Building End-to-End Apps with C# 11 and .NET 7

The complete guide to building web, desktop, and mobile apps

Arun Gupta

www.bpbonline.com

First published: 2024

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55516-374

www.bpbonline.com

Dedicated to

In loving memory of my father, your dedication, tireless work ethic and simple lifestyle continue to inspire me every day. Your life was a shining example of determination and passion, setting a standard that I strive to live up to. Like any devoted parent, you yearned for nothing more than to see your children excel, and you spared no effort to make that dream a reality. Though you are no longer with us, your legacy of love and ambition lives on in everything that I do.

Foreword

C# has been my true passion for over a decade, shaping my professional path. I've immersed myself in the .NET developer community, sharing insights worldwide on C#-centered app development strategies. From active GitHub contributions to collaborating with top engineers, I have championed feature adoption and innovation. My experience as a Solutions Architect overseeing design decisions for twenty teams resulted in an astonishing 20x speed boost in just 30 days. I am honored to hold titles like Google Developer Expert, two-time Microsoft Valuable Professional (MVP), Twilio Champion, and author of "Learning Blazor: Build Single-Page Apps with WebAssembly and C#" with O'Reilly Media. Co-hosting "On .NET Live" has also been an incredible journey, and I'm deeply grateful for these enriching experiences that celebrate diversified thoughts from amazing people around the .NET community.

App development in C# encompasses a wide array of possibilities, resonating uniquely with each developer. This language serves as a versatile paintbrush, enabling you to craft and collaborate with fellow enthusiasts in the ever-evolving realm of modern software development. What masterpiece will you bring to life, how brilliant will it shine, and whose world will it enhance? Within the pages of this book, you'll delve into the realm of C# development, exploring its real-world applications and the captivating stories it can tell. It is my firm belief that C# developers possess the ability to create truly exquisite and meaningful digital experiences.

C# has continually evolved with each new iteration of .NET over the years. As developers, we're confronted with the pivotal task of discerning which features to incorporate into our projects and, crucially, why we should do so. This decision-making process necessitates a deep understanding of the trade-offs involved—whether a feature enhances readability, boosts performance, or simplifies complexity. Notably, not all features will tick all these boxes, and that's perfectly acceptable. However, it remains invaluable to grasp the array of available features.

Certain features wield the power to profoundly influence how we craft our code. From libraries and services to frameworks and core functionalities, they can usher in significant shifts. This underscores the importance of cultivating robust development environments anchored by dependable codebases. Our reliance on cutting-edge tools, including world-class integrated development environments, enriched by AI-powered statement completions and language servers, fuels our productivity and ensures we stay at the forefront of innovation. In this book, you'll explore these evolving features, understand their implications, and navigate the dynamic landscape of C# development.

In this book, you'll explore an extensive array of app development scenarios catering to diverse client needs. We begin by thoroughly covering the features of C# 11, progressing into web app development with ASP.NET Core Web app and Web APIs. The journey continues with gRPC Services, the innovation of Blazor WebAssembly, and a nod to TypeScript and SPA development with Angular. We also delve into desktop applications using WPF, WinUI 3, and .NET MAUI. Just when it seems like we've covered it all, we introduce you to ML.NET for machine learning app development. This book is a comprehensive resource that helps you pinpoint which features align with your goals. I'm confident that you'll find tremendous value in this read, and I want to extend my gratitude to Arun for creating this exceptional book.

- David Pine

(Microsoft, Senior Content Developer)

About the Author

Arun Gupta brings over twenty-five years of valuable experience in software development. He has worked in diverse roles across medium and large organizations in services and product development industry of the likes of erstwhile CMC Limited, Patni Computers and Kronos Solutions. These days, besides technical consulting, he enjoys sharing his expertise by mentoring and helping individuals become better programmers and testers.

About the Reviewer

Marco Siccardi is a .NET developer with more than 10 years of experience, currently working primarily with Xamarin and .NET MAUI on iOS in his day job for a big Swiss transportation company. He has also worked with server, desktop, web and cloud technologies in the past, completing his expertise.

His side projects have a wide range from mobile apps to cloud based applications and open source libraries, utilising everything the .NET world has to offer (including Azure, ASP.NET, MS SQL). In all projects, Marco is trying to get the cleanest architecture, using established patterns like MVVM, the repository pattern and others.

Recently, he started to explore the Swift programming language to evolve his native knowledge of the Apple ecosystem.

Besides that, he shares his knowledge through his personal blog, is an active runner and loves to take snaps with his iPhone. He is father of two young adults and husband to a patient and understanding wife.

Acknowledgement

I would like to express my deep gratitude to several individuals whose unwavering support and contributions have played a pivotal role in bringing this book to fruition.

I extend my heartfelt thanks to my wife, Deepika, and my sons, Dhruv and Anmol, your patience, love, and the steadfast belief in my endeavors have been my guiding light. Even during the several missed family occasions and countless stern noes, your understanding and encouragement have kept my spirits high and my determination unshakable.

To my family, friends and colleagues, your encouragement and well-wishes provided the invisible force that propelled me to the finish line. Your belief in me has been a constant source of inspiration.

Special thanks to Marco for his meticulous technical review and invaluable comments, which improved the technical aspects of the content.

Specifically, for the ML.NET chapter, I would like to extend my heartfelt appreciation to my former colleague, Ajay Sharma, a distinguished authority in this domain. Ajay graciously steered me in the right direction on more than one occasion, for which I am grateful.

I am also grateful to the dedicated BPB staff for their professionalism and the process framework of seamlessly transforming individuals from the industry into first-time authors.

To each of you, your collective contributions have been indispensable in making this book a reality, and for that, I am deeply thankful.

Preface

Welcome to the world of C# programming, where learning is an immersive journey, and knowledge is acquired by doing. This book is not a conventional manual; instead, it is a collection of notes born from the essence of a computer lab experience. As you delve into these pages, you embark on a journey that sets a learning tone reminiscent of a technical blog, where we tackle real-world problems step-by-step, unravelling the intricacies of C# 11 and .NET 7 along the way.

The approach is unique. Instead of diving deep into a single topic, we take a panoramic view, exploring various aspects necessary to build end-to-end applications. Think of it as going breadthwise around a topic, appreciating its nuances, before diving deep. This approach does not negate the value of depth, but it lays a solid foundation by starting from the concrete and progressing towards the abstract rather than the other way around.

This book is not a primer, it assumes you already possess a basic understanding of C# and Visual Studio. However, it serves as a steppingstone for those looking to graduate to the new language features, application types, and ways of working within the Visual Studio IDE.

The vast sea of information scattered across the internet and Microsoft's website can be overwhelming. This book serves as a beacon, synthesizing information from multiple sources into a coherent picture in the context of solving a specific problem. It presents this wealth of knowledge as a concise summary, guiding you towards further exploration through credible resources.

Each chapter in this book is bound together by the project templates found in Visual Studio. We leverage these templates to build a variety of applications, progressing step by step. Throughout these pages, we reinforce key concepts by creating engaging projects centered around interesting topics such as a Digital Bookstore, game of Cows and Bulls, Sudoku, an Image Navigator, and more. We begin with the big picture followed by taking the step-by-step journey into thinking about solving the problem. Once you have mastered the projects detailed within these pages, you will be equipped to apply the same principles to your own projects. The aim is the proverbial "teach you how to fish" – get you started and bring you up to a level where you can find the answers yourself.

In today's world, books are just one of the many resources at our disposal to get information and build knowledge. I commend your determination to excel in this field and implore you not to let your initial enthusiasm wane. I encourage you to fully engage with the text, diligently work through all the examples, ponder over the exercises, and explore the references. This way, by the end you will not only have a deeper understanding of these technologies but also a newfound inquisitiveness to delve further into the subject matter.

So, dear reader, as you embark on this educational journey, remember that this book is your key to unlocking the world of C# programming. Approach it with curiosity, embrace the "learning by doing" philosophy, and let it be your guide to mastering C#, .NET, and Visual Studio, one step at a time. I invite you to explore, experiment, and excel in this exciting field, and may this book be your trusted companion on your path to success.

The chapter-wise details are as follows

Chapter 1: New Features in C# 11 - .NET version 6 released in Nov 2021 along with C# 10 since then there have been several advancements both in .NET platform and the C# language. This chapter describes the new features that have made their way in C# language. Also included are the code snippets demonstrating the working and the use of the new features.

Chapter 2: ASP.NET Core Web App - gives an overview of the different web applications that can be created in .NET 7 and then goes on to explain with the example of creating a Digital Book Store, Razor Pages web application end-to-end. Unit testing is demonstrated for the Book Details page and Selenium based functional test is written for the Book Index page.

Chapter 3: ASP.NET Core Web API - focuses on developing Web APIs using ASP.NET Core. It explains the various concepts around API development along with creating a simple Create Read Update and Delete (CRUD) Web API around the Digital Book Store.

Chapter 4: gRPC Service - introduces the ASP.NET Core gRPC services as a logical extension of WCF services and compares it to the other options of building services. It illustrates the concepts around gRPC services by creating one which will download a large file on the client through service method call using server streaming. MSTest based unit-test is written for the Employees Data Service.

Chapter 5: Blazor WebAssembly - introduces the different Blazor applications, describes the concepts around them and goes on to create a simulation for the game of Cows and Bulls using Blazor WebAssembly App project template, to highlight the intricacies of the development process. Unit-testing is demonstrated for the Razor components using the bUnit – a community driven project. For end-to-end testing scenario the use of Playwright .NET is described.

Chapter 6: SPA with Angular - develops a rich single-page application with Angular on the client side supported by ASP.NET core on the server side. The chapter explains the interplay of different technologies in various tiers followed by creating the Angular version of the Digital Book Store application starting with the ASP.NET Core with Angular project template.

Chapter 7: WPF Application - we begin exploration of the Microsoft's desktop applications starting with Windows Presentation Foundation (WPF). We will understand the different types of Desktop Applications that can be created and where WPF fits in the overall strategy. To understand the controls and layout, we will create a simple custom pizza order form followed by building a real world WPF application around the popular game of Sudoku. Here, to further re-enforce the concepts in building a WPF application we will make use of User Controls to design the UI, implement Model-View-ViewModel (MVVM) pattern, make use of resource dictionaries, and finally publish the application. Unit-testing of the MVVM application is demonstrated by adding MSTest based test for the View-Model.

Chapter 8: WinUI 3 - begins by describing the evolution of creating applications for the Windows platform and then goes on to elaborate on developing native Windows applications for Windows 10 and Windows 11 using Windows UI Library (WinUI) 3. To understand the concepts around building a real-world WinUI application we will create an Image Navigator application.

Chapter 9: .NET MAUI - describes the evolving landscape for mobile application development with .NET followed by an overview of developing cross-platform apps using the .NET MAUI application templates. We will examine the differences in the .NET MAUI App template and the .NET MAUI Blazor Hybrid App template. This will be followed by migrating the Sudoku application that we created using WPF into a .NET MAUI application, to further elaborate the concepts around .NET MAUI.

Chapter 10: ML.NET - gives an overview of building machine learning model in .NET using the ML.NET library. The chapter begins by explaining the overall machine learning process and goes on to illustrate the concepts by creating a program that uses binary classification algorithm to perform sentiment analysis. Another example uses the graphical Visual Studio extension Model Builder, to solve machine learning problems related to regression analysis. This is followed by working end-to-end on an anomaly detection problem with a reasonably large financial dataset including data preparation and model training followed by evaluation and prediction using the model.

Disclaimer

(Running examples on Mac)

We would like to bring to your attention that the programs developed in this book, along with the related code bundle, have been meticulously crafted using Visual Studio 2022 on a Windows 10 machine. While we have made every effort to ensure compatibility with other environments, it is essential to acknowledge that variances may arise when attempting to execute these programs on a Mac machine.

Our technical reviews have revealed that these differences can encompass a range of aspects, from variations in menu options to the absence of certain features. In such instances, we kindly urge our readers to embrace experimentation while keeping the end-goal in mind. We firmly believe that with some trial and error, you can still derive significant value from the topics discussed in this book.

For further information regarding Visual Studio on Mac and its capabilities, you may refer to the official documentation at **https://visualstudio.microsoft.com/vs/mac/**. Just to add, Microsoft is encouraging the use of Visual Studio Code for development on Mac as outlined at **https://learn.microsoft.com/en-us/visualstudio/mac/what-happened-to-vs-for-mac**.

Furthermore, it is crucial to note that the chapters pertaining to desktop development using WPF and WinUI are exclusively supported on Windows platforms.

We greatly appreciate your understanding and patience as you navigate through the diverse technical landscape, and we remain committed to assisting you in your journey of learning and exploration.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/61b4r71

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Building-End-to-End-Apps-with-CSharp11-and-.NET7**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

Chapter 1 New Features in C# 11

Introduction

The previous version of .NET, version 6, was released in Nov 2021 along with C# 10. Since then, there have been a number of advancements both in the .NET platform and the C# language. These advancements include performance improvements and newer constructs. This chapter explains the latest additions to C# 11, including the new features introduced through updated .NET platform APIs and language improvements within the C# language itself.

The chapter describes the major changes along with code snippets that make use of the new features. The examples are part of the companion source code bundle and can be referred to from there. Not every new enhancement is described here, and readers are encouraged to explore further from the websites given in the reference section at the end of the chapter. The exercise at the end gives the reader an opportunity to apply the newly learned concepts. The solutions to the exercise are also part of the source code bundle.

As we prepare to go to print, it is worth noting that the forthcoming releases of .NET and C# are imminent. Readers are encouraged to access further information regarding these upcoming versions through the provided references.

Structure

The chapter describes the following new features:

- Auto-default struct

- Extended **`nameof`** scope
- File-scoped types
- Generic attributes
- Generic math
- List patterns
- **`Microsecond`** and **`Nanosecond`** (.NET 7)
- Newlines in string interpolation expressions
- Pattern match Span `<char>` on a constant string
- Raw string literals
- Regex source generation (.NET 7)
- Required members
- Simplified ordering with **`System.LINQ`** (.NET 7)
- Stream reading - **`ReadExactly`** and **`ReadAtLeast`** (.NET 7)
- Tar APIs (.NET 7)
- Type converters - **`DateOnly`**, **`TimeOnly`**, **`Int128`**, **`UInt128`**, and **`Half`** (.NET 7)
- UTF-8 string literals
- Warning wave 7

Objectives

By the end of this chapter, the readers will be able to develop an understanding of the new features of C# 11 and the common scenarios where these new concepts can be applied. Some references to find more information on the topic has also been provided at the end of the chapter.

Let us now have a look at the new features of C# 11 which have been discussed as follows:

Auto-default struct

In the previous version of C#, structures where the constructor did not assign all the fields gave compile errors, but not anymore. In C# 11 any fields not explicitly initialized are set to their default value:

```
namespace _01_NewFeatures.Features;

internal class AutoDefaultStructure
{
```

```
5.      public void Demo()
6.      {
7.          Point p = new Point(-5);
8.          Console.WriteLine(p.ToString());
9.      }
10. }
11.
12. internal struct Point
13. {
14.     public int X { get; init; }
15.     public int Y { get; init; }
16.
17.     public Point(int x)
18.     {
19.         X = x;
20.     }
21.
22.     public override string ToString()
23.     {
24.         return $"[X = {X}, Y = {Y}]";
25.     }
26. }
```

In the preceding example, the **`Point`** constructor on line 17 gives compile error with C# 10 (.NET 6) - **`CS0843: Auto-implemented property '{0}' must be fully assigned before control is returned to the caller. Consider updating to language version '{1}' to auto-default the property`**. However, it works fine in C# 11 and the value of the property Y is set to default zero by the compiler.

Extended nameof scope

A **`nameof`** expression produces the name of a variable, type, or member as the string constant. In C# 11 this definition is extended to allow **`nameof`** operator to specify the name of a method parameter in an attribute on the method or parameter declaration. This feature is most often useful to add attributes for nullable static analysis.

```
namespace _01_NewFeatures.Features;

public class CustomNullCheck : Attribute
{
    private readonly string _paramName;
6.       public CustomNullCheck(string paramName)
7.       {
8.           _paramName = paramName;
9.       }
10.  }
11.
12.  internal class ExtendedNameOfScope
13.  {
14.      [CustomNullCheck(nameof(values))] //C# 11
15.      public void Demo(List<int> values)
16.      {
17.          Console.WriteLine(nameof(values));  //output: values
18.          Console.WriteLine(nameof(values.Count));  // output: Count
19.
20.          Name = null; //throws exception: Name cannot be null (Parameter 'value')
21.
22.          [CustomNullCheck(nameof(T))] //C# 11
23.          void LocalFunction<T>(T param)
24.          { }
25.
26.          var lambdaExpression = ([CustomNullCheck(nameof(someNumber))] int
     someNumber) => someNumber.ToString(); //C# 11
27.      }
28.
29.      private string _name;
30.      public string Name
31.      {
32.          get => _name;
33.          set => _name = value ?? throw new ArgumentNullException(nameof(value),
     $"{nameof(Name)} cannot be null");
34.      }
35.  }
```

In the preceding snippet, lines 17, 18 and 20 demonstrate existing usage of **`nameof`**. The new feature allows using **`CustomNullCheck(nameof(values))`** on line 14 instead of **`CustomNullCheck("values")`**. Lines 22 and 26 demonstrate the extended scope with local function and parameter declaration, respectively.

File-scoped types

C# 11 introduces a new access modifier **`file`**. The visibility of created type is scoped to the source file in which it is declared. This feature helps source generator authors avoid naming collisions. **`file`** local types cannot be nested, and no accessibility modifiers (**`private`**, **`public`**, **`protected`**, and so on.) can be used in combination with **`file`** on a type as it is treated as an independent concept from accessibility.

```csharp
//File1.cs

namespace _01_NewFeatures.Features;

internal class FileLocalTypes
{
}

file class Animal
{
}

// File2.cs

namespace SecondNS;

file class Animal // different symbol than the Animal in File1
{
}

// File3.cs
using SecondNS;

var animal = new Animal(); // error: The type or namespace name 'Animal' could
not be found.
```

The **`Animal`** class declared in `File1.cs` and `File2.cs` are different because of the **`file`** modifier. Trying to access either in `File3.cs` gives error.

Generic attributes

C# 11 now supports generics for attribute classes as well. Earlier attributes had to accept **`System.Type`** as a parameter and users passed a **`typeof`** expression to provide the attribute with types that it needs. Attribute can now use the existing system of type parameter constraints to express the requirements for the types they take as input.

```
1.  namespace _01_NewFeatures.Features;
2.
3.  // Before C# 11 -----------
4.  internal class BeforeCSharp11
5.  {
6.      [CustomTypeAttribute(typeof(string))]
7.      public string? Method() => default;
8.  }
9.
10. public class CustomTypeAttribute : Attribute
11. {
12.     public CustomTypeAttribute(Type t) => ParamType = t;
13.
14.     public Type ParamType { get; }
15. }
16.
17. // ------------------------
18.
19. internal class GenericAttributes
20. {
21.     [GenericAttribute<string>()]
22.     public string Method() => default;
23. }
24.
25. public class GenericAttribute<T> : Attribute { }
```

The preceding line, line 10 defines **`CustomType`** attribute in the older way. Notice the constructor accepting parameter **`t`** of type **`Type`**. When this attribute is used on line 6 **`typeof(string)`** is passed. The newer way allows defining attributes using generic as on line 25 notice the **`<T>`** with **`GenericAttribute`**. To use this **`<string>`** is passed on line 21.

Generic math

.NET 7 introduces new math-related generic interfaces to the base class library. In addition, C# 11 allows defining static abstract interface members. Together, these innovations enable performing mathematical operations generically — that is, without having to know the exact type. Earlier one had to create an overloaded method for each type. Now a single generic method can be written, where the type of the parameter is constrained to be a number-like type.

```
using System.Numerics;
namespace _01_NewFeatures.Features;

internal class GenericMath
{
    public void Demo()
    {
        Console.WriteLine("GenericMath.Demo() ---- ");

        //Example 1
        Vector v1 = new() { X = -2, Y = 8 };
        Vector v2 = new() { X = 6, Y = -1 };

        Console.WriteLine(v1 + v2); //Outputs: Vector (X: 4; Y: 7)
        Console.WriteLine(v1 - v2); //Outputs: Vector (X: -8; Y: 9)

        //Example 2
        Console.WriteLine(new List<int> { 5, -3, 0, 25 }.AddNumbers<int, long>()); //Outputs: 27
        Console.WriteLine(new List<double> { 5.5, 3.2, 4.6, 10.7 }.AddNumbers<double, double>()); //Outputs: 24
        Console.WriteLine("----------------------- ");
    }
}

public record Vector :
    IAdditionOperators<Vector, Vector, Vector>,
    ISubtractionOperators<Vector, Vector, Vector>
{
    public int X { get; set; }
    public int Y { get; set; }

    public static Vector operator +(Vector self, Vector other)
    {
        return new Vector { X = self.X + other.X, Y = self.Y + other.Y };
    }

    public static Vector operator -(Vector self, Vector other)
    {
        return new Vector { X = self.X - other.X, Y = self.Y - other.Y };
    }

    public override string ToString() => $"Vector (X: {X}; Y: {Y})";
}

internal static class MathExtensions
{
    public static TResult AddNumbers<T, TResult>(this IEnumerable<T> values)
    where T : INumber<T>
    where TResult : INumber<TResult>
    {
        TResult result = TResult.Zero;

        foreach (var value in values)
        {
            result += TResult.CreateChecked(value);
        }

        return result;
    }
}
```

The preceding snippet demonstrates two examples. In the first example, **`Vector`** record implements **`IAdditionOperators`** which contains the **`+`** operator declared as *static abstract* and is implemented explicitly in the **`Vector`** record. Similarly, **`Vector`** also implements **`ISubtractionOperators`** containing the **`-`** operator declared as *static abstract* and is implemented explicitly.

In the second example, **`AddNumbers`** extension method adds a set of values together. The method takes in an **`IEnumerable<T>`** where **`T`** must be a type that implements the **`INumber<T>`** interface. It returns a **`TResult`** with a similar constraint (it must be a type that implements **`INumber<TResult>`).** Because of the two generic parameters, it is allowed to return a different type than it takes as an input. For example, **`AddNumbers<int, long>`** allows summing the values of an **`int[]`** and returning a 64-bit result to help avoid overflow. **`TResult.Zero`** efficiently gives the value of 0 as a **`TResult`** and **`TResult.CreateChecked`** converts value from a **`T`** into a **`TResult`** throwing an **`OverflowException`** if it is too large or too small to fit in the destination format.

There may be a question why we cannot do simply as follows:

```
61.     public static T Sum<T>(this IEnumerable<T> source)
62.     {
63.         T sum = default;
64.         foreach (T v in source)
65.         {
66.             sum += v;
67.         }
68.
69.         return sum;
70.     }
```

The preceding code gives error on line 66 - **`CS0019: Operator '+=' cannot be applied to operands of type 'T' and 'T'`** because we cannot use addition assignment operators on a generic class **`T`** unless it implements the **`INumber`** interface which supports the addition operator defined as static abstract member.

List patterns

Patterns are used in C# to match an input expression against some characteristics (or pattern combination). C# supports multiple patterns; list is the latest addition in C# 11. List patterns evaluate sequence of elements in a list or an array to corresponding constant, type, property, or relational pattern.

```
namespace _01_NewFeatures.Features;

internal class ListPatterns
{
    public void Demo()
    {
        var numbers = new[] { -10, 0, 5, 7 };

        // Comparison with constant patterns
        Console.WriteLine(numbers is [-10, 0, 5, 7]); // True
        Console.WriteLine(numbers is [-10, 0, 5]); // False
        Console.WriteLine(numbers is [7, -10, 0, 5]);    // False

14.         // Comparison with discard patterns
15.         Console.WriteLine(numbers is [_, 0, _, 7]); // True
16.
17.         // Comparison with range pattern
18.         Console.WriteLine(numbers is [.., 5, _]);   // True
19.
20.         // Comparison with logical patterns
21.         Console.WriteLine(numbers is [_, <= 2, _, _]); // True
22.
23.         // Comparison with length pattern
24.         if (numbers is [< 0, .. { Length: 2 or 4 }, > 0]) //true
25.             Console.WriteLine("Valid");
26.         else
27.             Console.WriteLine("Invalid");
28.
29.         // Comparison with var pattern
30.         if ("Curious" is ['c' or 'C', 'u', .. var str, 'u', 's' or 'S']) //true
31.             Console.WriteLine($"Matches, inner string: {str}");
32.         else
33.             Console.WriteLine($"No match");
34.     }
35. }
```

The preceding snippet demonstrates comparing **`numbers`** which is a list of integers with various combinations of list patterns. The last example on line 30, compares a string with a list pattern and retrieves the value within the string using var pattern.

Microsecond and Nanosecond (.NET 7)

Before .NET 7, the lowest increment of time available in the various date and time structures was the *tick*, available as the **`Ticks`** property, one tick being 100 nanoseconds. To find anything lower than tick, developers had to perform calculations on the tick value. .NET 7 addresses this by introducing both microsecond and nanosecond to the date and time implementations – **`TimeSpan`**, **`TimeOnly`**, **`DateTime`**, and **`DateTimeOffset`**.

```
namespace _01_NewFeatures.Features;

internal class MicroAndNanoSeconds
{
    public void Demo()
    {
        const string dt = "0001-01-01"; //yyyy-mm-dd
        const string tm1 = "00:00:00.0001000"; //hh:mm:ss.ticks
        const string tm2 = "00:00:00.0000009"; //hh:mm:ss.ticks

        //1 sec = 10^3 milli sec = 10^6 micro sec = 10^7 ticks

        //DateTime examples
        Console.WriteLine(new DateTime(0001, 01, 01, 00, 00, 00, 00, 100).Ticks); //1000 (100 micro sec = 1000 ticks)
        Console.WriteLine(DateTime.Parse($"{dt} {tm1}").Microsecond); //100 (1000 ticks = 100 micro sec)
        Console.WriteLine(DateTime.Parse($"{dt} {tm2}").Nanosecond); //900 (9 ticks = 900 nano sec)
        Console.WriteLine(DateTime.MinValue.AddMicroseconds(100).Ticks); //1000 (100 micro sec = 1000 ticks)

        //DateTimeOffset examples
        Console.WriteLine(new DateTimeOffset(0001, 01, 01, 00, 00, 00, 00, 999, TimeSpan.FromHours(-8)).Ticks); //9990 (999 micro sec = 9990 ticks)
        Console.WriteLine(DateTimeOffset.Parse($"{dt} {tm1} -8").Microsecond); //100 (1000 ticks = 100 micro sec)
        Console.WriteLine(DateTimeOffset.Parse($"{dt} {tm2} -8").Nanosecond); //900 (9 ticks = 900 nano sec)
        Console.WriteLine(new DateTimeOffset().AddMicroseconds(100).Ticks); //1000 (100 micro sec = 1000 ticks)

        //TimeSpan examples
        Console.WriteLine(new TimeSpan(10 * TimeSpan.TicksPerMicrosecond).Ticks); //100 (10 * 10)
        Console.WriteLine(new TimeSpan(10 * TimeSpan.NanosecondsPerTick).Ticks); //1000 (10 * 100)
        Console.WriteLine(new TimeSpan(0, 0, 0, 1, 1, 1).Ticks); //10010010 (1 sec + 1 milli sec + 1 micro sec = 10^7 + 10^4 + 10 ticks)
        Console.WriteLine(TimeSpan.Parse($"{tm1}").Microseconds); //100 (1000 ticks = 100 micro sec)
        Console.WriteLine(TimeSpan.Parse($"{tm2}").Nanoseconds); //900 (9 ticks = 900 nano sec)
        Console.WriteLine(TimeSpan.FromMicroseconds(100).Ticks); //1000 (100 micro sec = 1000 ticks)
```

```
32.
33.         //TimeOnly examples
34.         Console.WriteLine(new TimeOnly(0, 0, 1, 1, 1).Ticks); //10010010 (1 sec
    + 1 milli sec + 1 micro sec = 10^7 + 10^4 + 10 ticks)
35.         Console.WriteLine(TimeOnly.Parse($"{tm1}").Microsecond); //100 (1000
    ticks = 100 micro sec)
36.         Console.WriteLine(TimeOnly.Parse($"{tm2}").Nanosecond); //900 (9 ticks =
    900 nano sec)
37.     }
38. }
```

The preceding snippet shows how to use the new **`Microsecond`** and **`Nanosecond`** properties in the date and time related structures. Also notice the new constructors that accept the microsecond parameter (lines 14, 20, 28 and 34) and the new related methods **`AddMicroseconds`** and **`FromMicroseconds`** (lines 17, 23 and 31).

Newlines in string interpolation expressions

Starting C# 11, the interpolated expressions can include newlines. The text between the curly braces **{}** must be valid C# expression including newlines to improve readability.

```
namespace _01_NewFeatures.Features;

internal class NewlineInStringInterpolation
{
    public void Demo()
    {
        // Example 1 - switch expression
        int statusCode = 302;
        string message = $"HTTP status code {statusCode} is for {statusCode switch
        {
            > 599 or < 100 => "Invalid",
            > 499 => "Server error",
            > 399 => "Client error",
            > 299 => "Redirection",
            > 199 => "Successful",
            > 99 => "Informational response"
        }}";
       Console.WriteLine(message); //Output: HTTP status code 302 is for Redirection

        // Example 2 - LINQ query
        int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        Console.WriteLine($"The even values of {nameof(numbers)} are {string.Join(", ",
            numbers.Where(n => n % 2 == 0))}."); //Output: The even values of numbers are 2, 4, 6, 8, 10.
    }
}
```

The preceding code shows examples that use **`switch`** statement and LINQ query spread across multiple lines within a string interpolation expression.

Pattern match Span<char> on a constant string

To encourage adoption of **`ReadOnlySpan<char>`** and **`Span<char>`** (for performance reasons) pattern matching these with a constant string is allowed in C# 11.

```
1.  namespace _01_NewFeatures.Features;
2.
3.  internal class PatternMatchingWithSpanChar
4.  {
5.      public void Demo()
6.      {
7.          var readOnlySpan = "Keep it simple!".AsSpan();
8.          if (readOnlySpan is "Keep it simple!")
9.          {
10.             Console.WriteLine("Simplicity!");
11.         }
12.
13.         Span<char> spanChar = new Span<char>(new char[] { 'a', 'b', 'c' });
14.         if (spanChar is "abc")
15.         {
16.             Console.WriteLine("Alphabets!");
17.         }
18.     }
19. }
```

Line 14 in the preceding code demonstrates the **`is`** pattern matching for a **`Span<char>`** with a string literal.

Raw string literals

C# 11 allows a new form of string literal that starts with a minimum of three double-quote characters **"""** followed by the content of the string, and then ends with the same number of quotes that the literal started with. The content can be any arbitrary text including whitespaces, new lines, embedded quotes, and other special characters without requiring escape sequences. Single line raw string literals require the opening and closing quote characters on the same line. Multi-line raw string literals require both opening and closing quote characters on their own line. The newlines following the opening quote and preceding the closing quote are not included in the final content.

If the text contains three (or more) repeated double quotes, then four (or more) double quotes must be used to escape them. Raw string literals can be combined with string interpolation to include braces in the output text. Multiple **$** characters denote how many consecutive braces start and end the interpolation.

```
namespace _01_NewFeatures.Features;

internal class RawStringLiterals
{
    public void Demo()
    {
        //Example 1: string with quote, newline and tab
        var xml = """
          <element attr="content">
            <body>
            </body>
          </element>
          """;
        Console.WriteLine(xml);

        //Example 2: string with four quotes
        var line = """""
                This line needs four quotes """" so
                    should terminate with five.
          """"";
        Console.WriteLine(line);

        //Example 3: string interpolation
        string key = "100", value = "C#";
        string jsonString =
            $$"""
            {
                "Key": {{key}},
                "Value": {{value}}
            }
        """;
        Console.WriteLine(jsonString);
    }
}
```

Notice in the preceding three examples, the opening and the closing quote are on their own separate line, but the newline is ignored in the output. The white spaces in the text between is however preserved in the output.

Regex source generation (.NET 7)

The .NET 7 SDK includes a source generator that recognizes the new **`GeneratedRegexAttribute`** on a partial method that returns Regex. During compilation source code of regular expression is generated using the **Roslyn Source Generator**. The source that is emitted is part of the project in a file `RegexGenerator.g.cs` under `System.Text.RegularExpressions.Generator` within `Analyzers` and is easily viewable and debuggable. This code is optimized and provides an implementation of the method that implements all the logic for the Regex.

```
using System.Text.RegularExpressions;
namespace _01_NewFeatures.Features;

internal partial class RegexSourceGenerator
{
    private static readonly Regex VowelRegex =
    new(pattern: "[aeiou]",
        options: RegexOptions.Compiled | RegexOptions.IgnoreCase);

    [GeneratedRegex("[aeiou]", RegexOptions.IgnoreCase | RegexOptions.Compiled, "en-US")]
    private static partial Regex VowelGeneratedRegex();

    private string IsVowel(char c)
    {
        var d = VowelRegex.IsMatch(c.ToString()) ? "" : " not";
        return $"VowelRegex: {c} is{d} vowel";
    }

    private string IsVowelGeneratedRegex(char c)
    {
        var d = VowelGeneratedRegex().IsMatch(c.ToString()) ? "" : " not";
        return $"VowelGeneratedRegex: {c} is{d} vowel";
    }

    public void Demo()
    {
        Console.WriteLine(IsVowel('A')); //VowelRegex: A is vowel
        Console.WriteLine(IsVowelGeneratedRegex('e')); //VowelGeneratedRegex: e is vowel

        Console.WriteLine(IsVowel('c')); //VowelRegex: c is not vowel
        Console.WriteLine(IsVowelGeneratedRegex('S')); //VowelGeneratedRegex: S is not vowel

        Console.WriteLine(IsVowel('o')); //VowelRegex: o is vowel
        Console.WriteLine(IsVowelGeneratedRegex('I')); //VowelGeneratedRegex: I is vowel
    }
}
```

The preceding code uses two different methods to check for vowels. The method **`IsVowel`** uses the traditional regular expression **`VowelRegex`** whereas **`IsVowelGeneratedRegex`** uses the source generated regular expression **`VowelGeneratedRegex`**. Both ways are equivalent, but the latter is more optimized.

Required members

Starting C# 11 the required modifier allows to mark fields and properties that must be initialized while declaring a type - **`class`**, **`struct`**, and **`record`** (but not **`interface`** types). Any expression that initializes a new instance of the type must initialize all the required members without this there is compilation error.

The required members in a type form a *required member list* which must be initialized during the construction and initialization of an instance of the type. This list is inherited by the derived types automatically that removes repetitive code.

This modifier cannot be applied on **`static`**, **`private`** or **`read-only`** members. Additionally, the **`SetsRequiredMembers`** attribute added to a constructor asserts to the compiler that the constructor does initialize all required members.

```
1.  using System.Diagnostics.CodeAnalysis;
2.
3.  namespace _01_NewFeatures.Features;
4.
5.  internal class RequiredMembers
6.  {
7.      public void Demo()
8.      {
9.          //valid
10.         var suzuki = new Car { Make = "Suzuki", Model = string.Empty, Color =
    "Blue", Year = 2022 };
11.         Car toyota = new("", "Camry");
12.
13.         //compilation error due to missing required members
14.         Car car = new();
15.         var someCar = new Car { Model = string.Empty, Color = "Red", Year = 2020
    };
16.     }
17. }
18. public class Car
19. {
20.     public Car() { }
21.
22.     [SetsRequiredMembers]
23.     public Car(string make, string model)
24.     {
25.         Make = make;
26.         Model = model;
27.     }
28.
29.     public required string Make { get; init; }
30.     public required string Model { get; set; }
31.     public string? Color { get; set; }
32.     public int? Year { get; set; }
33. }
```

In the preceding code, the **`Car`** class has two required properties **`Make`** and **`Model`**. Not initializing these properties results in compilation error on line numbers 14 and 15.

Simplified ordering with System.LINQ (.NET 7)

`System.Linq` now has the methods **`Order`** and **`OrderDescending`**, which are there to order an **`IEnumerable`** (and **`IQueryable`**) according to **`T`**.

```csharp
namespace _01_NewFeatures.Features;

internal class SimplifiedOrdering
{
    public void Demo()
    {
        var data = new[] { 2, 1, 3, 0, -10, 25, 8, -4, 22 };

        //Existing way (still supported)
        var sorted = data.OrderBy(e => e);
        var sortedDesc = data.OrderByDescending(e => e);

        //New simplified way
        var sortedSimplified = data.Order();
        var sortedDescSimplified = data.OrderDescending();
    }
}
```

The preceding code with the list of integer values is sorted using the existing way which requires passing the lambda expression **`e => e`** and the newer simplified way which sorts by the generic type **`T`**.

Stream reading - ReadExactly and ReadAtLeast (.NET 7)

There may be occasions when **`Stream.Read()`** may return less data than what is available in the stream or size of the buffer being passed. There are two new methods in **`System.IO.Stream`** class to help with such a situation. The new **`ReadExactly`** method is guaranteed to read exactly the number of bytes requested. If the stream ends before the requested bytes have been read, an **`EndOfStreamException`** is thrown. The new **`ReadAtLeast`** method will read at least the number of bytes requested. It can read more if more data is readily available, up to the size of the buffer. If the stream ends before the requested bytes have been read, an **`EndOfStreamException`** is thrown. Here, it is possible to opt out of throwing the exception to manage the end-of-stream scenario.

```csharp
namespace _01_NewFeatures.Features;

```