
{
    static void Main()
    {
        IDefaultInterfaceMethod anyClass = new AnyClass();
        anyClass.DefaultMethod();

        IOverrideDefaultInterfaceMethod anyClassOverridden = new AnyClass();
        anyClassOverridden.DefaultMethod();
    }
}

```

Output:

> I am an overridden default method!

> I am an overridden default method!



Console output for the next version C# 9:

> I am a default method in the interface!

> I am an overridden default method!

For more information, you can read more here:

*"Collision of lookup rules and decisions for base()"*

<https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-02-27.md#collision-of-lookup-rules-and-decisions-for-base>

The keywords **virtual** and **abstract** are redundant, and they can be removed from the interface; however, removing them does not have any effect on the compiled code.

*Modifier Override example*

The explicit access modifiers in the overridden method are not permitted.

Example

```

interface IOverrideDefaultInterfaceMethod : IDefaultInterfaceMethod
{
    public void IDefaultInterfaceMethod.DefaultMethod()
    {
        Console.WriteLine("I am an overridden default method");
    }
}

```

The above code produces a compile-time error: The modifier **public** is not valid for this item.

```

44 2 references
45 interface IOVERRIDEDefaultInterfaceMethod : IDefaultInterfaceMethod
46 {
47     3 references
48     public void IDefaultInterfaceMethod.DefaultMethod()
49     {
50         Console.WriteLine("I am an overridden default method");
51     }
52 }
53 2 references

```

Output

DefaultInterface

1 Error 0 Warnings 0 Messages

Build + IntelliSense

CS0106 The modifier 'public' is not valid for this item

Figure -5- shows the overridden explicit modifier error

### Diamond Problem

An ambiguity error that can arise because of allowing multiple inheritance. It is a big problem for languages (like C++) that allow multiple inheritance of state. In C#, however, multiple inheritance is unallowed for classes, but rather only for interfaces in a limited way, so that does not contain state.

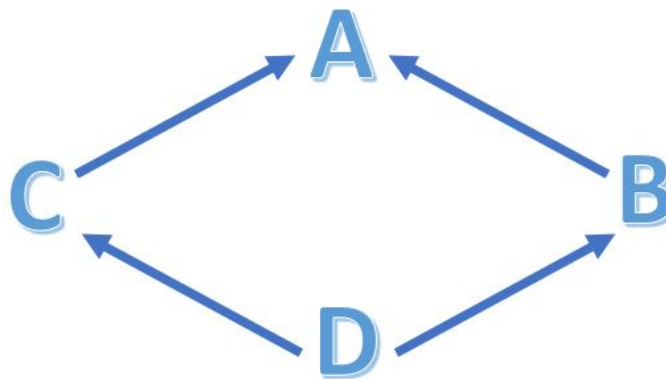


Figure -6- Diamond Problem dependencies

Consider the following situation:

```
// -----Diamond inheritance problem -----  
interface A  
{  
    void m();  
}  
  
interface B : A  
{  
    void A.m()  
    {  
        System.Console.WriteLine("interface B");  
    }  
}  
  
interface C : A  
{  
    void A.m()  
    {  
        System.Console.WriteLine("interface C");  
    }  
}  
  
class D : B, C  
{  
    static void Main()  
    {  
        C c = new D();  
        c.m();  
    }  
}
```

The above code will produce the compile-time error, which shown below in Figure -7-:

```

94 4 references
95 interface A
96 {
97     void m() { System.Console.WriteLine("interface A"); }
98 }
99 1 reference
100 interface B : A
101 {
102     2 references
103     void A.m() { System.Console.WriteLine("interface B"); }
104 }
105 2 references
106 interface C : A
107 {
108     2 references
109     void A.m() { System.Console.WriteLine("interface C"); }
110 }
111 1 reference
112 class D : B, C
113 {
114     0 references
115     static void Main()
116     {
117         C c = new D();
118         c.m();
119     }
120 }
121 //public interface IPlayer
122 //{
123 //    int Attack(int amount);
124 }

```

CS8505 Interface member 'A.m()' does not have a most specific implementation. Neither 'B.A.m()' nor 'C.A.m()' are most specific.

Figure -7- Diamond Problem error message

The .NET development team has decided to solve the Diamond Problem by taking the most specific override at runtime.

### "Diamonds with Classes

*A class implementation of an interface member should always win over a default implementation in an interface, even if it is inherited from a base class. Default implementations are always a fallback only for when the class does not have any implementation of that member at all."*

If you want to know more about this problem, you can find more information in the proposal: *"Default Interface Methods and C# Language Design Notes for Apr 19, 2017"*.

Back to our example, the problem is that the most specific override cannot be inferred from the compiler. However, you can add the method "m" in the class "D" as shown below, and now the compiler uses the class implementation to solve the diamond problem.

```

class D : B, C
{
    // Now the compiler uses the most specific override, which is defined in the
    // class "D".
    void A.m()
    {
        System.Console.WriteLine("I am in class D");
    }
    static void Main()
    {
        A a = new D();
        a.m();
    }
}

```

```
}
```

Output:

> I am in class D

Using "This" Keyword

The code below is an example that shows how to use 'this' keyword in the interfaces.

```
public interface IDefaultInterfaceWithThis
{
    internal int this[int x]
    {
        get
        {
            return x;
        }
        set
        {
            System.Console.WriteLine("SetX");
        }
    }

    void CallDefaultThis(int x)
    {
        this[0] = x;
    }
}

class DefaultMethodWithThis : IDefaultInterfaceWithThis
{
}
```

Using the code:

```
IDefaultInterfaceWithThis defaultMethodWithThis = new
DefaultMethodWithThis();

Console.WriteLine(defaultMethodWithThis[0]);

defaultMethodWithThis.CallDefaultThis(0);
```

Output:

0  
SetX



## The ILogger Example

The logger interface is an good example to explain the Default methods technique. I have defined below one abstract method named *WriteCore*. All other methods have a default implementation.

*ConsoleLogger* and *TraceLogger* classes are implementing the interface. If you look at the code below, you can see that the code is compact. In the past, it was mandatory to implement all the methods in a class that implements an interface unless that class is declared as an abstract, and this might make your code DRY.

```
enum LogLevel
{
    Information,
    Warning,
    Error
}

interface ILogger
{
    void WriteCore(LogLevel level, string message);

    void WriteInformation(string message)
    {
        WriteCore(LogLevel.Information, message);
    }

    void WriteWarning(string message)
    {
        WriteCore(LogLevel.Warning, message);
    }
    void WriteError(string message)
    {
        WriteCore(LogLevel.Error, message);
    }
}

class ConsoleLogger : ILogger
{
    public void WriteCore(LogLevel level, string message)
    {
        Console.WriteLine($"{level}: {message}");
    }
}

class TraceLogger : ILogger
{
    public void WriteCore(LogLevel level, string message)
    {
        switch (level)
        {
            case LogLevel.Information:
                Trace.TraceInformation(message);
                break;

            case LogLevel.Warning:
```

```
        Trace.TraceWarning(message);  
        break;  
  
        case LogLevel.Error:  
            Trace.TraceError(message);  
            break;  
    }  
}  
}
```

Using the code:

```
ILogger consoleLogger = new ConsoleLogger();  
consoleLogger.LogWarning("Cool no code duplication!");  
// Output: Warning: Cool no Code duplication!  
  
ILogger traceLogger = new TraceLogger();  
consoleLogger.WriteInformation("Cool no code duplication!");  
// Output: Information: Cool no Code duplication!
```

## The Player Game

A game, which contains different types of players. The power player causes the most attack damage; on the other side, the limited player can cause less damage.

```
public interface IPlayer  
{  
    int Attack(int amount);  
}  
  
public interface IPowerPlayer: IPlayer  
{  
    int IPlayer.Attack(int amount)  
    {  
        return amount + 50;  
    }  
}  
  
public interface ILimitedPlayer: IPlayer  
{  
    int IPlayer.Attack(int amount)  
    {  
        return amount + 10;  
    }  
}
```

```
public class WeakPlayer : ILimitedPlayer
{
}

public class StrongPlayer : IPowerPlayer
{
}
```

Using the code:

```
IPlayer powerPlayer = new StrongPlayer();
Console.WriteLine(powerPlayer.Attack(5)); // Output 55

IPlayer limitedPlayer = new WeakPlayer ();
Console.WriteLine(limitedPlayer.Attack(5)); // Output 15
```

As you see in the code example above, the default implementation is in the *IPowerPlayer* interface and the *ILimitedPlayer* interface. The limited player is doing less damage. If we define a new class, for example, *SuperDuperPlayer*, which inherited from the class *StrongPlayer*, then the new class automatically receives the default power attack behavior of the interface, as shown in the example below.

```
public class SuperDuperPlayer: StrongPlayer
{
}

IPlayer superDuperPlayer = new SuperDuperPlayer();
Console.WriteLine(superDuperPlayer.Attack(5)); // Output 55
```

### Limitations

There are some limitations and considerations that you first need to understand when using modifier keywords in the interfaces. In most cases, the compiler protects you against the wrong usage of this feature by detecting a lot of common errors, such as those listed below.

Consider the following code:

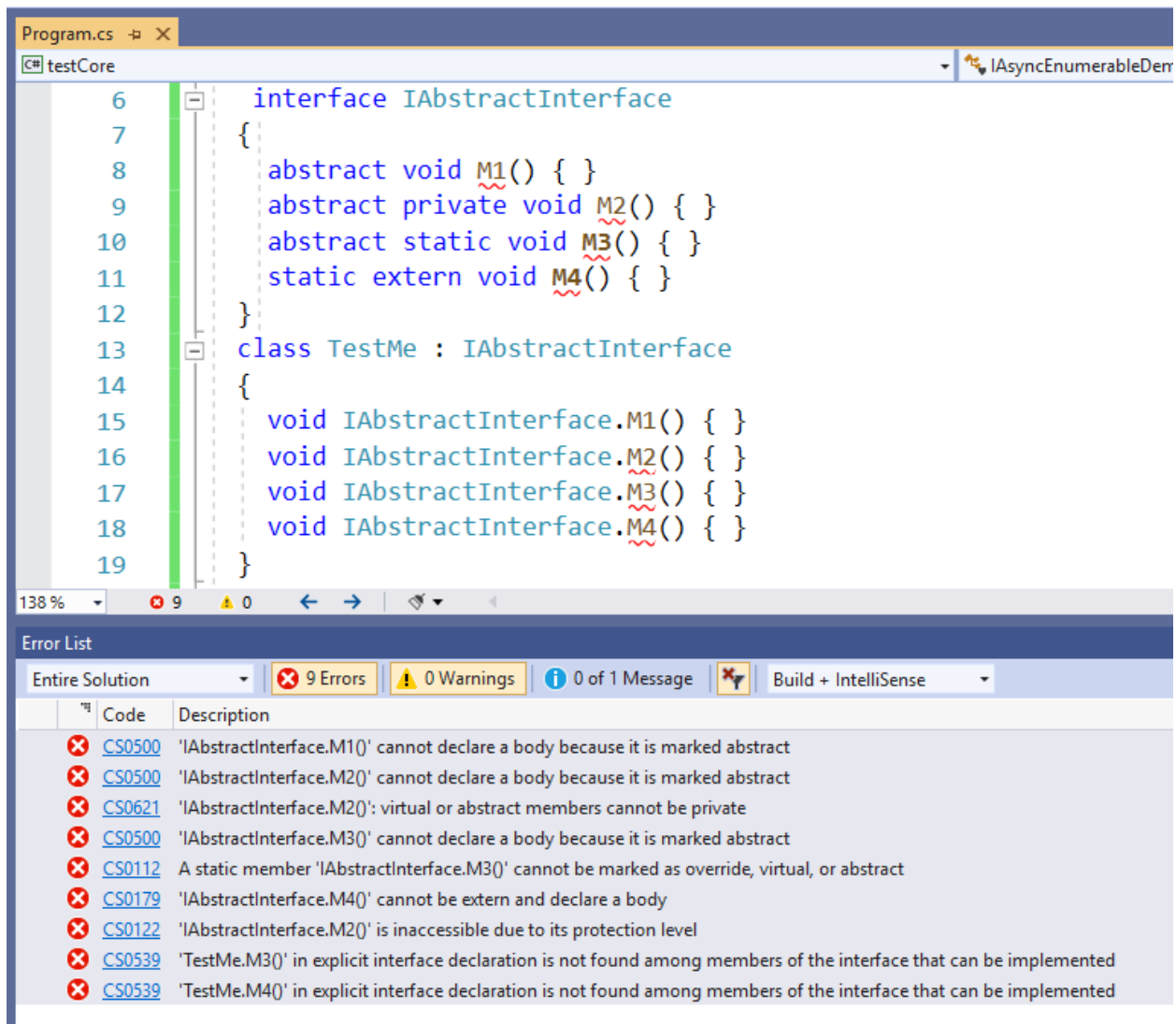
```
interface IAbstractInterface
{
    abstract void M1() {}
    abstract private void M2() {}
    abstract static void M3() {}
    static extern void M4() {}
}
```

```
}  
  
class TestMe : IAbstractInterface  
{  
    void IAbstractInterface.M1() {}  
    void IAbstractInterface.M2() {}  
    void IAbstractInterface.M3() {}  
    void IAbstractInterface.M4() {}  
}
```

The above code produces the below-listed compilation errors:

```
Error    CS0500  'IAbstractInterface.M1()' cannot declare a body because it is  
marked abstract...  
Error    CS0500  'IAbstractInterface.M2()' cannot declare a body because it is  
marked abstract...  
Error    CS0621  'IAbstractInterface.M2()': virtual or abstract members cannot  
be private...  
Error    CS0500  'IAbstractInterface.M3()' cannot declare a body because it is  
marked abstract...  
Error    CS0112  A static member 'IAbstractInterface.M3()' cannot be marked as  
override, virtual, or abstract...  
Error    CS0179  'IAbstractInterface.M4()' cannot be extern and declare a  
body...  
Error    CS0122  'IAbstractInterface.M2()' is inaccessible due to its  
protection level...  
Error    CS0539  'TestMe.M3()' in explicit interface declaration is not found  
among members of the interface that can be implemented...  
Error    CS0539  'TestMe.M4()' in explicit interface declaration is not found  
among members of the interface that can be implemented...
```

The error **CS0500**, means the default method *IAbstractInterface.M3()* cannot be abstract, and at the same time, it has a body. The error **CS0621**, says the method cannot be private and, at the same time, abstract.



The screenshot shows a C# program in Visual Studio. The code defines an interface `IAbstractInterface` with four methods: `M1()`, `M2()`, `M3()`, and `M4()`. A class `TestMe` implements these methods. The error list at the bottom shows the following errors:

Code	Description
CS0500	'IAbstractInterface.M1()' cannot declare a body because it is marked abstract
CS0500	'IAbstractInterface.M2()' cannot declare a body because it is marked abstract
CS0621	'IAbstractInterface.M2()': virtual or abstract members cannot be private
CS0500	'IAbstractInterface.M3()' cannot declare a body because it is marked abstract
CS0112	A static member 'IAbstractInterface.M3()' cannot be marked as override, virtual, or abstract
CS0179	'IAbstractInterface.M4()' cannot be extern and declare a body
CS0122	'IAbstractInterface.M2()' is inaccessible due to its protection level
CS0539	'TestMe.M3()' in explicit interface declaration is not found among members of the interface that can be implemented
CS0539	'TestMe.M4()' in explicit interface declaration is not found among members of the interface that can be implemented

Figure -8- Compilation Errors for the wrong modifiers in the Default Interface Methods

### Summary

This language feature allows you to add members to the interfaces and provide an implementation for those members. It enables API authors to add methods to an interface in later versions without breaking the source or binary compatibility with existing implementations of that interface.



You can add new functionality to the interface without breaking the compatibility with the older versions of those interfaces.



Please use this feature carefully. Otherwise, it can easily lead to violating the single responsibility principles.



It is a very arguable feature, and it left many open discussions among the .NET community. I have discussed this theme in-depth on Reddit:  
[https://www.reddit.com/r/csharp/comments/8xg2kx/default\\_interface\\_methods\\_in\\_c\\_8/](https://www.reddit.com/r/csharp/comments/8xg2kx/default_interface_methods_in_c_8/)



It is based on the Trait technique. Traits—Programming is a proven and powerful technique in OOP.

## Introduction to Asynchronous Streams

Async/Await is introduced with C# 5 to improve user interface responsiveness and web access to the resources. Async/Await, also known as Async Methods, has helped the developers to execute asynchronous operations that do not block threads and return one scalar result.

After many tries by Microsoft to simplify asynchronous operations, the *async/await* pattern has gained a good acceptance among developers thanks to its easy-to-understand approach.

One of the significant limitations of the existing Async Methods is the requirement that it must have a scalar return result (one value). Let us consider the following method **async Task<int> DoAnythingAsync()**. The result of *DoAnythingAsync* is an integer (One value).

Because of this limitation, you cannot use the **yield** inside Async Methods, or you cannot return an async enumeration, for example, *public async Task<IEnumerable<T>> DoAnythingAsync <T>()*.

If it were to be possible to combine async/awaiting feature with a yielding operator that can lead to a powerful programming model, which is known as **asynchronous data pull** or also known as **pull-based enumeration** and in F# this is called **async sequence**.

Async Streams in C# 8 removes the scalar result limitation and allows the async method to return multiple results.

This change makes the async pattern more flexible so that you can load data from the database asynchronously and show data that are partially loaded immediately, or you can download data from in an asynchronous sequence that returns the data in chunks when they become available.

A simple definition: Async Streams, allows you to use **async** keyword to iterate asynchronously over collections, for example:

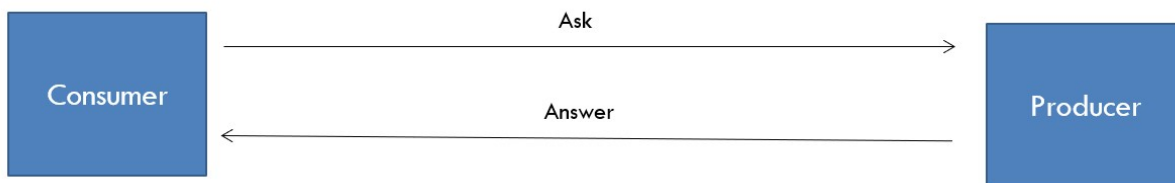
```
await foreach (var streamChunk in asyncStreams)
{
    Console.WriteLine($"Received data count = {streamChunk.Count}");
}
```

Reactive Extensions (Rx) is another approach to solve asynchronous programming problems. Rx is getting broader acceptance among developers. Many other programming languages like Java and JavaScript have implemented this technique (RxJava, RxJS). Rx is based on the **Push Programming Model** (Tell, Don't Ask system) and is also known as **reactive programming**. Reactive programming is a particular type of event driving programming, but it handles data rather than notifications.

Usually, in the Push Programming Model, you do not have to control the Publisher. The data are asynchronously pushed into a queue, and the Consumers consume the data when the data arrive. Unlike the Rx, the Async Streams can be called on-demand to generate multiple values until the end of the enumeration is reached.

Below you can find a comparison between the pull-based model and the push-based model, and you can see which scenario to which technique is a better fit from the other one. I have used many examples, and I have added a demo to show you the whole concept and the benefits. Finally, I have explained the Async Streams feature and demonstrate it.

#### Pull Programming Model vs. Push Programming Model



Pull-based Programming Model

**Good for Faster Producer and Slow Consumer scenario**



Push-based Programming Model

**Good for Slower Producer and Fast Consumer scenario**

Figure -9- Pull Programming Model vs. Push Programming Model

I have used the famous food example Producer/Consumer, but in our scenarios, the Producer will not generate food; instead of that, he generates data, and the Consumer consumes the generated data, as shown in Figure -9-. The Pull Model is easy to understand. The Consumer is asking and pulling the data from the Producer. The other approach is the Push Model. The Producer publishes the data in a queue, and the Consumer must subscribe to receive the pushed data.

In the *faster Producer and slow Consumer* use case, the pull model is suitable because the Consumer pulls its required data from the Producer to avoid the overflow issues. In the *slower Producer and faster Consumer* use case, the Push Model is suitable because the Producer pushes more data from the Producer to Consumer to avoid causing the Consumer unnecessary waiting time.

Rx and Akka Streams (**Stream Programming Model**) use the Backpressure technique, which is a flow control mechanism. It uses the Pull or Push Models dynamically to solve the Producer/Consumer problems mentioned above.

I have used in my example below a slow Consumer to pull the asynchronous sequence of data from a faster Producer. Once the Consumer processes an element, the Consumer asks the Producer again for the next element, and so on until the end of the sequence is reached.

### Motivation and Background

To understand the problem, *why we need Async Streams?* let's consider the following code.

I have extended the Console WriteLine/WriteLineAsync method so that it adds the current time and the current thread Id to the written message as shown below:

```
public static class ConsoleExt
{
    // Writes the message with timestamp and the thread Id.
    public static void WriteLine(object message)
    {
        Console.WriteLine(
            $"(Time: {DateTime.Now.ToShortTimeString()},
            Thread {Thread.CurrentThread.ManagedThreadId}):
{message} ");
    }

    // Writes the message with timestamp and the thread Id for the async
    methods.
    public static async void WriteLineAsync(object message)
    {
        await Task.Run(() => Console.WriteLine(
            $"(Time: {DateTime.Now.ToShortTimeString()},
            Thread {Thread.CurrentThread.ManagedThreadId}):
{message} ")
        );
    }
}
```

```
// Loops and sums the provided argument (count)
static int SumFromOneToCount(int count)
```

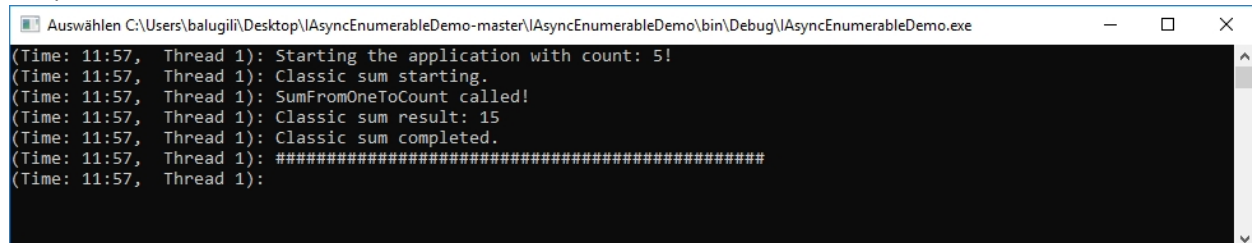


```
{
    ConsoleExt.WriteLine("SumFromOneToCount called!");
    var sum = 0;
    for (var i = 0; i <= count; i++)
    {
        sum = sum + i;
    }
    return sum;
}
```

Method call:

```
const int count = 5;
ConsoleExt.WriteLine($"Starting the application with count: {count}!");
ConsoleExt.WriteLine("Classic sum starting.");
ConsoleExt.WriteLine($"Classic sum result: {SumFromOneToCount(count)}");
ConsoleExt.WriteLine("Classic sum completed.");
ConsoleExt.WriteLine("#####");
ConsoleExt.WriteLine(Environment.NewLine);
```

Output:



We can make this method lazy by using the **yield** operator, as shown below.

```
static IEnumerable<int> SumFromOneToCountYield(int count)
{
    ConsoleExt.WriteLine("SumFromOneToCountYield called!");
    var sum = 0;
    for (var i = 0; i <= count; i++)
    {
        sum = sum + i;
        yield return sum;
    }
}
```

Calling the method:

```
const int count = 5;
ConsoleExt.WriteLine("Sum with yield starting.");
foreach (var i in SumFromOneToCountYield(count))
{
    ConsoleExt.WriteLine($"Yield sum: {i}");
}
```

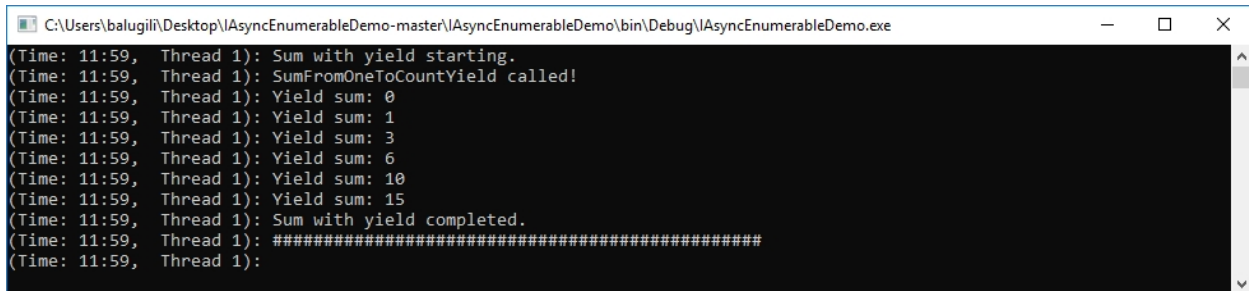
```

ConsoleExt.WriteLine("Sum with yield completed.");

ConsoleExt.WriteLine("#####");
ConsoleExt.WriteLine(Environment.NewLine);

```

Output:



```

C:\Users\balugili\Desktop\AsyncEnumerableDemo-master\AsyncEnumerableDemo\bin\Debug\AsyncEnumerableDemo.exe
(Time: 11:59, Thread 1): Sum with yield starting.
(Time: 11:59, Thread 1): SumFromOneToCountYield called!
(Time: 11:59, Thread 1): Yield sum: 0
(Time: 11:59, Thread 1): Yield sum: 1
(Time: 11:59, Thread 1): Yield sum: 3
(Time: 11:59, Thread 1): Yield sum: 6
(Time: 11:59, Thread 1): Yield sum: 10
(Time: 11:59, Thread 1): Yield sum: 15
(Time: 11:59, Thread 1): Sum with yield completed.
(Time: 11:59, Thread 1): #####
(Time: 11:59, Thread 1):

```

As you see above in the output window, the result is returned in parts and not as one value. The above shown accumulated results known as lazy enumeration. However, we still have a problem; the sum methods are blocking the code. If you look at the threads, then you can see that everything is running in the main thread.

Now let's use the magic word **async** and apply it to the first method *SumFromOneToCount* (without *yield*).

```

static async Task<int> SumFromOneToCountAsync(int count)
{
    ConsoleExt.WriteLine("SumFromOneToCountAsync called!");
    var result = await Task.Run(() =>
    {
        var sum = 0;
        for (var i = 0; i <= count; i++)
        {
            sum = sum + i;
        }
        return sum;
    });
    return result;
}

```

Calling the method:

```

const int count = 5;
ConsoleExt.WriteLine("async example starting.");
// Sum runs asynchronously! Not enough. We need sum to be async with lazy
behavior.
var result = await SumFromOneToCountAsync(count);
ConsoleExt.WriteLine("async Result: " + result);
ConsoleExt.WriteLine("async completed.");
ConsoleExt.WriteLine("#####");
ConsoleExt.WriteLine(Environment.NewLine);

```

Output:

```

C:\Users\balugili\Desktop\AsyncEnumerableDemo-master\AsyncEnumerableDemo\bin\Debug\AsyncEnumerableDemo.exe
(Time: 11:59, Thread 1): async example starting.
(Time: 11:59, Thread 1): SumFromOneToCountAsync called!
(Time: 11:59, Thread 3): async Result: 15
(Time: 11:59, Thread 3): async completed.
(Time: 11:59, Thread 3): #####
(Time: 11:59, Thread 3):

```

Impressive, we can see that computing is running in another thread, but we still have a problem with the result. The result is returned as one value!

Imagine that we can combine the lazy enumerations (**yield return**) with the Async Methods in an imperative style. This combination is known as Async Streams. It is the new C# 8 feature. The new feature gives us an excellent technique to solve the Pull Programming Model problems like download data from a website or to read in records from a file or a database in a modern way.

Let's try to do that with C# 7. I have added the **async** keyword to the method *SumFromOneToCountYield* as follows: *static async Task<IEnumerable<int>> SumFromOneToCountYield(int count)*

```

22
23 static async Task<IEnumerable<int>> SumFromOneToCountYield(int count)
24 {
25     ConsoleExt.WriteLine(message: "SumFromOneToCountYield called!");
26
27     var sum = 0;
28     for (var i = 0; i <= count; i++)
29     {
30         sum = sum + i;
31
32         yield return sum;
33     }
34 }
35

```

Errors List

2 Errors 0 of 1 Warning 0 of 8 Messages Build + IntelliSense

Code	Description
CS1624	The body of 'TaskExample.SumFromOneToCountYield(int)' cannot be an iterator block because 'Task<IEnumerable<int>>' is not an iterator interface type
CS8320	Feature 'async streams' is not available in C# 7.2. Please use language version 8.0 or greater.

Figure -10- The error occurs when **yield** with the **async** keyword is combined in C# 7

After Adding *async* to the *SumFromOneToCountYield* method, we get errors, as shown in Figure -10-.

Let's try something else. We can put **IEnumerable** in the **Task** and remove the **yield** keyword, as shown below:

```

static async Task<IEnumerable<int>> SumFromOneToCountTaskIEnumerable(int
count)

```

```
{
    ConsoleExt.WriteLine("SumFromOneToCountAsyncIEnumerable called!");
    var collection = new Collection<int>();
    var result = await Task.Run(() =>
    {
        var sum = 0;

        for (var i = 0; i <= count; i++)
        {
            sum = sum + i;
            collection.Add(sum);
        }
        return collection;
    });

    return result;
}
```

Calling the method:

```
const int count = 5;
ConsoleExt.WriteLine("SumFromOneToCountAsyncIEnumerable started!");

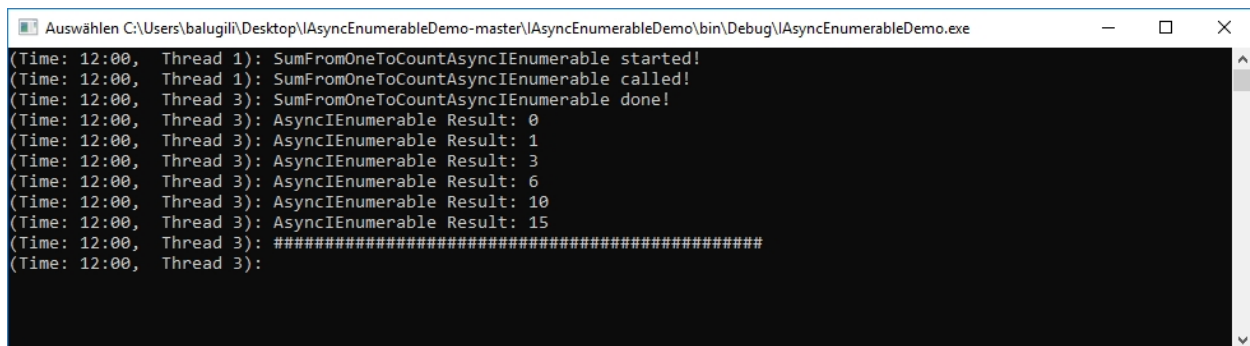
var scs = await SumFromOneToCountTaskIEnumerable(count);

ConsoleExt.WriteLine("SumFromOneToCountAsyncIEnumerable done!");

foreach (var sc in scs)
{
    // This is not what we need; we become the result in one block!
    ConsoleExt.WriteLine($"AsyncIEnumerable Result: {sc}");
}

ConsoleExt.WriteLine("#####");
ConsoleExt.WriteLine(Environment.NewLine);
```

Output:



```
(Time: 12:00, Thread 1): SumFromOneToCountAsyncIEnumerable started!
(Time: 12:00, Thread 1): SumFromOneToCountAsyncIEnumerable called!
(Time: 12:00, Thread 3): SumFromOneToCountAsyncIEnumerable done!
(Time: 12:00, Thread 3): AsyncIEnumerable Result: 0
(Time: 12:00, Thread 3): AsyncIEnumerable Result: 1
(Time: 12:00, Thread 3): AsyncIEnumerable Result: 3
(Time: 12:00, Thread 3): AsyncIEnumerable Result: 6
(Time: 12:00, Thread 3): AsyncIEnumerable Result: 10
(Time: 12:00, Thread 3): AsyncIEnumerable Result: 15
(Time: 12:00, Thread 3): #####
(Time: 12:00, Thread 3):
```

As you see in the threads, we calculate everything asynchronously, but we still have a problem. The results (all results are accumulated in the collection) return as one block. That is not our desired lazy behavior. If you remember, our goal is combining lazy behavior with an asynchronous computing style.

To achieve the desired behavior, you need to use an external library like Ix (part of the Rx), or you have to use the new C# 8 feature Async Streams.

Let's get back to our code example. I have used the Async Streams feature from C# 8.

```
static async Task ConsumeAsyncSumSequeunc(IAsyncEnumerable<int> sequence)
{
    ConsoleExt.WriteLineAsync("ConsumeAsyncSumSequeunc Called");

    await foreach (var value in sequence)
    {
        ConsoleExt.WriteLineAsync($"Consuming the value: {value}");

        // Simulate some delay!
        Task.Delay(TimeSpan.FromSeconds(1)).Wait();
    }
}

static async IAsyncEnumerable<int> ProduceAsyncSumSequeunc(int count)
{
    ConsoleExt.WriteLineAsync("ProduceAsyncSumSequeunc Called");
    var sum = 0;

    for (var i = 0; i <= count; i++)
    {
        sum = sum + i;

        //ConsoleExt.WriteLineAsync("Producing the value:" + i);

        // Simulate some delay!
        await Task.Delay(TimeSpan.FromSeconds(0.5));

        yield return sum;
    }
}
```

Calling the method:

```
const int count = 5;
ConsoleExt.WriteLine("Starting Async Streams Demo!");

// Start a new task to produce an async sequence of data!
IAsyncEnumerable<int> pullBasedAsyncSequence =
    ProduceAsyncSumSequeunc(count).ToAsyncEnumerable();

ConsoleExt.WriteLineAsync("X#X#X#X Doing some other work X#X#X#X");

// Start another task to consume the async data sequence!
```

```
C:\Users\balugili\Desktop\AsyncEnumerableDemo-master\AsyncEnumerableDemo\bin\Debug\AsyncEnumerableDemo.exe
```

```
(Time: 12:01, Thread 1): Starting Async Streams Demo!  
(Time: 12:01, Thread 3): X#X#X#X#X#X#X#X#X#X# Doing some other work X#X#X#X#X#X#X#X#X#X#  
(Time: 12:01, Thread 5): ConsumeAsyncSumSequeunc Called  
(Time: 12:01, Thread 3): ProduceAsyncSumSequeunc Called  
(Time: 12:01, Thread 5): Consuming the value: 0  
(Time: 12:01, Thread 3): Consuming the value: 1  
(Time: 12:01, Thread 3): Consuming the value: 3  
(Time: 12:01, Thread 3): Consuming the value: 6  
(Time: 12:01, Thread 3): Consuming the value: 10  
(Time: 12:01, Thread 3): Consuming the value: 15  
(Time: 12:01, Thread 4): Async Streams Demo Done!
```

Here is the Source Code:

## Asynchronous pull with Client/Server

### Client/Server synchronous call

The Client sends a request to the server, and the client must wait (the Client is blocked) until the response is coming from the server, as shown below in Figure -11-.

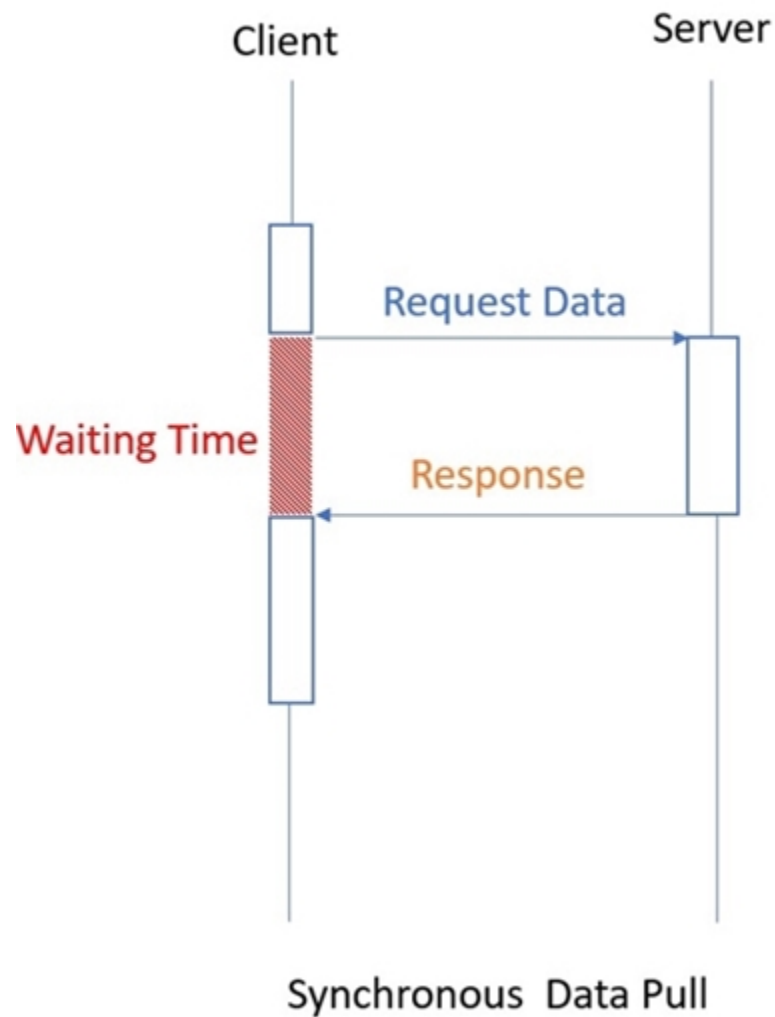


Figure -11- Synchronous Data Pull, the Client, wait until the request is finished

#### *Asynchronous data pull*

In this case, the Client asks for the data and continue further by doing something else. Once the data have arrived, then the client continues with his work.

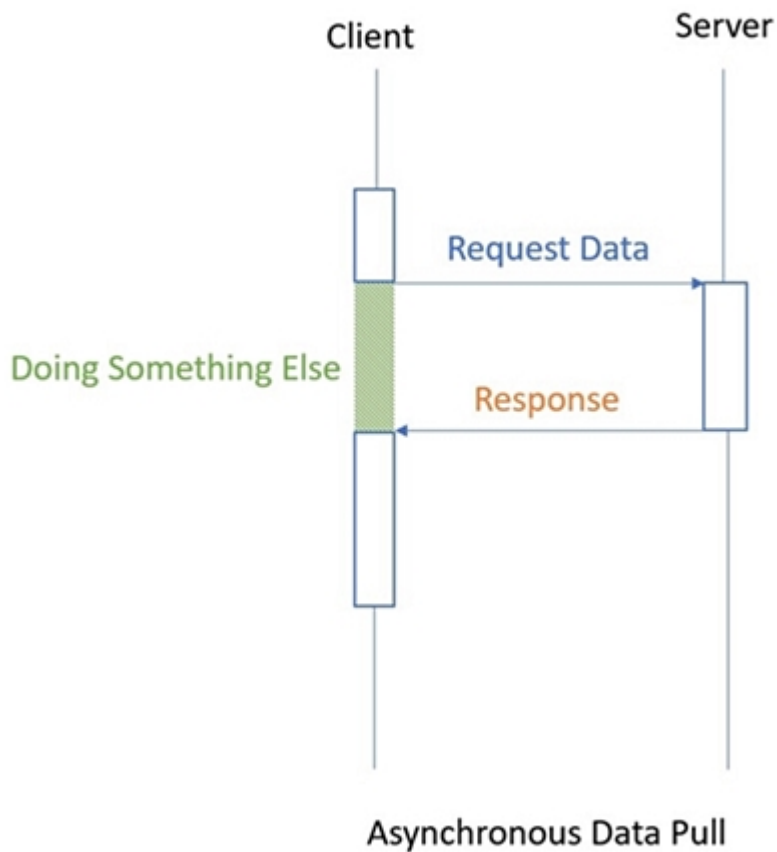


Figure -12- Asynchronous Data Pull, the client, can be doing something else while the data are requested

#### *Asynchronous sequence data pull*

In this case, the Client asks for a chunk of data and continue further by doing something else. Once the data chunk arrives. The Client process the received data and asking for the next data chunk and so on until the last data chunk. This scenario is precisely where the Async Streams idea is coming from. Figure - 12- shows that the Client can do something else or process the data chunk when any data is received.



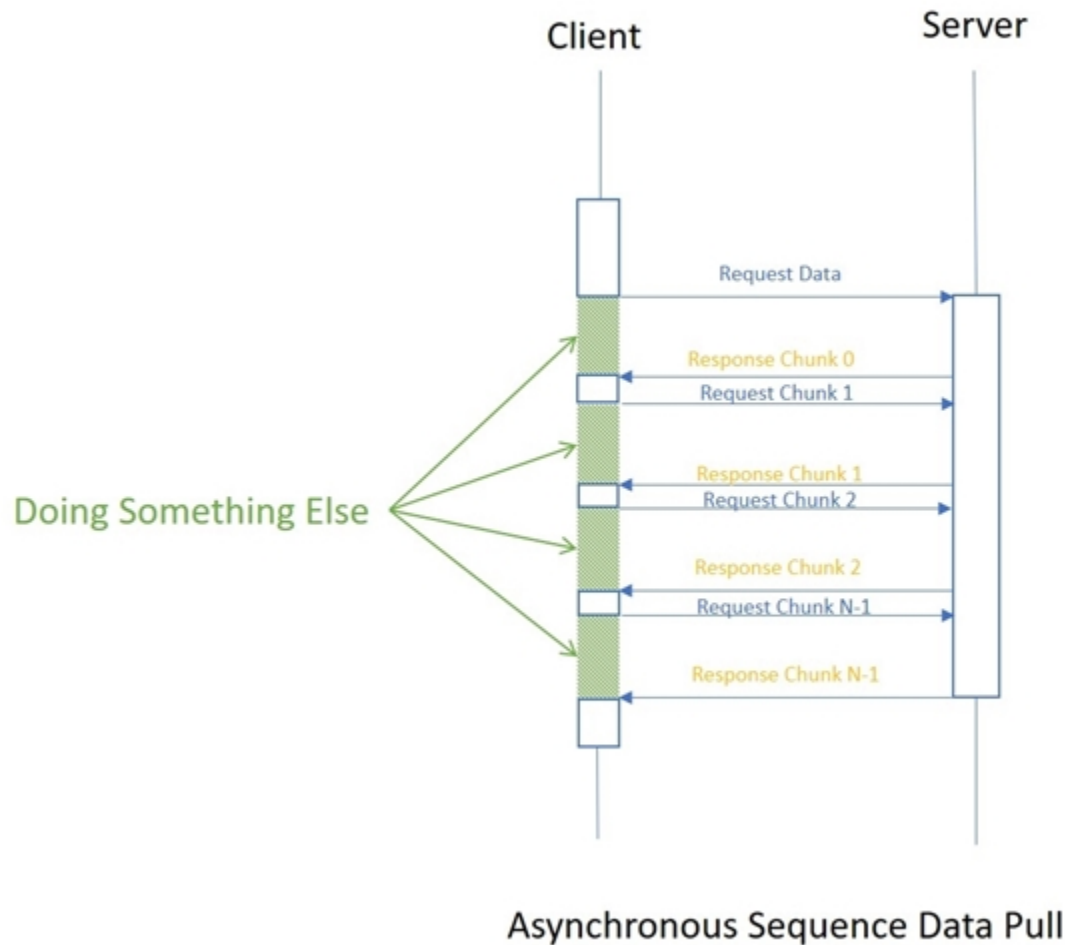


Figure -13- Asynchronous Sequence Data Pull (Async Streams). The Client is not blocked!

### Async Streams

In C# 8, similar to `IEnumerable<T>` and `IEnumerator<T>`; There are two new interfaces, **`IAsyncEnumerable<T>`** and **`IAsyncEnumerator<T>`** which are defined as below:

```
namespace System.Collections.Generic
{
    public interface IAsyncEnumerable<out T>
    {
        IAsyncEnumerator<T> GetAsyncEnumerator(CancellationTok
cancellationToken = default);
    }

    public interface IAsyncEnumerator<out T> : IAsyncDisposable
    {
        ValueTask<bool> MoveNextAsync();
        T Current { get; }
    }
}
```

```
}
```

**ValueTask<T>** is a struct that wraps **Task<T>**, and is behaviorally equivalent to **Task<T>**, except that it enables more efficient execution when the task completes synchronously (which can often happen when enumerating a sequence).

```
public interface IAsyncDisposable
{
    ValueTask DisposeAsync();
}
```

**IAsyncDisposable** is an asynchronous version of **IDisposable** and provides an opportunity to perform cleanup should you choose to implement the interfaces manually:

### Syntax

The **foreach** syntax has been extended with asynchronous functionality so that making use of the asynchronous interfaces when the **await** keyword is used:

```
await foreach (var dataChunk in asyncStreams)
{
    // Working the yield data chunk! Or doing something else!
}
```

As seen above in the example, instead of computing a single value, we can now to potentially compute many values, sequentially, while also being able to await other asynchronous operations.

Example:

To generate an asynchronous stream, you write a method that combines the principles of iterators and asynchronous methods. As I mentioned before, your method should include both yield return and await, and it should return **IAsyncEnumerable<T>**:

```
public static async IAsyncEnumerable<int> CreateSequence(int count)
{
    for (var i = 0; i < count; i++)
    {
        await Task.Delay(500); // Time simulation; Do something async
        yield return i; // produce the Async sequence
    }
}
```

Use the **await foreach** statement, to consume an asynchronous sequence:

```
static async Task Main(string[] args)
{
    await foreach (var number in CreateSequence(10))
```

```
{  
    Console.WriteLine(number); // use the Async sequence  
}  
}
```

Output:

0  
1  
2  
3  
4  
5  
6  
7  
8  
9

Consuming an asynchronous stream requires you to add the **await** keyword before the **foreach** keyword when you enumerate the elements of the stream. Adding the **await** keyword requires the method that enumerates the asynchronous stream to be declared with the **async** modifier, and to return a type allowed for an async method. Typically, that means returning a **Task** or **Task<TResult>**. It can also be a **ValueTask** or **ValueTask<TResult>**. A method can both consume and produce an asynchronous stream, which means it would return an **IAsyncEnumerable<T>**. The above code generates a sequence from 0 to 9 and waiting 500 milliseconds between each number:

### Cancellation

There are many ways to cancel the Async Streams. I have explained below the cancellation token. The cancellation token is a technique to cancel the async operations, and it can be passed to the extension method **WithCancellation**.

I have modified the example above that the asynchronous process should be canceled after one second.

```
public static async IAsyncEnumerable<int> CreateSequence(int count,  
[EnumeratorCancellation] CancellationToken cancellationToken = default)  
{  
    var i = 0;  
    while (i < count && !cancellationToken.IsCancellationRequested)  
    {  
        i++;  
        await Task.Delay(500); // Time simulation; Do something async  
        yield return i; // produce the Async sequence  
    }  
}  
  
static async Task Main(string[] args)  
{  
    var cts = new CancellationTokenSource();  
    cts.CancelAfter(1000);  
}
```

```
await foreach (var number in
CreateSequence(10).WithCancellation(cts.Token))
{
    Console.WriteLine(number); // Use the Async sequence
}

Console.WriteLine("Done!");
}
```

Output:

1

2

Done!

The **[EnumeratorCancellation]** attribute is very important. Placing this attribute on a parameter tells the compiler that if a token is passed to the async method, that token should be used instead of the value initially passed for the parameter.

### Summary

Async Streams is an excellent asynchronous pulling technique that can be used to write computations that generate multiple values asynchronously.

The programming concept behind Async Streams is the async pull model. We can demand the next element of the sequence, and we eventually get a reply. This technique is different from the push model of **IObservable<T>**, which generates values unrelatedly to the consumer's state. Async Streams provide an excellent way to represent asynchronous data sources that can be controlled by the consumer, for example, when the consumer isn't ready to handle more data. Examples include a web application or reading records from the database.



Async Streams provide an excellent way to represent asynchronous data sources that can be controlled by the consumer. For example, when downloading data from the web, we would like to create an asynchronous collection that returns the data in chunks as they become available.



Well accepted from .Net communities.



Async Pull Programming Model is known and used for years and already used in many other languages like Akka Streams.

## Indices and Ranges

Ranges and indices allow more natural syntax for accessing single items or ranges in a sequence. It consists of two new operators (Index operator **^** in **System.Index**) and (Range operator **..** in **System.Range**), which allow constructing **System.Index**, and **System.Range** objects, and using them to index/slice collections at runtime. The new operators are syntactic sugar and making your code cleaner and more readable.

### Index

Index feature is implemented in **System.Index**, and it is an excellent way to index a collection from the ending.

The end index operator **^** (*hat operator*) specifying that the index is relative to the end of the sequence.

```
var array= new[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Index 4 from the begin of the collection
Index i2 = 4;
Console.WriteLine($"{array[i2]}"); // Output: "4"

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

### Range

The Range is a more natural syntax for specifying or accessing subranges in a sequence.

The Range easily defines a sequence of data. It is a replacement for `Enumerable.Range()`, except that the Range, defines the start and stop points rather than start and count, and it helps you to write more readable code.

Example:

```
var array= new[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

foreach (var firstFiveNumbers in array[..5])
{
    Console.WriteLine(firstFiveNumbers);
}

// Output: 0, 1, 2, 3, 4
```

The Range feature is implemented in **System.Range** and it offer a new way to access *ranges* or *slices* of collections. This new way can help you to avoid LINQ and to make your code compact and more readable. You can compare this feature with Range in F#.

The range operator has a start and the end parameter, which specifies the start and end of a range as its operands.

Example:

```
Range range = 1..4; // 1 is the start parameter and 4 is the end parameter
var slice = array[range];
Console.WriteLine(slice); // output 1, 2, 3
```

Examples

Consider the following array:

```
var array = new int[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

C# 8 Style	C# 1..7.x Style
<code>var thirdItem = array [2];</code>	<code>var thirdItem = array [2];</code>
<code>var lastItem = array [^1];</code>	<code>var lastItem = array[array.Count -1];</code> <code>var lastItem = array.Last; // LINQ</code>
<code>var subCollection = array[2..^5];</code> <code>// Output: 2, 3, 4, 5</code>	<code>var subCollection =</code> <code>array.ToList().GetRange(2, 4);</code>
Notes: <code>array[2..^5]</code> ; as you see here, we have used both operators! Range and Index.  <b><code>array[2..</code></b> : means skip until index 2 from the beginning and <b><code>...^5]</code></b> : means ignore the first 5 elements from behind.	or with LINQ <code>var subCollection =</code> <code>array.Skip(2).Take(4);</code>

*var array* indexes overview:

Array Value	Array Index from start	Array Index from end
0	0	^11
1	1	^10
2	2	^9
3	3	^8

4	4	^7
5	5	^6
6	6	^5
7	7	^4
8	8	^3
9	9	^2
10	10	^1

We cut a slice from this array as below:

```
var slice= array[2..5];
```

<b>var slice values:</b>	<b>2</b>	<b>3</b>	<b>4</b>
We can access the slice values with the following start indexes:			
<b>var slice indexes</b>	<b>0</b>	<b>1</b>	<b>2</b>

Important Note: the **start index is inclusive** (included to the slice), and the **end index is exclusive** (excluded from the slice).

```
var slice1 = array [4..^2]; // Range.Create(4, new Index(2, true))
Output: 4, 5, 6, 7, 8
```

slice1 will be of type **Span<int>**. [4..^2] means Skip items from the begin until the index 4 and skip 2 items from the ending.

```
var slice2 = array [..^3]; // Range.EndAt(new Index(3, true))
Output: 0, 1, 2, 3, 4, 5, 6, 7


var slice3 = array [2..]; // Range.StartAt (2)
Output: 2, 3, 4, 5, 6, 7, 8, 9, 10

var slice4 = array [..]; // Range.All
Output: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

### Bounded Ranges

In the bounded ranges, the lower bound (start index) and the upper bound (end index) are known or predefined.

`array[start..end]` // Get items from **start** until **end-1**

The above **Range** syntax is inspired by Python. Python supports the following syntax(lower:upper:step), The step in Python is optional, and it is 1 by default.  C# 8 does not support steps, but there are some wishes in the C# community to use the F# syntax (lower..step..upper).

Range syntax in F#.

```
array { 5 .. 2 .. 20 } // where 2 = step [start .. step .. end]
```

Output:

5 7 9 11 13 15 17 19

Example for the bounded ranges in C# 8

```
var array = new int[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
var subarray = array[3..5]; // The selected items: 3, 4
var fiveToSeven = array[5..7]; // The selected items: 5, 6
```

The code above is equal to `array.ToList().GetRange(3, 2)`. If you compare `array.ToList().GetRange(3, 2)` with `array[3..5]` you can see that the new style is more expressive.



There is a feature request to use the Range in the “if” statement and with the Pattern Matching, for example, `(switch(array) [1..5] ..)`, but unfortunately, this is still not done yet.

### Unbounded Ranges

When the lower bound is omitted, it is interpreted to be zero, or the upper bound is omitted. It is interpreted to be the length of the receiving collection.

Examples:

```
array[start..] // Get items start with the rest of the array
array[..end] // Get items from the beginning until end-1
array[..] // A Get the whole array
var fiveToEnd = 5..; // Equivalent to Range.StartAt(5), Missing upper bound
var startToTen = ..1; // Equivalent to Range.EndAt(1), Missing lower bound.
var everything = ..; // Equivalent to Range.All.
```

### Range Chars

```
var collection = new [] { 'a', 'b', 'c' };
collection[2..]; // Selected chars: c
collection[..2]; // Selected chars: a, b
collection[..]; // Selected chars: a, b, c
```



### *Range with Strings*

Ranges allow creating substrings by using the indexer:

Example:

```
var helloWorldStr = "Hello, World!";  
var hello = helloWorldStr[..5]; // Take 5 from the begin  
Console.WriteLine(hello); // Output: Hello  
  
var world = helloWorldStr[7..]; // Skip 7  
Console.WriteLine(world); // Output: World!
```

Or you can write it like the following:

```
var world = helloWorldStr[^6..]; // Take the last 6 characters from behind  
Console.WriteLine(world); // Output: World!
```

### Ranges ForEach loops

#### *Range with IEnumerable Example*

Ranges implement **IEnumerable<int>**, which allowing the iteration over a sequence of data.

```
var array= new[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
foreach (var firstFiveNumbers in array[..5])  
{  
    Console.WriteLine(firstFiveNumbers);  
}  
// Output: 0, 1, 2, 3, 4
```

### Summary

The Range feature is very useful to generate sequences of numbers in the form of a collection or a list.

Range and Indexes make the C# syntax simpler and more readable.



Nice syntax to access the sequence of data from a collection.



An excellent feature and well accepted by many developers and especially for the functional programming programmers.



Nothing is new here. Similar technology already exists in other languages like Python.

## Pattern Matching

Pattern matching is one of the powerful constructs, which is available in many functional programming languages like F#. Furthermore, pattern matching provides the ability to deconstruct matched objects, giving you access to parts of their data structures. C# offers a rich set of patterns that can be used for matching.

Pattern matching was initially planned for C# 7, but after a while, the .Net team has found that they need more time to finish this feature. For this reason, they have divided it into two main parts. Basic Pattern Matching that is already delivered with C # 7 and the Advanced Patterns Matching delivered with C# 8. We have seen in C# 7 Const Pattern, Type Pattern, Var Pattern, and the Discard Pattern.

In C# 8, we have seen more patterns like Recursive Pattern, which consist of multiple sub-patterns like the Positional Pattern, Tuple, Pattern, and Property Pattern.

As I mentioned above, the power of the Pattern matching is to allow you to compare data with a logical structure or structures, decompose data into constituent parts, or to extract the information from data in various ways.

The main benefits of Pattern Matching: Pattern Matching provides a flexible and powerful way of testing data against a series of conditions (like if-else) and performing some computations based on the condition met.

To have a deep understanding of Pattern Matching and Recursive Patterns, we need many examples.

I have defined two classes *Employee* and *Company*, and I have used them to demonstrate the Pattern Matching concepts:

```
public class Employee
{
    public string Name
    {
        get;
        set;
    }

    public int Age
    {
        get;
        set;
    }

    public Company Company
    {
        get;
```

```
        set;
    }

    public void Deconstruct(out string name, out int age, out Company company)
    {
        name = Name;
        age = Age;
        company = Company;
    }
}

public class Company
{
    public string Name
    {
        get;
        set;
    }

    public string Website
    {
        get;
        set;
    }

    public string HeadOfficeAddress
    {
        get;
        set;
    }

    public void Deconstruct(out string name, out string website, out string
headOfficeAddress)
    {
        name = Name;
        website = Website;
        headOfficeAddress = HeadOfficeAddress;
    }
}
```

The Deconstruct method allows you to deconstruct the employ object to a tuple. Deconstruct returns void, and each value to be deconstructed is indicated by an out parameter in the method signature.

The following Deconstruct method of an Employ class returns the name, age, and company:

```
public void Deconstruct(out string name, out int age, out Company company)

// using the Deconstruct method
var (eName, eAge, eCompany) = employ;
```

## Switch expressions

Most of us are familiar with the original switch statement in C#, which returns one value while the new switch expression returns an expression. The new *switch* expressions syntax is a little bit different, but it is more succinct as compared to the old *switch* statement. Besides, *switch* expression allows you to use Pattern Matching within the switch scope.

The anatomy of the *switch* expression looks like this:

```
switch (value)
{
    case pattern guard => Code block to be executed
    ...
    case _ => default code
}
```

### Example:

```
var bassam = new Employee() {Age=42, Name="Bassam"};
var thomas = new Employee() {Age=43, Name="Thomas"};
var obj = new object();

Console.WriteLine(GetEmployee(bassam));
Console.WriteLine(GetEmployee(thomas));
Console.WriteLine(GetEmployee(obj));

static string GetEmployee(object o) => o switch
{
    Employee p when p.Age == 42 && p.Name
        == "Bassam" => $"{p.Name} Alugili",
    Employee p
        => $"{p.Name} Albrecht",
    _
        => "unknown"
};
```

### Output:

Bassam Alugili  
Thomas Albrecht  
Unknown

## Tuple Pattern

Tuple patterns allow you to switch based on multiple values expressed as a tuple.

### Example:

```
Console.WriteLine(GetEmployee("Bassam", 42));
Console.WriteLine(GetEmployee("Thomas", 43));
Console.WriteLine(GetEmployee(string.Empty, 0));
```

```
public static string GetEmployee(string name, int age) => (name, age) switch
{
    ("Bassam", 42) => $"{name} Alugili",
    ("Thomas", _) => $"{name} Alugili",
    (_, _) => "unknown"
};
```

Output:

Bassam Alugili  
Thomas Albrecht  
Unknown

*("Bassam", 42) => \$"{name} Alugili"*, the previous code line means: return the following *string*:*"Bassam Alugili"* only if the first tuple item is "Bassam" and the age is 42.

### Positional Pattern

Positional Pattern decomposes a matched type and performs further pattern matching on the values that are returned from it. The final value of this pattern/expression is true or false, which led to execute the code block or not.

```
if (employee is Employee(_, _, ("Stratec", _, _)) employeeTmp)
{
    Console.WriteLine($" The employee: {employeeTmp.Name}!");
}
```

Output:

The employee: Bassam Alugili

In this example, I have used the pattern matching recursively. The first part is the Positional Pattern *employee is Employee(...)*, and the second part is the sub-patterns within the brackets *(\_, \_ ("Stratec", \_, \_))*.

The code block after the **if** statement will only be executed if the conditions in the root Positional Pattern (employee object must be of type Employee) with its sub-pattern *(\_, \_ ("Stratec", \_, \_))* the company name must be "Stratec") are satisfied, and the rest is discarded. To explain it with simple words, it can be explained as follows: the **if** statement evaluates true when the company name is "Stratec" because everything else will be ignored/discarded.

### Property Pattern

The property Pattern is straight forward. You can access a type fields and properties and apply a further pattern matching against them.

```
if (bassam is Employee {Name: "Bassam Alugili", Age: 42})
{
    Console.WriteLine($"The employee: {bassam.Name} , Age {bassam.Age}");
}
```

Old style C# 6:

```
if (firstEmployee.GetType() == typeof(Employee))
{
    var employee = (Employee) firstEmployee;
    if (employee.Name == "Bassam Alugili" && employee.Age == 42)
    {
        Console.WriteLine($"The employee: {employee.Name} , Age {employee.Age}");
    }
}
```

// Or we can do it like this:

```
var employee = firstEmployee as Employee;
if (employee != null)
{
    if (employee.Name == "Bassam Alugili" && employee.Age == 42)
    {
        Console.WriteLine($"The employee: {employee.Name} , Age {employee.Age}");
    }
}
```

Compare the pattern matching code with C# 6 and look at how the C# 8 code is more clearly. The new style removes the redundant code and the typecasting or the ugly operators like **typeof** or **as**.

### Recursive Patterns

Recursive Pattern matching is a very powerful feature, which allows code to be written more elegantly, mainly when used together with recursion. The Recursive Patterns consist of multiple sub-patterns like as Positional Patterns, for example, `var isBassam = user is Employee ("Bassam",_,_)`, Property Patterns, `var isMAis= user is Employee {Name : "Mais"}`, Discard Pattern (`_`), and so forth.

MSDN:

*"In addition to new patterns in new places, C# 8.0 adds recursive patterns. The result of any pattern expression is an expression. A recursive pattern is simply a pattern expression applied to the output of another pattern expression."*

Example:

Recursive Patterns with tuples.

```
var employee = (Name: "Rami Alugili", Age: 10);
switch (employee)
```

```
{
    case (_, 10) employeeTmp when (employeeTmp.Name == "Rami Alugili"):
    {
        Console.WriteLine($"Hi {employeeTmp.Name}, you are now 10!");
    }
    break;

    // If the employee has any other information, then execute the code below.
    case (_, _):
        Console.WriteLine("any other person!");
        break;
}
```

The `case (_, 10)` is interpreted as follows. The first part, `_`, means the Name might be any string, and the second part, `10` means the Age, must be 10. If the employee tuple contains this pair of information (*any string, 10*), then the case block will be executed. Otherwise, the Discard Pattern will be executed case `(_, _)`.

Note: You can `(_, _)` discard with the default case.

Output:

Hi Rami Alugili, you are now 10!

#### More about Recursive Patterns

Recursive Patterns are nothing more than a combination of the above-described patterns. The type will be decomposed to subparts so that the subparts may be matched against sub-patterns. Behind the scene, this pattern deconstructs the type by using the *Deconstruct* method and applying on the deconstructed value further pattern matching if needed. If your type does not have a *Deconstruct* method or it is not a tuple, then you need to write it by yourself.

If you remove the *Deconstruct* method from the *Company* class above, then the following error occurs:

error **CS8129**: No suitable Deconstruct instance or extension method was found for type *Company*, with 0 out parameters and a void return type.

#### Pattern Matching examples

I have created two employees and two companies and have mapped each employee to a company:

```
var stratec = new Company
{
    Name = "Stratec",
    Website = "www.stratec.com",
    HeadOfficeAddress = "Birkenfeld",
};
```

```
var firstEmployee = new Employee
{
    Name = "Bassam Alugili",
    Age = 42,
    Company = stratec
};

var microsoft = new Company
{
    Name = "Microsoft",
    Website = "www.microsoft.com",
    HeadOfficeAddress = "Redmond, Washington",
};

var secondEmployee = new Employee
{
    Name = "Satya Nadella",
    Age = 52,
    Company = microsoft
};

Console.WriteLine(DumpEmployee(firstEmployee));
Console.WriteLine(DumpEmployee(secondEmployee));

public static string DumpEmployee(Employee employee) => employee switch
{
    (_, _, _) employeeTmp => $"The employee: {employeeTmp.Name}! ",
    _ => "Other company!"
};
```

## Output

The employee: Bassam Alugili!

The employee: Satya Nadella!

In the example above, the case condition is matching any employee with any data; it is a combination of the Deconstruction Pattern and the Discard Pattern. Now we can go one step further. We need to filter the *Stratec* employees.

In fact, there is more than one way to do this with Pattern Matching. We can replace/rewrite the below case condition from the example above with some different techniques:

```
(_, _, _) employeeTmp => $"The employee: {employeeTmp.Name}! ",
```

The first approach, by using the Recursive Patterns Matching (Deconstruction Pattern) in the switch statement like the following.



Replace the above code with the code below:

```
(_, _, ("Stratec", _, _)) employeeTmp => $"The employee: {employeeTmp.Name}! ",
```

Output:

The employee: Bassam Alugili!

Other company!

The second approach is by using guards (Constraints).

Replace the case expression with the following:

```
(_, _, (_, _, _)) employeeTmp when employeeTmp.Company.Name == "Stratec" =>  
$"The employee: {employeeTmp.Name}!",
```

Likewise, we can rewrite the case expression in different ways:

```
(_, _, _) employeeTmp when employeeTmp.Company.Name == "Stratec" => $"The  
employee: {employeeTmp.Name}! ",  
  
{ } employeeTmp when employeeTmp.Company.Name == "Stratec" => $"The employee:  
{employeeTmp.Name}! ",
```

Where {} is the object pattern!

We can also combine Deconstruction Pattern with Var Pattern like the following:

```
(_, _, var (companyNameTmp, _, _)) employeeTmp when companyNameTmp ==  
"Stratec"=> $"The employee: {employeeTmp.Name}! ",
```

Another approach to filter the data by using Property Pattern recursively as shown below:

```
{ Company: Company { Name: "Stratec" } } employeeTmp => $"The employee:  
{employeeTmp.Name}! ",
```

The Output for the all above examples:

The employee: Bassam Alugili!

Other company!

One important note I want to mention about using the switch statement with the pattern matching:

Look to the following Recursive Patterns matching examples:

```
public static string DumpEmployee(Employee employee) => employee switch
{
    {Name:"Bassam Alugili",Company: Company(_,_,_)} employeeTmp => $"The
employee: {employeeTmp.Name}!",

    (_,_,("Stratec",_,_)) employeeTmp => $"The employee: {employeeTmp.Name}!",

    (_,_,Company(_,_,_)) employeeTmp => $"The employee: {employeeTmp.Name}!",

    (_,_,_) employeeTmp => $"The employee: {employeeTmp.Name}!",

    _ => "Other company!"
};
```

The above switch is working fine. If we move one of those switch cases somewhere else up/down. Let's say you have moved `(_,_,_) employeeTmp => $"The employee: {employeeTmp.Name}"`, to the beginning (first case after the switch) as shown below:

```
public static string DumpEmployee(Employee employee) => employee switch
{
    (_,_,_) employeeTmp => $"The employee: {employeeTmp.Name}!",

    {Name:"Bassam Alugili",Company: Company(_,_,_)} employeeTmp => $"The
employee: {employeeTmp.Name}!",

    (_,_,("Stratec",_,_)) employeeTmp => $"The employee: {employeeTmp.Name}!",

    (_,_,Company(_,_,_)) employeeTmp => $"The employee: {employeeTmp.Name}!",

    _ => "Other company!"
};
```

Then we get the following errors:

error **CS8510** : The pattern has already been handled by a previous arm of the switch expression.

error **CS8510** : The pattern has already been handled by a previous arm of the switch expression.

error **CS8510** : The pattern has already been handled by a previous arm of the switch expression.

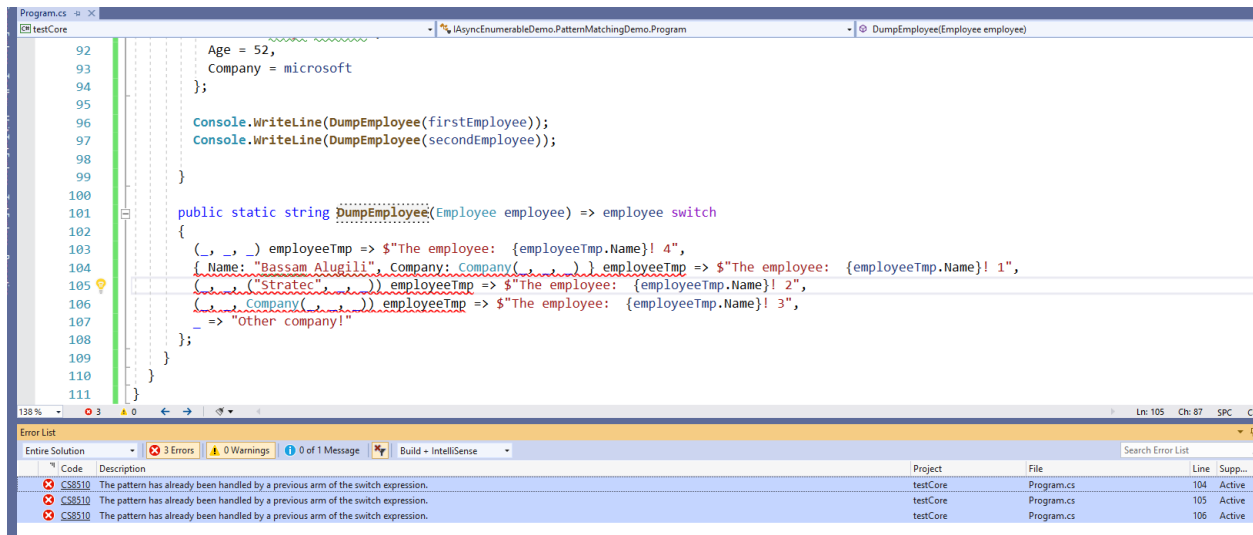


Figure -14- Error after changing the switch case position in Visual Studio

The compiler knows that the cases below your most generic *switch case* `(_, _, _) employeeTmp` cannot be reached (dead code), and the compiler informs you about (you are doing something wrong).

### Pattern Matching (C# 8) Playground

You can test it with Visual studio 2019 or in the web browser: <https://sharplab.io>

Code:

```
using System;
namespace PatternMatchingDemo
{
    public class Company
    {
        public string Name
        {
            get;
            set;
        }

        public string Website
        {
            get;
            set;
        }

        public string HeadOfficeAddress
        {
            get;
            set;
        }

        public void Deconstruct(out string name, out string website, out string headOfficeAddress)
    }
}
```

```
{
    name = Name;
    website = Website;
    headOfficeAddress = HeadOfficeAddress;
}

public class Employee
{
    public string Name
    {
        get;
        set;
    }

    public int Age
    {
        get;
        set;
    }

    public Company Company
    {
        get;
        set;
    }

    public void Deconstruct(out string name, out int age, out Company
company)
    {
        name = Name;
        age = Age;
        company = Company;
    }
}

class Program
{
    static void Main(string[] args)
    {
        var stratec = new Company
        {
            Name = "Stratec",
            Website = "www.stratec.com",
            HeadOfficeAddress = "Birkenfeld",
        };
        var firstEmployee = new Employee
        {
            Name = "Bassam Alugili",
            Age = 42,
            Company = stratec
        };
        var microsoft = new Company
        {
            Name = "Microsoft",
            Website = "www.microsoft.com",
```

```

        HeadOfficeAddress = "Redmond, Washington",
    };
    var secondEmployee = new Employee
    {
        Name = "Satya Nadella",
        Age = 52,
        Company = microsoft
    };
    Console.WriteLine(DumpEmployee(firstEmployee));
    Console.WriteLine(DumpEmployee(secondEmployee));

}

public static string DumpEmployee(Employee employee) => employee switch
{
    { Name: "Bassam Alugili", Company: Company(_, _, _) } employeeTmp =>
$"The employee: {employeeTmp.Name}! 1",
    (_, _, ("Stratec", _, _)) employeeTmp => $"The employee:
{employeeTmp.Name}! 2",
    (_, _, Company(_, _, _)) employeeTmp => $"The employee:
{employeeTmp.Name}! 3",
    (_, _, _) employeeTmp => $"The employee: {employeeTmp.Name}! 4",
    _ => "Other company!"
};
}
}

```

### Summary

Recursive Pattern is the core of the Pattern Matching. Pattern Matching helps you to compare the runtime data with any data structure and decompose it into constituent parts or extract sub-data from data in different ways, and the compiler is supporting you to check the logic of your code.

Pattern Matching is an excellent feature. It is giving you the flexibility to testing the data against a sequence of conditions and performing further computations based on the condition met.



Pattern matching helps you decompose and navigate data structures in a very convenient, compact syntax. While pattern matching is conceptually similar to a sequence of (if-then) statements, so it can help you to write the code in a functional style.



In complex expressions, the syntax might be tricky and difficult to understand.



Well accepted among .NET Community, and it can be very good used with Records C# 9 feature.



Pattern matching is a proven and known technology, which has been used for many years, especially in functional programming like F#.

## Using declarations

It enhances the using operator to use with Patterns and make it more natural.

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

In the example above, the repository is disposed when the closing brace for the method is reached.

## Enhancement of interpolated verbatim strings

Allows @\$"" as a verbatim interpolated string.

Example.

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

## Null-coalescing assignment

The ??= operator assigns a variable only if it's null.

You can use it to simplify 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
```

## Unmanaged constructed types

In C# 7.3 and earlier, a constructed type (a type that includes at least one type of argument) can't be an unmanaged type. Starting with C# 8.0, a constructed value type is unmanaged if it contains fields of unmanaged types only.

```
Public struct Foo<T>
{
    public T Var1;
    public T Var2;
}
```

Bar<int> type is an unmanaged type in C# 8.0. Like for any unmanaged type, you can create a pointer to a variable of this type or allocate a block of memory on the stack for instances of this type:

```
// Pointer
var foo = new Foo <int> { Var1 = 0, Var2 = 0 },
var bar = &foo; // C# 8

// Block of memory
Span< Foo<int>> bars = stackalloc[]
{
    new Foo <int> { Var1 = 0, Var2 = 0 },
    new Foo <int> { Var1 = 0, Var2 = 0 }
};
```

Notes:

- This feature is a performance enhancement.
- Constructed value types are now unmanaged if it only contains fields of unmanaged types.
- This feature means that you can do things like allocate instances on the stack.

## Static local functions

It 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;
}
```

## Readonly-Member

It allows you to apply the **readonly** modifier to any member of a struct. It indicates that the member doesn't modify the state.

```
public struct XValue
{
    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

Starting with C# 8.0, if the result of a **stackalloc** expression is of the **System.Span<T>** or **System.ReadOnlySpan<T>** type, you can use the **stackalloc** expression in other expressions:

```
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

Allow 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!
```



# C# 8 Feature Cheat Sheet

## Default interface methods

It allows you to add new functionality to the 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.

```
#nullable enable
string? nullableString = null; // Is Ok,
// nullableString it can be null and it is null.
string oldString = null; // Warning, you have to
// use here Nullable reference type.

// WARNING: possible of NullReferenceException!
Console.WriteLine(nullableString.Length);
```

## Using declarations

It 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`

## Pattern matching enhancements

It 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
};
```

## 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
```

## Asynchronous streams

Allows having enumerators that support `async` operations.

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

## Static local functions

It allows you to add the `static` modifier to the local functions.

```
int AddFiveAndSeven()
{
    int y = 5; int x = 7;
    return Add(x, y);

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

## Indices and ranges

It 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 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[3..6]; // { 3, 4, 5 }
```

## Disposable ref structs

It 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!
```

## 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
}
```

## Stackalloc in nested expressions

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

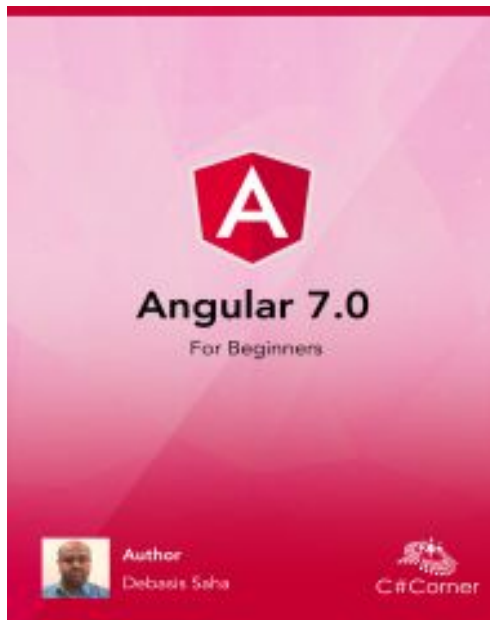
```
Span<int> set =
    stackalloc[] { 1, 2, 3, 4, 5, 6 };
var subSet = set.Slice(3, 2);
foreach (var n in subSet)
    Console.WriteLine(n); // Output: 4 5
```

## Readonly member

It allows you to apply the `readonly` modifier to any member of a `struct`.

```
public struct XValue
{
    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;
    }
}
```



Download



Download



Download



Download