**Csharp 08**

https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8

C# 8.0 adds the following features and enhancements to the C# language:

Readonly members

Default interface methods

Pattern matching enhancements:

Switch expressions

Property patterns

Tuple patterns

Positional patterns

Using declarations

Static local functions

Disposable ref structs

Nullable reference types

Asynchronous streams

Asynchronous disposable

Indices and ranges

Null-coalescing assignment

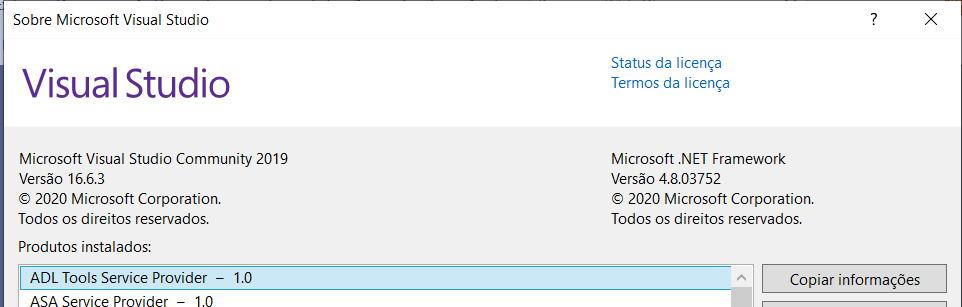
Unmanaged constructed types

Stackalloc in nested expressions

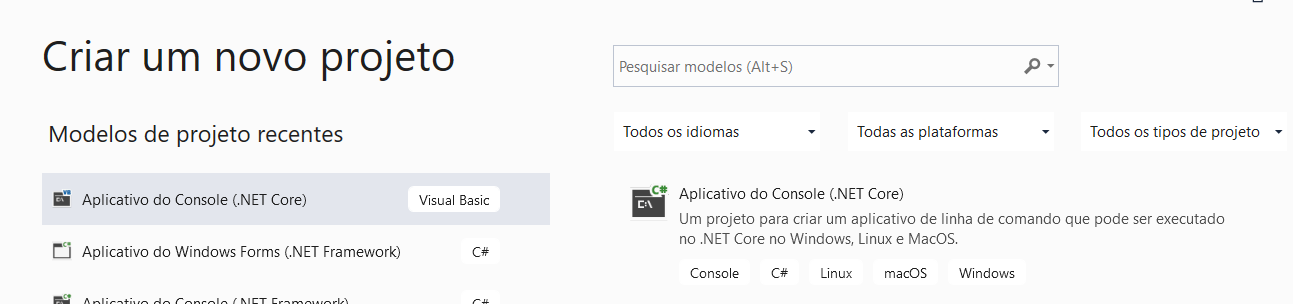
Enhancement of interpolated verbatim strings

## **HABILITANDO O C# 8**

A primeira etapa é garantir que estejamos usando o Visual Studio 2019 versão 16.3 ou superior. Verifique no menu Ajuda > Informações sobre o Visual Studio.



Em seguida, precisamos configurar o projeto para o C# 8. Se estivermos acostumados a trabalhar com o Visual Studio, talvez espere alterar de maneira simples a configuração de um projeto. Infelizmente, isso não funciona mais. Sob as novas regras, a versão padrão do C# é determinada por qual framework estamos direcionando. Somente o .NET Core 3.0 e o .NET Standard 2.1 possuem o C# 8, o restante começa com o C# 7.3. Crie um projeto console Core 3.0:



Podemos solucionar isso, editando o arquivo do projeto. Vamos abrir o arquivo .csproj do projeto e adicionar esta linha dentro do PropertyGroup.

<LangVersion>8.0</LangVersion>

Se estivermos usando um formato de projeto moderno, podemos abri-lo clicando duas vezes no projeto no Solution Explorer. Podemos reconhecer esse formato porque a raiz do arquivo XML é parecida com esta:

<Project Sdk="Microsoft.NET.Sdk">

Se estivermos usando o formato do projeto legado, podemos editá-lo diretamente, mas será um pouco mais complicado. Uma opção é fechar o Visual Studio e usar o bloco de notas ou algum editor de texto. Como alternativa, podemos instalar o Power Commands do Visual Studio, que adiciona um comando "Edit Project File". Para referência, a raiz do arquivo XML será mais ou menos assim:

<Project ToolsVersion="14.0" DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

**MEMBROS SOMENTE LEITURA**

Você pode aplicar o modificador readonlya membros de uma estrutura. Indica que o membro não modifica o estado. É mais granular do que aplicar o modificador readonly a uma declaração struct. Como a maioria dos structs, o método ToString()não modifica o estado. Você pode indicar isso adicionando o modificador readonly à declaração de ToString():

public struct Point

{

public double X { get; set; }

public double Y { get; set; }

public readonly double Distance => Math.Sqrt(X \* X + Y \* Y);

public readonly override string ToString() =>

$"({X}, {Y}) is {Distance} from the origin";

}

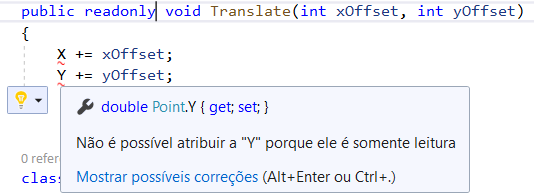
A alteração anterior gera um aviso do compilador, porque ToString acessa a propriedade Distance, que não está marcada readonly:

warning CS8656: Call to non-readonly member 'Point.Distance.get' from a 'readonly' member results in an implicit copy of 'this'

O compilador avisa quando é necessário criar uma cópia defensiva. A propriedade Distance não muda de estado, então você pode corrigir esse aviso adicionando o modificador readonly à declaração:

public readonly double Distance => Math.Sqrt(X \* X + Y \* Y)

Observe que o modificador readonly é necessário em uma propriedade somente leitura. O compilador não assume que os acessadores get não modificam o estado; você deve declarar readonly explicitamente. As propriedades implementadas automaticamente são uma exceção; o compilador tratará todos os getters implementados automaticamente como readonly, portanto, aqui não há necessidade de adicionar o readonly às modificador propriedades X e Y. O compilador impõe a regra de que os membros readonly não modificam o estado. O método a seguir não será compilado a menos que você remova o readonly modificador:



Ao retirar modificador readonly do método, podemos executar a Struct:

static void Main(string[] args)

{

var ponto = new Point();

ponto.X = 10;

ponto.Y = 20;

Console.WriteLine(ponto.ToString());

//(10, 20) is 22,360679774997898 from the origin

ponto.Translate(2, 8);

Console.WriteLine(ponto.ToString());

//(12, 28) is 30,463092423455635 from the origin

}

Este recurso permite que você especifique a intenção do projeto para que o compilador possa aplicá-la e fazer otimizações com base nessa intenção.

**MÉTODOS DE INTERFACE PADRÃO**

Agora você pode adicionar membros às interfaces e fornecer uma implementação para esses membros. Este recurso de linguagem permite que os autores da API adicionem métodos a uma interface em versões posteriores sem quebrar a compatibilidade de origem ou binária com as implementações existentes dessa interface. As implementações existentes herdam a implementação padrão. Esse recurso também permite que o C# interopere com APIs voltadas para Android ou Swift, que oferecem suporte a recursos semelhantes. Os métodos de interface padrão também permitem cenários semelhantes a um recurso de linguagem de "características".

Os métodos de interface padrão afetam muitos cenários e elementos de linguagem. Por esse tutorial (<https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/default-interface-methods-versions>) você pode definir uma implementação ao declarar um membro de uma interface. O cenário mais comum é adicionar membros com segurança a uma interface já lançada e usada por inúmeros clientes. Neste tutorial, você aprenderá como:

* Estenda interfaces com segurança adicionando métodos com implementações.
* Crie implementações parametrizadas para fornecer maior flexibilidade.
* Permita que os implementadores forneçam uma implementação mais específica na forma de uma substituição.

Uma empresa que construiu uma biblioteca pretendia que os clientes com aplicativos existentes adotassem sua biblioteca. Eles forneceram definições mínimas de interface para os usuários de sua biblioteca implementarem. Esta é a definição de interface para um cliente:

public interface ICustomer

{

IEnumerable<IOrder> PreviousOrders { get; }

DateTime DateJoined { get; }

DateTime? LastOrder { get; }

string Name { get; }

IDictionary<DateTime, string> Reminders { get; }

}

Eles definiram uma segunda interface que representa um pedido:

public interface IOrder

{

DateTime Purchased { get; }

decimal Cost { get; }

}

A partir dessas interfaces, a equipe poderia construir uma biblioteca para seus usuários, a fim de criar uma experiência melhor para seus clientes. Depois de um tempo, chega a hora de atualizar a biblioteca para a próxima versão.

Um dos recursos solicitados permite um desconto de fidelidade para clientes que têm muitos pedidos. Este novo desconto de fidelidade é aplicado sempre que um cliente faz um pedido. O desconto específico é propriedade de cada cliente individual. Cada implementação de ICustomer pode definir regras diferentes para o desconto de fidelidade. A maneira mais natural de adicionar essa funcionalidade é aprimorar a interface ICustomer com um método para aplicar qualquer desconto de fidelidade. Esta sugestão de design causou preocupação entre os desenvolvedores experientes: "As interfaces são imutáveis depois de serem lançadas! Esta é uma alteração “breaking change”!"

Para que isso não ocorra mais, o C# 8.0 adiciona implementações de interface padrão para atualização de interfaces. Os autores da biblioteca podem adicionar novos membros à interface e fornecer uma implementação padrão para esses membros. As implementações de interface padrão permitem que os desenvolvedores atualizem uma interface enquanto ainda permitem que qualquer implementador substitua essa implementação. Os usuários da biblioteca podem aceitar a implementação padrão como uma alteração ininterrupta. Se suas regras de negócios forem diferentes, eles podem ser substituidas.

**Adicionar Métodos à Interface Padrão**

A equipe concordou com a implementação padrão mais provável: um desconto de fidelidade para clientes. A atualização deve fornecer a funcionalidade para definir duas propriedades:

* o número de pedidos necessários para ter direito ao desconto
* a porcentagem do desconto.

Isso o torna um cenário perfeito para métodos de interface padrão. Você pode adicionar um método à ICustomerinterface e fornecer a implementação mais provável. Todas as implementações existentes e quaisquer novas podem usar a implementação padrão ou fornecer a sua própria. Primeiro, adicione o novo método à interface ICustomer, incluindo o corpo do método:

// Version 1 : HARD CODE

public decimal ComputeLoyaltyDiscount()

{

DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);

if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 1))

{

return 0.10m;

}

return 0;

}

O autor da biblioteca escreveu um primeiro teste para verificar a implementação:

static void Main(string[] args)

{

SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))

{

Reminders =

{

{ new DateTime(2010, 08, 12), "childs's birthday" },

{ new DateTime(1012, 11, 15), "anniversary" }

}

};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);

c.AddOrder(o);

o = new SampleOrder(new DateTime(2103, 7, 4), 25m);

c.AddOrder(o);

// Check the discount:

ICustomer theCustomer = c;

Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}"); // 0,10

}

O Cast de SampleCustomer para ICustomer é necessário. A classe SampleCustomer não precisa fornecer uma implementação para ComputeLoyaltyDiscount; que é fornecido pela interface ICustomer.

public class SampleCustomer : ICustomer

{

public SampleCustomer(string name, DateTime dateJoined) =>

(Name, DateJoined) = (name, dateJoined);

private List<IOrder> allOrders = new List<IOrder>();

public IEnumerable<IOrder> PreviousOrders => allOrders;

public DateTime DateJoined { get; }

public DateTime? LastOrder { get; private set; }

public string Name { get; }

private Dictionary<DateTime, string> reminders = new Dictionary<DateTime, string>();

public IDictionary<DateTime, string> Reminders => reminders;

public void AddOrder(IOrder order)

{

if (order.Purchased > (LastOrder ?? DateTime.MinValue))

LastOrder = order.Purchased;

allOrders.Add(order);

}

}

public class SampleOrder : IOrder

{

public SampleOrder(DateTime purchase, decimal cost) =>

(Purchased, Cost) = (purchase, cost);

public DateTime Purchased { get; }

public decimal Cost { get; }

}

No entanto, a classe SampleCustomer não herda membros de suas interfaces. Essa regra não mudou. Para chamar qualquer método declarado e implementado na interface, a variável deve ser o tipo da interface, ICustomer neste exemplo.

**Fornecer Parâmetros à Interface Padrão**

Porém, a implementação padrão é muito restritiva, pois os consumidores desse sistema gostariam de escolher limites diferentes para o número de compras, um período de assinatura diferente ou um desconto percentual diferente. O ideal seria ser possível fornecer uma maneira de definir esses parâmetros. Vamos adicionar um método estático que define esses três parâmetros que controlam a implementação padrão:

// Version 2: PARAMETERS !!!

public static void SetLoyaltyThresholds(

TimeSpan ago,

int minimumOrders = 1,

decimal percentageDiscount = 0.10m)

{

length = ago;

orderCount = minimumOrders;

discountPercent = percentageDiscount;

}

private static TimeSpan length = new TimeSpan(365 \* 2, 0, 0, 0); // two years

private static int orderCount = 1;

private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()

{

DateTime start = DateTime.Now - length;

if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))

{

return discountPercent;

}

return 0;

}

Existem muitos novos recursos de linguagem mostrados nesse pequeno fragmento de código. As interfaces agora podem incluir membros estáticos, incluindo campos e métodos. Diferentes modificadores de acesso também estão ativados. Os campos adicionais são privados, o novo método é público. Qualquer um dos modificadores é permitido em membros da interface.

Os aplicativos que usam a fórmula geral para calcular o desconto de fidelidade, mas parâmetros diferentes, não precisam fornecer uma implementação personalizada; eles podem definir os argumentos por meio de um método estático. Por exemplo, o código a seguir define uma "apreciação do cliente" que recompensa qualquer cliente com mais de um mês de assinatura:

ICustomer theCustomer = c;

ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);

Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}"); // 0,25

**Estenda a implementação padrão**

Para um recurso final, vamos refatorar o código um pouco para habilitar cenários onde os usuários podem querer construir sobre a implementação padrão. Considere uma startup que deseja atrair novos clientes. Eles oferecem um desconto de 50% no primeiro pedido de um novo cliente. Caso contrário, os clientes existentes obtêm o desconto padrão. O autor da biblioteca precisa mover a implementação padrão para um método protected static para que qualquer classe que implemente essa interface possa reutilizar o código em sua implementação. A implementação padrão do membro da interface também chama este método compartilhado:

// Version 3: EXTENDED DEFAULT INTERFACE !!!

public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);

protected static decimal DefaultLoyaltyDiscount(ICustomer c)

{

DateTime start = DateTime.Now - length;

if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))

{

return discountPercent;

}

return 0;

}

Em uma implementação de uma classe que implementa essa interface, a substituição pode chamar o método auxiliar estático e estender essa lógica para fornecer o desconto de "novo cliente", adicione na classe SampleCustomer:

public decimal ComputeLoyaltyDiscount()

{

if (PreviousOrders.Any() == false)

return 0.50m;

else

return ICustomer.DefaultLoyaltyDiscount(this);

}

Esses novos recursos significam que as interfaces podem ser atualizadas com segurança quando há uma implementação padrão razoável para esses novos membros. Projete interfaces cuidadosamente para expressar ideias funcionais únicas que podem ser implementadas por várias classes. Isso torna mais fácil atualizar essas definições de interface quando novos requisitos são descobertos para a mesma ideia funcional.

Pattern matching enhancements:

Switch expressions

Property patterns

Tuple patterns

Positional patterns

**EXPANDINDO CORRESPONDÊNCIA DE PATTERN**

A correspondência de Pattern fornece ferramentas para fornecer funcionalidade dependente da forma em diferentes tipos de dados relacionados. C# 7.0 introduziu a sintaxe para Pattern de tipo e Pattern constantes usando a expressão “is” e a instrução “switch”. Esses recursos representaram as primeiras etapas provisórias para suportar paradigmas de programação onde dados e funcionalidade vivem separados. À medida que a indústria avança em direção a mais microsserviços e outras arquiteturas baseadas em nuvem, outras ferramentas de linguagem são necessárias.

C# 8.0 expande esse vocabulário para que você possa usar mais expressões de Pattern em mais lugares em seu código. Considere esses recursos quando seus dados e funcionalidade forem separados. Considere a correspondência de Pattern quando seus algoritmos dependerem de um fato diferente do tipo de tempo de execução de um objeto. Essas técnicas fornecem outra maneira de expressar projetos.

Além de novos Pattern em novos lugares, C# 8.0 adiciona Pattern recursivos . O resultado de qualquer expressão de padrão é uma expressão. Um padrão recursivo é simplesmente uma expressão de padrão aplicada à saída de outra expressão de padrão.

**Expressão Switch**

Freqüentemente, expressões switch produzem um valor em cada um de seus blocos case. As novas expressões switch permitem que você use uma sintaxe de expressão mais concisa, com menos case-break e menos chaves. Como exemplo, considere o seguinte enum que lista as cores do arco-íris:

public enum Rainbow

{

Red,

Orange,

Yellow

}

public class RGBColor

{

public String ColorHex;

public RGBColor(byte R, byte G, byte B)

{

var color = System.Drawing.Color.FromArgb(R, G, B);

ColorHex = color.Name;

}

}

Se seu aplicativo definiu um tipo RGBColor que é construído a partir dos componentes R, Ge B, você poderia converter um valor Rainbow em seus valores RGB usando o seguinte método contendo uma expressão switch:

public static RGBColor FromRainbow(Rainbow colorBand) =>

colorBand switch

{

Rainbow.Red => new RGBColor(0xFF, 0x00, 0x00),

Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),

Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),

\_ => throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand)),

};

var cor = FromRainbow(Rainbow.Yellow);

Console.WriteLine("HEX:" + cor.ColorHex);

Onde:

* A variável vem antes da palavra - chave switch. A ordem diferente torna visualmente fácil distinguir a expressão switch da instrução switch.
* O elemento “case :” é substituído por “=>”. É mais conciso e intuitivo.
* O caso default é substituído por um “\_ “ .
* Os corpos são expressões, não afirmações.

**Pattern de propriedade**

O Pattern de propriedade permite que você combine as propriedades do objeto examinado. Considere um site de comércio eletrônico que deve calcular o imposto sobre vendas com base no endereço do comprador. Esse cálculo não é uma responsabilidade central de uma classe Address. Ele mudará com o tempo, provavelmente com mais frequência do que as alterações de formato de endereço. O valor do imposto sobre vendas depende da propriedade State do endereço. O método a seguir usa o Pattern de propriedade para calcular o imposto sobre vendas a partir do endereço e do preço:

public class Address

{

public string State { get; set; }

}

public static decimal ComputeSalesTax(Address location, decimal salePrice) =>

location switch

{

{ State: "WA" } => salePrice \* 0.06M,

{ State: "MN" } => salePrice \* 0.075M,

{ State: "MI" } => salePrice \* 0.05M,

// other cases removed for brevity...

\_ => 0M

};

var preco = ComputeSalesTax(new Address() { State= "MN" }, 3);

Console.WriteLine("Preço:" + preco);

A correspondência de padrões cria uma sintaxe concisa para expressar esse algoritmo.

**Pattern de tupla**

Alguns algoritmos dependem de várias entradas. Os Pattern de tupla permitem que você alterne com base em vários valores expressos como uma tupla. O código a seguir mostra uma expressão de switch para a pedra, papel e tesoura do jogo :

public static RGBColor FromRainbow(Rainbow colorBand) =>

colorBand switch

{

Rainbow.Red => new RGBColor(0xFF, 0x00, 0x00),

Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),

Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),

Rainbow.Green => new RGBColor(0x00, 0xFF, 0x00),

Rainbow.Blue => new RGBColor(0x00, 0x00, 0xFF),

Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),

Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),

\_ => throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand)),

};

var jankenpo = RockPaperScissors("paper", "scissors");

Console.WriteLine("Resultado:" + jankenpo);

As mensagens indicam o vencedor. O caso de descarte representa as três combinações de empates ou outras entradas de texto.

**Pattern** **posicionais**

Alguns tipos incluem um método Deconstruct que desconstrói suas propriedades em variáveis discretas. Quando um método Deconstruct está acessível, você pode usar Pattern posicionais para inspecionar propriedades do objeto e usar essas propriedades para um Pattern. Considere a seguinte classe Point que inclui um método Deconstruct para criar variáveis discretas para X e Y:

public class Point

{

public int X { get; }

public int Y { get; }

public Point(int x, int y) => (X, Y) = (x, y);

public void Deconstruct(out int x, out int y) =>

(x, y) = (X, Y);

}

Além disso, considere o seguinte enum que representa várias posições de um quadrante:

public enum Quadrant

{

Unknown,

Origin,

One,

Two,

Three,

Four,

OnBorder

}

O método a seguir usa o padrão posicional para extrair os valores de X e Y. Em seguida, ele usa uma cláusula when para determinar o Quadrant do ponto:

static Quadrant GetQuadrant(Point point) => point switch

{

(0, 0) => Quadrant.Origin,

var (x, y) when x > 0 && y > 0 => Quadrant.One,

var (x, y) when x < 0 && y > 0 => Quadrant.Two,

var (x, y) when x < 0 && y < 0 => Quadrant.Three,

var (x, y) when x > 0 && y < 0 => Quadrant.Four,

var (\_, \_) => Quadrant.OnBorder,

\_ => Quadrant.Unknown

};

var posicao = GetQuadrant(new Point(10, 2));

Console.WriteLine("Posição:" + posicao);

O padrão de descarte na opção anterior corresponde quando X ou Y é 0, mas não ambos. Uma expressão switch deve produzir um valor ou lançar uma exceção. Se nenhum dos casos corresponder, a expressão switch lançará uma exceção. O compilador gera um aviso para você se você não cobrir todos os casos possíveis em sua expressão switch. Você pode explorar técnicas de correspondência de padrões neste tutorial avançado sobre correspondência de padrões.

**DECLARAÇÕES using**

Uma declaração using é uma declaração de variável precedida pela palavra-chave using. Diz ao compilador que a variável que está sendo declarada deve ser descartada no final do escopo delimitador. Por exemplo, considere o seguinte código que grava um arquivo de texto:

static int WriteLinesToFile(IEnumerable<string> lines)

{

using var file = new System.IO.StreamWriter("WriteLines2.txt");

// Notice how we declare skippedLines after the using statement.

int skippedLines = 0;

foreach (string line in lines)

{

if (!line.Contains("Second"))

{

file.WriteLine(line);

}

else

{

skippedLines++;

}

}

// Notice how skippedLines is in scope here.

return skippedLines;

// file is disposed here

}

No exemplo anterior, o arquivo é descartado quando a chave de fechamento do método é atingida. Esse é o fim do escopo em que fileé declarado. Na nova sintaxe e na using clássica, o compilador gera a chamada para Dispose(). O compilador gera um erro se a expressão na instrução using não for descartável.

**FUNÇÕES LOCAIS ESTÁTICAS**

Agora você pode adicionar o modificador static às funções locais para garantir que a função local não capture (faça referência) nenhuma variável do escopo delimitador. Fazer isso gera:

"A static local function can't contain a reference to <variable>."

Considere o seguinte código. A função local LocalFunction acessa a variável y, declarada no escopo envolvente (o método M). Portanto, LocalFunction não pode ser declarado com o modificador static:

int M()

{

int y;

LocalFunction();

return y;

void LocalFunction() => y = 0;

}

O código a seguir contém uma função local estática. Pode ser estático porque não acessa nenhuma variável no escopo envolvente:

int M()

{

int y = 5;

int x = 7;

return Add(x, y);

static int Add(int left, int right) => left + right;

}

**REF STRUCTS DESCARTÁVEIS**

Um struct declarado com o modificador ref pode não implementar nenhuma interface e, portanto, não pode implementar IDisposable . Portanto, para permitir que um ref structseja descartado, ele deve ter um void método Dispose() acessível . Este recurso também se aplica as declarações readonly ref struct.

**TIPOS DE REFERÊNCIA ANULÁVEIS**

Dentro de um contexto de anotação anulável, qualquer variável de um tipo de referência é considerada um tipo de referência não anulável . Se você deseja indicar que uma variável pode ser nula, você deve anexar o nome do tipo com o ?para declarar a variável como um tipo de referência anulável .

Para tipos de referência não anuláveis, o compilador usa análise de fluxo para garantir que as variáveis locais sejam inicializadas com um valor não nulo quando declaradas. Os campos devem ser inicializados durante a construção. O compilador gera um aviso se a variável não for definida por uma chamada a qualquer um dos construtores disponíveis ou por um inicializador. Além disso, os tipos de referência não anuláveis não podem receber um valor que poderia ser nulo.

Os tipos de referência anuláveis não são verificados para garantir que não sejam atribuídos ou inicializados como nulos. No entanto, o compilador usa a análise de fluxo para garantir que qualquer variável de um tipo de referência anulável seja verificada em relação a nulo antes de ser acessada ou atribuída a um tipo de referência não anulável.

Você pode aprender mais sobre o recurso na visão geral dos tipos de referência anuláveis . Experimente você mesmo em um novo aplicativo neste tutorial de tipos de referência anuláveis . Aprenda sobre as etapas para migrar uma base de código existente para fazer uso de tipos de referência anuláveis na migração de um aplicativo para usar o tutorial de tipos de referência anuláveis .

**STREAMS ASSÍNCRONOS**

A partir do C# 8.0, você pode criar e consumir fluxos de forma assíncrona. Um método que retorna um fluxo assíncrono tem três propriedades:

* É declarado com o modificador async.
* Ele retorna um IAsyncEnumerable <T> .
* O método contém yield returninstruções para retornar elementos sucessivos no fluxo assíncrono.

Consumir um fluxo assíncrono requer que você adicione a awaitpalavra - chave antes da foreachpalavra - chave ao enumerar os elementos do fluxo. Adicionar a awaitpalavra - chave requer que o método que enumera o fluxo assíncrono seja declarado com o asyncmodificador e retorne um tipo permitido para um asyncmétodo. Normalmente, isso significa retornar uma Tarefa ou Tarefa <TResult> . Também pode ser ValueTask ou ValueTask <TResult> . Um método pode consumir e produzir um fluxo assíncrono, o que significa que ele retornaria um IAsyncEnumerable <T> . O código a seguir gera uma sequência de 0 a 19, esperando 100 ms entre a geração de cada número:

public static async System.Collections.Generic.IAsyncEnumerable<int> GenerateSequence()

{

for (int i = 0; i < 20; i++)

{

await Task.Delay(100);

yield return i;

}

}

Você enumeraria a sequência usando a await foreachinstrução:

await foreach (var number in GenerateSequence())

{

Console.WriteLine(number);

}

Você mesmo pode tentar fluxos assíncronos em nosso tutorial sobre como criar e consumir fluxos assíncronos . Por padrão, os elementos do fluxo são processados no contexto capturado. Se você deseja desativar a captura do contexto, use o método de extensão TaskAsyncEnumerableExtensions.ConfigureAwait . Para obter mais informações sobre contextos de sincronização e captura do contexto atual, consulte o artigo sobre como consumir o padrão assíncrono baseado em tarefas .

**DESCARTÁVEL ASSÍNCRONO**

A partir do C# 8.0, a linguagem oferece suporte a tipos descartáveis assíncronos que implementam a interface System.IAsyncDisposable . Você usa a await usinginstrução para trabalhar com um objeto descartável de forma assíncrona. Para obter mais informações, consulte o artigo Implementar um método DisposeAsync .

**ÍNDICES E INTERVALOS**

Índices e intervalos fornecem uma sintaxe sucinta para acessar elementos ou intervalos únicos em uma sequência.

Este suporte de idioma depende de dois novos tipos e dois novos operadores:

* System.Index representa um índice em uma sequência.
* O índice do operador final ^, que especifica que um índice é relativo ao final da sequência.
* System.Range representa um subfaixa de uma sequência.
* O operador de intervalo .., que especifica o início e o fim de um intervalo como seus operandos.

Vamos começar com as regras para índices. Considere uma matriz sequence. O índice 0 é o mesmo que sequence[0]. O índice ^0 é o mesmo que sequence[sequence.Length]. Observe que sequence[^0] lança uma exceção, assim como sequence[sequence.Length] faz. Para qualquer número n, o índice ^né o mesmo que sequence.Length - n.

Um intervalo especifica o início e o fim de um intervalo. O início do intervalo é inclusivo, mas o final do intervalo é exclusivo, o que significa que o início está incluído no intervalo, mas o final não está incluído no intervalo. O intervalo [0..^0] representa todo o intervalo, assim como [0..sequence.Length] representa todo o intervalo.

Vejamos alguns exemplos. Considere a seguinte matriz, anotada com seu índice do início e do final:

var words = new string[]

{

// index from start index from end

"The", // 0 ^9

"quick", // 1 ^8

"brown", // 2 ^7

"fox", // 3 ^6

"jumped", // 4 ^5

"over", // 5 ^4

"the", // 6 ^3

"lazy", // 7 ^2

"dog" // 8 ^1

}; // 9 (or words.Length) ^0

Você pode recuperar a última palavra com o ^1índice:

Console.WriteLine($"The last word is {words[^1]}");

// writes "dog"

O código a seguir cria um subintervalo com as palavras "rápido", "marrom" e "raposa". Inclui words[1]através words[3]. O elemento words[4]não está no intervalo.

var quickBrownFox = words[1..4];

O código a seguir cria um subintervalo com "preguiçoso" e "cachorro". Inclui words[^2]e words[^1]. O índice final words[^0]não está incluído:

var lazyDog = words[^2..^0];

Os exemplos a seguir criam intervalos que são abertos para o início, o fim ou ambos:

var allWords = words[..]; // contains "The" through "dog".

var firstPhrase = words[..4]; // contains "The" through "fox"

var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"

Você também pode declarar intervalos como variáveis:

Range phrase = 1..4;

O intervalo pode então ser usado dentro dos caracteres [e ]:

var text = words[phrase];

Não apenas as matrizes oferecem suporte a índices e intervalos. Você também pode usar índices e intervalos com string , Span <T> ou ReadOnlySpan <T> . Para obter mais informações, consulte Suporte de tipo para índices e intervalos .

Você pode explorar mais sobre índices e intervalos no tutorial sobre índices e intervalos .

**ATRIBUIÇÃO DE COALESCÊNCIA NULA**

C# 8.0 introduz o operador de atribuição de coalescência nula ??=. Você pode usar o ??=operador para atribuir o valor de seu operando à direita a seu operando à esquerda somente se o operando à esquerda for avaliado como null.

List<int> numbers = null;

int? i = null;

numbers ??= new List<int>();

numbers.Add(i ??= 17);

numbers.Add(i ??= 20);

Console.WriteLine(string.Join(" ", numbers)); // output: 17 17

Console.WriteLine(i); // output: 17

Para obter mais informações, consulte o ?? e ?? = artigo de operadores .

**TIPOS CONSTRUÍDOS NÃO GERENCIADOS**

No C# 7.3 e anteriores, um tipo construído (um tipo que inclui pelo menos um argumento de tipo) não pode ser um tipo não gerenciado . A partir do C# 8.0, um tipo de valor construído não é gerenciado se contiver campos apenas de tipos não gerenciados. Por exemplo, dada a seguinte definição do Coords<T>tipo genérico :

public struct Coords<T>

{

public T X;

public T Y;

}

o Coords<int>tipo é um tipo não gerenciado em C# 8.0 e posterior. Como para qualquer tipo não gerenciado, você pode criar um ponteiro para uma variável desse tipo ou alocar um bloco de memória na pilha para instâncias desse tipo:

Span<Coords<int>> coordinates = stackalloc[]

{

new Coords<int> { X = 0, Y = 0 },

new Coords<int> { X = 0, Y = 3 },

new Coords<int> { X = 4, Y = 0 }

};

Para obter mais informações, consulte Tipos não gerenciados .

**STACKALLOC EM EXPRESSÕES ANINHADAS**

A partir do C# 8.0, se o resultado de uma expressão stackalloc for do tipo System.Span <T> ou System.ReadOnlySpan <T> , você poderá usar a stackallocexpressão em outras expressões:

Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };

var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });

Console.WriteLine(ind); // output: 1

**APRIMORAMENTO DE STRINGS TEXTUAIS INTERPOLADAS**

A ordem dos tokens $e @em strings textuais interpoladas pode ser qualquer: ambos $@"..."e @$"..."são strings textuais interpoladas válidas. Em versões anteriores do C#, o $token deve aparecer antes do @token.