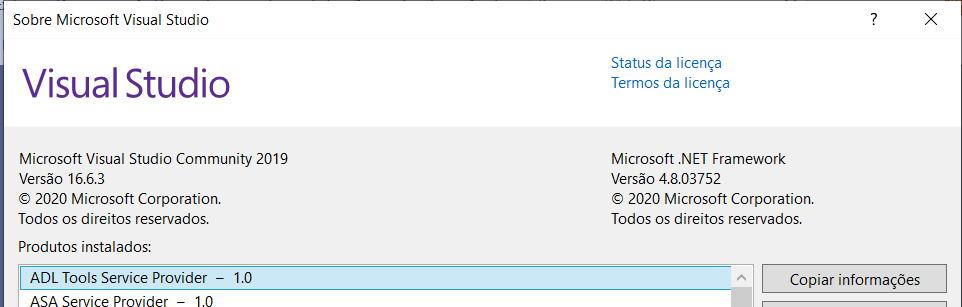
**Csharp 08**

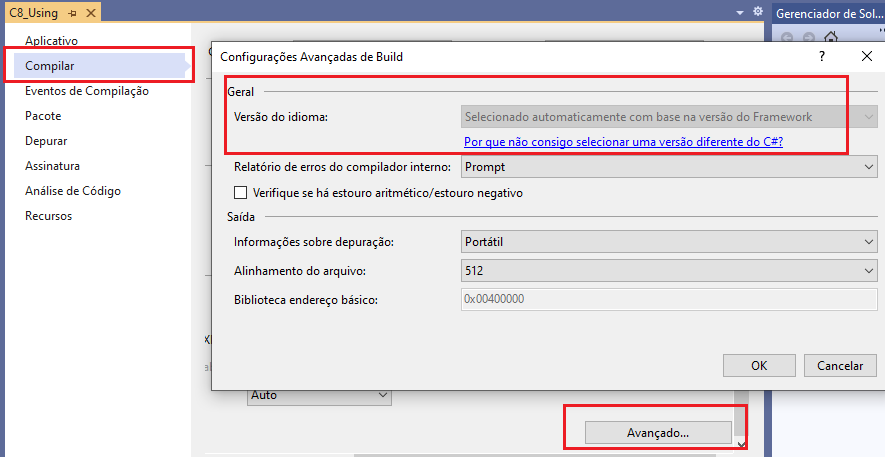
https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8

## **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. Antes era só ir para Compilar da janela de propriedades do projeto e selecionar a versão do idioma C# :



Infelizmente, isso não funciona mais. Sob as novas regras, a versão padrão do C# é determinada por qual framework estamos direcionando. Os compiladores do C# que fazem parte da instalação do Visual Studio 2017 ou de versões anteriores do SDK do .NET Core são direcionados ao C# 7.0 por padrão. O C# 8,0 tem suporte apenas no .NET Core 3. x e em versões mais recentes. Muitos dos recursos mais recentes exigem recursos de biblioteca e tempo de execução introduzidos no .NET Core 3. x:

* A implementação de interface padrão requer novos recursos no .net Core 3,0 CLR.
* Os fluxos assíncronos exigem os novos tipos:
  + System.IAsyncDisposable,
  + System.Collections.Generic.IAsyncEnumerable<T>
  + System.Collections.Generic.IAsyncEnumerator<T> .
* Índices e intervalos exigem os novos tipos System.Index e System.Range .
* Os tipos de referência anuláveis fazem uso de vários atributos para fornecer avisos melhores. Esses atributos foram adicionados no .NET Core 3,0. Outras estruturas de destino não foram anotadas com nenhum desses atributos. Isso significa que os avisos anuláveis podem não refletir com precisão possíveis problemas.

O C# 9,0 tem suporte apenas no .NET 5 e em versões mais recentes.

**Padrões do compilador**

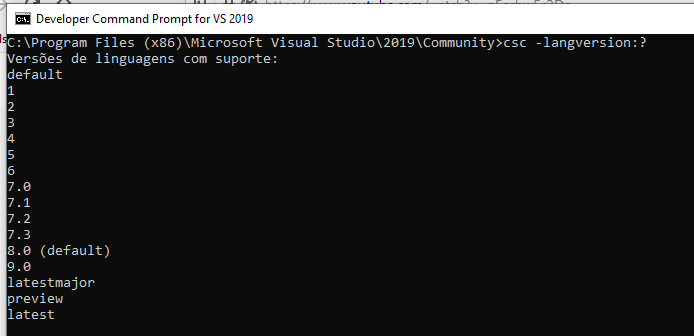
O compilador determina um padrão com base nestas regras:

|  |  |  |
| --- | --- | --- |
| **Estrutura de destino** | **version** | **Padrão da versão da linguagem C#** |
| .NET | 5 | C# 9,0 |
| .NET Core | 3.x | C# 8.0 |
| .NET Core | 2. x | C# 7.3 |
| .NET Standard | 2.1 | C# 8.0 |
| .NET Standard | 2.0 | C# 7.3 |
| .NET Standard | 1.x | C# 7.3 |
| .NET Framework | all | C# 7.3 |

Abra o prompt de comando do desenvolvedor para o Visual Studioe execute o comando a seguir para ver a lista de versões de idioma disponíveis em seu computador.

**csc -langversion:?**

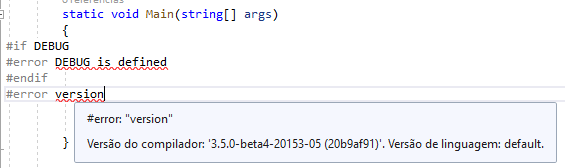
Questionando a opção de compilação -langversion como essa, imprimirá algo semelhante ao seguinte:



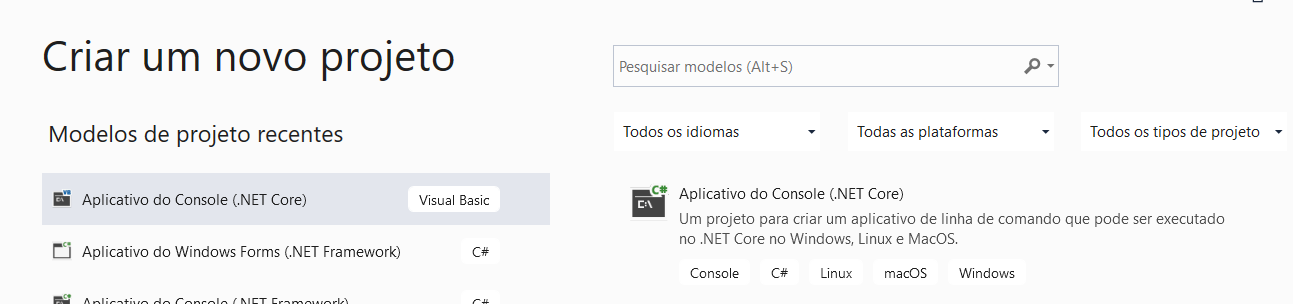
A tabela a seguir mostra todas as versões atuais da linguagem C#. Seu compilador pode não entender necessariamente todos os valores se for mais antigo. Se você instalar o .NET Core 3,0 ou posterior, terá acesso a todos os itens listados.

|  |  |
| --- | --- |
| **Valor** | **Significado** |
| preview | O compilador aceita todas as sintaxes de linguagem válidas da versão prévia mais recente. |
| latest | O compilador aceita a sintaxe da versão lançada mais recente do compilador (incluindo a versão secundária). |
| latestMajor (default) | O compilador aceita a sintaxe da versão principal mais recente lançada do compilador. |
| 8.0 | O compilador aceita somente a sintaxe incluída no C# 8.0 ou inferior. |
| 7.3 | O compilador aceita somente a sintaxe incluída no C# 7.3 ou inferior. |
| 7.2 | O compilador aceita somente a sintaxe incluída no C# 7.2 ou inferior. |
| 7.1 | O compilador aceita somente a sintaxe incluída no C# 7.1 ou inferior. |
| 7 | O compilador aceita somente a sintaxe incluída no C# 7.0 ou inferior. |
| 6 | O compilador aceita somente a sintaxe incluída no C# 6.0 ou inferior. |
| 5 | O compilador aceita somente a sintaxe incluída no C# 5.0 ou inferior. |
| 4 | O compilador aceita somente a sintaxe incluída no C# 4.0 ou inferior. |
| 3 | O compilador aceita somente a sintaxe incluída no C# 3.0 ou inferior. |
| ISO-2(ou 2) | O compilador aceita apenas a sintaxe que está incluída em ISO/IEC 23270:2006 C# (2,0). |

Para saber qual versão de idioma você está usando no momento, coloque #error em seu código. Isso faz com que o compilador produza um diagnóstico, CS8304, com uma mensagem que contém a versão do compilador que está sendo usada e a versão atual do idioma selecionado.



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:



**Substituir um padrão**

Se precisar especificar sua versão do C# explicitamente, poderá fazer isso de várias maneiras:

1. **Edite manualmente o arquivo de projeto (**csproj)**.**
2. **Definir a versão da linguagem para vários projetos em um subdiretório.**
3. **Configure a -langversion opção do compilador.**

**Editar o arquivo de projeto**

É possível definir a versão da linguagem em seu arquivo de projeto. Por exemplo, se você quiser explicitamente acesso às versões prévias dos recursos, adicione um elemento como este:

<PropertyGroup>

<LangVersion>preview</LangVersion> OU <LangVersion>8.0</LangVersion>

</PropertyGroup>

O valor preview usa a versão prévia mais recente da linguagem C# compatível com seu compilador. 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">

**Configurar vários projetos**

Para configurar vários projetos, você pode criar um arquivo Directory. Build. props que contém o <LangVersion> elemento. Normalmente, você faz isso no diretório da solução. Adicione o seguinte a um arquivo Directory. Build. props no diretório da solução:

<Project>

<PropertyGroup>

<LangVersion>preview</LangVersion>

</PropertyGroup>

</Project>

As compilações em todos os subdiretórios do diretório que contém esse arquivo usarão a versão preview C#. Para obter mais informações, confira o artigo sobre como personalizar o build. C # 8.0 adiciona os seguintes recursos e aprimoramentos à linguagem C #:

|  |  |
| --- | --- |
| Membros somente leitura | Readonly members |
| Métodos de interface padrão | Default interface methods |
| * Adicionar Métodos à Interface Padrão | * Add Methods to the Standard Interface |
| * Fornecer Parâmetros à Interface Padrão | * Provide Parameters to the Standard Interface |
| * Estenda a implementação padrão | * Extend the standard implementation |
| Aprimoramentos de correspondência de padrões | Pattern matching enhancements |
| * expressões Switch | * Switch expressions |
| * Padrões de propriedade | * Property patterns |
| * Padrões de tupla | * Tuple patterns |
| * Padrões posicionais | * Positional patterns |
| Declarações Using | Using declarations |
| Funções locais estáticas | Static local functions |
| Ref structs descartáveis | Disposable ref structs |
| Tipos de referência anuláveis | Nullable reference types |
| * Escolher uma estratégia para tipos de referência anuláveis | * Choose a strategy for nullable reference types |
| Streams assíncronos | Asynchronous streams |
| * CanceledToken | * CanceledToken |
| Descartável assíncrono | Asynchronous disposable |
| Índices e intervalos | Indices and ranges |
| Atribuição de coalescência nula | Null-coalescing assignment |
| * Tipos anuláveis | * Nullable types |
| * Operador de Coalescência Nula ?? | * Null Coalescence Operator ?? |
| Tipos construídos não gerenciados | Unmanaged constructed types |
| Stackalloc em expressões aninhadas | Stackalloc in nested expressions |
| * System.Span<T> ou System.ReadOnlySpan<T> | * System.Span <T> or System.ReadOnlySpan <T> |
| * Por um tipo ponteiro | * By a pointer type |
| Aprimoramento de strings textuais interpoladas | Enhancement of interpolated verbatim strings |

**MEMBROS SOMENTE LEITURA**

Você pode aplicar o modificador readonly a 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";

}

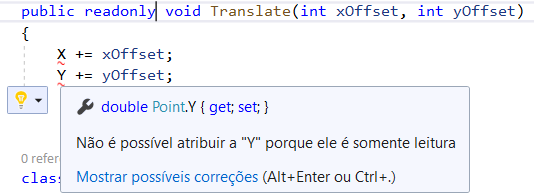
A alteração anterior gera um aviso do compilador, porque ToString acessa a propriedade Distance, que não está marcada readonly:

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 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 modificador readonly:



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.

**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

}

IEnumerable<string> m\_oEnum = new List<string>() { "1", "2", "3" };

IEnumerable<string> m\_oArray = new string[] { "1", "2", "3" };

IEnumerable<string> myStrings = new[] { "first item", "Second item" };

WriteLinesToFile(myStrings);

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:

public 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:

public int M\_Est()

{

int y = 5;

int x = 7;

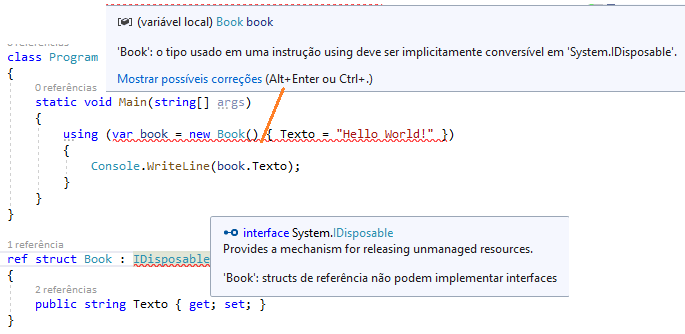
return Add(x, y);

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

}

**REF STRUCTS DESCARTÁVEIS**

Os Ref structs foram introduzidos no C # 7.2, e tinham algumas limitações severas, como não ser capaz de implementar interfaces. Os Ref structs não podem implementar interface porque os exporia à possibilidade de boxing. Mas por causa disso, não podemos torná-los implementando IDisposable e, portanto, não podemos usá-los nas instruções using.



Portanto, ao projetar um tipo que possui algum recurso, gostaríamos de ter alguma possibilidade de limpeza explícita. Em C#, a escolha é óbvia - temos um contrato bem conhecido na forma de interface IDisposable e seu método Dispose. Mas como já foi dito, até C# 8.0 ele não podia ser usado em estruturas Ref. As estruturas Ref agora podem ser descartáveis sem implementar a interface IDisposable. Para permitir que um ref struct seja descartado, ele deve ter um void método público Dispose(). Este recurso também se aplica as declarações readonly ref struct. Além disso, devido às mudanças na própria instrução using, agora podemos usar uma forma mais concisa de usar a instrução using:

class Program

{

static void Main(string[] args)

{

using (var book = new Book() { Texto = "Hello World!" })

Console.WriteLine(book.Texto);

}

}

ref struct Book

{

public string Texto { get; set; }

public void Dispose()

{

}

}

Poderíamos ter introduzido o método Dispose para limpá-lo no final do uso, assim:

var book1 = new Book() { Texto = "Hello World!" };

Console.WriteLine(book1.Texto);

book1.Dispose();

Isso é obviamente complicado, pois precisamos nos lembrar de como chamar Dispose. E o tratamento de exceções não é tratado adequadamente aqui. É por isso que using statement foi introduzido, para ter certeza de que será chamado por baixo.

Em geral a limpeza explícita (finalização determinística) é preferida em vez de implícita (finalização não determinística). Isso é de alguma forma intuitivo. É melhor fazer uma limpeza explicitamente assim que possível (chamando Close, Dispose ou usando a instrução), em vez de esperar pela limpeza não explícita que ocorrerá “em algum momento” (pelos finalizadores de execução do tempo de execução).

**TIPOS DE REFERÊNCIA ANULÁVEIS**

Tipos de referência anuláveis ​​já foram considerados nos estágios iniciais de desenvolvimento do C # 7.0, mas foram adiados para a próxima versão principal. O objetivo deste recurso é ajudar os desenvolvedores a evitar exceções NullReferenceException não tratadas e ajudam a evitar erros de referência nula em tempo de compilação.

Para lidar com isso, a análise estática para segurança nula pode ser habilitada seletivamente com uma chave do compilador no nível do projeto. A opção é mantida como uma propriedade no arquivo de projeto. Ainda não há interface do usuário no Visual Studio 2019 para alterar seu valor. Para habilitá-lo, adicione manualmente ao primeiro elemento PropertyGroup do arquivo de projeto. csproj.

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

<PropertyGroup>

<OutputType>Exe</OutputType>

<TargetFramework>netcoreapp3.1</TargetFramework>

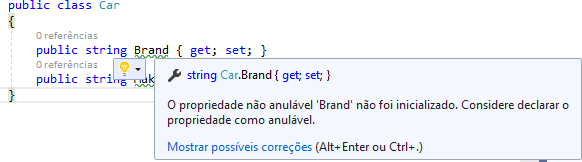
<Nullable>enable</Nullable>

</PropertyGroup>

</Project>

Definindo para um projeto usando o Nullable configura como o compilador interpreta a nulidade de tipos e quais avisos são gerados. As configurações válidas são:

|  |  |
| --- | --- |
| **Configuração** | **Descrição** |
| enable | O contexto de anotação anulável está habilitado. O contexto de aviso que permite valor nulo está habilitado.  Variáveis de um tipo de referência, string por exemplo, não permitem valor nulo. Todos os avisos de nulidade estão habilitados. |
| warnings | O contexto de anotação anulável está desabilitado. O contexto de aviso que permite valor nulo está habilitado.  As variáveis de um tipo de referência são óbvias. Todos os avisos de nulidade estão habilitados. |
| annotations | O contexto de anotação anulável está habilitado. O contexto de aviso que permite valor nulo está desabilitado.  Variáveis de um tipo de referência, String, por exemplo, são não anuláveis. Todos os avisos de nulidade estão desabilitados. |
| disable | O contexto de anotação anulável está desabilitado. O contexto de aviso que permite valor nulo está desabilitado.  As variáveis de um tipo de referência são alheias, como as versões anteriores do C#. Todos os avisos de nulidade estão desabilitados. |



Depois de adicionar essa tag, você verá dois avisos do compilador e efeitos visuais nas propriedades. Como uma alternativa mais rápida ao uso de um campo de apoio anulável, ou se a biblioteca que instancia sua classe não for compatível com essa abordagem, você pode simplesmente inicializar a propriedade como nula diretamente, com a ajuda do operador de tolerância nula (!):

public class Car

{

public string Brand { get; set; } = null!;

public string Make { get; set; } = null!;

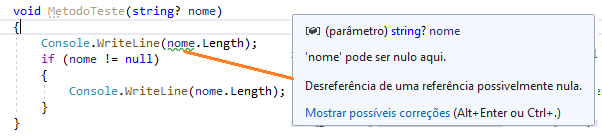
}

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 ? (Elvis operator) para declarar a variável como um tipo de referência anulável. A ideia central é permitir que as definições de tipo de variável especifiquem se podem ter um valor nulo atribuído a elas ou não:

string nome = null; Warning: Assignment of null to non-nullable reference type

string? nome = null; OK

Atribuir um valor nulo ou um valor nulo potencial a uma variável não anulável resulta em um aviso do compilador. Da mesma forma, avisos são gerados desreferenciar a referência de uma variável anulável sem verificar primeiro o valor nulo:



Uma propriedade não pode ser nula, mas o compilador ainda avisa. Então você pode dizer ao compilador que essa propriedade nunca é nula usando o ! operador.

void MetodoTeste(string? nome)

{

Console.WriteLine(nome!.Length);

}

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. Quando variáveis não devem ser nulas, o compilador impõe regras que garantem que é seguro desreferenciar essas variáveis sem primeiro verificar se ela não é nula:

* A variável deve ser inicializada para um valor não nulo.
* A variável nunca pode receber o valor null.

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. Quando variáveis puderem ser nulas, o compilador imporá diferentes regras para garantir que você verificou corretamente se há uma referência nula:

* A variável só poderá ser desreferenciada quando o compilador puder garantir que o valor não é nulo.
* Essas variáveis podem ser inicializadas com o valor null padrão e receber o valor null em outro código.

Para obter mais granularidade, as diretivas de aviso #pragma podem ser usadas para desativar e reativar avisos individuais para um bloco de código. Como alternativa, uma nova diretiva #nullable foi adicionada.

|  |  |
| --- | --- |
| **Diretivas** | **Descrição** |
| #nullable enable | Define o contexto de anotação anulável e o contexto de aviso anulável como habilitado. |
| #nullable disable | Define o contexto de anotação anulável e o contexto de aviso anulável como desabilitado. |
| #nullable restore | Restaura o contexto de anotação anulável e o contexto de aviso anulável para as configurações do projeto. |
| #nullable disable warnings | Defina o contexto de aviso anulável como desabilitado. |
| #nullable enable warnings | Defina o contexto de aviso anulável como habilitado. |
| #nullable restore warnings | Restaura o contexto de aviso anulável para as configurações do projeto. |
| #nullable disable annotations | Defina o contexto de anotação anulável como desabilitado. |
| #nullable enable annotations | Defina o contexto de anotação anulável como habilitado. |
| #nullable restore annotations | Restaura o contexto de aviso de anotação para as configurações do projeto. |

Ele pode ser usado para ativar o suporte a tipos de referência anuláveis ​​para um bloco de código, mesmo se estiver desativado no nível do projeto:

#nullable enable

string cantBeNull = null; /\* Aviso\*/

string? canBeNull = null; /\*OK\*/

cantBeNull = canBeNull; // Aviso

#nullable restore

É uma boa ideia usar #nullable restore em vez de #nullable disable para desativar os tipos de referência anuláveis ​​para o código a seguir. Isso garantirá que as verificações permaneçam habilitadas para o resto do arquivo se você decidir habilitar o recurso para todo o projeto posteriormente. Usar #nullable disable desabilitaria as verificações mesmo nesse caso.

**Escolher uma estratégia para tipos de referência anuláveis**

A primeira opção é se os tipos de referência anuláveis devem estar ativados ou desativados por padrão. Você tem duas estratégias:

* Habilite tipos de referência anuláveis para o projeto inteiro e desabilite-o no código que não está pronto.
* Só habilite tipos de referência anuláveis para o código anotado para tipos de referência anuláveis.

A primeira estratégia funciona melhor quando você está adicionando outros recursos à biblioteca ao atualizá-los para tipos de referência anuláveis. Todo o novo desenvolvimento tem reconhecimento anulável. À medida que você atualiza o código existente, você habilita os tipos de referência anuláveis nessas classes.

Seguindo essa primeira estratégia, faça o seguinte:

1. Habilite tipos de referência anuláveis para todo o projeto <Nullable>enable</Nullable> adicionando o elemento aos seus arquivos csproj .
2. Adicione o #nullable disable pragma a cada arquivo de origem em seu projeto.
3. Conforme você trabalha em cada arquivo, remova o pragma e resolva os avisos.

Essa primeira estratégia tem mais trabalho antecipado para adicionar o pragma a cada arquivo. A vantagem é que todos os novos arquivos de código adicionados ao projeto serão habilitados para permitir valor nulo. Qualquer trabalho novo terá reconhecimento de Nullable; somente o código existente deve ser atualizado.

A segunda estratégia funciona melhor se a biblioteca costuma ser estável e o foco principal do desenvolvimento é adotar tipos de referência anuláveis. Você ativa os tipos de referência anuláveis ao anotar APIs. Quando tiver terminado, habilite os tipos de referência anuláveis para o projeto inteiro.

Seguindo essa segunda estratégia, você faz o seguinte:

1. Adicione o #nullable enable pragma ao arquivo que você deseja que o reconheça de forma anulável.
2. Resolva os avisos.
3. Continue essas duas primeiras etapas até que você tenha feito o reconhecimento de toda a biblioteca.
4. Habilite tipos anuláveis para todo o projeto adicionando <Nullable>enable</Nullable> o elemento aos seus arquivos csproj .
5. Remova os #nullable enable pragmas, pois eles não são mais necessários.

Essa segunda estratégia tem menos trabalho de antecedência. A desvantagem é que a primeira tarefa quando você cria um novo arquivo é adicionar o pragma e torná-lo indesejado de forma anulável. Se algum desenvolvedor da sua equipe esquecer, esse novo código estará no registro posterior do trabalho para tornar todos os incompatíveis com o código anulável.

Qual dessas estratégias você escolhe depende de quanto o desenvolvimento ativo está ocorrendo em seu projeto. Quanto mais maduro e estável for seu projeto, melhor a segunda estratégia. Quanto mais recursos estiverem sendo desenvolvidos, melhor será a primeira estratégia.

**STREAMS ASSÍNCRONOS**

O C# 5 já era possível ter suporte para iteradores e recursos assíncronos, como aguardar um resultado com await e async. Portanto, uma armadilha comum para cair pode ser que você queira usar um tipo de retorno de Task <IEnumerable <T>> e torná-lo assíncrono. Algo assim:

public static async void EscreverForeach()

{

foreach (var dataPoint in await FetchIOTData())

{

Console.WriteLine("Foreach: " + dataPoint);

}

}

static async Task<IEnumerable<int>> FetchIOTData()

{

List<int> dataPoints = new List<int>();

for (int i = 1; i <= 10; i++)

{

await Task.Delay(1000);//Simulate waiting for data to come through.

dataPoints.Add(i);

}

return dataPoints;

}

Se rodássemos esse aplicativo, nada apareceria por 10 segundos, e então veríamos todos os nossos pontos de dados de uma vez. Não será o bloqueio de thread, ainda é assíncrono, mas não recebemos os dados assim que os recebemos. Observe que estamos percorrendo todos os resultados em um loop while, instanciando todos os objetos de produto, colocando-os em um List<int> e, por fim, retornamos tudo. Isso é bastante ineficiente, especialmente para conjuntos de dados maiores. É menos IEnumerable e mais parecido com um List<T>. O que realmente queremos é ser capaz de usar a palavra-chave yield, para retornar os dados conforme os recebemos para serem processados imediatamente.

Em vez de retornar a Task<IEnumerable<T>>, nosso método agora pode retornar IAsyncEnumerable<T>e usar yield return para emitir dados. No C# 8, em vez de retornar a Task<IEnumerable<T>>, nosso método agora pode retornar IAsyncEnumerable<T> (baseados nas interfaces IEnumerable e IEnumerator) e usar yield return para emitir dados. Os dois combinados em fluxos assíncronos podem ser muito útil se você estiver desenvolvendo um aplicativo IoT e estiver lidando com muitas chamadas assíncronas retornando dados. Com o IOT se tornando cada vez maior, faz sentido para C# adicionar uma maneira de iterar em um IEnumerable de maneira assíncrona enquanto usa a palavra-chave yield para obter dados conforme eles chegam. Por exemplo, recuperar sinais de dados de um de IOT, deseja receber dados e processá-los à medida que são recuperados, mas não de uma forma que bloqueie a CPU enquanto esperamos. Um método que retorna um fluxo assíncrono tem três propriedades:

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

public interface IAsyncEnumerable<out T>

{

IAsyncEnumerator<T> GetAsyncEnumerator();

//OU IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken cancellationToken = default);

}

public interface IAsyncEnumerator<out T> : IAsyncDisposable

{

T Current { get; }

ValueTask<bool> MoveNextAsync();

}

Além disso, uma versão assíncrona da interface IDisposable é necessária para consumir os iteradores assíncronos:

public interface IAsyncDisposable

{

ValueTask DisposeAsync();

}

Isso permite que o seguinte código seja usado para iterar os itens:

public static async void EscreverWhile()

{

var asyncEnumerator = GenerateSequence().GetAsyncEnumerator();

try

{

while (await asyncEnumerator.MoveNextAsync())

{

var value = asyncEnumerator.Current;

Console.WriteLine("While: " + value);

}

}

finally

{

await asyncEnumerator.DisposeAsync();

}

}

É muito semelhante ao código que usamos para consumir iteradores síncronos regulares. No entanto, não parece familiar porque normalmente usamos apenas a instrução foreach. Uma versão assíncrona da instrução foreach está disponível para iteradores assíncronos:

public static async void EscreverForeach()

{

await foreach (var number in GenerateSequence())

{

Console.WriteLine("Foreach: " + number);

}

}

Consumir um fluxo assíncrono requer que você adicione a palavra-chave await antes da palavra-chave foreach ao enumerar os elementos do fluxo. Adicionar a palavra-chave await requer que o método que enumera o fluxo assíncrono seja declarado com o modificador async e retorne um tipo permitido para um método async. Normalmente, isso significa retornar uma Taskou Task<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 IAsyncEnumerable<int> GenerateSequence()

{

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

{

await Task.Delay(100);

yield return i;

}

}

**CancelamentoToken**

Você deve ter notado o parâmetro CancelamentoToken do método GetAsyncEnumerator da interface IAsyncEnumerable<T>. Como seria de se esperar, ele pode ser usado para suportar o cancelamento de fluxos assíncronos. Em qualquer ponto durante a enumeração, se o cancelamento for solicitado, uma chamada MoveNextAsync subsequente ou em andamento pode ser interrompida e lançar uma OperationCanceledException (ou algum tipo derivado , como uma TaskCanceledException). Isso levanta duas questões:

* Se o token precisa ser passado para GetAsyncEnumerator, mas é o compilador que está gerando a chamada GetAsyncEnumerator para meu await foreach, como faço para passar um token?
* Se o token for passado para GetAsyncEnumerator e for o compilador que está gerando a implementação GetAsyncEnumerator para meu método iterador assíncrono, de onde obtenho o token passado?

A resposta para a primeira pergunta é que há um método de extensão WithCancellation para IAsyncEnumerable <T>. Ele aceita um CancellationToken como argumento e retorna um tipo de struct personalizado que aguarda foreach se vincula por meio de um padrão em vez de por meio da interface IAsyncEnumerable <T>, permitindo que você escreva código como o seguinte:

await foreach (var number in RangeAsync(1, 10).WithCancellation(token))

{

Console.WriteLine("Foreach: " + number);

}

Essa mesma vinculação baseada em padrão também é usada para habilitar um método ConfigureAwait, que também pode ser encadeado em um design fluente com WithCancellation:

await foreach (var number in RangeAsync(1, 10).WithCancellation(token).ConfigureAwait(false))

{

Console.WriteLine("Foreach: " + number);

}

A resposta à segunda pergunta é um novo atributo [EnumeratorCancellation]. Você pode adicionar um parâmetro CancellationToken ao método iterador assíncrono e anotá-lo com este atributo. Ao fazer isso, o compilador gerará um código que fará com que o token passado para GetAsyncEnumerator fique visível para o corpo do iterador assíncrono como esse argumento:

static async IAsyncEnumerable<int> RangeAsync(int start, int count,

[EnumeratorCancellation] CancellationToken cancellationToken = default)

{

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

{

await Task.Delay(i, cancellationToken);

yield return start + i;

}

}

Isso significa que se você escrever este código:

var cts = new CancellationTokenSource(TimeSpan.FromSeconds(0.2));

await foreach (var number in RangeAsync(1, 10).WithCancellation(cts.Token).ConfigureAwait(false))

{

Console.WriteLine("Foreach: " + number);

}

O código dentro de RangeAsync verá seu parâmetro cancellationToken igual a cts.Token. Claro, porque o token é um parâmetro normal para o método iterador, também é possível passar o token diretamente como um argumento:

await foreach (var number in RangeAsync(1, 10, cts.Token).ConfigureAwait(false))

{

Console.WriteLine("Foreach: " + number);

}

Nesse caso, o corpo do iterador assíncrono verá cts.Token como seu cancellationToken. Passar o token diretamente para o método é mais fácil, mas não funciona quando você recebe um IAsyncEnumerable <T> arbitrário de alguma outra fonte, mas ainda deseja poder solicitar o cancelamento de tudo o que o compõe. Em casos extremos, também pode ser vantajoso passar o token para GetAsyncEnumerator, pois isso evita "queimar" o token no caso em que o único enumerável será enumerado várias vezes: ao passá-lo para GetAsyncEnumerator, um token diferente pode ser passado cada vez. Claro, também é possível que dois tokens diferentes acabem sendo passados ​​para o mesmo iterador, um como um argumento para o iterador e outro via GetAsyncEnumerator. Nesse caso, o código gerado pelo compilador lida com isso criando um novo token vinculado que terá o cancelamento solicitado quando um dos dois tokens tiver o cancelamento solicitado, e esse novo token “combinado” será aquele que o corpo do iterador vê.

**DESCARTÁVEL ASSÍNCRONO**

Todo desenvolvedor .NET irá, mais cedo ou mais tarde, usar o método Dispose() (indiretamente com uma instrução using) e implementar o IDisposable. O padrão para descartar um objeto está disponível como um conceito no .NET há muito tempo, desde o .NET Framework 1.1. Considere estes dois métodos DoWorkAsync:

public static Task DoWorkAsync()

{

var arg1 = ComputeArg();

var arg2 = ComputeArg();

return AwaitableMethodAsync(arg1, arg2);

}

public static async Task DoWork2Async()

{

var arg1 = ComputeArg();

var arg2 = ComputeArg();

await AwaitableMethodAsync(arg1, arg2);

}

static public Task ComputeArg()

{

return Task.Run(() =>

{

Console.WriteLine($"{nameof(ComputeArg)} starting...");

Task.Delay(1000).Wait();

Console.WriteLine($"{nameof(ComputeArg)} ending...");

});

}

public static async Task<string> AwaitableMethodAsync(Task arg1, Task arg)

{

using (System.Net.Http.HttpClient client = new System.Net.Http.HttpClient())

{

string result = await client.GetStringAsync("http://www.microsoft.com");

return result;

}

}

O primeiro é um método síncrono que retorna uma Tarefa. A tarefa pode ou não ter sido concluída quando o método retornar. O segundo como um método assíncrono que retorna o resultado da espera de outro trabalho. Esses dois métodos parecem quase iguais, mas o código gerado pelo compilador para eles é muito diferente. Na maioria dos casos, você deve preferir escrever a primeira versão, quando possível. O método é muito mais simples e muito mais fácil de raciocinar. É um método síncrono que retorna um objeto que representa um trabalho que pode estar em andamento.

O segundo constrói uma máquina de estado. Ele gerencia a reentrada para o código que deve ser executado quando a tarefa esperada terminar. Quando finaliza a tarefa, ele retorna e retoma a execução. Você pode escrever a primeira versão para qualquer método de retorno de tarefa que poderia ser um método síncrono. É o caso quando:

* O método não faz nenhum trabalho depois que o único método de retorno de tarefa é chamado.
* O retorno do único método de retorno de tarefa corresponde à assinatura deste método.

A introdução de uma variável local que se refere a um objeto que o IDisposable implementa ficaria assim:

public class DisposableFoo : IDisposable

{

private bool disposed = false;

private HttpClient client;

public DisposableFoo()

{

client = = new System.Net.Http.HttpClient();

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (disposed)

return;

if (disposing)

{

client.Dispose();

// Free any other managed objects here.

}

disposed = true;

}

public async Task<string> AwaitableMethodAsync(Task arg1, Task arg)

{

string result = await client.GetStringAsync("http://www.microsoft.com");

return result;

}

}

Você poderia fazer o seguinte para descartar um objeto de forma assíncrona em um contexto async/await (não recomendado):

var disposableObj = new DisposableFoo();

var arg1 = ComputeArg();

var arg2 = ComputeArg();

await disposableObj.AwaitableMethodAsync(arg1, arg2);

await Task.Run(() => disposableObj.Dispose());

O uso incorreto de Task.Run() pode causar " thread pool starvation" - threads do pool de threads são um recurso compartilhado globalmente. Além disso, você poderia usar disposableObj em uma instrução using:

public static Task DoWorkFooAsync()

{

using (var service = new DisposableFoo())

{

var arg1 = ComputeArg();

var arg2 = ComputeArg();

return service.AwaitableMethodAsync(arg1, arg2);

}

}

public static async Task DoWorkFoo2Async()

{

using (var service = new DisposableFoo())

{

var arg1 = ComputeArg();

var arg2 = ComputeArg();

await service.AwaitableMethodAsync(arg1, arg2);

}

}

Nesta implementação significa que você deve usar a segunda versão, onde o compilador gera a máquina de estado e uma continuação. O primeiro método é síncrono, o que significa que não há continuações. O objeto de serviço será Disposed() assim que retornar de AwaitableMethodAsync(). O objeto é descartado se o trabalho assíncrono for concluído, antes que o método retorne a tarefa (possivelmente ainda em execução). Há uma alta probabilidade de que resulte em um ObjectDisposedException em alguns casos.

O método assíncrono gera o código para que o Dispose gerada pelo compilador seja executada somente após a conclusão da tarefa retornada de AwaitableMethodAsync(). O serviço será descartado apenas quando terminar de fazer todo o seu trabalho. Não é facilmente visível em seu código-fonte e difícil de detectar em testes de unidade automatizados. Freqüentemente, escrevemos testes de unidade para métodos assíncronos que sempre retornam de forma síncrona, usando Task.FromResult (). Esses testes são bons e verificam se o atalho funciona corretamente.

Você também deve escrever testes que verificam o caminho lento, em que uma tarefa não foi concluída de forma síncrona. Não precisa ser mensuravelmente lento. Basta polvilhar uma instrução ‘await Task.Yield()’ em sua implementação simulada e você forçará o caminho lento.

Desde então, muito evoluiu - novas coisas como async/await foram adicionadas e os sistemas multi-threaded não são mais indispensáveis e mais importantes hoje em dia com a nuvem. Assim, há a necessidade de fazer o descarte também de forma assíncrona. O DisposeAsync() cumpre exatamente a mesma finalidade que o método Dispose() de IDisposable e deve seguir as mesmas regras de implementação:

* DisposeAsync/Dispose pode ser chamado várias vezes, chamadas subsequentes devem ser ignoradas
* DisposeAsync/Dispose não deve lançar exceção
* DisposeAsync/Dispose deve ser implementado quando contém outro objeto descartável ou/e recursos não gerenciados

A partir do C# 8.0, a linguagem oferece suporte a tipos descartáveis assíncronos que implementam a interface System.IAsyncDisposable. Você usa a instrução await using para trabalhar com um objeto descartável de forma assíncrona. Abaixo você pode encontrar um exemplo muito simples de uso assíncrono descartável em C# 8.0.

static async Task Main(string[] args)

{

await using (var disposableObject = new Foo())

{

Console.WriteLine("Hello World!");

}

Console.WriteLine("Done!");

}

class Foo : IAsyncDisposable

{

public async ValueTask DisposeAsync()

{

Console.WriteLine("Delaying!");

await Task.Delay(1000);

Console.WriteLine("Disposed!");

}

}

É claro que há um atraso visível entre Delaying! e Disposed! desde que colocamos em um Task.Delay lá para mostrar o aspecto assíncrono do descarte. A proposta de valor é a capacidade de fazer o trabalho de limpeza assíncrona em nossos recursos (em vez de ter que bloquear no tradicional IDisposable), bem como a sintaxe concisa agradável de C # 8.0 - await using. Semelhante ao comando non await using, ele se expande para (internamente):

Foo disposableObject = null;

try

{

disposableObject = new Foo();

//...

}

finally

{

if (disposableObject != null)

await disposableObject.DisposeAsync();

}

Vamos ver como você poderia implementar IAsyncDisposable em uma classe que já implementa IDisposable em sua própria biblioteca:

public class DisposableFoo : IAsyncDisposable, IDisposable

{

private bool disposed = false;

private Utf8JsonWriter \_jsonWriter = new Utf8JsonWriter(new MemoryStream()); //IAsyncDisposable

private HttpClient client = new System.Net.Http.HttpClient(); //IDisposable

public void Dispose()

{

Dispose(disposing: true);

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (disposed)

return;

if (disposing)

{

client?.Dispose();

(\_jsonWriter as IDisposable)?.Dispose();

}

\_jsonWriter = null;

client = null;

disposed = true;

}

public async ValueTask DisposeAsync()

{

await DisposeAsyncCore();

Dispose(disposing: false);

GC.SuppressFinalize(this);

}

protected virtual async ValueTask DisposeAsyncCore()

{

if (\_jsonWriter != null)

{

await \_jsonWriter.DisposeAsync().ConfigureAwait(false);

}

if (client is IAsyncDisposable disposable)

{

await disposable.DisposeAsync().ConfigureAwait(false);

}

else

{

client.Dispose();

}

\_jsonWriter = null;

client = null;

}

public async Task<string> AwaitableMethodAsync(Task arg1, Task arg)

{

string result = await client.GetStringAsync("http://www.microsoft.com");

return result;

}

}

Normalmente, você pode chamar o método dispose existente, mas isso nem sempre é possível - depende da estrutura/propósito de sua classe. O DisposeAsync retorna um ValueTask e não uma Tarefa por causa do desempenho. ValueTask é uma estrutura que não adiciona pressão ao GC alocando no heap. Isso pode ser importante quando muitos objetos são dispostos em um loop apertado. O try/catch só é necessário quando a classe não está selada ou/e você não tem certeza de que em nenhuma circunstância uma exceção é lançada. Além disso, não há suporte para CancelamentoToken para DisposeAsync porque não existe um cenário conhecido para cancelar o processo de limpeza e deixar o objeto em um estado inconsistente.

IAsyncDisposable não é herdado de IDisposable por intensão, permitindo que os desenvolvedores escolham entre implementar um ou ambos. Dependendo do uso / propósito de sua classe, é mais apropriado oferecer apenas DisposeAsync quando provavelmente for usado de forma assíncrona. Outro benefício de ter IAsyncDisposable é realizar uma operação de descarte com uso intensivo de recursos sem bloquear o thread principal de um aplicativo GUI por um longo tempo.

Se uma implementação de IAsyncDisposable for selada, o método DisposeAsyncCore() não será necessário e a limpeza assíncrona poderá ser realizada diretamente no método IAsyncDisposable.DisposeAsync().

Uma diferença principal no padrão de descarte assíncrono em comparação com o padrão de descarte é que a chamada de DisposeAsync() para o método de sobrecarga Dispose(bool) recebe false como argumento. Ao implementar o método IDisposable.Dispose (), entretanto, true é passado. Isso ajuda a garantir a equivalência funcional com o padrão de descarte síncrono e ainda garante que os caminhos do código do finalizador ainda sejam chamados. Em outras palavras, o método DisposeAsyncCore() descartará recursos gerenciados de forma assíncrona, portanto, você não deseja descartá-los de forma síncrona também. Portanto, chame Dispose(false) em vez de Dispose(true).

**ÍNDICES E INTERVALOS**

Trabalhar com matrizes torna-se mais expressivo com os novos tipos e operadores introduzidos para índices e intervalos. Anteriormente, quando tínhamos que endereçar itens com base em sua posição no final de uma matriz, o código resultante parecia um tanto bagunçado:

var numbers = Enumerable.Range(1, 10).ToArray();

var lastItem = numbers[numbers.Length - 1];

Além disso, quando tivemos que dividir uma matriz com base na posição inicial e final, houve uma certa cerimônia envolvida, por exemplo, para calcular o comprimento, pré-inicializar uma matriz ou retornar para:

var (start, end) = (1, 7);

var length = end - start;

// Using LINQ

var subset1 = numbers.Skip(start).Take(length);

subset1.ToList().ForEach(w => Console.Write(w)); //234567

// Or using Array.Copy

var subset2 = new int[length];

Array.Copy(numbers, start, subset2, 0, length);

subset2.ToList().ForEach(w => Console.Write(w)); //234567

O último método usando Array.Copy é aproximadamente o que o compilador emitirá nos bastidores ao usar os novos tipos Index e Range. 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:

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

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

Vamos começar com as regras para índices. O índice 0 é o mesmo que words[0]. O índice ^0 é o mesmo que words[words.Length]. Observe que words[^0] lança uma exceção, assim como words[words.Length]. Para qualquer número n, o índice ^n é o mesmo que words.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.. words.Length] representa todo o intervalo.

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

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

O código a seguir cria um subintervalo com as palavras "quick", "brown" e "fox".

Console.WriteLine(string.Join(" ", words[1..4])); //quick brown fox

O código a seguir cria um subintervalo com "lazy" e "dog".

Console.WriteLine(string.Join(" ", words[^2..^0])); //lazy dog

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

//The quick brown fox jumped over the lazy dog

Console.WriteLine(string.Join(" ", words[..]));

Console.WriteLine(string.Join(" ", words[..4])); //The quick brown fox

Console.WriteLine(string.Join(" ", words[6..])); //the lazy dog

Você também pode declarar intervalos como variáveis, e ser usado dentro dos caracteres [e]:

Range range = 1..4;

Console.WriteLine(string.Join(" ", words[range])); //quick brown fox

Não apenas as matrizes oferecem suporte a índices e intervalos. Você também pode usar índices e intervalos com string, Span<T> ou ReadOnlySpan<T>.

var array = new[] { 0, 1, 2, 3, 4, 5 };

var span = array.AsSpan(1, 4); // = { 1, 2, 3, 4 }

var subSpan = span[1..^1]; // = { 2, 3 }

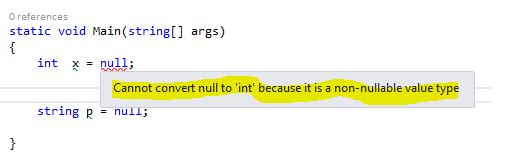
var substring = "Thequickbrownfoxjumpedoverthelazydog"[1..4];

Console.WriteLine(substring); //"heq"

**ATRIBUIÇÃO DE COALESCÊNCIA NULA**

**Tipos anuláveis**

Os tipos de dados em C # são divididos em duas categorias principais: Tipo de valor e Tipo de referência . Uma variável de tipo de valor não pode ser nula, mas podemos atribuir um valor nulo na variável de tipo de referência. Vamos verificar o que acontecerá quando atribuirmos null a um tipo de valor.



Esse é um tipo de erro comum que geralmente enfrentamos durante a codificação. Existem duas maneiras de resolver este problema.

1. **Nullable<int> x = null;**
2. **int ? x = null;**

Os exemplos acima mostram duas maneiras de converter um tipo de valor não anulável em tipo de valor anulável. A partir disso, concluímos que: Um tipo é considerado anulável se pode receber um valor ou pode ser nulo, o que significa que o tipo não tem valor algum. Por padrão, todos os tipos de referência, como strings, são anuláveis, mas todos os tipos de valor, como Int32, não.

Existem dois membros de um tipo anulável.

1. **HasValue**: é do tipo valor booleano. É definido como verdadeiro quando a variável contém um valor não nulo.
2. **Value**: é do tipo booleano. Ele contém os dados armazenados no tipo anulável.

int? x = 5;

if (x.HasValue)

{

Console.WriteLine("contain not nullable value: " + x.Value.ToString());

}

else

{

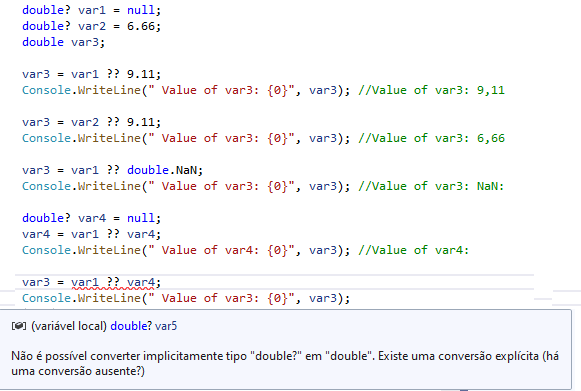
Console.WriteLine("contain Null value.");

}

Isso é a saber sobre tipos anuláveis ​​em C#. Agora vamos entender um pouco sobrer o operador Null Coalescing em C#.

**Operador de Coalescência Nula ??**

O operador de coalescência nulo foi introduzido no C# 7.0. Este operador retorna o valor do operando à esquerda se ele for "não nulo", caso contrário, o valor do operando à direita é avaliado. Se o operando à esquerda for "não nulo", ele nem mesmo avalia o operando da direita. Vamos considerar um exemplo do programa a seguir, que cobre os dois cenários.



**Note:**

1. operando var1 à esquerda é nulo, portanto, o valor de 9.11 é avaliado e atribuído à variável de saída.
2. operando var2 à esquerda não é nulo, portanto, o valor de var2 é avaliado e atribuído à variável de saída.
3. operando var1 à esquerda e operando var4 são nulos, o valor nulo é atribuído à variável anulável.
4. operando var1 à esquerda e operando var4 são nulos, o valor não é atribuído à variável não anulável.

Os operadores de coalescência nula são associativos à direita. Ou seja, as expressões do formulário são avaliadas como

|  |  |
| --- | --- |
| **Expressão** | **Avaliação** |
| a ?? b ?? c | a ?? (b ?? c) |
| d ??= e ??= f | d ??= (e ??= f) |

Os operador de coalescência nulo operadores e podem ser úteis nos seguintes cenários:

1. Fornecer uma expressão alternativa a ser avaliada, caso o resultado da expressão com operações condicionais nulas seja null:

private static double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)

{

return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;

}

var sum = SumNumbers(null, 0);

Console.WriteLine(sum); // output: NaN

1. Fornecer um valor de um tipo de valor subjacente, caso um valor de tipo anulável seja null :

int? a = null;

int b = a ?? -1;

Console.WriteLine(b); // output: -1

1. Usar uma expressão throw como o operando à direita do ?? operador para tornar o código de verificação de argumento mais conciso.

public string Name

{

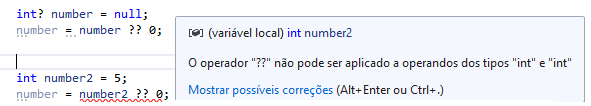
get => name;

set => name = value ?? throw new ArgumentNullException(nameof(value), "Name cannot be null");

}

No C # 7.3 e anteriores, o tipo de operando esquerdo do operador ?? deve ser um tipo de referência ou um tipo de valor anulável. A partir do C # 8.0, esse requisito foi substituído pelo seguinte:

**O tipo de operando esquerdo dos operadores ?? e ?? = tem que ser do tipo de valor anulável**



Isso amplia a utilização do operador que pode ser uma variável, uma propriedade ou um elemento indexador. 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);

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

numbers.Add(i ??= 20);

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

Em particular, a partir do C# 8,0, você pode usar os operadores de coalescência nulo com parâmetros de tipo irrestrito, pois o parâmetro de tipo irrestrito T existe, é um tipo anulável e não é um tipo de referência. Um parâmetro de tipo genérico irrestrito não pode usar os operadores == e != para comparar duas instâncias do parâmetro de tipo, no caso de o tipo não ser compatível com esses operadores. Essas verificações são necessárias para genéricos, mas não para modelos porque os genéricos podem ser especializados em runtime com qualquer classe que atenda às restrições, quando for tarde demais para verificar o uso de membros inválidos.

private static void Display<T>(T a, T backup)

{

Console.WriteLine(a ?? backup);

}

A partir do C# 8,0, você pode usar o ??= operador para comparar e atribuir valor as variáveis:

List<int> numbers = null;

int? variable = null;

int expression = 5;

if (variable is null)

{

variable = expression;

}

(numbers = numbers ?? new List<int>()).Add(5);

numbers.Add((variable = variable ?? 0).Value);

pelo código a seguir:

variable ??= expression;

(numbers ??= new List<int>()).Add(5);

numbers.Add(variable ??= 0);

**Pontos importantes:**

* O operando à esquerda do operador ??= deve ser uma variável, uma propriedade ou um elemento indexador.
* É associativo à direita.
* Você não pode sobrecarregar o operador ??=.
* Você tem permissão para usar o operador ??= com tipos de referência e tipos de valor.

**TIPOS CONSTRUÍDOS NÃO GERENCIADOS**

As definições de tipos não gerenciados e construídos citados pelas especificações de linguagem:

* **Tipo construído:** se for genérico e o parâmetro de tipo já estiver definido. Por exemplo, List <string> é um tipo construído, enquanto List <T> não é.
* **Tipo não gerenciado**: especificação C# 6 não é um tipo de referência ou tipo construído e não contém tipo de referência ou campos de tipo construído em qualquer nível de aninhamento. É quando pode ser usado em um contexto não seguro. Isso é verdadeiro para muitos tipos básicos integrados. A documentação oficial inclui a lista destes tipos:
* sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal ou bool
* Qualquer tipo de Enumeração
* Qualquer tipo de ponteiro (int\*, char\*, int\*[], void\*, etc). O tipo especificado antes do \* em um tipo de ponteiro é chamado de tipo referent. Somente um tipo não gerenciado pode ser um tipo referent. Os tipos de ponteiro não são herdados de object e não há nenhuma conversão entre tipos de ponteiro e object.
* Qualquer tipo de struct definido pelo usuário que contém campos de tipos não gerenciados somente e, em C# 7,3 e anterior, não é um tipo construído (um tipo que inclui pelo menos um argumento de tipo)

A partir do C# 7,3, você pode usar a restrição unmanaged para especificar que um parâmetro de tipo é um tipo não gerenciado, não-ponteiro e não anulável. Mas no C# 7,3, um tipo construído (um tipo que inclui pelo menos um argumento de tipo) não pode ser um tipo não gerenciado. Nesta versão, os usuários podem declarar structs e obter o endereço dele (em um contexto não seguro), desde que não seja considerado um tipo "gerenciado".

O compilador atualmente relata structs genéricos como tipos "gerenciados", embora não sejam rastreados por GC. No entanto, não há nada no tempo de execução que impeça um usuário de obter o endereço de uma estrutura genérica e, em certos cenários, pode ser desejável permitir isso. Um exemplo é o tipo System.Runtime.Intrinsics.Vector128 <T>, que não contém objetos rastreados por GC. O tipo é projetado para ser usado em cenários de alto desempenho e geralmente inseguros, mas existem certas operações (como stackalloc, pinning, etc) que não podiam ser feitas em C#.

A partir do C# 8,0, foi removida a restrição de que um tipo não gerenciado não pode ser um tipo construído. Em vez disso, os tipos construídos não seriam gerenciados se atendessem aos requisitos de tipos de estrutura gerais definidos pelo usuário. Por exemplo, um tipo struct construído que contém campos de tipos não gerenciados também é não gerenciado, como mostra o exemplo a seguir:

public struct Coords<T>

{

public T X;

public T Y;

}

class Program

{

static void Main(string[] args)

{

DisplaySize<Coords<int>>();

DisplaySize<Coords<double>>();

Console.WriteLine("Hello World!");

}

private unsafe static void DisplaySize<T>() where T : unmanaged

{

Console.WriteLine($"{typeof(T)} is unmanaged and its size is {sizeof(T)} bytes");

}

}

No exemplo, um tipo struct genérica pode ser a fonte de tipos construídos não gerenciados e gerenciados. O exemplo anterior define uma struct genérica Coords<T> e apresenta os exemplos de tipos construídos não gerenciados. O exemplo de um tipo gerenciado seria Coords<object>, porque tem os campos do tipo gerenciados object. Se você quiser que todos os tipos construídos sejam tipos não gerenciados, use a unmanaged restrição na definição de uma estrutura genérica:

public struct Coords<T> where T : unmanaged

{

public T X;

public T Y;

}

Como para qualquer tipo não gerenciado, agora você pode criar um ponteiro para uma variável desse tipo:

unsafe

{

int length = 3;

Coords<int>\* numbers = stackalloc Coords<int>[length];

for (var i = 0; i < length; i++)

{

numbers[i] = new Coords<int> { X = 0, Y = i };

}

}

Ou alocar um bloco de memória na pilha para instâncias desse tipo:

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

{

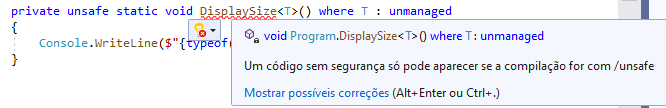
new Coords<int> { X = 0, Y = 0 },

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

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

};

O compilador pode precisar fazer uma validação adicional para validar se uma estrutura genérica definida pelo usuário está "ok" para usar.

****

**STACKALLOC EM EXPRESSÕES ANINHADAS**

Uma expressão stackalloc aloca um bloco de memória na pilha, que é criado durante a execução do método e é descartado automaticamente quando esse método é retornado. Não é possível liberar explicitamente a memória alocada com stackalloc. Um bloco de memória alocado de pilha não está sujeito à coleta de lixo e não precisa ser fixado com uma instrução fixed.

A principal razão para usar stackalloc é o desempenho (para cálculos ou interoperabilidade). Usando stackalloc em vez de um array de heap alocado nativo(como malloc ou o equivalente .Net):

* Cria menos pressão GC (roda menos), você não precisa fixar os arrays
* Automaticamente liberado na saída do método. As matrizes alocadas no heap são desalocadas apenas quando o GC é executado
* Ganha velocidade pois a chance de acertos do cache na CPU devido à localidade dos dados é maior
* Habilita automaticamente os recursos de detecção de estouro de buffer no CLR (Common Language Runtime). Se for detectada um estouro de buffer, o processo será encerrado assim que possível para minimizar a chance de o código mal-intencionado ser executado.

Você pode atribuir o resultado de uma expressão stackalloc a uma variável de um dos seguintes tipos:

* **Por System.Span<T> ou System.ReadOnlySpan<T>**
* **Por um** [**tipo ponteiro**](https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/unsafe-code-pointers/pointer-types)**,**

**System.Span<T> ou System.ReadOnlySpan<T>**

A partir do C# 7,2, é a forma recomendável para trabalhar com memória alocada na pilha, pois não precisa usar um contexto unsafe como mostra o exemplo a seguir:

//Antes do C# 7,2

unsafe

{

int\* Array1 = stackalloc int[3] { 5, 6, 7 };

}

int length = 3;

Span<int> numbers7\_2 = stackalloc int[length];

for (var i = 0; i < length; i++)

{

numbers7\_2[i] = i;

}

Ao trabalhar com esses tipos, você pode usar uma expressão stackalloc em condicional ou expressões de atribuição, como mostra o seguinte exemplo:

length = 1000;

const int MaxStackLimit = 1024;

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

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 expressão stackalloc 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

No exemplo acima, usamos stackalloc como coleção de dados int e passamos stackalloc como expressão para o método IndexOfAny, que retornará o resultado da posição encontrada.

**Por um** [**tipo ponteiro**](https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/unsafe-code-pointers/pointer-types)

Precisa usar um contexto unsafe ao trabalhar com tipos de ponteiro. Nestes casos, você pode usar uma expressão stackalloc somente em uma declaração de variável local para inicializar a variável.

unsafe

{

int lengthpont = 3;

int\* numberspont = stackalloc int[lengthpont];

for (var i = 0; i < lengthpont; i++)

{

numberspont[i] = i;

}

}

Como a quantidade de memória disponível na pilha é limitada, para evitar que um StackOverflowException seja lançado:

* Limite a quantidade de memória que você aloca com stackallo. A quantidade de memória depende do ambiente no qual o código é executado, seja conservador quando você define o valor de limite real. No exemplo acima, foi limitado com a constante MaxStackLimit.
* Evite usar stackalloc loops internos. Aloque o bloco de memória fora de um loop e reutilize-o dentro do loop.

O conteúdo da memória recém-alocada é indefinido. Você deve inicializá-lo antes do uso. Por exemplo, você pode usar o Span<T>.Clear método que define todos os itens com o valor padrão do tipo T .

A partir do C# 7,3, você pode usar a sintaxe do inicializador de matriz para definir o conteúdo da memória alocada recentemente. O seguinte exemplo demonstra várias maneiras de fazer isso:

Span<int> first = stackalloc int[3] { 1, 2, 3 };

Span<int> second = stackalloc int[] { 1, 2, 3 };

ReadOnlySpan<int> third = stackalloc[] { 1, 2, 3 };

**APRIMORAMENTO DE STRINGS TEXTUAIS INTERPOLADAS**

Uma string que começa com $, era a forma de identificar a string como uma string interpolada, esta string contém a expressão de interpolação e outras que são substituídas pelo resultado real. Veja o exemplo abaixo:

string szName = "ABC";

int iCount = 15;

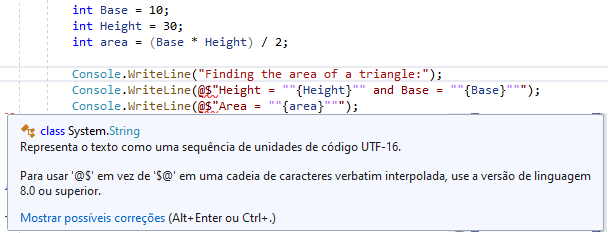
//Olá, ABC! Você tem 15 maçãs

Console.WriteLine($"Olá, {szName}! Você tem {iCount} maçãs");

Console.WriteLine($@"Olá, {szName}! Você tem {iCount} maçãs");

Console.WriteLine(@$"Olá, {szName}! Você tem {iCount} maçãs");

Agora em C# 8, $@"" or @$"" são válidos significa que você pode iniciar a string por @ ou $. Em versões anteriores do C#, o $ token devia aparecer antes do @ token. Nas versões abaixo de C# 8, quando usamos um token @ antes de $ token, o compilador lança um erro.

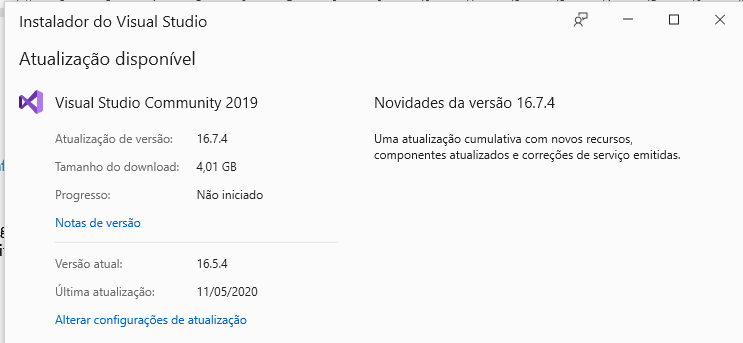


**Csharp 9.0**

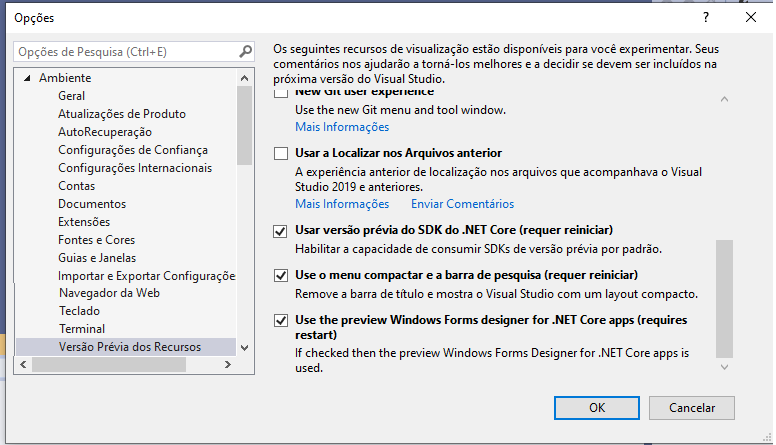
**CONFIGURANDO O C# 9**

Configurar com C # 9 é basicamente o mesmo que configurar com .NET Preview 5. E, felizmente para você, temos um guia brilhante aqui: https://dotnetcoretutorials.com/2020/05/22/ Getting-started-with-net-5-and-the-msbuild-net-5-cliffnotes/. Mas para as notas precipitadas desse post:

* Certifique-se de que sua versão do Visual Studio 2019 seja pelo menos 16.7 clicando em Help => Check For Updates dentro do Visual Studio. Em caso de dúvida, atualize.

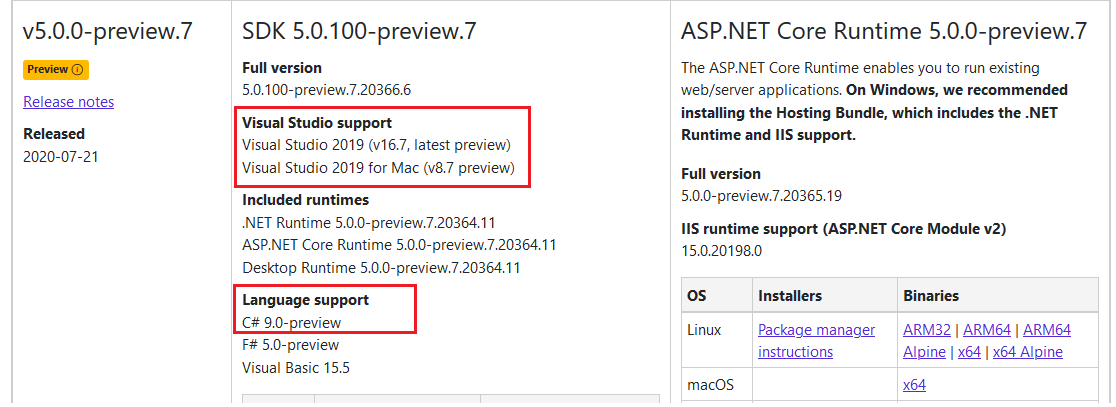


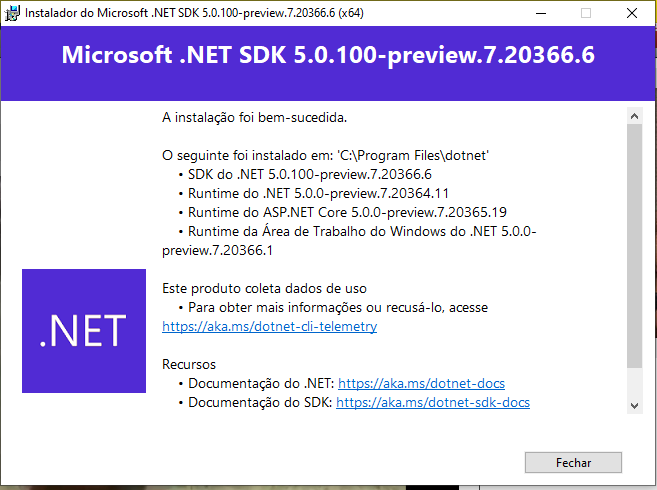
* Vá em Ferramentas => Opções dentro de Ambiente, selecione “Versão Prévia de recursos” e marque a caixa que diz “Usar versão prévia do SDK do .NET Core”. Em seguida, reinicie o Visual Studio.



* Baixe e instale um SDK do .NET 5 compatível com a versão de VS 2019 instalada:

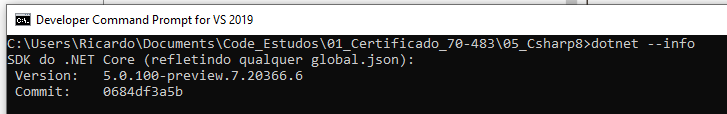
<https://dotnet.microsoft.com/download/dotnet/5.0>



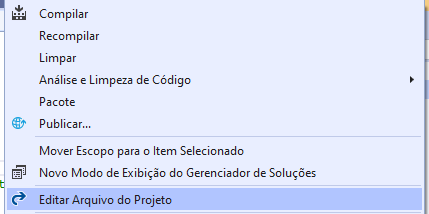


* Após a instalação, você pode executar o comando dotnet info em um prompt de comando:

**dotnet –info**



Depois que o SDK do .NET 5 Preview estiver instalado e configurado. Para qualquer projeto existente que você deseja testar em execução no .NET 5 (por exemplo, um pequeno aplicativo de console), tudo o que você precisa fazer é brir o arquivo .csproj. Para abrir o arquivo de projeto \* .csproj: Clique com o botão direito do mouse no Gerenciador de Soluções e escolha Editar Arquivo de Projeto.:



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

<PropertyGroup>

<OutputType> Exe </OutputType>

<TargetFramework> netcoreapp3.1 </TargetFramework>

</PropertyGroup>

</Project>

Para :

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

<PropertyGroup>

<OutputType> Exe </OutputType>

<TargetFramework> net5.0 </TargetFramework>

<LangVersion> 9.0 </LangVersion>

</PropertyGroup>

</Project>

Observe que alguns modelos, como aplicativos de console, não solicitam a versão do SDK ao criar um novo projeto, eles apenas usam o SDK mais recente disponível. Nesse caso, seu “padrão” para Visual Studio repentinamente se torna uma prévia do SDK do .NET Core.

**Solução de problemas**

Se você fez tudo isso e obteve um destes erros:

* Os assemblies de referência para .NET Framework, Version = v5.0 não foram encontrados.
* Opção inválida '9.0' para / langversion. Use '/ langversion :?' para listar os valores suportados.

Então aqui está sua lista de solução de problemas rápida e fácil:

* Tem certeza de que tem o SDK do .NET 5 mais recente instalado? Lembre-se de que é o SDK, não o runtime, e é .NET 5.
* Tem certeza de que o Visual Studio está atualizado? Mesmo se você tiver o SDK instalado, o Visual Studio segue suas próprias regras e precisa ser atualizado.
* Você está usando o Visual Studio 2019? Qualquer versão anterior não funcionará.
* Tem certeza de que ativou os SDKs de visualização? Lembre-se de Ferramentas => Opções e, em seguida, “Visualizar Recurso”
* Para rodar par Visual Studio 2019 version 16.8 e superior, requer o .NET 5 SDK preview 8. NoVisual Studio 2019 version 16.7 e superior, requer o .NET 5 SDK preview 7. O próprio SDK define um VS mínimo que ele suporta e não carregará nas versões do VS antes desse valor para evitar comportamento interrompido. Cada pré-lançamento contínuo do .NET e do SDK será enviado junto com um pré-lançamento do 16.8 VS com o qual são testados. Por exemplo, .NET 5 preview 8 fornecido com o Visual Studio preview 2. RC1 enviado com preview 3. Recomendamos que você use essas combinações juntas para teste, pois eles têm alguns componentes compartilhados e dependências que farão os cenários funcionarem melhor E2E.

|  |  |
| --- | --- |
| **Releases** | **Visual Studio 2019 Preview** |
| September 14, 2020 | version 16.8 Preview 3.1New release icon |
| August 31, 2020 | Visual Studio 2019 version 16.8 Preview 2.1 |
| August 25, 2020 | Visual Studio 2019 version 16.8 Preview 2 |
| August 5, 2020 | Visual Studio 2019 version 16.8 Preview 1 |

Uma nova versão de amostra do .NET 5/C# 9 é lançada a cada poucos meses. Portanto, se você estiver lendo sobre um novo recurso C# 9 ou .NET 5 que alguém está usando, mas parece que não consegue fazê-lo funcionar, volte sempre para https://dotnet.microsoft.com/download/dotnet/5.0 e baixe a última versão de visualização e instale. Semelhante ao Visual Studio, embora normalmente não seja um problema, tente mantê-lo atualizado, já que muitas vezes os recursos mais recentes que não funcionam são simplesmente porque estou em uma versão que estava perfeitamente bem um mês atrás, mas agora está desatualizada.

O c# 9,0 adiciona os seguintes recursos e aprimoramentos à linguagem C#:

|  |  |
| --- | --- |
| init somente Setters | Init only setters |
| Registros | Records |
| 1. Copiar e clonar Membros | 1. Copy and clone Members |
| * Expressões With | * With expressions |
| 1. Métodos para comparações de igualdade com base em valor como Equals(objeto), IEquatable<T> | 1. Methods for equality comparisons based on value such as Equals (object), IEquatable <T> |
| * sobrecargas Equals | * Equals overloads |
| * GetHashCode () | * GetHashCode () |
| * operator == e operator != | * operator == and operator! = |
| * EqualityContract | * EqualityContract |
| 1. Override de GetHashCode() | 1. Override GetHashCode() |
| 1. PrintMembers e ToString() | 1. PrintMembers and ToString () |
| 1. Método Construct/Deconstruct com registros posicionais simplificados | 1. Construct/Deconstruct method with simplified positional registers |
| * Registros Posicionais | * Positional Records |
| Instruções de nível superior | Top-level statements |
| Melhorias na correspondência de padrões | Pattern matching enhancements |
| Desempenho e interoperabilidade | Performance and interoperability |
| * Inteiros de tamanho nativo | * Native sized integers |
| * Ponteiros de função | * Function pointers |
| * Suprimir emissão do sinalizador localsinit | * Suppress emitting localsinit flag |
| Aprimoramento da tipagem | Typing improvement |
| * Expressões new com Target-typing | * Target-typed new expressions |
| * Expressões condicionais Target-typing | * Target-typed conditional expressions |
| * Tipos de retorno covariantes | * Covariant return types |
| funções anônimas estáticas | static anonymous functions |
| GetEnumeratorSuporte de extensão para foreach loops | Extension GetEnumerator support for foreach loops |
| Parâmetros discard de lambda | Lambda discard parameters |
| Atributos em funções locais | Attributes on local functions |
| Suporte para geradores de código | Support for code generators |
| * Inicializadores de módulo | * Module initializers |
| * Novos recursos para métodos parciais | * New features for partial methods |

**SETTERS INIT**

O C# 9 está lançando uma série de recursos somente init. Isso inclui propriedades somente init, acessadores init e campos somente leitura. Em C#, é comum declarar uma classe com getters e setters:

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

}

Claro, você pode lançar um set privado lá para controle de acesso, mas suas opções normalmente são limitadas para seus getters e setters. No entanto, isso força a mutabilidade quando você executa a inicialização do objeto. Para que isso funcione, as propriedades devem ser mutáveis. Para definir valores em uma nova pessoa, você precisará chamar o construtor do objeto (neste caso, como na maioria dos casos, um construtor sem parâmetros) e, em seguida, executar a atribuição dos configuradores.

var person = new Person

{

FirstName = "Tony",

LastName = "Stark",

};

E como isso é mutável, você pode fazer isso facilmente:

Console.WriteLine(person.FirstName); // Tony

person.FirstName = "Howard";

Console.WriteLine(person.FirstName); // Howard

Com C# 9, podemos mudar isso com um acessador init. Variante do acessador set, um acessador init só pode ser chamado para propriedades e indexadores durante a inicialização do objeto, isso significa que você só pode criar e definir uma propriedade ao inicializar o objeto. Se modificarmos nosso modelo Person assim, podemos evitar que o FirstName seja alterado:

public class Person

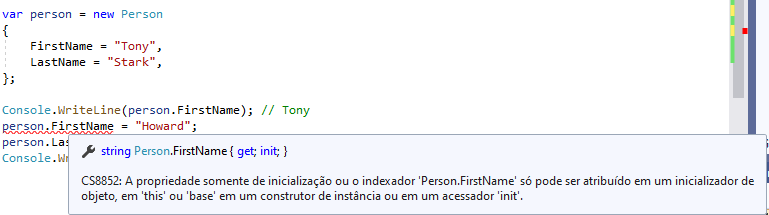
{

public string FirstName { get; init; }

public string LastName { get; set; }

}

Com essa declaração, essas propriedades são ReadOnly quando a construção é concluída, qualquer atribuição subsequente a propriedade FirstName é um erro.



Como os acessadores init só podem ser chamados durante a inicialização, eles têm permissão para modificar os campos somente leitura da classe envolvente, assim como você pode em um construtor. Você pode declarar setters somente init em qualquer tipo que você escreve. Com o uso de campos privados definirá um valor durante a inicialização - caso contrário, podemos lançar uma ArgumentNullException:

public class Person

{

private readonly string lastName;

public string FirstName { get; init; }

public string LastName

{

get => lastName;

init => lastName = (value ?? throw new ArgumentNullException(nameof(LastName)));

}

}

Os setters somente init podem ser úteis para definir propriedades de classe base de classes derivadas. Eles também podem definir propriedades derivadas por meio de auxiliares em uma classe base. Registros posicionais declaram propriedades usando somente init setters. Esses setters são usados em expressões with. Você pode declarar setters somente init para qualquer um class ou struct.

**REGISTROS**

O C# 9,0 apresenta tipos de registro, que são um tipo de referência que fornece métodos sintetizados para fornecer a semântica de valor para igualdade. Os registros são imutáveis por padrão. Imutabilidade é uma palavra com a simples premissa: uma vez instanciados ou inicializados, os tipos imutáveis nunca mudam. Por exemplo, System.DateTime é imutável, assim como strings.

Historicamente, os tipos .NET são classificados em grande parte como tipos de referência (incluindo classes e tipos anônimos) e tipos de valor (incluindo structs e tuplas). Embora tipos de valor imutáveis sejam recomendados, os tipos de valores mutáveis geralmente não introduzem erros. As variáveis de tipo de valor contêm os valores para que as alterações sejam feitas em uma cópia dos dados originais quando os tipos de valor são passados para métodos.

A imutabilidade é uma propriedade interessante, pois tende a simplificar a maneira como pensamos sobre um método. Um objeto imutável tem apenas um estado, o estado que você especificou quando criou o objeto. Quando algo é imutável, não precisamos nos preocupar se outra parte do aplicativo pode sofrer mutação. Elimina a necessidade de usar bloqueios e simplifica todas as partes altamente simultâneas do seu sistema. Essas vantagens são mais pronunciadas em programas simultâneos com dados compartilhados.

Os registros do C # 9.0 nos ajudarão a criar tipos imutáveis que são muito úteis em grandes arquiteturas distribuídas ao usar conceitos como mensagens e microsserviços, pois eles são seguros e thread-safe, sem necessidade de sincronização. Dito de outra forma: tipos imutáveis reduzem o risco, são mais seguros e ajudam a prevenir muitos bugs desagradáveis que ocorrem quando você atualiza seus objetos.

Você poderia simular a imutabilidade com structs e até mesmo classes, mas a ideia é ter uma construção que seja simples e direta de implementar. Um dos maiores difereciais de registros sobre estruturas é a reduzida alocação de memória necessária. Como os registros C# são compilados para referenciar tipos nos bastidores, eles são acessados ​​por uma referência e não como uma cópia. Como resultado, nenhuma alocação de memória adicional é necessária além da alocação de registro original. É imutável, pois nenhuma das propriedades pode ser modificada depois de ser criada. Quando você define um tipo de registro, o compilador sintetiza vários outros métodos para você:

1. **Copiar e clonar Membros**
2. **Métodos para comparações de igualdade com base em valor como Equals(objeto), IEquatable<T>**
3. **Override de GetHashCode()**
4. **PrintMembers e ToString()**
5. **Método Construct/Deconstruct com registros posicionais simplificados**

Os registros fornecem uma declaração de tipo para um tipo de referência imutável que usa semântica de valor para igualdade. Os métodos sintetizados para códigos de igualdade e hash consideram dois registros iguais se suas propriedades forem iguais. Considere esta definição:

public record Person

{

public string LastName { get; }

public string FirstName { get; }

public Person(string first, string last) => (FirstName, LastName) = (first, last);

}

A definição de registro cria um tipo Person que contém duas propriedades ReadOnly: FirstName e LastName. O tipo Person é um tipo de referência. O Registro tem também suporte à herança.

* Os registros não podem herdar de classes, a menos que a classe seja object
* Classes não podem herdar de registros.
* Os registros podem ser herdados de outros registros.

Você pode declarar um novo registro derivado do da Person seguinte maneira:

public record Teacher : Person

{

public string Subject { get; }

public decimal Price { get; }

public Teacher(string first, string last, string sub, decimal price)

: base(first, last) => (Subject, Price) = (sub, price);

}

Você também pode criar registros selados para evitar uma maior derivação:

public sealed record Student : Person

{

public int Level { get; }

public Student(string first, string last, int level)

: base(first, last) => Level = level;

}

O compilador sintetiza versões diferentes dos métodos acima. As assinaturas de método dependem de se o tipo de registro é selada e se a classe base direta é Object.

**1 - Copiar e clonar Membros**

Os registros dão suporte à construção de cópia. A construção correta da cópia deve incluir hierarquias de herança e propriedades adicionadas por desenvolvedores. Os registros podem ser copiados com modificações. Essas operações de cópia e modificação oferecem suporte a mutação não destrutiva. Um tipo de registro contém dois membros de cópia:

* Um construtor que assume um único argumento do tipo de registro. Ele é chamado de "Construtor de cópia".
* Um método de "clonagem" de instância pública resumida com um nome reservado de compilador. O método "clone" sintetizado oferece suporte à construção de cópias para hierarquias de registros. O termo "clone" está entre aspas porque o nome real é gerado pelo compilador. Você não pode criar um método denominado Clone em um tipo de registro.

A finalidade do construtor de cópia é copiar o estado do parâmetro para a nova instância que está sendo criada. Esse construtor não executa nenhum campo de instância/inicializadores de propriedade presentes na declaração de registro. Se o construtor não for declarado explicitamente, um construtor será sintetizado pelo compilador. Se o registro estiver selado, o Construtor será privado, caso contrário, ele será protegido.

Vamos supor que em algum momento você deseje criar esse novo objeto Friend e precise alterar o sobrenome desse amigo para Mueller. Observe a atribuição do valor da propriedade FirstName e Middlename do primeiro objeto Friend às propriedades do segundo objeto Friend:

public class Friend

{

public string FirstName { get; init; }

public string MiddleName { get; init; }

public string LastName { get; init; }

}

var friend = new Friend

{

FirstName = "Thomas",

MiddleName = "Claudius",

LastName = "Huber"

};

var newFriend = new Friend

{

FirstName = friend.FirstName,

MiddleName = friend.MiddleName,

LastName = "Mueller"

};

Você deve copiar todas as propriedades do antigo objeto Friend. Isso significa que quanto mais propriedades você tem, mais difícil fica. Claro que você pode implementar alguma lógica de cópia com reflexão ou serialização, ou pode usar uma biblioteca de mapeamento automático. Mas o C # 9.0 tem uma maneira melhor de trabalhar com classes de dados imutáveis: Registros.

**Expressões With**

Os registros têm suporte expressões with. Uma expressão with instrui o compilador a criar uma cópia de um registro, mas com as propriedades especificadas modificadas:

* Uma expressão with não é permitida como uma instrução.
* Uma expressão with permite uma "mutação não destrutiva", projetada para produzir uma cópia da expressão do destinatário com modificações em atribuições no inicializador de objeto para indicar o que é diferente no novo objeto do antigo.
* Um expressão a with válida tem um receptor com um tipo não void. O tipo de receptor deve ser um registro.
* No lado direito da with expressão, há uma lista de inicializador de objeto com uma sequência de atribuições para o identificador, que deve ser um campo de instância acessível ou Propriedade do tipo do destinatário.

Um registro define implicitamente um "construtor de cópia" protegido - um construtor que pega um objeto de registro existente e o copia campo por campo para o novo. Em seguida, cada inicializador de objeto é processada da mesma forma que uma atribuição para um campo ou acesso de Propriedade do resultado da conversão. As atribuições são processadas na ordem léxica.

Person person2 = new Teacher("Bill", "Wagner", "Math", 11.99m);

var newperson2 = person2 with { LastName = "VideoGame" };

Console.WriteLine("Type: " + newperson2.GetType().Name); //Type: Teacher

Console.WriteLine("FirstName: " + newperson2.FirstName); //FirstName: Bill

Console.WriteLine("LastName: " + newperson2.LastName); //LastName: VideoGame

//Console.WriteLine("Subject: " + newperson2.Subject);

//Console.WriteLine("Price: " + newperson2.Price);

A linha acima cria um novo Person registro no qual a LastName propriedade é uma cópia de Person e o FirstName é "Bill". Você pode definir qualquer número de propriedades em uma expressão with. Qualquer um dos membros sintetizados, exceto o método "clone", pode ser escrito por você. Se um tipo de registro tiver um método que corresponda à assinatura de qualquer método sintetizado, o compilador não sintetizará esse método.

Podemos retornar ao tipo original Teacher fazendo o cast do tipo:

var teacher2 = (Teacher)newperson2; //available after casting

Console.WriteLine("Type: " + teacher2.GetType().Name); //Type: Teacher

Console.WriteLine("FirstName: " + teacher2.FirstName); //FirstName: Bill

Console.WriteLine("LastName: " + teacher2.LastName); //LastName: VideoGame

Console.WriteLine("Subject: " + teacher2.Subject); //Subject: Math

Console.WriteLine("Price: " + teacher2.Price); //Price: 11,99

O método "clone" sintetizado retorna o tipo de registro que está sendo copiado usando o envio virtual. O compilador adiciona modificadores diferentes para o método "clone" dependendo dos modificadores de acesso no registro:

* Se o tipo de registro for abstrato, o método "clone" também será abstrato. Se o tipo base não for objeto, o método também será sobrescrito.
* Para tipos de registro que não são abstratos quando o tipo base é objeto:
  + Se o registro for selado, nenhum modificador adicional será adicionado ao método "clone" (o que significa que não é virtual).
  + Se o registro não estiver selado, o método "clone" é virtual.
* Para tipos de registro que não são abstratos quando o tipo base não é objeto:
  + Se o registro estiver selado, o método "clone" também será selado.
  + Se o registro não estiver selado, o método "clone" será substituído.

**2 - Métodos para comparações de igualdade com base em valor como Equals(objeto), IEquatable<T>**

A igualdade é baseada em valor e inclui uma verificação de que os tipos correspondem. Por exemplo, um Student não pode ser igual a a Person, mesmo que os dois registros compartilhem o mesmo nome. Tecnicamente os registros são um tipo de classe, o que também significa que são tecnicamente, tipos de referência. Todos os objetos herdam um método virtual Equals (objeto) da classe de objeto. Isso é usado como base para o método estático Object.Equals (objeto, objeto) quando ambos os parâmetros são não nulos. C**omo structs, os registros** **fozem override do método Equals** (objeto) que cada classe tem, para alcançar o valor que buscamos. Structs sobrescrevem isso para ter “igualdade baseada em valor”, comparando cada campo da struct chamando Equals neles recursivamente. Os registros fazem o mesmo. Isso significa que, de acordo com seu “valor”, dois objetos de registro podem ser iguais um ao outro sem serem o mesmo objeto. Por exemplo, se modificarmos o sobrenome da pessoa modificada novamente:

var originalPerson = otherPerson with { LastName = "Hunter" };

Teríamos agora ReferenceEquals (person, originalPerson) = false (eles não são o mesmo objeto), mas Equals (person, originalPerson) = true (eles têm o mesmo valor).

var person = new Person("Tony", "Stark", "10880 Malibu Point", "Malibu", "red");

var newPerson = person with { FirstName = "Howard", City = "Pasadena" };

Console.WriteLine(Object.ReferenceEquals(person, newPerson)); // false

Console.WriteLine(Object.Equals(person, newPerson)); // false

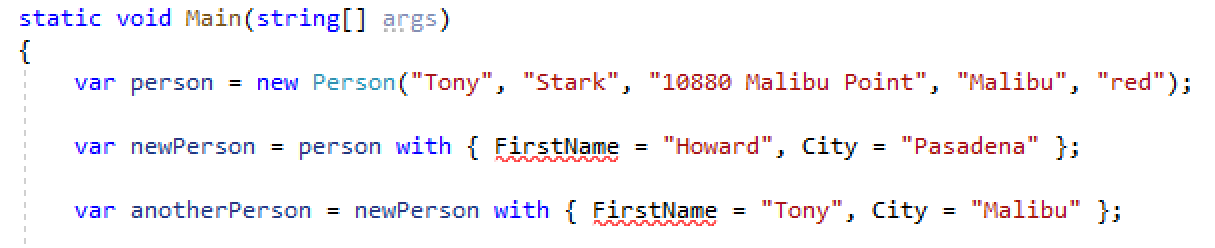
var anotherPerson = newPerson with { FirstName = "Tony", City = "Malibu" };

Console.WriteLine(Object.ReferenceEquals(person, anotherPerson)); // false

Console.WriteLine(Object.Equals(person, anotherPerson)); // true

Isso significa que também podemos trabalhar com **igualdade baseada em valores.** Por exemplo, se criarmos dois novos objetos, sabemos que eles têm referências diferentes na memória, então uma chamada ReferenceEquals (ou mesmo uma chamada ==) retornará falso mesmo se eles tiverem os mesmos valores. Isso é diferente de structs - porque structs são tipos de valor, isso não ocorrerá. Com os registros, compararemos valores. Observe o que acontece enquanto:

1. Crie uma nova pessoa chamada pessoa, Tony Stark
2. Crie outra pessoa, chamada newPerson, Howard Stark, com duas propriedades diferentes (nome e cidade)
3. Crie uma terceira pessoa chamada anotherPerson e defina anotherPerson com os mesmos valores da pessoa original



Repare que está indicando erro. Isso porque em nossas propriedades temos:

public record Person

{

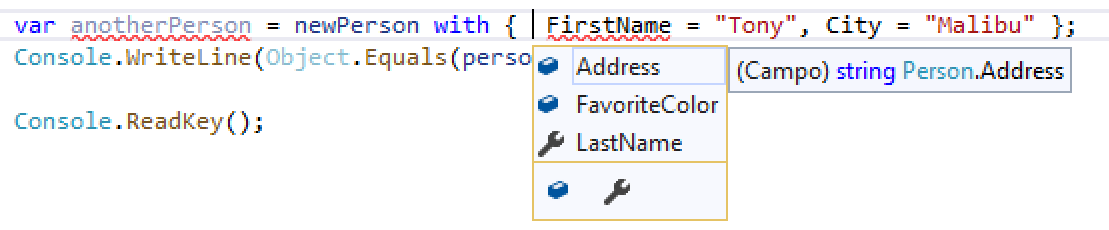
public string LastName { get; }

public string FirstName { get; }

public string Address;

public string City;

public string FavoriteColor;



Alguns são campos, outros propriedade... Mas está havendo erro ao setar o valor a uma propriedade. Para corrigir isso temos que utilizar init:

public record Person

{

public string FirstName { get; init; }

public string LastName { get; init; }

public string Address { get; init; }

public string City { get; init; }

public string FavoriteColor { get; init; }

}

Se você não gosta do comportamento de comparação campo por campo padrão da substituição Equals gerada, você pode escrever o seu próprio. Você só precisa ter cuidado para entender como a igualdade baseada em valores funciona em registros, especialmente quando a herança está envolvida.

Junto com Equals baseado em valor, há também uma substituição GetHashCode () baseada em valor para acompanhá-lo.

sobrecargas Equals

public record Teacher : Person, IEquatable<Person>

{

public string Subject { get; init; }

public decimal Price { get; init; }

public void Teachclass() => Console.WriteLine("It's class teaching time");

public bool Equals([AllowNull] Person other)

{

throw new NotImplementedException();

}

public Teacher(string first, string last, string sub, decimal price)

: base(first, last) => (Subject, Price) = (sub, price);

}

GetHashCode ()

public override int GetHashCode()

{

return Combine(EqualityComparer<Type>.Default.GetHashCode(EqualityContract),

EqualityComparer<T1>.Default.GetHashCode(P1));

}

operator == e operator !=

public static bool operator !=(Person r1, Person r2)

{

return !(r1 == r2);

}

public static bool operator ==(Person r1, Person r2)

{

return (object)r1 == r2 || (r1?.Equals(r2) ?? false);

}

EqualityContract

Além das sobrecargas Equals conhecidas, operator == e operator != , o compilador sintetiza uma nova propriedade EqualityContract. A propriedade retorna um Type objeto que corresponde ao tipo do registro.

* Se o tipo base for object, a propriedade será virtual.
* Se o tipo base for outro tipo de registro, a propriedade será um override.
* Se o tipo de registro for sealed, a propriedade será sealed.

product2 “pode pensar” product1 é o mesmo porque irá comparar propriedades compartilhadas (Name e CategoryId), mas product1 pode pensar que product2 é diferente porque há uma propriedade ausente (ISBN).

EqualityContract existe para “arbitrar um consenso”, então product1.Equals (product2) e product2.Equals (product1) devem retornar false;

using System.Runtime.CompilerServices;

protected virtual Type EqualityContract

{

[CompilerGenerated]

get

{

return typeof(Person);

}

}

A igualdade é implementada consistentemente em qualquer hierarquia de tipos de registro. Dois registros são iguais entre si se suas propriedades forem iguais e seus tipos forem os mesmos, conforme mostrado no exemplo a seguir:

var person = new Person("Bill", "Wagner");

var student = new Student("Bill", "Wagner", 11);

Console.WriteLine(student == person); // false

**3 - Override de GetHashCode()**

Os registros têm uma representação de numérica gerada.

var person = new Person("Bill", "Wagner");

var student = new Student("Bill", "Wagner", 11);

Console.WriteLine(student.GetHashCode()); // -2031643194

Console.WriteLine(person.GetHashCode()); // 327587142

O sintetizado GetHashCode usa o GetHashCode de todas as propriedades e campos declarados no tipo base e no tipo de registro. Esses métodos sintetizados impõem a igualdade baseada em valor em uma hierarquia de herança. Isso significa que um Student nunca será considerado igual a um Person com o mesmo nome. Os tipos dos dois registros devem corresponder e todas as propriedades compartilhadas entre os tipos de registro são iguais.

public override int GetHashCode()

{

return ((EqualityComparer<Type>.Default.GetHashCode(EqualityContract) \* -1521134295

+ EqualityComparer<string>.Default.GetHashCode(FirstName)) \* -1521134295

+ EqualityComparer<string>.Default.GetHashCode(Address)) \* -1521134295

+ EqualityComparer<string>.Default.GetHashCode(LastName);

}

**4 - PrintMembers e ToString()**

O compilador sintetiza dois métodos que dão suporte à saída impressa:

1. **Override do ToString()**
2. **PrintMembers.**

O PrintMembers usa System.Text.StringBuilder como argumento. Ele acrescenta uma lista separada por vírgulas de nomes de propriedade e valores para todas as propriedades no tipo de registro. PrintMembers chama a implementação de base para todos os registros derivados de outros registros. O override do ToString() retorna a cadeia de caracteres produzida por PrintMembers , cercada por { e } .

public record Person

{

string LastName { get; }

string FirstName { get; }

public void TakeNotes() => Console.WriteLine("It's note taking time");

public Person(string first, string last) => (FirstName, LastName) = (first, last);

protected virtual bool PrintMembers(StringBuilder builder)

{

builder.Append("FirstName");

builder.Append(" = ");

builder.Append((object)FirstName);

builder.Append(", ");

builder.Append("LastName");

builder.Append(" = ");

builder.Append((object)LastName);

return true;

}

public override string ToString()

{

var builder = new StringBuilder();

builder.Append(nameof(Person));

builder.Append(" { ");

if (PrintMembers(builder))

builder.Append(" ");

builder.Append("}");

return builder.ToString();

}

}

public record Teacher : Person

{

public string Subject { get; }

public decimal Price { get; }

public void Teachclass() => Console.WriteLine("It's class teaching time");

protected override bool PrintMembers(StringBuilder builder)

{

if (base.PrintMembers(builder))

builder.Append(", ");

builder.Append(nameof(Subject));

builder.Append(" = ");

builder.Append(this.Subject); // or builder.Append(this.P2); if P2 has a value type

builder.Append(", ");

builder.Append(nameof(Price));

builder.Append(" = ");

builder.Append(this.Price); // or builder.Append(this.P3); if P3 has a value type

return true;

}

public override string ToString()

{

var builder = new StringBuilder();

builder.Append(nameof(Teacher));

builder.Append(" { ");

if (PrintMembers(builder))

builder.Append(" ");

builder.Append("}");

return $"{builder.ToString()} is a Class";

}

public Teacher(string first, string last, string sub, decimal price)

: base(first, last) => (Subject, Price) = (sub, price);

}

public sealed record Student : Person

{

public int Level { get; }

protected override bool PrintMembers(StringBuilder builder)

{

if (base.PrintMembers(builder))

builder.Append(", ");

builder.Append(nameof(Level));

builder.Append(" = ");

builder.Append(this.Level);

return true;

}

public override string ToString()

{

var builder = new StringBuilder();

builder.Append(nameof(Student));

builder.Append(" { ");

if (PrintMembers(builder))

builder.Append(" ");

builder.Append("}");

return $"{builder.ToString()}";

}

public Student(string first, string last, int level)

: base(first, last) => Level = level;

}

Por exemplo, o método ToString() retorna um string semelhante ao seguinte código:

var person = new Person("Bill", "Wagner");

var student = new Student("Jean", "Woo", 11);

//Person { FirstName = Bill, LastName = Wagner }

Console.WriteLine(person.ToString());

//Student { FirstName = Jean, LastName = Woo, Level = 11 }

Console.WriteLine(student.ToString());

var teacher = new Teacher("Mike", "Todd", "Math", 11.99m);

//Teacher { FirstName = Mike, LastName = Todd, Subject = Math, Price = 11,99 } is a Class

Console.WriteLine(teacher.ToString());

**5 - Método Construct/Deconstruct**

Os registros também têm um Construtor sintetizado e um método de "clonagem" para a criação de cópias. O Construtor sintetizado tem um argumento do tipo de registro. Ele produz um novo registro com os mesmos valores para todas as propriedades do registro. Esse construtor é private se o registro estiver selado, caso contrário, será protect.

Todos os registros dão suporte à desconstrução. O compilador produz um Deconstruct método para registros posicionais.

public record Person

{

public string LastName { get; }

public string FirstName { get; }

public string Address;

public string City;

public string FavoriteColor;

public Person(string firstName, string lastName, string address, string city, string favoriteColor)

=> (FirstName, LastName, Address, City, FavoriteColor) = (firstName, lastName, address, city, favoriteColor);

public Person(string first, string last) => (FirstName, LastName) = (first, last);

public void Deconstruct(out string firstName, out string lastName, out string address,

out string city, out string favoriteColor)

=> (firstName, lastName, address, city, favoriteColor) = (FirstName, LastName, Address, City, FavoriteColor);

}

O Deconstruct método tem parâmetros que correspondem aos nomes de todas as propriedades públicas no tipo de registro. O Deconstruct método pode ser usado para desconstruir o registro em suas propriedades de componente:

var person = new Person("Bill", "Wagner");

var (first, last, address, city, color) = person;

Console.WriteLine(person.FirstName); // Bill

Console.WriteLine(person.LastName); // Wagner

Console.WriteLine(first); // Bill

Console.WriteLine(last); // Wagner

Console.WriteLine(address); //

Console.WriteLine(city); //

Console.WriteLine(color); //

var hero = new Person("Tony", "Stark", "10880 Malibu Point", "Malibu", "red");

var (firsth, lasth, addressh, cityh, colorh) = hero;

Console.WriteLine(hero.FirstName); // Tony

Console.WriteLine(hero.LastName); // Stark

Console.WriteLine(firsth); // Tony

Console.WriteLine(lasth); // Stark

Console.WriteLine(addressh); // 10880 Malibu Point

Console.WriteLine(cityh); // Malibu

Console.WriteLine(colorh); // red

Console.ReadKey();

**Registros Posicionais**

Os exemplos mostrados até agora usam a sintaxe tradicional para declarar Propriedades. Há uma forma mais concisa chamada registros posicionais. Aqui estão os três tipos de registro definidos anteriormente como registros posicionais:

public record Person2(string FirstName, string LastName);

public record Teacher2(string FirstName, string LastName, string Subject, decimal Price) : Person(FirstName, LastName);

public sealed record Student2(string FirstName, string LastName, int Level) : Person(FirstName, LastName);

Os Registros posicionais, permite uma sintaxe mais curta por uma posição específica de membros: e torna as propriedades FirstName e LastName registro imutável e sua atribuição de valor é determinada por sua posição. A construção (por posição) e a desconstrução (por posição) funcionarão bem com a sintaxe que você já conhece nas versões anteriores do C#.

Essas declarações criam a mesma funcionalidade da versão anterior. Essas declarações terminam com um ponto e vírgula em vez de colchetes porque esses registros não adicionam outros métodos. Você pode adicionar um corpo e também incluir outros métodos:

public record Person

{

string LastName { get; }

string FirstName { get; }

public void TakeNotes() => Console.WriteLine("It's note taking time");

public Person(string first, string last) => (FirstName, LastName) = (first, last);

protected virtual bool PrintMembers(StringBuilder builder)

{

builder.Append("FirstName");

builder.Append(" = ");

builder.Append((object)FirstName);

builder.Append(", ");

builder.Append("LastName");

builder.Append(" = ");

builder.Append((object)LastName);

return true;

}

public override string ToString()

{

var builder = new StringBuilder();

builder.Append(nameof(Person));

builder.Append(" { ");

if (PrintMembers(builder))

builder.Append(" ");

builder.Append("}");

return builder.ToString();

}

}

public record Teacher : Person

{

public string Subject { get; }

public decimal Price { get; }

public void Teachclass() => Console.WriteLine("It's class teaching time");

public Teacher(string first, string last, string sub, decimal price)

: base(first, last) => (Subject, Price) = (sub, price);

}

**INSTRUÇÕES DE NÍVEL SUPERIOR**

Quando criamos um novo aplicativo de console C#, é criado esta estrutura:

using System;

namespace C9\_TopLevel

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hello World!");

}

}

}

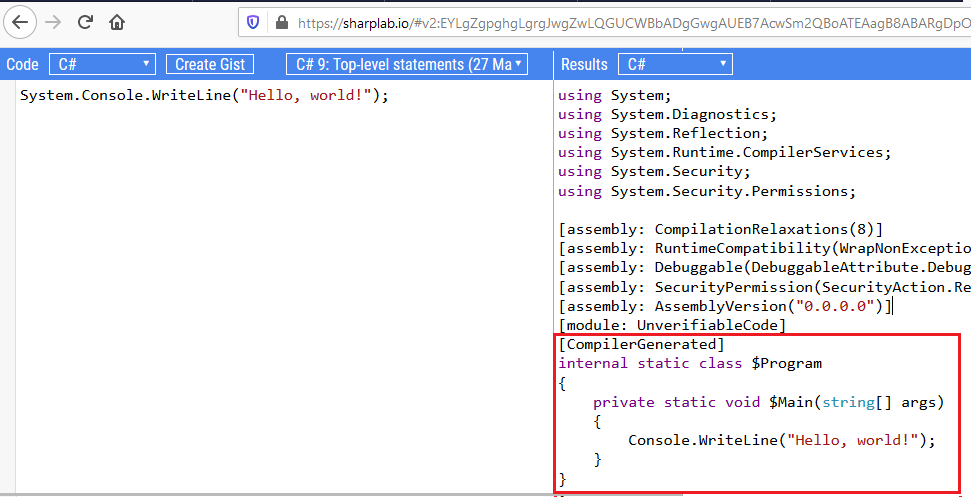
Imagine que você está tentando ensinar a alguém como um programa funciona. Antes mesmo de executar uma linha de código, você precisa falar sobre:

* O que são declarações using e por que precisamos de referência a System?
* O que são namespace, é somente o nome do programa?
* O que é uma classe?
* O que é uma função, static, void?
* O que são parâmetros, o que é uma matriz de string args?

Dificultando o entendimento para novos programadores, especialmente se comparado a linguagens Python ou JavaScript. As instruções de nível superior removem a cerimônia desnecessária de muitos aplicativos, se você quisesse um programa de uma linha, poderia remover todo o código e colocar a diretiva using de modo totalmente qualificado:

System.Console.WriteLine("Hello World!");

Se você olhar o que Roslyn gera, a partir do Sharplab(https://sharplab.io):



Ele irá gerar uma classe de programa e o método principal tradicional para você. Somente um arquivo em seu aplicativo pode usar instruções de nível superior. Se o compilador encontrar instruções de nível superior em vários arquivos de origem, será um erro. Também será um erro se você combinar instruções de nível superior com um método de ponto de entrada de programa declarado, normalmente um Main método. De certa forma, você pode imaginar que um arquivo contém as instruções que normalmente estaria no Main método de uma Program classe.

Um dos usos mais comuns para esse recurso é a criação de materiais de ensino. No entanto, os desenvolvedores experientes também encontrarão muitos usos para esse recurso. As instruções de nível superior permitem uma experiência semelhante a um script para experimentação semelhante ao que os notebooks Jupyter fornecem. As instruções de nível superior são ótimas para pequenos programas de console e utilitários. O Azure Functions é um caso de uso ideal para instruções de nível superior.

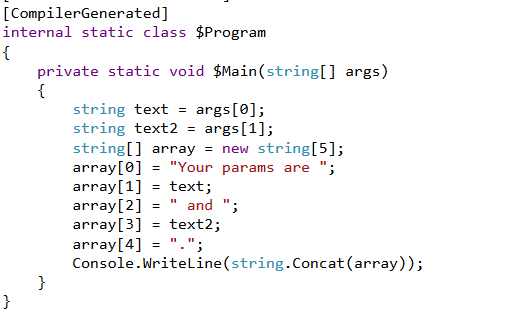
O mais importante é que as instruções de nível superior não limitam o escopo ou a complexidade do aplicativo. Essas instruções podem:

* Acessar ou usar qualquer classe .NET.
* Normalmente, isso é feito analisando os acessar uma matriz de cadeias de caracteres denominadas args[] que você passa para o método Main como vimos anteriormente. Agora os args estão disponíveis como um parâmetro oculto, o que significa que você deve ser capaz de acessá-los sem transmiti-los. Digamos que eu quisesse algo assim:

var param1 = args[0];

var param2 = args[1];

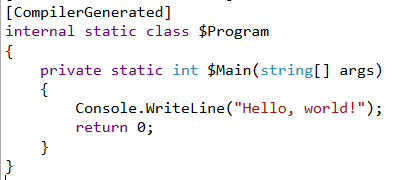
System.Console.WriteLine($"Your params are {param1} and {param2}.");



* Não limitam o uso de valores de retorno. Se retornarem um valor inteiro, esse valor se tornará o código de retorno de inteiro de um método sintetizado Main.

System.Console.WriteLine("Hello, world!");

return 0;



* Podem conter expressões assíncronas utilizando await.

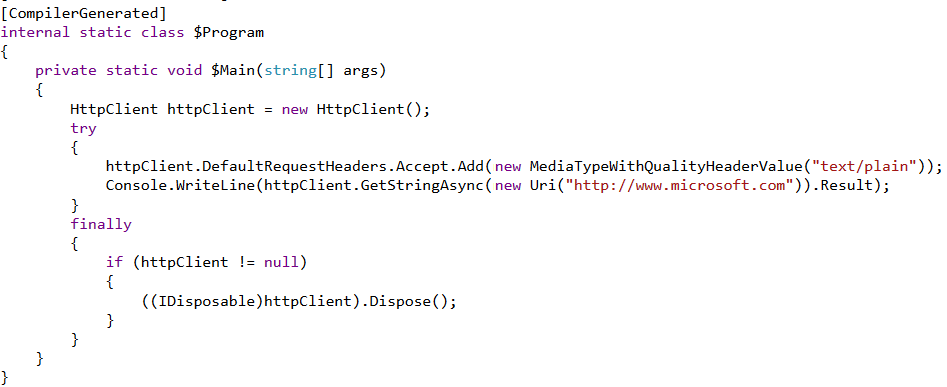
using (var httpClient = new HttpClient())

{

httpClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("text/plain"));

Console.WriteLine(httpClient.GetStringAsync(new Uri("http://www.microsoft.com")).Result);

}



Se as expressões assíncronas retorna um Task, ou Task<int>.

async System.Threading.Tasks.Task<string> AwaitableMethodAsync()

{

using (var httpClient = new System.Net.Http.HttpClient())

{

httpClient.DefaultRequestHeaders.Accept.Add(new System.Net.Http.Headers.MediaTypeWithQualityHeaderValue("text/plain"));

string tarefa = await httpClient.GetStringAsync(new System.Uri("http://www.microsoft.com"));

return tarefa;

}

}

var textao = await AwaitableMethodAsync();

System.Console.WriteLine(textao);

Se você olhar o que Roslyn gera, a partir do Sharplab, temos um extenso código que utiliza um TaskAwaiter e um AsyncStateMachine.

**MELHORIAS NA CORRESPONDÊNCIA DE PADRÕES**

Este não é um recurso completamente novo, mas algo que evoluiu desde que foi lançado no C # 7, embora na forma básica. A correspondência de padrões melhorou muito no C# 8, com o operador is e com expressões de switch. O C# 9 inclui novas melhorias de correspondência de padrões, muitos deles são baseados na sintaxe de switch aprimorada do C# 8.

string languagePhrase = languageInput switch

{

"C#" => "C# is fun!",

"JavaScript" => "JavaScript is mostly fun!",

\_ => throw new Exception("You code in something else I don't recognize."),

};

A correspondência de padrões permite que você simplifique os cenários em que é necessário gerenciar de forma coesa os dados de diferentes fontes. Um exemplo óbvio é quando você chama uma API externa onde você não tem nenhum controle sobre os tipos de dados que está recebendo. Claro, normalmente você criaria tipos em seu aplicativo para todos os diferentes tipos que poderia obter dessa API. Então, você construiria um modelo de objeto a partir desses tipos. Imagine se você estiver trabalhando com várias APIs.

\*\*\* and Padrões de conjuntiva\*\*\* exigem os dois padrões para corresponder

\*\*\* or Padrões de disjunctive\*\*\* exigem qualquer padrão para corresponder

Partindo de um código C# 8 switch:.

var person = new Person

{

FirstName = "Tony",

LastName = "Ramos",

Age = 45

};

string Switch\_C8(object pessoa) => pessoa switch

{

Person s when s.Age < 10 => "Criança",

Person s when s.Age <= 40 => "Adulto",

Person \_ => "Senior",

\_ => throw new ArgumentException("I do not know this one", nameof(pessoa))

};

Console.WriteLine(Switch\_C8(person)); "Senior"

A correspondência de padrões pode ser usado em qualquer contexto em que os padrões são permitidos:

* **Expressões is**
* **Expressões switch**
* **Padrões aninhados**
* **Padrão switch/case.**

C # 9 inclui novas melhorias de correspondência de padrões:

**Padrões de tipo simples**

Atualmente, um padrão de tipo precisa declarar um identificador quando o tipo corresponder, mesmo se esse identificador for um discard \_ , como em Person \_ => "Senior". Mas agora você pode simplesmente escrever o tipo:

static string SwitchTipoSimples\_C9(object simply) => simply switch

{

Person s when s.Age < 10 => "Criança",

Person s when s.Age <= 40 => "Adulto",

Person => "Senior",

\_ => throw new ArgumentException("I do not know this one", nameof(simply))

};

Permitimos um tipo como padrão, podemos também escrever:

void TypePattern(object tipo1, object tipo2)

{

var t = (tipo1, tipo2);

if (t is (int, string)) {

Console.WriteLine("É inteiro ou string!");

}

switch (tipo1)

{

case int:

Console.WriteLine("É inteiro!");

break;

case System.String:

Console.WriteLine("É string!");

break;

}

}

**Padrões relacionais**

Podemos simplificar esse switch usando os operadores relacionais, como <, <=, > e >=. Podemos

static string SwitchRelacional\_C9(Person hero) => hero.Age switch

{

< 10 => "Criança",

<= 40 => "Adulto",

\_ => "Senior"

};

**Padrões lógicos**

Podemos usar operadores lógicos conjuntivos (and), disjuntivos (or) ou de negação (not) como uma opção mais legível.

static string SwitchLogico\_C9(Person hero) => hero.Age switch

{

1 or 2 => "Bebê",

< 10 and < 18 => "Criança",

<= 40 => "Adulto",

\_ => "Senior"

};

É conveniente se você usar o padrão de constante nula em conjunto com padrão not para dividir o tratamento de casos desconhecidos dependendo de serem nulos ou em não nulos

static string SwitchTipoSimples\_C9(object simply) => simply switch

{

Person s when s.Age < 10 => "Criança",

Person s when s.Age <= 40 => "Adulto",

Person => "Senior",

not null => throw new ArgumentException($"I do not know this one: {simply}", nameof(simply)),

null => throw new ArgumentNullException(nameof(simply))

};

O not é útil também em condições if contendo expressões is onde, oferecendo maior clareza sobre a lógica da negação.

if (!(person is Person)) { }

if (person is not Person) { }

Como alternativa, com parênteses opcionais para deixá-lo claro que and tem precedência maior do que or:

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public int Age { get; set; }

public bool IsChild() => Age is >= 10 and <= 18;

public bool TakeNotes(char c) => c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';

public bool TakeNotesOrSeparator(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';

}

**DESEMPENHO E INTEROPERABILIDADE**

Três novos recursos melhoram o suporte para a interoperabilidade nativa e bibliotecas de nível baixo que exigem alto desempenho:

* **Inteiros de tamanho nativo**
* **Ponteiros de função**
* **Suprimir emissão do sinalizador localsinit**

**Inteiros de tamanho nativo**

Os inteiros de tamanho nativo são projetados para ser um número inteiro cujo tamanho é específico para a plataforma. Em outras palavras, uma instância desse tipo deve ser de 32 bits em hardwares/sistemas operacionais de 32 bits, 64 bits em hardwares/sistemas operacionais de 64 bits. Inteiros de tamanho nativo nint e nuint, são tipos inteiros. Os tipos nint e nuint são representados pelos tipos subjacentes System.IntPtr e System.UIntPtr com o compilador mostrando conversões e operações adicionais para esses tipos como ints nativos.

* As constantes nint estão no intervalo [int.MinValue, int.MaxValue].
* As constantes nuint estão no intervalo [uint.MinValue, uint.MaxValue].
* Não há MinValue ou MaxValue o nint ou nuint porque esses valores não podem ser emitidos como constantes pois dependem do tamanho nativo de um inteiro no computador de destino. Exceto nuint.MinValue, que tem um MinValue de 0

O compilador executa o dobramento constante para todos os operadores unários e binários usando os tipos System.Int32 e System.UInt32, se o resultado não couber em 32 bits, a operação será executada em tempo de execução e não será considerada uma constante.

O CLR/JIT/MSIL suporta a definição e o uso de inteiros nativos/inteiros sem sinal. Desde o CLR do .NET 4.0, é possível adicionar/subtrair um inteiro de um System.IntPtr/System.UIntPtr, e é possível fazer comparações == e != com outro System.IntPtr/System.UIntPtr, mas qualquer outra operação de comparação não é possível, ou seja, eles não podem ser comparados com >, > = etc. Então System.IntPtr/System.UIntPtr permanece muito básico na quantidade de aritmética de ponteiro.

C# 9 traz o que mono trouxe antes: suporte de idioma para tipos inteiros assinados e não assinados de tamanho nativo com palavras-chave nint e nuint. A motivação aqui é para cenários de interoperabilidade e para bibliotecas de baixo nível, portanto, o uso não é muito frequente. Os inteiros de tamanho nativo podem aumentar o desempenho em cenários em que a matemática de inteiros é usada extensivamente e precisa ter o desempenho mais rápido possível.

nint x = 3;

int y = 3;

long v = 10;

Console.WriteLine(nint.Equals(x, y)); // False

Console.WriteLine(nint.Equals(x, (nint)y)); // True

Console.WriteLine(y + 1 > x); // True

Console.WriteLine(y - 1 == x); // False

Console.WriteLine(typeof(nint)); // System.IntPtr

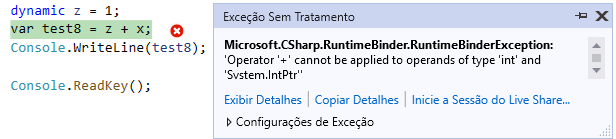
Console.WriteLine(typeof(nuint)); // System.UIntPtr

Console.WriteLine((x + 1).GetType()); // System.IntPtr

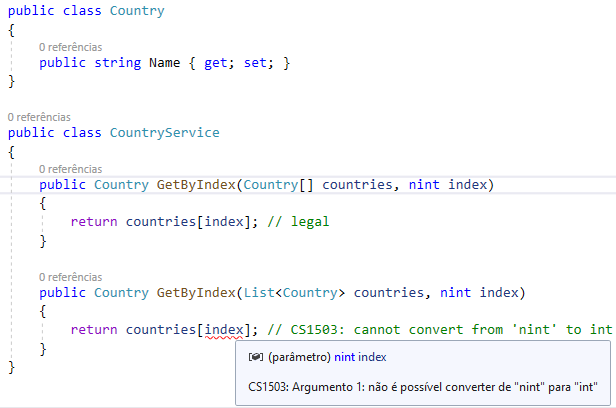
Console.WriteLine((x + y).GetType()); // System.IntPtr

Console.WriteLine((x + v).GetType()); // System.Int64

Quando você adiciona um int a um nint, o resultado é um nint, mas se você adicionar um long a um nint, o resultado será um longo. Isso ocorre porque o nativo dependendo da plataforma pode ser um inteiro de 32 bits ou um inteiro de 64 bits.



Você pode notar que os arrays suportam tipo assinado de tamanho nativo como índice, mas não listas, exemplo:



**Ponteiros de função**

Os ponteiros de função do C # 9.0, fornecem uma maneira eficiente de declarar ponteiros para chamar funções nativas gerenciadas e não gerenciadas. Ponteiros de função é uma variável que armazena o endereço de uma função que pode ser chamada posteriormente por meio desse ponteiro de função. Os ponteiros de função podem ser invocados e passados argumentos exatamente como em uma chamada de função normal.

O runtime oferece suporte e complementar as partes relacionadas à interoperabilidade do recurso, fornecendo uma solução simétrica para chamar funções gerenciadas a partir do código nativo.

UnmanagedCallersOnlyAttribute

UnmanagedCallersOnlyAttribute indica que uma função será chamada apenas a partir do código nativo, permitindo que o runtime reduza o custo de chamar a função gerenciada. Para limitar a complexidade do cenário, o uso deste atributo é restrito a métodos que devem:

* Seja estático
* Só tem argumentos blittable
  + Remove a dependência de qualquer lógica especial de empacotamento
* Não ser chamado de código gerenciado
  + Limita os cenários que precisam ser tratados (por exemplo, nenhuma chamada por meio de reflexão), permitindo que o foco permaneça na redução do custo de chamar a função gerenciada a partir do código nativo

Um cenário de uso básico de passar um retorno de chamada gerenciado para uma função nativa teria, sem UnmanagedCallersOnlyAttribute, a seguinte aparência:

private static void Callback(int i)

{

throw new NotImplementedException();

}

private delegate void CallbackDelegate(int i);

private static CallbackDelegate s\_callback = new CallbackDelegate(Callback);

static void Main(string[] args)

{

IntPtr callback = Marshal.GetFunctionPointerForDelegate(s\_callback);

NativeFunctionWithCallback(callback);

}

[DllImport("NativeLib")]

private static extern void NativeFunctionWithCallback(IntPtr callback);

O acima requer a alocação de um delegado e o marshalling desse delegado para um ponteiro de função. Se a função nativa que está sendo chamada puder reter o retorno de chamada, também precisamos garantir que o delegado não seja coletado como lixo. Esse detalhe costuma ser esquecido, levando a travamentos intermitentes de “retorno de chamada no delegado coletado”. Com a combinação de ponteiros de função e UnmanagedCallersOnlyAttribute, isso pode ser reescrito como:

Ponteiros de função fornecem uma sintaxe fácil para acessar os opcodes de IL ldftn e calli. Você pode declarar ponteiros de função usando a sintaxe de tipo de ponteiro new delegate\*. Invocar o tipo delegate\* usa calli, em contraste com um delegado que usa callvirt no método Invoke(). Sintaticamente, as invocações são idênticas. Invocação de ponteiro de função usa a Convenção managed de chamada. Você adiciona a palavra-chave unmanaged após a sintaxe delegate\* para declarar que deseja a Convenção de chamada unmanaged. Outras convenções de chamada podem ser especificadas usando atributos na declaração delegate\*.

**Suprimir emissão do sinalizador localsinit**

Por fim, você pode adicionar o System.Runtime.CompilerServices.SkipLocalsInitAttribute para instruir o compilador C# Roslyn a não emitir o localsinit sinalizador. Esse sinalizador instrui o CLR a inicializar zero todas as variáveis locais. O localsinit sinalizador tem sido o comportamento padrão para C# desde 1,0. No entanto, a inicialização zero extra pode ter um impacto mensurável no desempenho em alguns cenários. Em particular, quando você usa o stackalloc. Nesses casos, você pode adicionar o SkipLocalsInitAttribute ao Core CLR. Você pode adicioná-lo a um único método ou propriedade, ou a um class, struct interface ou até mesmo a um módulo. Esse atributo não afeta abstract os métodos; ele afeta o código gerado para a implementação.

Esses recursos podem melhorar o desempenho em alguns cenários. Eles devem ser usados somente após um parâmetro de comparação cuidadoso antes e depois da adoção. O código que envolve inteiros de tamanho nativo deve ser testado em várias plataformas de destino com tamanhos de inteiro diferentes. Os outros recursos exigem código não seguro.

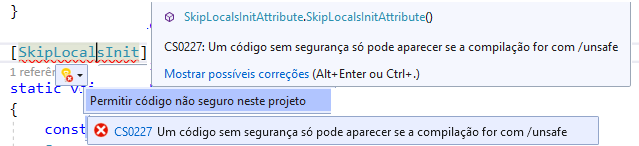
No aplicativo de alto desempenho, o custo da inicialização zero forçada pode ser perceptível. É particularmente perceptível quando o stackalloc é usado. Em alguns casos, o JIT pode Elide inicialização zero inicial de locais individuais quando tal inicialização é "eliminada" por atribuições subsequentes. Nem todos os JITs fazem isso e essa otimização tem limites. Ele não ajuda com o stackalloc .

Para ilustrar que o problema é real, há um bug conhecido em que um método que não contém nenhum IL local não teria um localsinit sinalizador. O bug já está sendo explorado pelos usuários, colocando stackalloc -se em tais métodos, intencionalmente, para evitar os custos de inicialização. Isso é apesar do fato de que a ausência de IL locais é uma métrica instável e pode variar dependendo das alterações na estratégia de codegen. O bug deve ser corrigido e os usuários devem ter uma maneira mais documentada e confiável de suprimir o sinalizador.

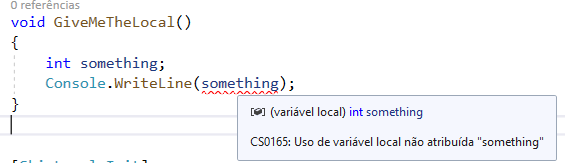
Desvantagens

* Os compiladores antigos/outros podem não honrar o atributo. Ignorar o atributo é um comportamento compatível. Pode resultar apenas em uma ligeira queda de desempenho.
* O código sem localinits sinalizador pode disparar falhas de verificação. Os usuários que solicitam esse recurso geralmente não são preocupados com a capacidade de verificação.
* A aplicação do atributo em níveis mais altos do que um método individual tem efeito não local, que é observável quando stackalloc é usado. Ainda assim, esse é o cenário mais solicitado.

Esse novo recurso só pode ser usado com a opção unsafe do compilador:

****

O CLR suporta um atributo .localsinit para métodos que, quando definido, garante que todas as variáveis ​​sejam zeradas antes de executar o método. Veja o seguinte erro:



Isso não é válido em C# porque ele verifica a atribuição definitiva. A análise de atribuição definitiva é um processo que verifica se todas as variáveis ​​foram atribuídas antes de serem utilizadas. Isso me ajuda a evitar erros como desenvolvedor, pois eu nunca deveria realmente querer usar variáveis ​​antes de atribuí-las, mesmo que seja uma pequena questão de escrever int something = default para deixar as coisas claras.

A equipe C# tem nos últimos anos priorizado o desempenho da linguagem para promover o C# como um sério competidor para aplicativos de alto desempenho. (por exemplo, em structs e readonly). Por exemplo, os desenvolvedores de jogos precisam chegar à velocidade do metal para fornecer uma experiência de usuário tranquila e, portanto, procurar qualquer oportunidade para remover ineficiências. Portanto, não é surpreendente que alguns vejam o sinalizador localsinit como um obstáculo desnecessário ao desempenho, especialmente quando se considera que o C# tem Atribuição definida embutida e, portanto, mesmo se os locais não fossem zerados, eles ainda seriam sobrescritos na maioria das vezes.

Assim nasceu SkipLocalsInitAttribute para instruir Roslyn a evitar a emissão .localsinit de métodos críticos de desempenho. O maior ganho aqui é para métodos com um quadro de pilha grande, portanto, especialmente para métodos stackalloc que usam o qual aloca um array na pilha. O CLR zera esse array se .localsinit estiver definido, e se o array for grande e o programa estiver prestes a atribuir todos os slots do array de qualquer maneira, é um desperdício total. Vamos iniciar a pré-visualização do VS2019 com um projeto unsafe e e criar um método que mantém uma contagem de invocação na pilha usando stackalloc:

static void Main(string[] args)

{

for (int i = 0; i < 10; i++)

{

PrintRunningCount();

}

}

[SkipLocalsInit]

static void PrintRunningCount()

{

const int Magic = 123214;

Span<int> arr = stackalloc int[2];

if (arr[0] != Magic)

{

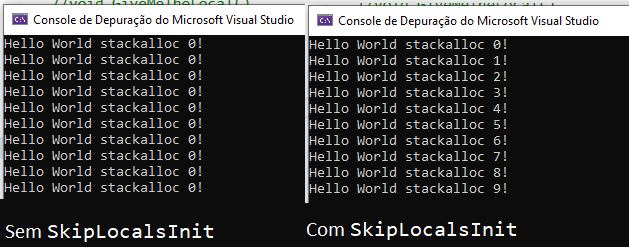
arr[0] = Magic;

arr[1] = 0;

}

Console.WriteLine($"Hello World stackalloc {arr[1]++}!");

}



Podemos ver que cada vez que PrintRunningCount é invocado, ele cria um array que contém 2 valores. O primeiro arr[0]é um valor mágico para controlar se o método está sendo executado pela primeira vez. Como localsinit não está mais lá para ajudar, não podemos presumir que nosso contador comece em 0. O segundo valor contém a contagem real.

Este exemplo é bastante seguro se você me perguntar; o desenvolvedor optou por usar [SkipLocalsInit] e, portanto, pode contar com uma atribuição definida, exceto para stackalloc. No entanto, agora nos movemos para o território mais inseguro ... para quando a Análise de Atribuição Definida falhar! As regras são bem claras quando se trata de variáveis ​​atribuídas como um todo ( variable = ...). No entanto, as estruturas são especiais e podem ser atribuídas por partes, como no exemplo a seguir.

static void Main(string[] args)

{

for (int i = 0; i < 10; i++)

{

PrintRunningCountStruct();

}

}

struct StackCounter

{

public long A;

public static long Increment(ref StackCounter c) { return c.A++; }

}

[SkipLocalsInit]

static void PrintRunningCountStruct()

{

StackCounter counter;

counter.A = default;

// counter is definitely assigned since we've set all of it's fields

var count = StackCounter.Increment(ref counter);

Console.WriteLine($"Hello World {count}!");

}

Compile o acima e conforme o esperado, você observará impresso algumas vezes no console Hello World 0!. No entanto, esta regra para atribuição definitiva tem um problema. E se alguém mover StackCounter para outra biblioteca e compilar o programa de console como está? A Atribuição Definida é verificada no momento da compilação e, portanto, o programa de console é compilado com sucesso.

E se alguém "atualizar" a biblioteca, recompilá-la e sobrescrever o arquivo dll antigo? Esse tipo de coisa acontece com frequência - por exemplo, um projeto pode fazer referência a uma versão mais antiga de uma biblioteca na mesma solução de um projeto que faz referência a uma nova versão e, quando tudo é construído, o mais novo é usado para ambos. No meu caso, eu "atualizei" StackCounter da seguinte maneira.

public struct StackCounter

{

public long A;

public long Count;

public long Magic;

public static long Increment(ref StackCounter c)

{

const long Magic = 23872139472367;

if (c.Magic != Magic)

{

c.Magic = Magic;

c.Count = 0;

}

return c.Count++;

}

}

[SkipLocalsInit]

static void PrintRunningCountStruct()

{

long count;

StackCounter counter;

//StackCounter counter = default(StackCounter);

//StackCounter counter = new StackCounter();

counter.A = default;

counter.Count = default;

counter.Magic = default;

// counter is definitely assigned since we've set all of it's fields

count = StackCounter.Increment(ref counter);

Console.WriteLine($"Hello World {count}!");

}

Se executarmos o mesmo programa agora, veremos que a contagem aumenta de Hello World 0! a Hello World 9!. Para quem está olhando para a fonte, este exemplo é muito menos claramente quebrado do que o stackallocexemplo. É o desenvolvedor da biblioteca que agora armazena o estado na pilha de outro método.

**APRIMORAMENTO DO TYPING**

C # 9 inclui suporte aprimorado para target-typing. Target-typing o que C # usa, normalmente em expressões, para obter um tipo de seu contexto. Um exemplo comum seria o uso da palavra-chave var. O tipo pode ser inferido de seu contexto, sem que você precise declará-lo explicitamente.

O aprimoramento do target-typing no C # 9 vem em:

* **Expressão new**
* **Expressão condicional de Target-typing ?? e ?. (Ainda não funciona)**

**Expressão new**

Muitos dos outros recursos ajudam a escrever código com mais eficiência. No C# 9,0, você pode omitir o tipo em uma expressão new quando o tipo do objeto criado já é conhecido. O uso mais comum está em declarações de campo:

Person person = new("Bob", "Dyla");

List<Person> \_observations = new();

var personList = new List<Person>

{

new ("Bob", "Marei"),

new ("Bob", "Quened"),

new ("Uncle", "Bob"),

new ("Rick", "Bob")

};

O tipo de destino New também pode ser usado quando você precisa criar um novo objeto para passar como um parâmetro para um método. Considere um método Encontro() com a seguinte assinatura:

public class Person

{

//....

public void Encontro(DateTime dia, Opcoes options) => Console.WriteLine("It's class teaching time: " + dia.ToString() + "-" + options.texto);

}

public class Opcoes

{

private int \_num;

private string \_txt;

public Opcoes(){ }

public Opcoes(int numero, string texto)

{

\_num = numero;

\_txt = texto;

}

public int numero { get { return \_num; } set { \_num = value; } }

public string texto { get { return \_txt; } set { \_txt = value; } }

}

Você pode chamá-lo da seguinte maneira:

person.Encontro(DateTime.Now.AddDays(2), new(1, "oi"));

Outro bom uso para esse recurso é combiná-lo com propriedades init somente para inicializar um novo objeto:

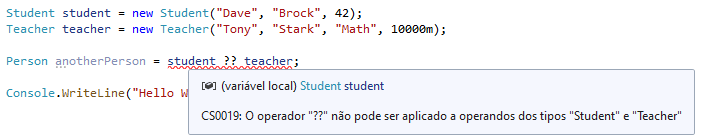
Opcoes opcoes = new() { numero = 42, texto = "Seattle, WA" };

Você pode retornar uma instância criada pelo construtor padrão usando uma expressão return new();.

**Expressões Condicionais de Target-typing**

Um recurso semelhante melhora a resolução de tipo de destino de expressões condicionais. Com essa alteração, as duas expressões não precisam ter uma conversão implícita de uma para a outra, mas podem ter conversões implícitas em um tipo comum. Você provavelmente não perceberá essa alteração. O que você observará é que algumas expressões condicionais que antes exigiam conversões ou que não compilaram agora só funcionam.

Por falar em declarações ternárias, agora podemos inferir tipos usando os operadores condicionais. Isso funciona bem com ??, o operador de coalescência nula. O ?? operador retorna o valor do que está à esquerda se não for nulo. Caso contrário, o lado direito é avaliado e retornado. Então, imagine que temos alguns objetos que compartilham a mesma classe base, como este:



**Tipos de Retorno Covariantes**

Com a covariância do tipo de retorno, você pode substituir um método de classe base (que possui um tipo menos específico) por um que retorna um tipo mais específico. Você não precisará mais implementar soluções alternativas de interface. Antes do C # 9, você teria que retornar o mesmo tipo em uma situação como esta:

public class Person

{

private string \_firstName;

private string \_lastName;

public Person(string firstName, string lastName)

{

\_firstName = firstName;

\_lastName = lastName;

}

public virtual Person GetPerson()

{

return new Person(\_firstName, \_lastName);

}

}

public class Student : Person

{

private string \_favoriteSubject;

private string \_firstName;

private string \_lastName;

public Student(string firstName, string lastName, string favoriteSubject) : base(firstName, lastName)

{

\_favoriteSubject = favoriteSubject;

\_firstName = firstName;

\_lastName = lastName;

}

public override Person GetPerson()

{

// you can return the child class, but still return a Person

return new Student(\_firstName, \_lastName, "None");

}

}

Agora, você pode retornar o tipo mais específico em C# 9.

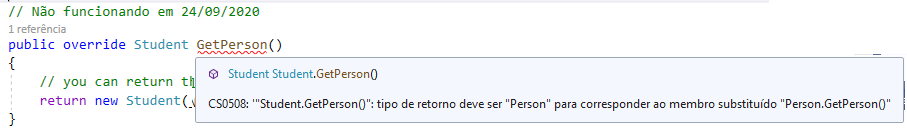
public override Student GetPerson()

{

// you can return the child class, but still return a Person

return new Student(\_firstName, \_lastName, "None");

}



Isso pode ser útil para registros e para outros tipos que dão suporte a métodos de clonagem ou de alocador virtual.

**FUNÇÕES ANÔNIMAS ESTÁTICAS**

A partir do C# 9,0, você pode adicionar o static modificador a expressões lambda ou a métodos anônimos. As expressões lambda estáticas são análogas às static funções locais: uma função lambda ou anônima estática não pode capturar variáveis locais ou estado de instância. O static modificador impede capturar inadvertidamente o estado do contexto envolvente, o que pode resultar na retenção inesperada de objetos capturados (captura acidental de outras variáveis) ou em alocações adicionais inesperadas.

static void Main(string[] args)

{

int y = 10;

someMethod(x => x + y); // captures 'y', causing unintended allocation.

}

private static void someMethod(Func<int, int> soma\_func)

{

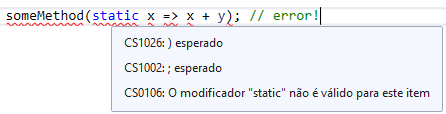
Console.WriteLine(soma\_func(2));//12

Console.WriteLine(soma\_func(12));//22

}

Em C #, as funções anônimas que se referem a variáveis locais requerem a alocação de um objeto temporário. O parâmetro local é então movido para fora do método e para dentro do objeto, de forma que continuará a existir após o término da função atualmente em execução. Isso é necessário porque uma função anônima pode existir por mais tempo do que a função que a criou.

Adicionar a palavra-chave static indica que a função anônima impede essa alocação de memória.



Para corrigir o erro, a variável y precisa ser transformada em um campo constante ou estático.

const int y = 10;

algumMétodo (estático x => x + y); // OK :-)

Abaixo estão as principais regras para este recurso:

* O modificador estático em um lambda ou método anônimo indica é uma função anônima estática.
* Uma função anônima estática pode fazer referência a membros estáticos do escopo delimitador.
* Uma função anônima estática não pode capturar o estado do escopo delimitador. Como resultado, locais, parâmetros e isso do escopo delimitador não estão disponíveis em uma função anônima estática.
* Uma função anônima estática não pode fazer referência a membros de instância de uma referência implícita ou explícita this ou base.
* Uma função anônima estática pode fazer referência a membros estáticos do escopo delimitador.
* Uma função anônima estática pode fazer referência a definições constantes do escopo delimitador.
* nameof () em uma função anônima estática pode referenciar locais, parâmetros, ou this ou base do escopo envolvente.
* As regras de acessibilidade para membros privados no escopo delimitador são as mesmas para funções anônimas estáticas e não estáticas.
* Nenhuma garantia é feita se uma definição de função anônima estática é emitida como um método estático em metadados. Isso é deixado para a implementação do compilador para otimizar.
* Uma função local não estática ou função anônima pode capturar o estado de uma função anônima estática envolvente, mas não pode capturar o estado fora da função anônima estática envolvente.
* Remover o modificador estático de uma função anônima em um programa válido não altera o significado do programa.

**PARÂMETROS DISCARD DE LAMBDA**

Permitir usar os descartes como parâmetros para:

* Expressões Lambdas: (\_, \_) => 0, (int \_, int \_) => 0
* Métodos anônimos: delegate (int \_, int \_) { return 0; }

Essa conveniência permite que você evite nomear o argumento, e o compilador pode evitar usá-lo. Você usa o \_ para qualquer argumento. Permite que o lambda tenha várias declarações dos parâmetros denominados \_. Nesse caso, os parâmetros são "descartados" e não podem ser usados dentro do lambda.

**ATRIBUTOS EM FUNÇÕES LOCAIS**

**Função Local Estática**

A função local estática é um novo recurso introduzido no C # 8.0. Antes do C # 8.0, não podíamos ter uma função local como uma função estática. Em C # 7.x, o código abaixo não é compilado.

int num1 = 5, num2 = 7;

static int Sum(int x, int y)

{

return x + y;

};

var result = Sum(num1, num2);

Console.WriteLine(result); //12

Enquanto no C # 8.0, o código acima é compilado com sucesso. Mas como sabemos que existem algumas condições com função estática, essas condições também se aplicam à função estática local. O último exemplo de código que tomamos também pode ser reescrito conforme mostrado abaixo:

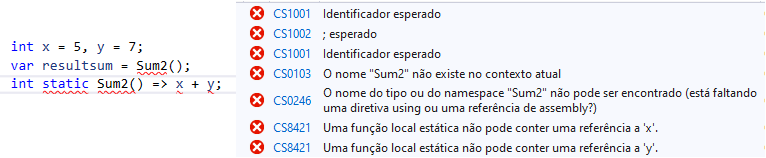
int num1 = 5, num2 = 7;

int Sum2() => num1 + num2;

var resultsum = Sum2();

Console.WriteLine(resultsum); //12

Portanto, no trecho de código acima, podemos ver que estamos usando a variável do método pai em uma função local e essas variáveis são variáveis não estáticas. Mas se tentarmos colocar uma palavra-chave estática com a função local, que está acessando variáveis do método pai, então não funcionará. A seguir está a captura de tela para este princípio.



Por fim, agora você pode aplicar atributos a funções locais. Por exemplo, você pode aplicar anotações de atributo anulável ou condicional a funções locais.

int num1 = 5, num2 = 7;

int Sum2() => num1 + num2;

var resultsum = Sum2();

Console.WriteLine(resultsum); //12

Sum\_C9(num1, num2);

[Conditional("DEBUG")]

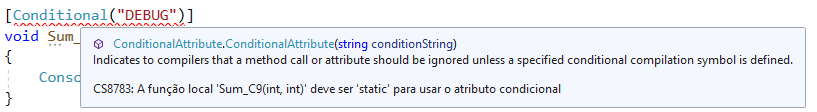
static void Sum\_C9([NotNull] int x, [NotNull] int y)

{

Console.WriteLine(x + y); //12

}

Se retirar o modificador static emitirá erro no atributo Condicional, mas não terá problemas no atributo NotNull



Outro exemplo de uso com EnumeratorCancellation no parâmetro CancelamentoToken de uma função local que implementa um iterador assíncrono, que é comum ao implementar operadores de consulta.

public static IAsyncEnumerable<int> Where(this IAsyncEnumerable<int> source, Func<int, int> predicate)

{

if (source == null)

throw new ArgumentNullException(nameof(source));

if (predicate == null)

throw new ArgumentNullException(nameof(predicate));

return Core();

async IAsyncEnumerable<int> Core([EnumeratorCancellation] CancellationToken token = default)

{

await foreach (var item in source.WithCancellation(token))

{

if (predicate(item) == 1)

{

yield return item;

}

}

}

}

**SUPORTE A EXTENSÃO GETENUMERATOR PARA FOREACH LOOPS**

Alguns tipos de coleção oferecem o método GetEnumerator. Este método retorna um objeto Enumerator que pode ser usado para percorrer a coleção. Não é necessário conhecimento completo do tipo de coleção real.

O exemplo abaixo mostra como funciona o método GetEnumerator no tipo List. Apartir de uma Lista(int) de inteiros o GetEnumerator retorna um objeto Enumerador List (int). Este objeto implementa IEnumerator (int).

static void Main(string[] args)

{

List<int> list = new List<int>();

list.Add(1);

list.Add(5);

list.Add(9);

List<int>.Enumerator e = list.GetEnumerator();

Write(e);

Console.ReadKey();

}

static void Write(IEnumerator<int> e)

{

while (e.MoveNext())

{

int value = e.Current;

Console.WriteLine(value);

}

}

Podemos então escrever um método Write que recebe IEnumerator(int) e percorre seus elementos com uma construção while e a instrução MoveNext. O IEnumerator é uma interface implementada por objetos Enumerator e é diferente de IEnumerable. IEnumerable é geralmente é usado em uma coleção que pode ser repetida com foreach-loops. Outros tipos de coleção também fornecem GetEnumerator. Por exemplo,

* LinkedList
* Dictionary

**O loop foreach**

O loop foreach é usado para iterar sobre uma coleção e automaticamente armazena o item atual em uma variável de loop. O loop foreach monitora onde está a coleção e protege você contra a iteração após o final da coleção. O C# fornece a instrução foreach para iterar com coleções de itens em que a quantidade não é conhecida no tempo de execução, como alocações dinâmicas com base na entrada do usuário. Por exemplo, você pode criar uma matriz de caracteres a partir dos caracteres individuais de uma sequência de texto inserida por um usuário em tempo de execução. Outras possibilidades podem ser um conjunto de dados criado após o acesso a um banco de dados. Nos dois casos, você não saberá o número de valores no momento em que escreve o código. O código abaixo mostra um exemplo de como usar o loop foreach.

int[] values = { 1, 2, 3, 4, 5, 6 };

foreach (int i in values)

{

Console.Write(i);

}

Como você pode ver, o loop foreach armazena automaticamente o item atual em uma variável fortemente tipada. Você pode usar as instruções continue e interromper para influenciar o funcionamento do loop foreach. Coleções são tipicamente matrizes, mas também outros objetos .NET que implementaram as interfaces IEnumerable. A variável de loop não pode ser modificada. Você pode fazer modificações no objeto para o qual a variável aponta, mas não pode atribuir um novo valor a ele. O código a seguir mostra essas diferenças.

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

}

static void CannotChangeForeachIterationVariable()

{

var people = new List<Person>

{

new Person() { FirstName ="John", LastName ="Doe"},

new Person() { FirstName ="Jane", LastName = "Doe"},

};

foreach (Person p in people)

{

p.LastName = "Changed"; // This is allowed

//p = new Person(); // This gives a compile error

}

}

Você pode entender esse comportamento quando souber como o foreach realmente funciona. Quando o compilador encontra uma instrução foreach, ele gera algum código em seu nome; foreach é um açúcar sintático que permite escrever um código de maneira agradável. O exemplo abaixo mostra o que está acontecendo.

var people = new List<Person>{

new Person() { FirstName ="John", LastName ="Doe"},

new Person() { FirstName ="Jane", LastName = "Doe"}};

List<Person>.Enumerator e = people.GetEnumerator();

try

{

Person v;

while (e.MoveNext())

{

v = e.Current;

}

}

finally

{

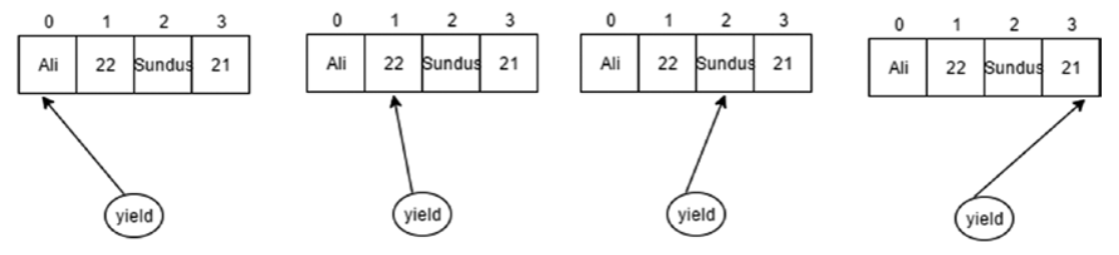
System.IDisposable d = e as System.IDisposable;

if (d != null) d.Dispose();

}

Se você alterar o valor de e.Current para outra coisa, o padrão do iterador não pode determinar o que fazer quando o e.MoveNext é chamado. É por isso que não é permitido alterar o valor da variável de iteração em uma instrução foreach.

É importante saber que o método foreach da linguagem C# trabalha com todos os tipos que implementam a interface IEnumerable, ou seja, ArrayList e Queue.O loop foreach chama o método GetEnumerator da "lista", que gera um valor de retorno do índice de cada matriz em cada iteração. Portanto, myArrayList agora se tornou uma coleção personalizada devido ao IEnumerable. Na imagem a seguir, você pode ver um valor yield de retorno do índice de cada matriz em cada iteração.



A partir do C# 9,0, o loop foreach reconhecerá e usará um método de extensão GetEnumerator que, de outra forma, atende ao foreach padrão. Essa alteração significa que o foreach pode ser consistente com outras construções baseadas em padrões, como o padrão assíncrono e a desconstrução baseada em padrões. Na prática, essa alteração significa que você pode adicionar suporte foreach a qualquer tipo.

**SUPORTE PARA GERADORES DE CÓDIGO**

Dois recursos finais dão suporte a geradores de código C#. Os geradores de código C# são um componente que você pode escrever que é semelhante a um analisador de Roslyn ou a uma correção de código. A diferença é que os geradores de código analisam o código e gravam novos arquivos de código-fonte como parte do processo de compilação. Um gerador de código típico pesquisa código em busca de atributos ou outras convenções.

Um gerador de código lê atributos ou outros elementos de código usando as APIs de análise de Roslyn. A partir dessas informações, ele adiciona um novo código à compilação. Os geradores de origem só podem adicionar código; Eles não têm permissão para modificar nenhum código existente na compilação.

Os dois recursos adicionados para geradores de código são:

1. Extensões para sintaxe de método parcial
2. Inicializadores de módulo.

**Novos recursos para métodos parciais**

Antes do C# 9,0, haviam algumas restrições em relação aos métodos parciais:

* Devem ter um tipo de retorno void
* Não podem ter parâmetros de saída out
* Por padrão são private, e não podem especificar um modificador de acesso (pública, privada, protegida, etc)

Abaixo estão alguns exemplos do que acontece quando um método parcial tem ou não uma palavra-chave de acessibilidade, tem ou não um parâmetro out, um tipo void ou not return, implementa uma interface:

partial class MyService : IMyService

{

partial void MyFirstFunction(); // Ok

// CS0750 A partial method cannot have access modifiers or the virtual, abstract, override, new, sealed, or extern modifiers

private partial void MySecondFunction();

// CS0750 A partial method cannot have .....

private partial void MyThirdFunction();

// CS0750 E CS0766 Partial methods must have a void return type

private partial object MyFourthFunction();

// CS0750 E CS0766 Partial methods must have a void return type

private partial object MyFifthFunction();

// CS0750 E CS0752 A partial method cannot have out parameters

private partial void MySixthFunction(out int result);

// CS0750 A partial method cannot have .....

public partial void MySeventhFunction();

}

partial class MyService

{

// CS0750 A partial method cannot have .....

private partial void MyThirdFunction() { }

// CS0750 E CS0766 Partial methods must have a void return type

private partial object MyFifthFunction() { return new { }; }

// CS0750 E CS0752 A partial method cannot have out parameters

private partial void MySixthFunction(out int result) { result = 42; return; }

// CS0750 A partial method cannot have .....

public partial void MySeventhFunction() { }

}

public interface IMyService

{

void MySeventhFunction();

}

Como, até o C# 8, não oferecia suporte a palavra-chave de acessibilidade em métodos parciais, na implementação da interface é impossível implementar o método parcial a partir de sua assinatura de interface, porque sem nenhuma palavra-chave, o método é implicitamente um método privado.

Essas restrições destinam-se que, se nenhuma implementação de método for fornecida, o compilador removerá todas as chamadas para o método parcial. O C# 9,0 remove essas restrições, mas requer que as declarações de método parciais tenham uma implementação. Os geradores de código podem fornecer essa implementação. Para evitar a introdução de uma alteração significativa, o compilador considera qualquer método parcial sem um modificador de acesso para seguir as regras antigas. Se o método parcial incluir o private modificador de acesso, as novas regras regem esse método parcial.

partial class MyService : IMyService

{

partial void MyFirstFunction(); // Ok

// CS8795 Partial method must have an implementation part because it has accessibility modifiers

private partial void MySecondFunction();

private partial void MyThirdFunction(); // Ok

// CS8795 Partial method must have an implementation part ....

private partial object MyFourthFunction();

private partial object MyFifthFunction(); // Ok

private partial void MySixthFunction(out int result);

public partial void MySeventhFunction(); // Ok

}

partial class MyService

{

private partial void MyThirdFunction() { } // Ok

private partial object MyFifthFunction() { return new { }; } // Ok

private partial void MySixthFunction(out int result) { result = 42; return; } // Ok

public partial void MySeventhFunction() { } // Ok

}

public interface IMyService

{

void MySeventhFunction();

}

C# 9 permite agora o que faltava em C# 8, mas agora requer uma implementação de métodos que são definidos com:

* void or not return type
* parâmetros de saída
* palavra-chave de acessibilidade (pública, privada, protegida, etc ...)

Como você pode ver acima, C# 9 traz um novo código de erro com sua mensagem caso as condições anteriores não sejam atendidas:

**CS8795: Partial method must have an implementation part because it has accessibility modifiers**

**Inicializadores de módulo**

Um "inicializador de módulo" é uma função executada quando um assembly é carregado pela primeira vez. Eles são métodos estáticos globais chamados .cctor cuja execução é garantida antes que qualquer outro código (inicializadores de tipo, construtores estáticos) em um assembly seja executado. Em muitos aspectos, é como um construtor estático em C#, mas em vez de se aplicar a uma classe, ele se aplica a todo o assembly. Esse recurso existe no CLR desde o início, mas até agora não estava disponível diretamente em C# ou VB.NET. Para utilizar este recurso era necessário utilizar ferramentas de terceiros para “injetar” os inicializadores de módulo.

O C# 9 adicionou um segundo novo recurso para geradores de código que é inicializadores de módulo para:

* Permitir que as bibliotecas façam uma inicialização rápida, única quando carregadas, com sobrecarga mínima e sem o usuário precisar chamar explicitamente qualquer coisa
* Um ponto problemático específico das abordagens do Construtor static atual é que o tempo de execução deve fazer verificações adicionais no uso de um tipo com um construtor estático, a fim de decidir se o construtor estático precisa ser executado ou não. Isso adiciona sobrecarga mensurável.
* Habilitar os geradores de origem para executar alguma lógica de inicialização global sem que o usuário precise chamar explicitamente nada. Um inicializador de módulo é previsível; todo o código nele é executado sequencialmente. Com construtores estáticos, a ordem em que eles são executados não é determinística do ponto de vista da montagem. Dependendo do código do cliente, o construtor da Classe A pode ser executado antes ou depois da Classe B.

Inicializadores de módulo são métodos que têm o ModuleInitializerAttribute atributo anexado a eles.

class C

{

[ModuleInitializer]

internal static void M1()

{

// ...

}

}

Como você pode ver neste exemplo, um atributo de nível de módulo é marcado com um nome de classe. O construtor estático dessa classe é então promovido ao nível do inicializador de módulo. Esse recurso pode resultar em ganhos de desempenho em relação aos construtores estáticos normais. Esses métodos serão chamados em tempo de execução quando o assembly for carregado. Atualmente, o tempo de execução leva um bloqueio de inicialização que é usado para verificar se a lógica de construção estática foi processada. Adicionando até mesmo um campo estático somente leitura a uma classe.

Alguns requisitos são impostos sobre o método de destino com este atributo:

* O método deve ser static .
* O método deve ser sem parâmetros.
* O método deve retornar void.
* O método não deve ser genérico ou estar contido em um tipo genérico.
* O método deve ser acessível do módulo que o contém.
  + Isso significa que a acessibilidade efetiva do método deve ser internal ou public .
  + Isso também significa que o método não pode ser uma função local.

Quando um ou mais métodos válidos com esse atributo forem encontrados em uma compilação, o compilador emitirá um inicializador de módulo que chama cada um dos métodos atribuídos. As chamadas serão emitidas em uma ordem reservada, mas determinística.