

# Documentação do C#

Saiba como escrever qualquer aplicativo usando a linguagem de programação C# na plataforma .NET.

## Aprender a programar em C#

### COMEÇAR AGORA

[Aprenda C# | Tutoriais, cursos, vídeos e muito mais ↗](#)

### VÍDEO

[Série de vídeos para iniciantes no C# ↗](#)

[Fluxo para iniciantes no C# ↗](#)

[Série de vídeos para usuários intermediários do C#](#)

### TUTORIAL

[Tutoriais autoguiados](#)

[Tutorial no navegador](#)

### REFERÊNCIA

[C# no Q&A](#)

[Linguagens em fóruns da comunidade de tecnologia do .NET ↗](#)

[C# no Stack Overflow ↗](#)

[C# no Discord ↗](#)

## Princípios básicos do C#

### VISÃO GERAL

[Um tour pelo C#](#)

[Por dentro de um programa em C#](#)

[Série de vídeos sobre os destaques do C# ↗](#)

## CONCEITO

Sistema de tipos

Programação orientada a objeto

Técnicas funcionais

Exceções

Estilo de codificação

---

## TUTORIAL

Exibir linha de comando

Introdução às classes

C# orientado a objeto

Converter tipos

Correspondência de padrões

Usar o LINQ para consultar dados

## Principais conceitos

---

### VISÃO GERAL

Conceitos de programação

---

### INÍCIO RÁPIDO

Métodos

Propriedades

Indexadores

Iterators

Delegados

Eventos

---

## CONCEITO

Tipos de referência anuláveis

Migrações de referência anuláveis

[Resolver avisos anuláveis](#)

[LINQ \(Consulta Integrada à Linguagem\)](#)

[Controle de versão](#)

## Novidades

### NOVIDADES

[Novidades do C# 11](#)

[Novidades do C# 10](#)

[Novidades do C# 9.0](#)

[Novidades no C# 8.0](#)

### TUTORIAL

[Explorar tipos de registro](#)

[Explorar instruções de nível superior](#)

[Explorar novos padrões](#)

[Atualizar interfaces com segurança](#)

[Criar mixins com interfaces](#)

[Explore os índices e os intervalos](#)

[Tipos de referência anuláveis](#)

[Explorar fluxos assíncronos](#)

[Escrever um manipulador de interpolação de cadeia de caracteres personalizado](#)

### REFERÊNCIA

[Alterações de falha no compilador C#](#)

[Compatibilidade de versões](#)

## Referência da linguagem C#

### REFERÊNCIA

[Referência de linguagem](#)

[Palavras-chave de C#](#)

[Operadores C#](#)

[Configurar versão da linguagem](#)

[Especificação da linguagem C# – rascunho do C# 7 em andamento](#)

## Mantenha contato

---

 [REFERÊNCIA](#)

[.NET Developer Community](#) ↗

[YouTube](#) ↗

[Twitter](#) ↗

# Um tour pela linguagem C#

Artigo • 15/02/2023

O C# (pronuncia-se "C Sharp") é uma linguagem de programação moderna, orientada a objeto e fortemente tipada. O C# permite que os desenvolvedores criem muitos tipos de aplicativos seguros e robustos que são executados no .NET. O C# tem suas raízes na família de linguagens C e os programadores em C, C++, Java e JavaScript a reconhecerão imediatamente. Este tour dá uma visão geral dos principais componentes da linguagem em C# 8 e anteriores. Se quiser explorar a linguagem por meio de exemplos interativos, experimente os tutoriais de [Introdução à linguagem C#](#).

C# é uma linguagem de programação orientada a objetos e **orientada a componentes**. C# fornece construções de linguagem para dar suporte diretamente a esses conceitos, tornando C# uma linguagem natural para criação e uso de componentes de software. Desde sua origem, o C# adicionou recursos para dar suporte a novas cargas de trabalho e práticas emergentes de design de software. Em sua essência, o C# é uma linguagem **orientada a objeto**. Você define os tipos e o comportamento deles.

Vários recursos do C# ajudam a criar aplicativos robustos e duráveis. [A coleta de lixo](#) recupera automaticamente a memória ocupada por objetos não utilizados inacessíveis. [Tipos anuláveis](#) são protegidos contra variáveis que não se referem a objetos alocados. [O tratamento de exceções](#) fornece uma abordagem estruturada e extensível para detecção e recuperação de erros. [As expressões Lambda dão suporte a técnicas](#) de programação funcional. [Consulta Integrada à Linguagem \(LINQ\)](#) a sintaxe cria um padrão comum para trabalhar com dados de qualquer fonte. O suporte à linguagem para [operações assíncronas fornece sintaxe](#) para a criação de sistemas distribuídos. C# tem um [sistema de tipo unificado](#). Todos os tipos do C#, incluindo tipos primitivos, como `int` e `double`, herdam de um único tipo de `object` raiz. Todos os tipos compartilham um conjunto de operações comuns. Valores de qualquer tipo podem ser armazenados, transportados e operados de maneira consistente. Além disso, o C# dá suporte a [tipos de referência](#) e [tipos de valor](#) definidos pelo usuário. O C# permite a alocação dinâmica de objetos e o armazenamento em linha de estruturas leves. O C# dá suporte a métodos e tipos genéricos, que fornecem maior segurança e desempenho do tipo. O C# fornece iteradores, que permitem que os implementadores de classes de coleção definam comportamentos personalizados para o código do cliente.

O C# enfatiza o **controle de versão** para garantir que programas e bibliotecas possam evoluir ao longo do tempo de maneira compatível. Aspectos do design do C# que foram diretamente influenciados pelas considerações de controle de versão incluem os

modificadores separados `virtual` e `override`, as regras de resolução de sobrecarga de método e suporte para declarações explícitas de membro de interface.

## Arquitetura do .NET

Programas C# são executados no .NET, um sistema de execução virtual chamado CLR (Common Language Runtime) e um conjunto de bibliotecas de classes. O CLR é a implementação pela Microsoft da CLI (Common Language Infrastructure), um padrão internacional. A CLI é a base para criar ambientes de execução e desenvolvimento nos quais as linguagens e bibliotecas funcionam em conjunto perfeitamente.

O código-fonte escrito em C# é compilado em uma [IL \(linguagem intermediária\)](#) que está em conformidade com a especificação da CLI. O código IL e os recursos, como bitmaps e cadeias de caracteres, são armazenados em um assembly, normalmente com uma extensão de `.dll`. Um assembly contém um manifesto que fornece informações sobre os tipos, a versão e a cultura.

Quando o programa C# é executado, o assembly é carregado no CLR. Em seguida, o CLR executará a compilação JIT (Just-In-Time) para converter o código de IL em instruções nativas da máquina. O CLR também oferece outros serviços relacionados à coleta automática de lixo, tratamento de exceções e gerenciamento de recursos. O código que é executado pelo CLR é, às vezes, chamado de "código gerenciado". "Código não gerenciado" é compilado em linguagem de máquina nativa e visa uma plataforma específica.

Interoperabilidade de linguagem é um recurso importante do .NET. O código IL produzido pelo compilador C# está em conformidade com a CTS (Common Type Specification). O código IL gerado a partir do C# pode interagir com o código que foi gerado a partir das versões do .NET do F#, do Visual Basic, do C++. Há mais de 20 outras linguagens compatíveis com CTS. Um único assembly pode conter vários módulos gravados em diferentes idiomas do .NET. Os tipos podem fazer referência uns aos outros como se fossem escritos na mesma linguagem.

Além dos serviços de tempo de execução, o .NET também inclui bibliotecas extensas. Essas bibliotecas dão suporte a muitas cargas de trabalho diferentes. Eles são organizados em namespaces que fornecem uma ampla variedade de funcionalidades úteis. As bibliotecas incluem desde a entrada e a saída do arquivo até a manipulação de cadeia de caracteres até a análise de XML até estruturas de aplicativos Web para controles de Windows Forms. O aplicativo típico em C# usa bastante a biblioteca de classes para lidar com tarefas comuns de "conexão".

Para obter mais informações sobre o .NET, consulte [Visão geral do .NET](#).

# Hello world

O programa "Hello, World" é usado tradicionalmente para introduzir uma linguagem de programação. Este é para C#:

```
C#  
  
using System;  
  
class Hello  
{  
    static void Main()  
    {  
        Console.WriteLine("Hello, World");  
    }  
}
```

O programa "Hello, World" começa com uma diretiva `using` que faz referência ao namespace `System`. Namespaces fornecem um meio hierárquico de organizar bibliotecas e programas em C#. Os namespaces contêm tipos e outros namespaces — por exemplo, o namespace `System` contém uma quantidade de tipos, como a classe `Console` referenciada no programa e diversos outros namespaces, como `IO` e `Collections`. A diretiva `using` que faz referência a um determinado namespace permite o uso não qualificado dos tipos que são membros desse namespace. Devido à diretiva `using`, o programa pode usar `Console.WriteLine` como um atalho para `System.Console.WriteLine`.

A classe `Hello` declarada pelo programa "Hello, World" tem um único membro, o método chamado `Main`. O método `Main` é declarado com o modificador `static`. Embora os métodos de instância possam fazer referência a uma determinada instância de objeto delimitador usando a palavra-chave `this`, métodos estáticos operam sem referência a um objeto específico. Por convenção, um método estático denominado `Main` serve como ponto de entrada de um programa C#.

A saída do programa é produzida pelo método `WriteLine` da classe `Console` no namespace `System`. Essa classe é fornecida pelas bibliotecas de classe padrão, que, por padrão, são referenciadas automaticamente pelo compilador.

## Tipos e variáveis

Um *tipo* define a estrutura e o comportamento de qualquer dado em C#. A declaração de um tipo pode incluir seus membros, tipo base, interfaces implementadas e operações

permitidas para esse tipo. Uma *variável* é um rótulo que se refere a uma instância de um tipo específico.

Há dois tipos em C#: *tipos de referência* e *tipos de valor*. Variáveis de tipos de valor contêm diretamente seus dados. Variáveis de tipos de referência armazenam referências a seus dados, o último sendo conhecido como objetos. Com tipos de referência, é possível que duas variáveis referenciem o mesmo objeto e, portanto, é possível que operações em uma variável afetem o objeto referenciado por outra variável. Com tipos de valor, cada variável tem sua própria cópia dos dados e não é possível que operações em uma variável afetem a outra (exceto em variáveis de parâmetros `ref` e `out`).

Um *identificador* é um nome de variável. Um identificador é uma sequência de caracteres unicode sem nenhum espaço em branco. Um identificador pode ser uma palavra reservada em C#, se for prefixado por `@`. Usar uma palavra reservada como identificador pode ser útil ao interagir com outros idiomas.

Os tipos de valor do C# são divididos em *tipos simples*, *tipos de enumeração*, *tipos struct* e *tipos de valor anulável* e *tipos de valor de tupla*. Os tipos de referência do C# são divididos em *tipos de classe*, *tipos de interface*, *tipos de matriz* e *tipos delegados*.

A estrutura de tópicos a seguir fornece uma visão geral do sistema de tipos do C#.

- [Tipos de valor](#)
  - [Tipos simples](#)
    - [Integral com sinal](#): `sbyte`, `short`, `int`, `long`
    - [Integral sem sinal](#): `byte`, `ushort`, `uint`, `ulong`
    - [Caracteres Unicode](#): `char`, que representa uma unidade de código UTF-16
    - [Ponto flutuante binário de IEEE](#): `float`, `double`
    - [Ponto flutuante decimal de alta precisão](#): `decimal`
    - [Booliano](#): `bool`, que representa valores booleanos — valores que são `true` ou `false`
  - [Tipos enum](#)
    - Tipos definidos pelo usuário do formulário `enum E { ... }`. Um tipo `enum` é um tipo distinto com constantes nomeadas. Cada tipo `enum` tem um tipo subjacente, que deve ser um dos oito tipos integrais. O conjunto de valores de um tipo `enum` é o mesmo que o conjunto de valores do tipo subjacente.
  - [Tipos struct](#)
    - Tipos definidos pelo usuário do formulário `struct S { ... }`
  - [Tipos de valor anuláveis](#)
    - Extensões de todos os outros tipos de valor com um valor `null`
  - [Tipos de valor de tupla](#)

- Tipos definidos pelo usuário do formulário (`T1, T2, ...`)
- [Tipos de referência](#)
  - [Tipos de aula](#)
    - Classe base definitiva de todos os outros tipos: `object`
    - [Cadeias de caracteres Unicode](#): `string`, que representa uma sequência de unidades de código UTF-16
    - Tipos definidos pelo usuário do formulário `class C {...}`
  - [Tipos de interface](#)
    - Tipos definidos pelo usuário do formulário `interface I {...}`
  - [Tipos de matriz](#)
    - Unidimensional, multidimensional e irregular. Por exemplo: `int[], int[,]` e `int[][]`
  - [Tipos delegados](#)
    - Tipos definidos pelo usuário do formulário `delegate int D(...)`

Os programas em C# usam *declarações de tipos* para criar novos tipos. Uma declaração de tipo especifica o nome e os membros do novo tipo. Seis das categorias do C# de tipos são tipos definidos pelo usuário: tipos de classe, tipos struct, tipos de interface, tipos enum, tipos delegados e tipos de valor de tupla. Você também pode declarar tipos `record`, `record struct` ou `record class`. Os tipos de registro têm membros sintetizados pelo compilador. Você usa registros principalmente para armazenar valores, com o mínimo de comportamento associado.

- Um tipo `class` define uma estrutura de dados que contém membros de dados (campos) e membros de função (métodos, propriedades e outros). Os tipos de classe dão suporte à herança única e ao polimorfismo, mecanismos nos quais as classes derivadas podem estender e especializar as classes base.
- Um tipo `struct` é semelhante a um tipo de classe que representa uma estrutura com membros de dados e membros da função. No entanto, diferentemente das classes, structs são tipos de valor e, normalmente, não precisam de alocação de heap. Os tipos de estrutura não dão suporte à herança especificada pelo usuário, e todos os tipos de structs são herdados implicitamente do tipo `object`.
- Um tipo `interface` define um contrato como um conjunto nomeado de membros públicos. Um `class` ou `struct` que implementa um `interface` deve fornecer implementações de membros da interface. Um `interface` pode herdar de várias interfaces base e um `class` ou `struct` pode implementar várias interfaces.
- Um tipo `delegate` representa referências aos métodos com uma lista de parâmetros e tipo de retorno específicos. Delegados possibilitam o tratamento de métodos como entidades que podem ser atribuídos a variáveis e passadas como parâmetros. Os delegados são análogos aos tipos de função fornecidos pelas

linguagens funcionais. Eles também são semelhantes ao conceito de ponteiros de função encontrado em algumas outras linguagens. Ao contrário dos ponteiros de função, os delegados são orientados a objetos e fortemente tipados.

Os tipos `class`, `struct`, `interface` e `delegate` dão suporte a genéricos e podem ser parametrizados com outros tipos.

O C# dá suporte a matrizes unidimensionais e multidimensionais de qualquer tipo. Ao contrário dos tipos listados acima, os tipos de matriz não precisam ser declarados antes de serem usados. Em vez disso, os tipos de matriz são construídos seguindo um nome de tipo entre colchetes. Por exemplo, `int[]` é uma matriz unidimensional de `int`, `int[,]` é uma matriz bidimensional de `int`, e `int[][]` é uma matriz unidimensional da matriz unidimensional, ou uma matriz "denteada", de `int`.

Tipos anuláveis não exigem uma definição separada. Para cada tipo não nulo `T` há um tipo anulável correspondente `T?`, que pode conter um valor adicional, `null`. Por exemplo, `int?` é um tipo que pode conter qualquer inteiro de 32 bits ou o valor `null` e `string?` é um tipo que pode conter qualquer `string` ou o valor `null`.

O sistema de tipos do C# é unificado, de modo que um valor de qualquer tipo pode ser tratado como um `object`. Cada tipo no C#, direta ou indiretamente, deriva do tipo de classe `object`, e `object` é a classe base definitiva de todos os tipos. Os valores de tipos de referência são tratados como objetos simplesmente exibindo os valores como tipo `object`. Os valores de tipos de valor são tratados como objetos, executando *conversão boxing* e *operações de conversão unboxing*. No exemplo a seguir, um valor `int` é convertido em `object` e volta novamente ao `int`.

```
C#
```

```
int i = 123;
object o = i;    // Boxing
int j = (int)o; // Unboxing
```

Quando um valor de um tipo de valor é atribuído a uma referência `object`, uma "caixa" é alocada para conter o valor. Essa caixa é uma instância de um tipo de referência e o valor é copiado nessa caixa. Por outro lado, quando uma referência `object` é lançada em um tipo de valor, `object` é feita uma verificação de que o referenciado é uma caixa do tipo de valor correto. Se a verificação for bem-sucedida, o valor na caixa será copiado para o tipo de valor.

O sistema de tipo unificado do C# significa efetivamente que os tipos de valor são tratados como referências `object` "sob demanda". Devido à unificação, as bibliotecas de

uso geral que usam o tipo `object` podem ser usadas com todos os tipos derivados de `object`, incluindo tipos de referência e tipos de valor.

Existem vários tipos de *variáveis* no C#, incluindo campos, elementos de matriz, variáveis locais e parâmetros. As variáveis representam locais de armazenamento. Cada variável tem um tipo que determina quais valores podem ser armazenados na variável, conforme mostrado abaixo.

- Tipo de valor não nulo
  - Um valor de tipo exato
- Tipos de valor anulável
  - Um valor `null` ou um valor do tipo exato
- objeto
  - Uma referência `null`, uma referência a um objeto de qualquer tipo de referência ou uma referência a um valor de qualquer tipo de valor demarcado
- Tipo de classe
  - Uma referência `null`, uma referência a uma instância desse tipo de classe ou uma referência a uma instância de uma classe derivada desse tipo de classe
- Tipo de interface
  - Uma referência `null`, uma referência a uma instância de um tipo de classe que implementa esse tipo de interface ou uma referência a um valor demarcado de um tipo de valor que implementa esse tipo de interface
- Tipo de matriz
  - Uma referência `null`, uma referência a uma instância desse tipo de matriz ou uma referência a uma instância de um tipo de matriz compatível
- Tipo delegado
  - Uma referência `null` ou uma referência a uma instância de um tipo de delegado compatível

## Estrutura do programa

Os principais conceitos organizacionais em C# são *programas, namespaces, tipos, membros* e *assemblies*. Os programas declaram tipos que contêm membros e podem ser organizados em namespaces. Classes, structs e interfaces são exemplos de tipos. Campos, métodos, propriedades e eventos são exemplos de membros. Quando os programas em C# são compilados, eles são empacotados fisicamente em assemblies. Os assemblies normalmente têm a extensão de arquivo `.exe` ou `.dll`, dependendo se eles implementam *aplicativos* ou *bibliotecas*, respectivamente.

Como um pequeno exemplo, considere um assembly que contém o seguinte código:

C#

```
namespace Acme.Collections;

public class Stack<T>
{
    Entry _top;

    public void Push(T data)
    {
        _top = new Entry(_top, data);
    }

    public T Pop()
    {
        if (_top == null)
        {
            throw new InvalidOperationException();
        }
        T result = _top.Data;
        _top = _top.Next;

        return result;
    }

    class Entry
    {
        public Entry Next { get; set; }
        public T Data { get; set; }

        public Entry(Entry next, T data)
        {
            Next = next;
            Data = data;
        }
    }
}
```

O nome totalmente qualificado dessa classe é `Acme.Collections.Stack`. A classe contém vários membros: um campo chamado `_top`, dois métodos chamados `Push` e `Pop` e uma classe aninhada chamada `Entry`. A classe `Entry` ainda contém três membros: um campo chamado `Next`, um campo chamado `Data` e um construtor. `Stack` é uma classe *genérica*. Ele tem um parâmetro de tipo, `T` que é substituído por um tipo concreto quando é usado.

Uma *pilha* é uma coleção "primeiro a entrar - último a sair" (FILO). O elemento adicionado à parte superior da stack. Quando um elemento é removido, ele é removido da parte superior da pilha. O exemplo anterior declara o tipo `Stack` que define o

armazenamento e o comportamento de uma pilha. Você pode declarar uma variável que se refere a uma instância do tipo `Stack` para usar essa funcionalidade.

Os assemblies contêm código executável na forma de instruções de IL (Linguagem Intermediária) e informações simbólicas na forma de metadados. Antes de ser executado, o compilador Just-In-Time (JIT) do .NET Common Language Runtime converte o código IL em um assembly em código específico do processador.

Como um assembly é uma unidade autodescritiva da funcionalidade que contém o código e os metadados, não é necessário de diretivas `#include` e arquivos de cabeçalho no C#. Os tipos públicos e os membros contidos em um assembly específico são disponibilizados em um programa C# simplesmente fazendo referência a esse assembly ao compilar o programa. Por exemplo, esse programa usa a classe

`Acme.Collections.Stack` do assembly `acme.dll`:

C#

```
class Example
{
    public static void Main()
    {
        var s = new Acme.Collections.Stack<int>();
        s.Push(1); // stack contains 1
        s.Push(10); // stack contains 1, 10
        s.Push(100); // stack contains 1, 10, 100
        Console.WriteLine(s.Pop()); // stack contains 1, 10
        Console.WriteLine(s.Pop()); // stack contains 1
        Console.WriteLine(s.Pop()); // stack is empty
    }
}
```

Para compilar este programa, você precisaria fazer *referência* ao assembly que contém a classe de pilha definida no exemplo anterior.

Os programas C# podem ser armazenados em vários arquivos de origem. Quando um programa C# é compilado, todos os arquivos de origem são processados juntos e os arquivos de origem podem se referenciar livremente. Conceitualmente, é como se todos os arquivos de origem fossem concatenados em um arquivo grande antes de serem processados. Declarações de encaminhamento nunca são necessárias em C#, porque, com poucas exceções, a ordem de declaração é insignificante. O C# não limita um arquivo de origem para declarar somente um tipo público nem requer o nome do arquivo de origem para corresponder a um tipo declarado no arquivo de origem.

Outros artigos neste tour explicam esses blocos organizacionais.

**Próximo**

# Tipos e membros de C#

Artigo • 02/06/2023

Por ser uma linguagem orientada a objeto, o C# oferece suporte aos conceitos de encapsulamento, herança e polimorfismo. Uma classe pode herdar diretamente de uma classe pai e pode implementar qualquer quantidade de interfaces. Métodos que substituem métodos virtuais em uma classe pai exigem a palavra-chave `override` como uma forma de evitar uma redefinição acidental. Em C#, um struct é como uma classe simplificada. É um tipo alocado na pilha que pode implementar interfaces, mas não oferece suporte à herança. O C# fornece os tipos `record class` e `record struct`, cuja finalidade é armazenar, principalmente, valores de dados.

Todos os tipos são inicializados por meio de um *construtor*, um método responsável por inicializar uma instância. Duas declarações de construtor têm um comportamento exclusivo:

- Um *construtor sem parâmetros*, que inicializa todos os campos para seu valor padrão.
- Um *construtor primário*, que declara os parâmetros necessários para uma instância desse tipo.

## Classes e objetos

As *classes* são os tipos do C# mais fundamentais. Uma classe é uma estrutura de dados que combina ações (métodos e outros membros da função) e estado (campos) em uma única unidade. Uma classe fornece uma definição das *instâncias* da classe, também conhecidas como *objetos*. As classes dão suporte à *herança* e *polimorfismo*, mecanismos nos quais *classes derivadas* podem estender e especializar *classes base*.

Novas classes são criadas usando declarações de classe. Uma declaração de classe começa com um cabeçalho. O cabeçalho especifica:

- Os atributos e modificadores da classe.
- O nome da classe.
- A classe base (ao herdar de uma [classe base](#)).
- As interfaces implementadas pela classe.

O cabeçalho é seguido pelo corpo da classe, que consiste em uma lista de declarações de membro escrita entre os delimitadores `{` e `}`.

O código a seguir mostra uma declaração de uma classe simples chamada `Point`:

C#

```
public class Point
{
    public int X { get; }
    public int Y { get; }

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

Instâncias de classes são criadas usando o operador `new`, que aloca memória para uma nova instância, chama um construtor para inicializar a instância e retorna uma referência à instância. As instruções a seguir criam dois objetos `Point` e armazenam referências a esses objetos em duas variáveis:

C#

```
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```

A memória ocupada por um objeto é recuperada automaticamente quando o objeto não está mais acessível. Não é necessário nem possível desalocar explicitamente os objetos em C#.

C#

```
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```

Aplicativos ou testes para algoritmos podem precisar criar vários objetos `Point`. A classe a seguir gera uma sequência de pontos aleatórios. O número de pontos é definido pelo parâmetro do *construtor primário*. O parâmetro do construtor primário `numPoints` está no escopo de todos os membros da classe:

C#

```
public class PointFactory(int numPoints)
{
    public IEnumerable<Point> CreatePoints()
    {
        var generator = new Random();
        for (int i = 0; i < numPoints; i++)
        {
            yield return new Point(generator.Next(), generator.Next());
        }
    }
}
```

```
    }  
}
```

Use a classe conforme mostrado no código a seguir:

C#

```
var factory = new PointFactory(10);  
foreach (var point in factory.CreatePoints())  
{  
    Console.WriteLine($"{point.X}, {point.Y}");  
}
```

## Parâmetros de tipo

Classes genérico definem *parâmetros de tipo*. Os parâmetros de tipo são uma lista de nomes de parâmetros de tipo entre colchetes angulares. Os parâmetros de tipo seguem o nome da classe. Em seguida, os parâmetros de tipo podem ser usados no corpo das declarações de classe para definir os membros da classe. No exemplo a seguir, os parâmetros de tipo de `Pair` são `TFirst` e `TSecond`:

C#

```
public class Pair<TFirst, TSecond>  
{  
    public TFirst First { get; }  
    public TSecond Second { get; }  
  
    public Pair(TFirst first, TSecond second) =>  
        (First, Second) = (first, second);  
}
```

Um tipo de classe que é declarado para pegar parâmetros de tipo é chamado de *tipo de classe genérica*. Tipos struct, interface e delegate também podem ser genéricos. Quando a classe genérica é usada, os argumentos de tipo devem ser fornecidos para cada um dos parâmetros de tipo:

C#

```
var pair = new Pair<int, string>(1, "two");  
int i = pair.First; //TFirst int  
string s = pair.Second; //TSecond string
```

Um tipo genérico com argumentos de tipo fornecidos, como `Pair<int, string>` acima, é chamado de *tipo construído*.

## Classes base

Uma declaração de classe pode especificar uma classe base. Após o nome da classe e os parâmetros de tipo, insira dois-pontos e o nome da classe base. Omitir uma especificação de classe base é o mesmo que derivar do `object` de tipo. No exemplo a seguir, a classe base de `Point3D` é `Point`. No primeiro exemplo, a classe base de `Point` é `object`:

C#

```
public class Point3D : Point
{
    public int Z { get; set; }

    public Point3D(int x, int y, int z) : base(x, y)
    {
        Z = z;
    }
}
```

Uma classe herda os membros de sua classe base. Herança significa que uma classe contém implicitamente quase todos os membros da classe base. Uma classe não herda a instância, os construtores estáticos e o finalizador. Uma classe derivada pode adicionar novos membros aos que ela herda, mas não pode remover a definição de um membro herdado. No exemplo anterior, `Point3D` herda os membros `X` e `Y` de `Point` e cada instância `Point3D` contém três propriedades: `X`, `Y` e `Z`.

Existe uma conversão implícita de um tipo de classe para qualquer um de seus tipos de classe base. Uma variável de um tipo de classe pode referenciar uma instância dessa classe ou uma instância de qualquer classe derivada. Por exemplo, dadas as declarações de classe anteriores, uma variável do tipo `Point` podem referenciar um `Point` ou um `Point3D`:

C#

```
Point a = new(10, 20);
Point b = new Point3D(10, 20, 30);
```

## Estruturas

As classes definem tipos que dão suporte à herança e ao polimorfismo. Elas permitem que você crie comportamentos sofisticados com base nas hierarquias das classes derivadas. Por outro lado, os tipos `struct` são mais simples e a finalidade principal deles

é armazenar valores de dados. Structs não podem declarar um tipo base; eles derivam implicitamente de [System.ValueType](#). Você não pode derivar outros tipos `struct` de um tipo `struct`. Eles estão implicitamente selados.

C#

```
public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y) => (X, Y) = (x, y);
}
```

## Interfaces

Uma [interface](#) define um contrato que pode ser implementado por classes e structs. Você define uma *interface* para declarar recursos compartilhados entre tipos distintos. Por exemplo, a interface [System.Collections.Generic.IEnumerable<T>](#) define uma maneira consistente de percorrer todos os itens de uma coleção, como uma matriz. Uma interface pode conter métodos, propriedades, eventos e indexadores. Geralmente, uma interface não fornece implementações dos membros que define. Ela simplesmente especifica os membros que devem ser fornecidos por classes ou structs que implementam a interface.

As interfaces podem empregar a *herança múltipla*. No exemplo a seguir, a interface `IComboBox` herda de `ITextBox` e `IListBox`.

C#

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }
```

Classes e structs podem implementar várias interfaces. No exemplo a seguir, a classe `EditBox` implementa `IControl` e `IDataBound`.

```
C#  
  
interface IDataBound  
{  
    void Bind(Binder b);  
}  
  
public class EditBox : IControl, IDataBound  
{  
    public void Paint() { }  
    public void Bind(Binder b) { }  
}
```

Quando uma classe ou struct implementa uma interface específica, as instâncias dessa classe ou struct podem ser convertidas implicitamente para esse tipo de interface. Por exemplo

```
C#  
  
EditBox editBox = new();  
IControl control = editBox;  
IDataBound dataBound = editBox;
```

## Enumerações

Um tipo `Enum` define um conjunto de valores constantes. O `enum` a seguir declara constantes que definem diversos vegetais de raiz:

```
C#  
  
public enum SomeRootVegetable  
{  
    HorseRadish,  
    Radish,  
    Turnip  
}
```

Você também pode definir `enum` a ser usado na combinação como sinalizadores. A declaração a seguir define um conjunto de sinalizadores das quatro estações. Qualquer combinação das estações pode ser aplicada, incluindo um valor `All`, que inclui todas as estações:

C#

```
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}
```

O exemplo a seguir mostra declarações de ambas as enumerações anteriores:

C#

```
var turnip = SomeRootVegetable.Turnip;

var spring = Seasons.Spring;
var startingOnEquinox = Seasons.Spring | Seasons.Autumn;
var theYear = Seasons.All;
```

## Tipos anuláveis

Variáveis de qualquer tipo podem ser declaradas como *não anuláveis* ou *anuláveis*. Uma variável anulável pode conter um valor adicional `null`, indicando nenhum valor. Tipos de valor anulável (structs ou enums) são representados por `System.Nullable<T>`. Tipos de referência não anuláveis e anuláveis são representados pelo tipo de referência subjacente. A distinção é representada por metadados lidos pelo compilador e algumas bibliotecas. O compilador fornece avisos quando as referências anuláveis são desreferenciadas sem antes verificar o valor delas em `null`. O compilador também fornece avisos quando referências não anuláveis são atribuídas a um valor que pode ser `null`. O exemplo a seguir declara um *inteiro anulável*, inicializando-o como `null`. Em seguida, ele define o valor como `5`. Ele demonstra o mesmo conceito com uma *cadeia de caracteres anulável*. Para saber mais, confira [tipos de valor anulável](#) e [tipos de referência nula](#).

C#

```
int? optionalInt = default;
optionalInt = 5;
string? optionalText = default;
optionalText = "Hello World.;"
```

# Tuplas

O C# dá suporte a **tuplas**, que fornecem sintaxe concisa para agrupar vários elementos de dados em uma estrutura de dados leve. Você cria uma instância de uma tupla declarando os tipos e nomes dos membros entre `(` e `)`, conforme mostrado no exemplo a seguir:

C#

```
(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
//Output:
//Sum of 3 elements is 4.5.
```

As tuplas são uma alternativa para a estrutura de dados com vários membros, sem usar os blocos de construção descritos no próximo artigo.

[Anterior](#)

[Avançar](#)

# Blocos de construção de programas C#

Artigo • 07/04/2023

Os tipos descritos no [artigo anterior](#) desta série de Tour pelo C# são criados usando estes blocos de construção:

- [Membros](#), como propriedades, campos, métodos e eventos.
- [Expressões](#)
- [Instruções](#)

## Membros

Os membros de um `class` são **membros estáticos** ou **membros de instância**. Os membros estáticos pertencem às classes e os membros de instância pertencem aos objetos (instâncias de classes).

A lista a seguir fornece uma visão geral dos tipos de membros que uma classe pode conter.

- **Constantes**: valores constantes associados à classe
- **Campos**: variáveis associadas à classe
- **Métodos**: ações que podem ser executadas pela classe
- **Propriedades**: ações associadas à leitura e à gravação de propriedades nomeadas da classe
- **Indexadores**: ações associadas à indexação de instâncias da classe como uma matriz
- **Eventos**: notificações que podem ser geradas pela classe
- **Operadores**: conversões e operadores de expressão com suporte na classe
- **Construtores**: ações necessárias para inicializar instâncias da classe ou a própria classe
- **Finalizadores**: ações feitas antes das instâncias da classe serem descartadas permanentemente
- **Tipos**: Tipos aninhados declarados pela classe

## Acessibilidade

Cada membro de uma classe tem uma acessibilidade associada, que controla as regiões do texto do programa que podem acessar o membro. Existem seis formas possíveis de acessibilidade. Os modificadores de acesso são resumidos abaixo.

- `public`: acesso ilimitado.
- `private`: acesso limitado a essa classe.
- `protected`: acesso limitado a essa classe ou a classes derivadas dessa classe.
- `internal`: acesso limitado ao assembly atual (`.exe` ou `.dll`).
- `protected internal`: acesso limitado a essa classe, a classes derivadas dessa classe ou a classes dentro do mesmo assembly.
- `private protected`: acesso limitado a essa classe ou a classes derivadas desse tipo dentro do mesmo assembly.

## Campos

Um *campo* é uma variável que está associada a uma classe ou a uma instância de uma classe.

Um campo declarado com o modificador estático define um campo estático. Um campo estático identifica exatamente um local de armazenamento. Não importa quantas instâncias de uma classe são criadas, há sempre apenas uma cópia de um campo estático.

Um campo declarado sem o modificador estático define um campo de instância. Cada instância de uma classe contém uma cópia separada de todos os campos de instância dessa classe.

No seguinte exemplo, cada instância da classe `Color` tem uma cópia separada dos campos de instância `R`, `G` e `B`, mas há apenas uma cópia dos campos estáticos `Black`, `White`, `Red`, `Green` e `Blue`:

```
C#  
  
public class Color  
{  
    public static readonly Color Black = new(0, 0, 0);  
    public static readonly Color White = new(255, 255, 255);  
    public static readonly Color Red = new(255, 0, 0);  
    public static readonly Color Green = new(0, 255, 0);  
    public static readonly Color Blue = new(0, 0, 255);  
  
    public byte R;  
    public byte G;  
    public byte B;  
  
    public Color(byte r, byte g, byte b)  
    {  
        R = r;  
        G = g;
```

```
B = b;  
}  
}
```

Conforme mostrado no exemplo anterior, os *campos somente leitura* podem ser declarados com um modificador `readonly`. A atribuição a um campo somente leitura só pode ocorrer como parte da declaração do campo ou em um construtor da mesma classe.

## Métodos

Um *método* é um membro que implementa um cálculo ou uma ação que pode ser executada por um objeto ou classe. Os *métodos estáticos* são acessados pela classe. Os *métodos de instância* são acessados pelas instâncias da classe.

Os métodos podem ter uma lista de *parâmetros*, que representam valores ou referências de variáveis passadas para o método. Os métodos têm um *tipo de retorno*, que especifica o tipo do valor calculado e retornado pelo método. O tipo de retorno do método será `void` se ele não retornar um valor.

Como os tipos, os métodos também podem ter um conjunto de parâmetros de tipo, para que os quais os argumentos de tipo devem ser especificados quando o método é chamado. Ao contrário dos tipos, os argumentos de tipo geralmente podem ser inferidos de argumentos de uma chamada de método e não precisam ser fornecidos explicitamente.

A *assinatura* de um método deve ser exclusiva na classe na qual o método é declarado. A assinatura de um método consiste no nome do método, número de parâmetros de tipo e número, modificadores e tipos de parâmetros. A assinatura de um método não inclui o tipo de retorno.

Quando um corpo do método é apenas uma expressão, o método pode ser definido usando um formato de expressão compacta, conforme mostrado no seguinte exemplo:

C#

```
public override string ToString() => "This is an object";
```

## Parâmetros

Os parâmetros são usados para passar valores ou referências de variável aos métodos. Os parâmetros de um método obtêm seus valores reais de *argumentos* que são

especificados quando o método é invocado. Há quatro tipos de parâmetros: parâmetros de valor, parâmetros de referência, parâmetros de saída e matrizes de parâmetros.

Um *parâmetro de valor* é usado para passar argumentos de entrada. Um parâmetro de valor corresponde a uma variável local que obtém seu valor inicial do argumento passado para o parâmetro. As modificações em um parâmetro de valor não afetam o argumento passado para o parâmetro.

Os parâmetros de valor podem ser opcionais, especificando um valor padrão para que os argumentos correspondentes possam ser omitidos.

Um *parâmetro de referência* é usado para passar argumentos por referência. O argumento passado para um parâmetro de referência deve ser uma variável com um valor definido. Durante a execução do método, o parâmetro de referência representa o mesmo local de armazenamento que a variável de argumento. Um parâmetro de referência é declarado com o modificador `ref`. O exemplo a seguir mostra o uso de parâmetros `ref`.

```
C#  
  
static void Swap(ref int x, ref int y)  
{  
    int temp = x;  
    x = y;  
    y = temp;  
}  
  
public static void SwapExample()  
{  
    int i = 1, j = 2;  
    Swap(ref i, ref j);  
    Console.WriteLine($"{i} {j}");    // "2 1"  
}
```

Um *parâmetro de saída* é usado para passar argumentos por referência. Ele é semelhante a um parâmetro de referência, exceto que ele não requer que você atribua explicitamente um valor ao argumento fornecido pelo chamador. Um parâmetro de saída é declarado com o modificador `out`. O exemplo a seguir mostra o uso de parâmetros `out`.

```
C#  
  
static void Divide(int x, int y, out int quotient, out int remainder)  
{  
    quotient = x / y;  
    remainder = x % y;  
}
```

```
public static void OutUsage()
{
    Divide(10, 3, out int quo, out int rem);
    Console.WriteLine($"{quo} {rem}"); // "3 1"
}
```

Uma *matriz de parâmetros* permite que um número variável de argumentos sejam passados para um método. Uma matriz de parâmetro é declarada com o modificador `params`. Somente o último parâmetro de um método pode ser uma matriz de parâmetros e o tipo de uma matriz de parâmetros deve ser um tipo de matriz unidimensional. Os métodos `Write` e `WriteLine` da classe `System.Console` são bons exemplos de uso da matriz de parâmetros. Eles são declarados conforme a seguir.

C#

```
public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}
```

Dentro de um método que usa uma matriz de parâmetros, a matriz de parâmetros se comporta exatamente como um parâmetro regular de um tipo de matriz. No entanto, em uma invocação de um método com uma matriz de parâmetros, é possível passar apenas um argumento do tipo da matriz de parâmetro ou qualquer número de argumentos do tipo de elemento da matriz de parâmetros. No último caso, uma instância de matriz é automaticamente criada e inicializada com os argumentos determinados. Esse exemplo

C#

```
int x, y, z;
x = 3;
y = 4;
z = 5;
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

é equivalente ao escrito a seguir.

C#

```
int x = 3, y = 4, z = 5;

string s = "x={0} y={1} z={2}";
```

```
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

## Corpo do método e variáveis locais

Um corpo do método especifica as instruções a executar quando o método é invocado.

Um corpo de método pode declarar variáveis que são específicas para a invocação do método. Essas variáveis são chamadas de *variáveis locais*. Uma declaração de variável local especifica um nome de tipo, um nome de variável e, possivelmente, um valor inicial. O exemplo a seguir declara uma variável local `i` com um valor inicial de zero e uma variável local `j` sem valor inicial.

C#

```
class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i++;
        }
    }
}
```

O C# requer que uma variável local seja *atribuída definitivamente* antes de seu valor poder ser obtido. Por exemplo, se a declaração do anterior `i` não incluísse um valor inicial, o compilador relataria um erro para usos subsequentes de `i` porque `i` não seria definitivamente atribuído a esses pontos do programa.

Um método pode usar instruções `return` para retornar o controle é pelo chamador. Em um método que retorna `void`, as instruções `return` não podem especificar uma expressão. Em um método que retorna não nulo, as instruções `return` devem incluir uma expressão que calcula o valor retornado.

## Métodos estáticos e de instância

Um método declarado com um modificador `static` é um *método estático*. Um método estático não funciona em uma instância específica e pode acessar diretamente apenas membros estáticos.

Um método declarado sem o modificador `static` é um *método de instância*. Um método de instância opera em uma instância específica e pode acessar membros estáticos e de instância. A instância em que um método de instância foi invocado pode ser explicitamente acessada como `this`. É um erro fazer referência a `this` um método estático.

A seguinte classe `Entity` tem membros estáticos e de instância.

C#

```
class Entity
{
    static int s_nextSerialNo;
    int _serialNo;

    public Entity()
    {
        _serialNo = s_nextSerialNo++;
    }

    public int GetSerialNo()
    {
        return _serialNo;
    }

    public static int GetNextSerialNo()
    {
        return s_nextSerialNo;
    }

    public static void SetNextSerialNo(int value)
    {
        s_nextSerialNo = value;
    }
}
```

Cada instância `Entity` contém um número de série (e, presumivelmente, algumas outras informações que não são mostradas aqui). O construtor `Entity` (que é como um método de instância) inicializa a nova instância com o próximo número de série disponível. Como o construtor é um membro de instância, ele tem permissão para acessar tanto o campo de instância `_serialNo` quanto o campo estático `s_nextSerialNo`.

Os métodos estáticos `GetNextSerialNo` e `SetNextSerialNo` podem acessar o campo estático `s_nextSerialNo`, mas seria um erro para eles acessar diretamente o campo de instância `_serialNo`.

O exemplo a seguir mostra o uso da classe `Entity`.

C#

```
Entity.SetNextSerialNo(1000);
Entity e1 = new();
Entity e2 = new();
Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
```

Os métodos estáticos `SetNextSerialNo` e `GetNextSerialNo` são invocados na classe, enquanto o método de instância `GetSerialNo` é invocado em instâncias da classe.

## Métodos abstratos, virtuais e de substituição

Você usa métodos virtuais, de substituição e abstratos para definir o comportamento de uma hierarquia de tipos de classe. Como uma classe pode derivar de uma classe base, essas classes derivadas podem precisar modificar o comportamento implementado na classe base. Um método ***virtual*** é um declarado e implementado em uma classe base em que qualquer classe derivada pode fornecer uma implementação mais específica. Um método ***override*** é um método implementado em uma classe derivada que modifica o comportamento da implementação da classe base. Um método ***abstract*** é um método declarado em uma classe base que *deve* ser substituído em todas as classes derivadas. Na verdade, métodos abstratos não definem uma implementação na classe base.

As chamadas de método para métodos de instância podem ser resolvidas para implementações de classe base ou de classe derivada. O tipo de uma variável determina seu *tipo em tempo de compilação*. O *tipo em tempo de compilação* é o tipo que o compilador usa para determinar os membros do item. No entanto, uma variável pode ser atribuída a uma instância de qualquer tipo derivado do *tipo em tempo de compilação* dela. O *tipo de tempo de execução* é o tipo da instância real à qual uma variável se refere.

Quando um método virtual é invocado, o *tipo de tempo de execução* da instância para o qual essa invocação ocorre determina a implementação real do método para invocar. Em uma invocação de método não virtual, o *tipo de tempo de compilação* da instância é o fator determinante.

Um método virtual pode ser *substituído* em uma classe derivada. Quando uma declaração de método de instância inclui um modificador de substituição, o método substitui um método virtual herdado com a mesma assinatura. Uma declaração de método virtual introduz um novo método. Uma declaração de método de substituição especializa um método virtual herdado existente fornecendo uma nova implementação desse método.

Um *método abstrato* é um método virtual sem implementação. Um método abstract é declarado com o modificador `abstract` e é permitido somente em classes abstratas. Um método abstrato deve ser substituído em cada classe derivada não abstrata.

O exemplo a seguir declara uma classe abstrata, `Expression`, que representa um nó de árvore de expressão e três classes derivadas, `Constant`, `VariableReference` e `Operation`, que implementam nós de árvore de expressão para operações aritméticas, referências de variável e constantes. (Este exemplo é semelhante aos tipos de árvore de expressão, mas não está relacionado a eles).

C#

```
public abstract class Expression
{
    public abstract double Evaluate(Dictionary<string, object> vars);
}

public class Constant : Expression
{
    double _value;

    public Constant(double value)
    {
        _value = value;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        return _value;
    }
}

public class VariableReference : Expression
{
    string _name;

    public VariableReference(string name)
    {
        _name = name;
    }

    public override double Evaluate(Dictionary<string, object> vars)
```

```

    {
        object value = vars[_name] ?? throw new Exception($"Unknown
variable: {_name}");
        return Convert.ToDouble(value);
    }
}

public class Operation : Expression
{
    Expression _left;
    char _op;
    Expression _right;

    public Operation(Expression left, char op, Expression right)
    {
        _left = left;
        _op = op;
        _right = right;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        double x = _left.Evaluate(vars);
        double y = _right.Evaluate(vars);
        switch (_op)
        {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;

            default: throw new Exception("Unknown operator");
        }
    }
}

```

As quatro classes anteriores podem ser usadas para modelar expressões aritméticas. Por exemplo, usando instâncias dessas classes, a expressão `x + 3` pode ser representada da seguinte maneira.

C#

```

Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));

```

O método `Evaluate` de uma instância `Expression` é chamado para avaliar a expressão especificada e produzir um valor `double`. O método recebe um argumento `Dictionary` que contém nomes de variáveis (como chaves das entradas) e valores (como valores das

entradas). Como `Evaluate` é um método abstrato, classes não abstratas derivadas de `Expression` devem substituir `Evaluate`.

Uma implementação de `Evaluate` do `Constant` retorna apenas a constante armazenada. Uma implementação de `VariableReference` consulta o nome de variável no dicionário e retorna o valor resultante. Uma implementação de `Operation` primeiro avalia os operandos esquerdo e direito (chamando recursivamente seus métodos `Evaluate`) e, em seguida, executa a operação aritmética determinada.

O seguinte programa usa as classes `Expression` para avaliar a expressão  $x * (y + 2)$  para valores diferentes de `x` e `y`.

C#

```
Expression e = new Operation(
    new VariableReference("x"),
    '*',
    new Operation(
        new VariableReference("y"),
        '+',
        new Constant(2)
    )
);
Dictionary<string, object> vars = new();
vars["x"] = 3;
vars["y"] = 5;
Console.WriteLine(e.Evaluate(vars)); // "21"
vars["x"] = 1.5;
vars["y"] = 9;
Console.WriteLine(e.Evaluate(vars)); // "16.5"
```

## Sobrecarga de método

A *sobrecarga* de método permite que vários métodos na mesma classe tenham o mesmo nome, contanto que tenham assinaturas exclusivas. Ao compilar uma invocação de um método sobrecarregado, o compilador usa a *resolução de sobrecarga* para determinar o método específico para invocar. A resolução de sobrecarga encontra um método que melhor corresponde aos argumentos. Se nenhuma correspondência melhor puder ser encontrada, um erro será relatado. O exemplo a seguir mostra a resolução de sobrecarga em vigor. O comentário para cada invocação no método `UsageExample` mostra qual método é invocado.

C#

```

class OverloadingExample
{
    static void F() => Console.WriteLine("F()");
    static void F(object x) => Console.WriteLine("F(object)");
    static void F(int x) => Console.WriteLine("F(int)");
    static void F(double x) => Console.WriteLine("F(double)");
    static void F<T>(T x) => Console.WriteLine($"F<T>(T), T is
{typeof(T)}");
    static void F(double x, double y) => Console.WriteLine("F(double,
double)");

    public static void UsageExample()
    {
        F();                  // Invokes F()
        F(1);                // Invokes F(int)
        F(1.0);              // Invokes F(double)
        F("abc");             // Invokes F<T>(T), T is System.String
        F((double)1);         // Invokes F(double)
        F((object)1);         // Invokes F(object)
        F<int>(1);            // Invokes F<T>(T), T is System.Int32
        F(1, 1);              // Invokes F(double, double)
    }
}

```

Conforme mostrado pelo exemplo, um método específico sempre pode ser selecionado por meio da conversão explícita dos argumentos para os tipos de parâmetro e argumentos de tipo exatos.

## Outros membros da função

Os membros que contêm código executável são conhecidos coletivamente como *membros de função* de uma classe. A seção anterior descreve os métodos, que são os principais tipos de membros de função. Esta seção descreve os outros tipos de membros da função com suporte do C#: construtores, propriedades, indexadores, eventos, operadores e finalizadores.

O exemplo a seguir mostra uma classe genérica chamada `MyList<T>`, que implementa uma lista de objetos crescente. A classe contém vários exemplos dos tipos mais comuns de membros da função.

C#

```

public class MyList<T>
{
    const int DefaultCapacity = 4;

    T[] _items;

```

```
int _count;

public MyList(int capacity = DefaultCapacity)
{
    _items = new T[capacity];
}

public int Count => _count;

public int Capacity
{
    get => _items.Length;
    set
    {
        if (value < _count) value = _count;
        if (value != _items.Length)
        {
            T[] newItems = new T[value];
            Array.Copy(_items, 0, newItems, 0, _count);
            _items = newItems;
        }
    }
}

public T this[int index]
{
    get => _items[index];
    set
    {
        if (!object.Equals(_items[index], value)) {
            _items[index] = value;
            OnChanged();
        }
    }
}

public void Add(T item)
{
    if (_count == Capacity) Capacity = _count * 2;
    _items[_count] = item;
    _count++;
    OnChanged();
}

protected virtual void OnChanged() =>
    Changed?.Invoke(this, EventArgs.Empty);

public override bool Equals(object other) =>
    Equals(this, other as MyList<T>);

static bool Equals(MyList<T> a, MyList<T> b)
{
    if (Object.ReferenceEquals(a, null)) return
Object.ReferenceEquals(b, null);
    if (Object.ReferenceEquals(b, null) || a._count != b._count)
        return false;
```

```

        for (int i = 0; i < a._count; i++)
    {
        if (!object.Equals(a._items[i], b._items[i]))
        {
            return false;
        }
    }
    return true;
}

public event EventHandler Changed;

public static bool operator ==(MyList<T> a, MyList<T> b) =>
    Equals(a, b);

public static bool operator !=(MyList<T> a, MyList<T> b) =>
    !Equals(a, b);
}

```

## Construtores

O C# dá suporte aos construtores estáticos e de instância. Um *construtor de instância* é um membro que implementa as ações necessárias para inicializar uma instância de uma classe. Um *construtor estático* é um membro que implementa as ações necessárias para inicializar uma classe quando ele for carregado pela primeira vez.

Um construtor é declarado como um método sem nenhum tipo de retorno e o mesmo nome que a classe contém. Se uma declaração de Construtor incluir o modificador `static`, ele declara um construtor estático. Caso contrário, ela declara um construtor de instância.

Construtores de instância podem ser sobrecarregados e ter parâmetros opcionais. Por exemplo, a classe `MyList<T>` declara um construtor de instância com um único parâmetro `int` opcional. Os construtores de instância são invocados usando o operador `new`. As seguintes instruções alocam duas instâncias `MyList<string>` usando o construtor da classe `MyList` com e sem o argumento opcional.

C#

```

MyList<string> list1 = new();
MyList<string> list2 = new(10);

```

Ao contrário de outros membros, os construtores de instância não são herdados. Uma classe não tem construtores de instância diferentes dos construtores realmente

declarados na classe. Se nenhum construtor de instância for fornecido para uma classe, então um construtor vazio sem parâmetros será fornecido automaticamente.

## Propriedades

As *propriedades* são uma extensão natural dos campos. Elas são denominadas membros com tipos associados, e a sintaxe para acessar os campos e as propriedades é a mesma. No entanto, ao contrário dos campos, as propriedades não denotam locais de armazenamento. Em vez disso, as propriedades têm *acessadores* que especificam as instruções executadas quando seus valores são lidos ou gravados. Um *acessador get* lê o valor. Um *acessador set* grava o valor.

Uma propriedade é declarada como um campo, exceto que a declaração termina com um acessador get ou um acessador set escrito entre os delimitadores { e } em vez de terminar com um ponto e vírgula. Uma propriedade que tem um acessador get e um acessador set é uma *propriedade de leitura/gravação*. Uma propriedade que tem apenas um acessador get é uma *propriedade somente leitura*. Uma propriedade que tem apenas um acessador set é uma *propriedade somente gravação*.

Um acessador get corresponde a um método sem parâmetros com um valor retornado do tipo de propriedade. Um acessador set corresponde a um método com um parâmetro único chamado valor e nenhum tipo de retorno. O acessador get obtém o valor da propriedade. O acessador set fornece um novo valor para a propriedade. Quando a propriedade é o destino de uma atribuição, ou usa os operandos ++ ou --, o acessador set é invocado. Em outros casos em que a propriedade é referenciada, o acessador get é invocado.

A classe `MyList<T>` declara duas propriedades, `Count` e `Capacity`, que são somente leitura e leitura/gravação, respectivamente. O seguinte código é um exemplo de uso dessas propriedades:

C#

```
MyList<string> names = new();
names.Capacity = 100;    // Invokes set accessor
int i = names.Count;    // Invokes get accessor
int j = names.Capacity; // Invokes get accessor
```

Como nos campos e métodos, o C# dá suporte a propriedades de instância e a propriedades estáticas. As propriedades estáticas são declaradas com o modificador estático e as propriedades de instância são declaradas sem ele.

Os acessadores de uma propriedade podem ser virtuais. Quando uma declaração de propriedade inclui um modificador `virtual`, `abstract` ou `override`, ela se aplica aos acessadores da propriedade.

## Indexadores

Um *indexador* é um membro que permite que objetos sejam indexados da mesma forma que uma matriz. Um indexador é declarado como uma propriedade, exceto se o nome do membro for `this` seguido por uma lista de parâmetros escrita entre os delimitadores `[` e `]`. Os parâmetros estão disponíveis nos acessadores do indexador. Semelhante às propriedades, os indexadores podem ser de leitura-gravação, somente leitura e somente gravação, e os acessadores de um indexador pode ser virtuais.

A classe `MyList<T>` declara um indexador único de leitura-gravação que usa um parâmetro `int`. O indexador possibilita indexar instâncias `MyList<T>` com valores `int`. Por exemplo:

C#

```
MyList<string> names = new();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++)
{
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Os indexadores podem ser sobrecarregados. Uma classe pode declarar vários indexadores, desde que o número ou os tipos de parâmetros deles sejam diferentes.

## Eventos

Um *evento* é um membro que permite que uma classe ou objeto forneça notificações. Um evento é declarado como um campo, exceto que a declaração inclui uma palavra-chave `event` e o tipo deve ser um tipo delegado.

Dentro de uma classe que declara um membro de evento, o evento se comporta exatamente como um campo de um tipo delegado (desde que o evento não seja abstrato e não declare acessadores). O campo armazena uma referência a um delegado que representa os manipuladores de eventos que foram adicionados ao evento. Se nenhum manipulador de evento estiver presente, o campo será `null`.

A `MyList<T>` classe declara um único membro de evento chamado `Changed`, que indica que um novo item foi adicionado à lista ou um item de lista foi alterado usando o acessador do conjunto de indexadores. O evento Alterado é gerado pelo método virtual `OnChanged`, que primeiro verifica se o evento é `null` (o que significa que nenhum manipulador está presente). A noção de gerar um evento é precisamente equivalente a invocar o delegado representado pelo evento. Não há construções de linguagem especiais para gerar eventos.

Os clientes reagem a eventos por meio de *manipuladores de eventos*. Os manipuladores de eventos são conectados usando o operador `+=` e removidos usando o operador `-=`. O exemplo a seguir anexa um manipulador de eventos para o evento `Changed` de um `MyList<string>`.

C#

```
class EventExample
{
    static int s_changeCount;

    static void ListChanged(object sender, EventArgs e)
    {
        s_changeCount++;
    }

    public static void Usage()
    {
        var names = new MyList<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(s_changeCount); // "3"
    }
}
```

Para cenários avançados em que o controle do armazenamento subjacente de um evento é desejado, uma declaração de evento pode fornecer explicitamente os acessadores `add` e `remove`, que são semelhantes ao acessador `set` de uma propriedade.

## Operadores

Um *operador* é um membro que define o significado da aplicação de um operador de expressão específico para instâncias de uma classe. Três tipos de operadores podem ser definidos: operadores unários, operadores binários e operadores de conversão. Todos os operadores devem ser declarados como `public` e `static`.

A classe `MyList<T>` declara dois operadores: `operator ==` e `operator !=`. Esses operadores substituídos dão um novo significado a expressões que aplicam esses operadores a instâncias `MyList`. Especificamente, os operadores definem a igualdade de duas instâncias `MyList<T>` ao comparar cada um dos objetos contidos usando os métodos `Equals` deles. O exemplo a seguir usa o operador `==` para comparar duas instâncias `MyList<int>`.

C#

```
MyList<int> a = new();
a.Add(1);
a.Add(2);
MyList<int> b = new();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b); // Outputs "True"
b.Add(3);
Console.WriteLine(a == b); // Outputs "False"
```

O primeiro `Console.WriteLine` gera `True` porque as duas listas contêm o mesmo número de objetos com os mesmos valores na mesma ordem. Como `MyList<T>` não definiu `operator ==`, o primeiro `Console.WriteLine` geraria `False` porque `a` e `b` referenciam diferentes instâncias `MyList<int>`.

## Finalizadores

Um *finalizador* é um membro que implementa as ações necessárias para finalizar uma instância de uma classe. Normalmente, um finalizador é necessário para liberar recursos não gerenciados. Os finalizadores não podem ter parâmetros, eles não podem ter modificadores de acessibilidade e não podem ser invocados explicitamente. O finalizador de uma instância é invocado automaticamente durante a coleta de lixo. Para obter mais informações, confira o artigo sobre [finalizadores](#).

O coletor de lixo tem latitude ampla ao decidir quando coletar objetos e executar os finalizadores. Especificamente, o tempo das invocações de finalizadores não é determinístico e eles podem ser executados em qualquer thread. Para esses e outros motivos, as classes devem implementar os finalizadores apenas quando não houver outras soluções viáveis.

A instrução `using` fornece uma abordagem melhor para a destruição de objetos.

## Expressões

Expressões são construídas a partir de *operandos* e *operadores*. Os operadores de uma expressão indicam quais operações devem ser aplicadas aos operandos. Exemplos de operadores incluem `+`, `-`, `*`, `/` e `new`. Exemplos de operandos incluem literais, campos, variáveis locais e expressões.

Quando uma expressão contém vários operadores, a *precedência* dos operadores controla a ordem na qual os operadores individuais são avaliados. Por exemplo, a expressão `x + y * z` é avaliada como `x + (y * z)` porque o operador `*` tem precedência maior do que o operador `+`.

Quando ocorre um operando entre dois operadores com a mesma precedência, a *associatividade* dos operadores controla a ordem na qual as operações são executadas:

- Exceto para os operadores de atribuição e os operadores de avaliação de nulo, todos os operadores binários são *associativos à esquerda*, o que significa que as operações são executadas da esquerda para a direita. Por exemplo, `x + y + z` é avaliado como `(x + y) + z`.
- Os operadores de atribuição, os operadores de avaliação de nulo `??` e os operadores `??=`, bem como o operador condicional `?:`, são *associativo à direita*, o que significa que as operações são executadas da direita para a esquerda. Por exemplo, `x = y = z` é avaliado como `x = (y = z)`.

Precedência e associatividade podem ser controladas usando parênteses. Por exemplo, `x + y * z` primeiro multiplica `y` por `z` e, em seguida, adiciona o resultado a `x`, mas `(x + y) * z` primeiro adiciona `x` e `y` e, em seguida, multiplica o resultado por `z`.

A maioria dos operadores pode ser *sobre carregada*. A sobrecarga de operador permite que implementações de operador definidas pelo usuário sejam especificadas para operações em que um ou ambos os operandos são de um tipo struct ou de classe definida pelo usuário.

O C# fornece operadores para executar operações aritméticas, lógicas, de bits e deslocamentos e comparações de igualdade e de ordem.

Para obter a lista completa de operadores do C# ordenada pelo nível de precedência, confira [Operadores do C#](#).

## Instruções

As ações de um programa são expressas usando *instruções*. O C# oferece suporte a vários tipos diferentes de instruções, algumas delas definidas em termos de instruções inseridas.

- Um *bloco* permite a produção de várias instruções em contextos nos quais uma única instrução é permitida. Um bloco é composto por uma lista de instruções escritas entre os delimitadores { e }.
- *Instruções de declaração* são usadas para declarar constantes e variáveis locais.
- *Instruções de expressão* são usadas para avaliar expressões. As expressões que podem ser usadas como instruções incluem chamadas de método, alocações de objeto usando o operador new, atribuições usando = e os operadores de atribuição compostos, operações de incremento e decremento usando os operadores ++ e -- e as expressões await.
- *Instruções de seleção* são usadas para selecionar uma dentre várias instruções possíveis para execução com base no valor de alguma expressão. Esse grupo contém as instruções if e switch.
- *Instruções de iteração* são usadas para executar repetidamente uma instrução inserida. Esse grupo contém as instruções while, do, for e foreach.
- *Instruções de salto* são usadas para transferir o controle. Esse grupo contém as instruções break, continue, goto, throw, return e yield.
- A instrução try... catch é usada para capturar exceções que ocorrem durante a execução de um bloco, e a instrução try... finally é usada para especificar o código de finalização que é executado sempre, se uma exceção ocorrer ou não.
- As instruções checked e unchecked são usadas para controlar o contexto de verificação de estouro para operações e conversões aritméticas do tipo integral.
- A instrução lock é usada para obter o bloqueio de exclusão mútua para um determinado objeto, executar uma instrução e, em seguida, liberar o bloqueio.
- A instrução using é usada para obter um recurso, executar uma instrução e, em seguida, descartar esse recurso.

A seguinte lista mostra os tipos de instruções que podem ser usados:

- Declaração de variável local.
- Declaração constante local.
- Instrução de expressão.
- Instrução if.
- Instrução switch.
- Instrução while.
- Instrução do.
- Instrução for.
- Instrução foreach.
- Instrução break.
- Instrução continue.
- Instrução goto.

- Instrução `return`.
- Instrução `yield`.
- Instruções `throw` e instruções `try`.
- Instruções `checked` e `unchecked`.
- Instrução `lock`.
- Instrução `using`.

[Anterior](#)

[Avançar](#)

# Principais áreas da linguagem C#

Artigo • 20/03/2023

Este artigo apresenta os principais recursos da linguagem C#.

## Matrizes, coleções e LINQ

C# e .NET fornecem muitos tipos de coleção diferentes. As matrizes têm a sintaxe definida pela linguagem. Tipos de coleção genéricos são listados no namespace [System.Collections.Generic](#). As coleções especializadas incluem [System.Span<T>](#) para acessar a memória contínua no registro de ativação e [System.Memory<T>](#) para acessar a memória contínua no heap gerenciado. Todas as coleções, incluindo matrizes, [Span<T>](#) e [Memory<T>](#) compartilham um princípio unificador para iteração. Você usa a interface [System.Collections.Generic.IEnumerable<T>](#). Esse princípio unificador significa que qualquer um dos tipos de coleção pode ser usado com consultas LINQ ou outros algoritmos. Você escreve métodos usando [IEnumerable<T>](#) e esses algoritmos funcionam com qualquer coleção.

### Matrizes

Uma **matriz** é uma estrutura de dados que contém algumas variáveis acessadas por meio de índices calculados. As variáveis contidas em uma matriz, também chamadas de **elementos** da matriz, são do mesmo tipo. Esse tipo é chamado de **tipo do elemento** da matriz.

Os tipos de matriz são tipos de referência, e a declaração de uma variável de matriz simplesmente reserva espaço para uma referência a uma instância de matriz. As instâncias de matriz reais são criadas dinamicamente em tempo de operação usando o operador `new`. A operação `new` especifica o **comprimento** da nova instância de matriz, que é fixo durante todo o tempo de vida da instância. Os índices dos elementos de uma matriz variam de `0` a `Length - 1`. O operador `new` inicializa automaticamente os elementos de uma matriz usando o valor padrão, que, por exemplo, é zero para todos os tipos numéricos e `null` para todos os tipos de referência.

O exemplo a seguir cria uma matriz de elementos `int`, inicializa a matriz e imprime o conteúdo dela.

C#

```
int[] a = new int[10];
for (int i = 0; i < a.Length; i++)
```

```
{  
    a[i] = i * i;  
}  
for (int i = 0; i < a.Length; i++)  
{  
    Console.WriteLine($"a[{i}] = {a[i]}");  
}
```

Este exemplo cria e opera em uma *matriz unidimensional*. O C# também oferece suporte a *matrizes multidimensionais*. O número de dimensões de um tipo de matriz, também conhecido como *Ordem* do tipo de matriz, é uma unidade a mais que o número de vírgulas entre os colchetes do tipo de matriz. O exemplo a seguir aloca uma matriz unidimensional, bidimensional e tridimensional, respectivamente.

C#

```
int[] a1 = new int[10];  
int[,] a2 = new int[10, 5];  
int[,,] a3 = new int[10, 5, 2];
```

A matriz `a1` contém 10 elementos, a matriz `a2` contém 50 ( $10 \times 5$ ) elementos e a matriz `a3` contém 100 ( $10 \times 5 \times 2$ ) elementos. O tipo do elemento de uma matriz pode ser qualquer tipo, incluindo um tipo de matriz. Uma matriz com elementos do tipo matriz geralmente é chamada de *matriz denteada*, pois os comprimentos das matrizes dos elementos variam. O exemplo a seguir aloca uma matriz de matrizes de `int`:

C#

```
int[][] a = new int[3][];  
a[0] = new int[10];  
a[1] = new int[5];  
a[2] = new int[20];
```

A primeira linha cria uma matriz com três elementos, cada um do tipo `int[]`, e cada um com um valor inicial de `null`. Em seguida, as próximas linhas inicializam os três elementos com referências a instâncias de matriz individuais de comprimentos variados.

O operador `new` permite que os valores iniciais dos elementos da matriz sejam especificados usando um *inicializador de matriz*, que é uma lista de expressões escritas entre os delimitadores `{` e `}`. O exemplo a seguir aloca e inicializa um `int[]` com três elementos.

C#

```
int[] a = new int[] { 1, 2, 3 };
```

O comprimento da matriz é inferido do número de expressões entre { e }. A inicialização da matriz pode ser reduzida ainda mais, de modo que o tipo de matriz não precisa ser reafirmado.

C#

```
int[] a = { 1, 2, 3 };
```

Ambos os exemplos anteriores são equivalentes ao seguinte código:

C#

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

A instrução `foreach` pode ser usada para enumerar os elementos de qualquer coleção. O seguinte código enumera a matriz do exemplo anterior:

C#

```
foreach (int item in a)
{
    Console.WriteLine(item);
}
```

A instrução `foreach` usa a interface `IEnumerable<T>`, para que possa funcionar com qualquer coleção.

## Interpolação de cadeia de caracteres

A [\*interpolação de cadeia de caracteres\*](#) do C# permite formatar cadeias de caracteres definindo expressões cujos resultados são colocados em uma cadeia de caracteres de formato. Por exemplo, a seguinte demonstração imprime a temperatura em determinado dia de um conjunto de dados meteorológicos:

C#

```
Console.WriteLine($"The low and high temperature on {weatherData.Date:MM-dd-yyyy}");
Console.WriteLine($"      was {weatherData.LowTemp} and
```

```
{weatherData.HighTemp} .");
// Output (similar to):
// The low and high temperature on 08-11-2020
//      was 5 and 30.
```

Uma cadeia de caracteres interpolada é declarada usando o token `$`. A interpolação de cadeia de caracteres avalia as expressões entre `{` e `}`, em seguida, converte o resultado em um `string` e substitui o texto entre os colchetes pelo resultado da cadeia de caracteres da expressão. O `:` na primeira expressão, `{weatherData.Date:MM-dd-yyyy}` especifica a *cadeia de caracteres de formato*. No exemplo anterior, ele especifica que a data deve ser impressa no formato "MM-dd-aaaa".

## Correspondência de padrões

A linguagem C# fornece expressões de *padrões correspondentes* para consultar o estado de um objeto e executar o código com base nesse estado. Você pode inspecionar tipos e valores de propriedades e campos para determinar qual ação deve ser tomada. Você também pode inspecionar os elementos de uma lista ou matriz. A expressão `switch` é a expressão primária para correspondência de padrões.

## Delegados e expressões lambda

Um *tipo delegado* representa referências aos métodos com uma lista de parâmetros e tipo de retorno específicos. Delegados possibilitam o tratamento de métodos como entidades que podem ser atribuídos a variáveis e passadas como parâmetros. Delegados são semelhantes ao conceito de ponteiros de função encontrado em algumas outras linguagens. Ao contrário dos ponteiros de função, os delegados são orientados a objetos e fortemente tipados.

O exemplo a seguir declara e usa um tipo delegado chamado `Function`.

C#

```
delegate double Function(double x);

class Multiplier
{
    double _factor;

    public Multiplier(double factor) => _factor = factor;

    public double Multiply(double x) => x * _factor;
}
```

```

class DelegateExample
{
    static double[] Apply(double[] a, Function f)
    {
        var result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    public static void Main()
    {
        double[] a = { 0.0, 0.5, 1.0 };
        double[] squares = Apply(a, (x) => x * x);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}

```

Uma instância do tipo delegado `Function` pode fazer referência a qualquer método que usa um argumento `double` e retorna um valor `double`. O método `Apply` aplica um determinado `Function` aos elementos de um `double[]`, retornando um `double[]` com os resultados. No método `Main`, `Apply` é usado para aplicar três funções diferentes para um `double[]`.

Um delegado pode referenciar uma expressão lambda para criar uma função anônima (como `(x) => x * x` no exemplo anterior), um método estático (como `Math.Sin` no exemplo anterior) ou um método de instância (como `m.Multiply` no exemplo anterior). Um delegado que referencia um método de instância também referencia um objeto específico, e quando o método de instância é invocado por meio do delegado, esse objeto se torna `this` na invocação.

Os delegados também podem ser criados usando funções anônimas ou expressões lambda, que são "métodos em linha" criados quando declarados. As funções anônimas podem ver as variáveis locais dos métodos ao redor. O seguinte exemplo não cria uma classe:

C#

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

Um delegado não sabe nem se importa com a classe do método a que ele faz referência. O método referenciado deve ter os mesmos parâmetros e tipo de retorno que o delegado.

# async / await

O C# dá suporte a programas assíncronos com duas palavras-chave: `async` e `await`. Você adiciona o modificador `async` a uma declaração de método para declarar que o método é assíncrono. O operador `await` informa ao compilador para aguardar de maneira assíncrona um resultado para concluir. O controle é retornado ao chamador e o método retorna uma estrutura que gerencia o estado do trabalho assíncrono. A estrutura normalmente é um `System.Threading.Tasks.Task<TResult>`, mas pode ser qualquer tipo que dê suporte ao padrão `awaiter`. Esses recursos permitem que você escreva código que lê como seu equivalente síncrono, mas é executado de maneira assíncrona. Por exemplo, o seguinte código baixa a home page para [documentos da Microsoft](#):

C#

```
public async Task<int> RetrieveDocsHomePage()
{
    var client = new HttpClient();
    byte[] content = await
client.GetByteArrayAsync("https://learn.microsoft.com/");
    Console.WriteLine($"{nameof(RetrieveDocsHomePage)}: Finished
downloading.");
    return content.Length;
}
```

Este pequeno exemplo mostra os principais recursos para programação assíncrona:

- A declaração do método inclui o modificador `async`.
- O corpo do método `await` é o retorno do método `GetByteArrayAsync`.
- O tipo especificado na instrução `return` corresponde ao argumento de tipo da declaração `Task<T>` do método. (Um método que retorna `Task` usaria instruções `return` sem nenhum argumento).

## Atributos

Tipos, membros e outras entidades em um programa C# dão suporte a modificadores que controlam determinados aspectos de seu comportamento. Por exemplo, a acessibilidade de um método é controlada usando os modificadores `public`, `protected`, `internal` e `private`. O C# generaliza essa funcionalidade, de modo que os tipos definidos pelo usuário de informações declarativas podem ser anexados a entidades de

programa e recuperados no tempo de execução. Os programas especificam essas informações declarativas definindo e usando [atributos](#).

O exemplo a seguir declara um atributo `HelpAttribute` que pode ser colocado em entidades de programa para fornecem links para a documentação associada.

C#

```
public class HelpAttribute : Attribute
{
    string _url;
    string _topic;

    public HelpAttribute(string url) => _url = url;

    public string Url => _url;

    public string Topic
    {
        get => _topic;
        set => _topic = value;
    }
}
```

Todas as classes de atributo derivam da classe base [Attribute](#), fornecida pela biblioteca .NET. Os atributos podem ser aplicados, fornecendo seu nome, junto com quaisquer argumentos, dentro dos colchetes pouco antes da declaração associada. Se o nome de um atributo termina em `Attribute`, essa parte do nome pode ser omitida quando o atributo é referenciado. Por exemplo, o atributo `HelpAttribute` pode ser usado da seguinte maneira.

C#

```
[Help("https://learn.microsoft.com/dotnet/csharp/tour-of-csharp/features")]
public class Widget
{
    [Help("https://learn.microsoft.com/dotnet/csharp/tour-of-
csharp/features",
        Topic = "Display")]
    public void Display(string text) { }
```

Este exemplo anexa um `HelpAttribute` à classe `Widget`. Ele adiciona outro `HelpAttribute` ao método `Display` na classe. Os construtores públicos de uma classe de atributo controlam as informações que devem ser fornecidas quando o atributo é anexado a uma entidade de programa. As informações adicionais podem ser fornecidas

ao referenciar propriedades públicas de leitura-gravação da classe de atributo (como a referência anterior à propriedade `Topic`).

Os metadados definidos por atributos podem ser lidos e manipulados em tempo de execução usando reflexão. Quando um atributo específico é solicitado usando essa técnica, o construtor da classe de atributo é invocado com as informações fornecidas na origem do programa. A instância de atributo resultante é retornada. Se forem fornecidas informações adicionais por meio de propriedades, essas propriedades serão definidas para os valores fornecidos antes que a instância do atributo seja retornada.

O exemplo de código a seguir demonstra como obter as instâncias `HelpAttribute` associadas à classe `Widget` e seu método `Display`.

C#

```
Type widgetType = typeof(Widget);

object[] widgetClassAttributes =
widgetType.GetCustomAttributes(typeof(HelpAttribute), false);

if (widgetClassAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)widgetClassAttributes[0];
    Console.WriteLine($"Widget class help URL : {attr.Url} - Related topic :
{attr.Topic}");
}

System.Reflection.MethodInfo displayMethod =
widgetType.GetMethod(nameof(Widget.Display));

object[] displayMethodAttributes =
displayMethod.GetCustomAttributes(typeof(HelpAttribute), false);

if (displayMethodAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)displayMethodAttributes[0];
    Console.WriteLine($"Display method help URL : {attr.Url} - Related topic
: {attr.Topic}");
}
```

## Saiba mais

Você pode explorar mais sobre o C# experimentando um de nossos [tutoriais](#).

[Anterior](#)

# Introdução ao C#

Artigo • 15/02/2023

Bem-vindo aos tutoriais de introdução ao C#. Essas lições começam com código interativo que pode ser executado em seu navegador. Você pode aprender as noções básicas do C# na [série de vídeos de introdução ao C#](#) antes de iniciar essas lições interativas.

<https://docs.microsoft.com/shows/CSharp-101/What-is-C/player>

As primeiras lições explicam os conceitos de C# usando pequenos snippets de código. Você aprenderá os conceitos básicos da sintaxe de C# e como trabalhar com tipos de dados como cadeias de caracteres, números e valores booleanos. É tudo interativo e você começará a gravar e executar o código em questão de minutos. Estas primeiras lições não exigem conhecimento prévio de programação ou da linguagem C#.

Você pode experimentar esses tutoriais em ambientes diferentes. Os conceitos que você aprenderá são os mesmos. A diferença é qual experiência você prefere:

- [No navegador, na plataforma de documentos](#): essa experiência incorpora uma janela de código C# executável em páginas de documentos. Você escreve e executa o código C# no navegador.
- [Na experiência do Microsoft Learn](#). Este roteiro de aprendizagem contém vários módulos que ensinam as noções básicas de C#.
- [No Jupyter no Binder](#). Você pode experimentar o código C# em um notebook Jupyter no Binder.
- [Em seu computador local](#). Depois de explorar online, você pode [baixar](#) o SDK do .NET e criar programas em seu computador.

Todos os tutoriais de introdução posteriores à lição Olá, Mundo estão disponíveis por meio da experiência de navegador online ou [em seu próprio ambiente de desenvolvimento local](#). No final de cada tutorial, você decidirá se deseja continuar com a próxima lição online ou no próprio computador. Há links para ajudar você a configurar seu ambiente e continuar com o próximo tutorial no computador.

## Olá, Mundo

No tutorial [Olá, Mundo](#), você criará o programa C# mais básico. Você explorará o tipo `string` e como trabalhar com texto. Você também pode usar o caminho no [Microsoft Learn](#) ou o [Jupyter no Binder](#).

## Números em C#

No tutorial Números em C#, você aprenderá como os computadores armazenam números e como executar cálculos com diferentes tipos de número. Você aprenderá os conceitos básicos de arredondamento e como executar cálculos matemáticos usando C#. Este tutorial também está disponível [para execução local no seu computador](#).

Esse tutorial pressupõe que você já tenha concluído a lição Olá, Mundo.

## Loops e branches

O tutorial Branches e loops ensina os conceitos básicos da seleção de diferentes caminhos de execução de código com base nos valores armazenados em variáveis. Você aprenderá os conceitos básicos do fluxo de controle, que são os fundamentos de como os programas tomam decisões e escolhem ações diferentes. Este tutorial também está disponível [para execução local no seu computador](#).

Esse tutorial pressupõe que você já tenha concluído as lições Olá, Mundo e Números em C#.

## Coleções de lista

A lição Coleções de lista fornece um tour pelo tipo Coleções de lista que armazena as sequências de dados. Você aprenderá a adicionar e remover itens, pesquisar itens e classificar listas. Você explorará os diferentes tipos de listas. Este tutorial também está disponível [para execução local no seu computador](#).

Esse tutorial pressupõe que você já tenha concluído as lições listadas acima.

## Exemplos básicos do LINQ ↗

Este exemplo requer a ferramenta global [dotnet-try](#) ↗. Depois de instalar a ferramenta e clonar o repositório [try-samples](#) ↗, você poderá aprender LINQ (Consulta Integrada à Linguagem) por meio de um conjunto de amostras básicas que você pode executar interativamente. Você pode explorar diferentes maneiras de consultar, explorar e transformar sequências de dados.

# Configurar o ambiente local

Artigo • 10/05/2023

A primeira etapa para executar um tutorial em seu computador é configurar um ambiente de desenvolvimento.

- Recomendamos o [Visual Studio](#) para Windows ou Mac. Você pode baixar uma versão gratuita na [página de downloads do Visual Studio](#). O Visual Studio inclui o SDK do .NET.
- Você também pode usar o editor do [Visual Studio Code](#). Será preciso instalar o [SDK do .NET](#) mais recente separadamente.
- Se preferir outro editor, você precisará instalar o [SDK do .NET](#) mais recente.

## Fluxo de desenvolvimento de aplicativos básico

As instruções nesses tutoriais pressupõem que você esteja usando a CLI do .NET para criar, compilar e executar aplicativos. Você usará os comandos a seguir:

- [dotnet new](#) cria um aplicativo. Este comando gera os arquivos e ativos necessários para o seu aplicativo. Todos os tutoriais de introdução ao C# usam o tipo de aplicativo `console`. Depois de conhecer as noções básicas, você poderá expandir para outros tipos de aplicativo.
- [dotnet build](#) cria o executável.
- [dotnet run](#) executa o executável.

Se você usar o Visual Studio 2019 para estes tutoriais, escolherá uma seleção de menu do Visual Studio quando um tutorial o orientar a executar um destes comandos da CLI:

- **File>New>Project** cria um aplicativo.
  - O modelo de projeto `Console Application` é recomendado.
  - Você terá a opção de especificar uma estrutura de destino. Os tutoriais abaixo funcionam melhor ao direcionar o .NET 5 ou superior.
- **Build>Build Solution** cria o executável.
- **Debug>Start Without Debugging** executa o executável.

## Escolha seu tutorial

Você pode iniciar com qualquer um dos seguintes tutoriais:

# Números em C#

No tutorial [Números em C#](#), você aprenderá como os computadores armazenam números e como executar cálculos com diferentes tipos de número. Você aprenderá os conceitos básicos de arredondamento e como executar cálculos matemáticos usando C#.

Esse tutorial pressupõe a conclusão da lição [Olá, Mundo](#).

# Loops e branches

O tutorial [Branches e loops](#) ensina os conceitos básicos da seleção de diferentes caminhos de execução de código com base nos valores armazenados em variáveis. Você aprenderá os conceitos básicos do fluxo de controle, que são os fundamentos de como os programas tomam decisões e escolhem ações diferentes.

Esse tutorial pressupõe a conclusão das lições [Olá, Mundo](#) e [Números em C#](#).

# Coleções de lista

A lição [Colecções de lista](#) fornece um tour pelo tipo Coleções de lista que armazena as sequências de dados. Você aprenderá a adicionar e remover itens, pesquisar itens e classificar listas. Você explorará os diferentes tipos de listas.

Esse tutorial pressupõe a conclusão das lições listadas acima.

# Como usar números inteiros e de ponto flutuante em C#

Artigo • 10/05/2023

Este tutorial ensina tipos numéricos em C#. Você escreverá pequenas quantidades de código, depois compilará e executará esse código. O tutorial contém uma série de lições que exploram números e operações matemáticas em C#. Estas lições ensinam os princípios básicos da linguagem C#.

## 💡 Dica

Para colar um snippet de código dentro do **modo de foco**, você deve usar o atalho de teclado (`ctrl` + `v` ou `cmd` + `v`).

## Pré-requisitos

Este tutorial espera que você tenha um computador configurado para desenvolvimento local. Consulte [Configurar seu ambiente local](#) para obter instruções de instalação e uma visão geral do desenvolvimento de aplicativos no .NET.

Se você não quiser configurar um ambiente local, consulte a [versão interativa no navegador deste tutorial](#).

## Explorar a matemática de inteiros

Crie um diretório chamado *numbers-quickstart*. Torne esse o diretório atual e execute o seguinte comando:

CLI do .NET

```
dotnet new console -n NumbersInCSharp -o .
```

## ⓘ Importante

Os modelos C# para o .NET 6 usam *instruções de nível superior*. Se você já tiver atualizado para o .NET 6, talvez seu aplicativo não corresponda ao código descrito neste artigo. Para obter mais informações, consulte o artigo sobre [Novos modelos C# geram instruções de nível superior](#)

O SDK do .NET 6 também adiciona um conjunto de diretivas `implícitas global using` para projetos que usam os seguintes SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Essas diretivas `implícitas global using` incluem os namespaces mais comuns para o tipo de projeto.

Abra `Program.cs` em seu editor favorito e substitua o conteúdo do arquivo pelo seguinte código:

C#

```
int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

Execute este código digitando `dotnet run` na janela de comando.

Você viu apenas uma das operações matemáticas fundamentais com números inteiros. O tipo `int` representa um **inteiro**, zero, um número inteiro positivo ou negativo. Você usa o símbolo `+` para adição. Outras operações matemáticas comuns para inteiros incluem:

- `-` para subtração
- `*` para multiplicação
- `/` para divisão

Comece explorando essas diferentes operações. Adicione estas linhas após a linha que grava o valor de `c`:

C#

```
// subtraction
c = a - b;
Console.WriteLine(c);

// multiplication
c = a * b;
Console.WriteLine(c);

// division
```

```
c = a / b;  
Console.WriteLine(c);
```

Execute este código digitando `dotnet run` na janela de comando.

Você também pode experimentar executar várias operações matemáticas na mesma linha, se quiser. Experimente `c = a + b - 12 * 17;`, por exemplo. É permitido misturar variáveis e números constantes.

### 💡 Dica

À medida que explora C# (ou qualquer linguagem de programação), você cometerá erros ao escrever o código. O **compilador** encontrará esses erros e os reportará a você. Quando a saída contiver mensagens de erro, analise atentamente o código de exemplo e o código em sua janela para ver o que deve ser corrigido. Esse exercício ajudará você a conhecer a estrutura do código C#.

Você terminou a primeira etapa. Antes de iniciar a próxima seção, vamos passar o código atual para um *método* separado. Um método é uma série de instruções agrupadas que receberam um nome. Você nomeia um método escrevendo o nome do método seguido por `()`. Organizar seu código em métodos torna mais fácil começar a trabalhar com um novo exemplo. Quando você terminar, seu código deverá ter a seguinte aparência:

```
C#  
  
WorkWithIntegers();  
  
void WorkWithIntegers()  
{  
    int a = 18;  
    int b = 6;  
    int c = a + b;  
    Console.WriteLine(c);  
  
    // subtraction  
    c = a - b;  
    Console.WriteLine(c);  
  
    // multiplication  
    c = a * b;  
    Console.WriteLine(c);  
  
    // division  
    c = a / b;
```

```
        Console.WriteLine(c);
    }
```

A linha `WorkWithIntegers();` invoca o método. O código a seguir declara o método e o define.

## Explorar a ordem das operações

Comente a chamada para `WorkingWithIntegers()`. Isso tornará a saída menos congestionada enquanto você trabalha nesta seção:

```
C#
//WorkWithIntegers();
```

O `//` inicia um **comentário** em C#. Os comentários são qualquer texto que você queira manter em seu código-fonte, mas não queria executar como código. O compilador não gera nenhum código executável a partir dos comentários. Como `WorkWithIntegers()` é um método, você precisa apenas comentar uma linha.

A linguagem C# define a precedência de operações matemáticas diferentes com regras consistentes às regras que você aprendeu em matemática. Multiplicação e divisão têm precedência sobre adição e subtração. Explore isso adicionando o seguinte código a `WorkWithIntegers()` após a chamada e executando `dotnet run`:

```
C#
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

A saída demonstra que a multiplicação é executada antes da adição.

Você pode forçar uma ordem diferente de operações, adicionando parênteses para delimitar a operação, ou operações, que você quer realizar primeiro. Adicione as seguintes linhas e execute novamente:

```
C#
d = (a + b) * c;
Console.WriteLine(d);
```

Explore mais, combinando várias operações diferentes. Adicione algo semelhante às linhas a seguir. Tente `dotnet run` novamente.

C#

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

Talvez você tenha observado um comportamento interessante com relação aos números inteiros. A divisão de inteiros sempre produz um resultado inteiro, mesmo quando você espera que o resultado inclua uma parte decimal ou fracionária.

Se você ainda não viu esse comportamento, tente o seguinte:

C#

```
int e = 7;
int f = 4;
int g = 3;
int h = (e + f) / g;
Console.WriteLine(h);
```

Digite `dotnet run` novamente para ver os resultados.

Antes de avançarmos, pegue todo código que você escreveu nesta seção e coloque-o em um novo método. Chame esse novo método de `OrderPrecedence`. Seu código deve ter a seguinte aparência:

C#

```
// WorkWithIntegers();
OrderPrecedence();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);

    // subtraction
    c = a - b;
    Console.WriteLine(c);

    // multiplication
    c = a * b;
    Console.WriteLine(c);
```

```

// division
c = a / b;
Console.WriteLine(c);
}

void OrderPrecedence()
{
    int a = 5;
    int b = 4;
    int c = 2;
    int d = a + b * c;
    Console.WriteLine(d);

    d = (a + b) * c;
    Console.WriteLine(d);

    d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
    Console.WriteLine(d);

    int e = 7;
    int f = 4;
    int g = 3;
    int h = (e + f) / g;
    Console.WriteLine(h);
}

```

## Explorar a precisão de inteiros e limites

Esse último exemplo mostrou que uma divisão de inteiros trunca o resultado. Você pode obter o **restante** usando o operador **module**, o caractere `%`. Tente o seguinte código após a chamada de método para `OrderPrecedence()`:

```
C#
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

O tipo de inteiro C# difere dos inteiros matemáticos de outra forma: o tipo `int` tem limites mínimo e máximo. Adicione este código para ver esses limites:

```
C#
```

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

Se um cálculo produzir um valor que excede esses limites, você terá uma condição de **estouro negativo** ou **estouro**. A resposta parece quebrar de um limite para o outro. Adicione estas duas linhas para ver um exemplo:

C#

```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

Observe que a resposta é muito próxima do mínimo inteiro (negativo). É o mesmo que `min + 2`. A operação de adição **estourou** os valores permitidos para números inteiros. A resposta é um número negativo muito grande, pois um estouro "envolve" do maior valor de inteiro possível para o menor.

Há outros tipos numéricos com limites e precisão diferentes que você usaria quando o tipo `int` não atendesse às suas necessidades. Vamos explorar esses outros tipos em seguida. Antes de iniciar a próxima seção, mova o código que você escreveu nesta seção para um método separado. Nomeie-o como `TestLimits`.

## Trabalhar com o tipo Double

O tipo numérico `double` representa um número de ponto flutuante de precisão dupla. Esses termos podem ser novidade para você. Um número de **ponto flutuante** é útil para representar números não inteiros que podem ser muito grandes ou pequenos em magnitude. **Precisão dupla** é um termo relativo que descreve o número de dígitos binários usados para armazenar o valor. Os números de **precisão dupla** têm o dobro do número de dígitos binários do que os de **precisão simples**. Em computadores modernos, é mais comum usar números de precisão dupla do que de precisão simples. Números de **precisão simples** são declarados usando a palavra-chave `float`. Vamos explorar. Adicione o seguinte código e veja o resultado:

C#

```
double a = 5;
double b = 4;
double c = 2;
double d = (a + b) / c;
Console.WriteLine(d);
```

Observe que a resposta inclui a parte decimal do quociente. Experimente uma expressão ligeiramente mais complicada com duplos:

```
C#  
  
double e = 19;  
double f = 23;  
double g = 8;  
double h = (e + f) / g;  
Console.WriteLine(h);
```

O intervalo de um valor duplo é muito maior do que valores inteiros. Experimente o código a seguir abaixo do código que você escreveu até o momento:

```
C#  
  
double max = double.MaxValue;  
double min = double.MinValue;  
Console.WriteLine($"The range of double is {min} to {max}");
```

Esses valores são impressos em notação científica. O número à esquerda do E é o significando. O número à direita é o expoente, como uma potência de 10. Assim como os números decimais em matemática, os duplos em C# podem ter erros de arredondamento. Experimente esse código:

```
C#  
  
double third = 1.0 / 3.0;  
Console.WriteLine(third);
```

Você sabe que 0.3 repetido um número finito de vezes não é exatamente o mesmo que 1/3.

### *Desafio*

Experimente outros cálculos com números grandes, números pequenos, multiplicação e divisão usando o tipo double. Experimente cálculos mais complicados. Após algum tempo no desafio, pegue o código que você escreveu e coloque-o em um novo método. Chame esse novo método de WorkWithDoubles.

## Trabalhar com tipos decimais

Você viu os tipos numéricos básicos em C#: inteiros e duplos. Ainda há outro tipo: o tipo `decimal`. O tipo `decimal` tem um intervalo menor, mas precisão maior do que `double`. Vamos dar uma olhada:

C#

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

Observe que o intervalo é menor do que o tipo `double`. Veja a precisão maior com o tipo `decimal` experimentando o código a seguir:

C#

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

O sufixo `M` nos números é o modo como você indica que uma constante deve usar o tipo `decimal`. Caso contrário, o compilador assumirá o tipo `double`.

### ① Observação

A letra `M` foi escolhida como a letra mais visualmente distinta entre as palavras-chave `double` e `decimal`.

Observe que o cálculo usando o tipo `decimal` tem mais dígitos à direita da vírgula decimal.

### *Desafio*

Agora que você viu os diferentes tipos numéricos, escreva um código que calcula a área de um círculo cujo raio é de 2,50 centímetros. Lembre-se de que a área de um círculo é o quadrado do raio multiplicado por PI. Uma dica: o .NET contém uma constante para PI, `Math.PI`, que você pode usar para esse valor. `Math.PI`, como todas as constantes declaradas no namespace `System.Math`, é um valor `double`. Por esse motivo, você deve usar valores `double` em vez de `decimal` para esse desafio.

Você deve obter uma resposta entre 19 e 20. Confira sua resposta [analisando o código de exemplo finalizado no GitHub](#).

Experimente outras fórmulas, se quiser.

Você concluiu o início rápido "Números em C#". Continue com o início rápido [Branches e loops](#) em seu próprio ambiente de desenvolvimento.

Saiba mais sobre os números em C# nos artigos a seguir:

- [Tipos numéricos integrais](#)
- [Tipos numéricos de ponto flutuante](#)
- [Conversões numéricas internas](#)

# Instruções e loops em C# `if` – tutorial de lógica condicional

Artigo • 10/05/2023

Este tutorial ensina a escrever código C# que examina variáveis e muda o caminho de execução com base nessas variáveis. Escreva o código em C# e veja os resultados da compilação e da execução. O tutorial contém uma série de lições que exploram construções de branches e loops em C#. Estas lições ensinam os princípios básicos da linguagem C#.

## Dica

Para colar um snippet de código dentro do **modo de foco**, você deve usar o atalho de teclado (`Ctrl` + `V` ou `cmd` + `V`).

## Pré-requisitos

Este tutorial espera que você tenha um computador configurado para desenvolvimento local. Consulte [Configurar seu ambiente local](#) para obter instruções de instalação e uma visão geral do desenvolvimento de aplicativos no .NET.

Se você preferir executar o código sem precisar configurar um ambiente local, consulte a [versão interativa no navegador deste tutorial](#).

## Tome decisões usando a instrução `if`

Crie um diretório chamado *branches-tutorial*. Torne esse o diretório atual e execute o seguinte comando:

CLI do .NET

```
dotnet new console -n BranchesAndLoops -o .
```

## Importante

Os modelos C# para o .NET 6 usam *instruções de nível superior*. Se você já tiver atualizado para o .NET 6, talvez seu aplicativo não corresponda ao código descrito

neste artigo. Para obter mais informações, consulte o artigo sobre [Novos modelos C# geram instruções de nível superior](#)

O SDK do .NET 6 também adiciona um conjunto de diretivas *implícitas global using* para projetos que usam os seguintes SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Essas diretivas *implícitas global using* incluem os namespaces mais comuns para o tipo de projeto.

Esse comando cria um novo aplicativo de console .NET no diretório atual. Abra *Program.cs* em seu editor favorito e substitua o conteúdo pelo seguinte código:

C#

```
int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

Experimente este código digitando `dotnet run` na sua janela do console. Você deverá ver a mensagem "A resposta é maior que 10" impressa no console. Modifique a declaração de `b` para que a soma seja inferior a 10:

C#

```
int b = 3;
```

Digite `dotnet run` novamente. Como a resposta é inferior a 10, nada é impresso. A condição que você está testando é falsa. Não há qualquer código para execução, porque você escreveu apenas uma das ramificações possíveis para uma instrução `if`: a ramificação verdadeira.

### Dica

À medida que explora C# (ou qualquer linguagem de programação), você cometerá erros ao escrever o código. O compilador encontrará e reportará esses erros. Verifique atentamente a saída do erro e o código que gerou o erro. O erro do compilador geralmente pode ajudá-lo a localizar o problema.

Este primeiro exemplo mostra o poder dos tipos `if` e Booleano. Um *Booleano* é uma variável que pode ter um dos dois valores: `true` ou `false`. C# define um tipo especial, `bool` para variáveis Booleanas. A instrução `if` verifica o valor de um `bool`. Quando o valor é `true`, a instrução após `if` é executada. Caso contrário, ela é ignorada. Esse processo de verificação de condições e execução de instruções com base nessas condições é eficiente.

## Faça if e else funcionam juntas

Para executar um código diferente nos branches `true` e `false`, crie um branch `else` que será executado quando a condição for `false`. Experimente uma ramificação `else`. Adicione as duas últimas linhas do código abaixo (você já deve ter as quatro primeiras):

C#

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

A instrução após a palavra-chave `else` é executada somente quando a condição que estiver sendo testada for `false`. A combinação de `if` e `else` com condições Booleanas fornece todos os recursos que você precisa para lidar com uma condição `true` e `false`.

### ⓘ Importante

O recuo sob as instruções `if` e `else` é para leitores humanos. A linguagem C# não considera recuos ou espaços em branco como significativos. A instrução após a palavra-chave `if` ou `else` será executada com base na condição. Todos os exemplos neste tutorial seguem uma prática comum para recuar linhas com base no fluxo de controle de instruções.

Como o recuo não é significativo, você precisa usar `{` e `}` para indicar quando você quer que mais de uma instrução faça parte do bloco executado condicionalmente. Os programadores em C# normalmente usam essas chaves em todas as cláusulas `if` e `else`. O exemplo a seguir é igual ao que você acabou de criar. Modifique o código acima para coincidir com o código a seguir:

C#

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

### 💡 Dica

No restante deste tutorial, todos os exemplos de código incluem as chaves, seguindo as práticas aceitas.

Você pode testar condições mais complicadas. Adicione o código a seguir após o que você escreveu até o momento:

C#

```
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

O símbolo `==` testa a *igualdade*. Usar `==` distingue o teste de igualdade de atribuição, que você viu em `a = 5`.

O `&&` representa "e". Isso significa que as duas condições devem ser verdadeiras para executar a instrução no branch verdadeiro. Estes exemplos também mostram que você pode ter várias instruções em cada branch condicional, desde que você coloque-as entre `{ e }`. Você também pode usar `||` para representar "ou". Adicione o código a seguir após o que você escreveu até o momento:

C#

```
if ((a + b + c > 10) || (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not equal to the second");
}
```

Modifique os valores de `a`, `b` e `c` e alterne entre `&&` e `||` para explorar. Você obterá mais compreensão de como os operadores `&&` e `||` funcionam.

Você terminou a primeira etapa. Antes de iniciar a próxima seção, vamos passar o código atual para um método separado. Isso facilita o começo do trabalho com um exemplo novo. Coloque o código existente em um método chamado `ExploreIf()`.

Chame-o pela parte superior do programa. Quando você terminar essas alterações, seu código deverá estar parecido com o seguinte:

C#

```
ExploreIf();

void ExploreIf()
{
    int a = 5;
    int b = 3;
    if (a + b > 10)
    {
        Console.WriteLine("The answer is greater than 10");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
    }

    int c = 4;
    if ((a + b + c > 10) && (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("And the first number is greater than the
second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("Or the first number is not greater than the
second");
    }
}
```

```
if ((a + b + c > 10) || (a > b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is greater than the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not greater than the second");
}
```

Comente a chamada para `ExploreIf()`. Isso tornará a saída menos congestionada enquanto você trabalha nesta seção:

C#

```
//ExploreIf();
```

O `//` inicia um **comentário** em C#. Os comentários são qualquer texto que você queira manter em seu código-fonte, mas não queria executar como código. O compilador não gera qualquer código executável a partir dos comentários.

## Use loops para repetir operações

Nesta seção, você usa **loops** repetir as instruções. Adicione esse código após a chamada para `ExploreIf`:

C#

```
int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}
```

A instrução `while` verifica uma condição e executa a instrução, ou bloco de instruções, após o `while`. Ela verifica repetidamente a condição e executa essas instruções até que a condição seja falsa.

Há outro operador novo neste exemplo. O `++` após a variável `counter` é o operador **increment**. Ele adiciona 1 ao valor de `counter` e armazena esse valor na variável

counter.

### ⓘ Importante

Verifique se a condição de loop `while` muda para false ao executar o código. Caso contrário, crie um **loop infinito**, para que seu programa nunca termine. Isso não é demonstrado neste exemplo, porque você tem que forçar o programa a encerrar usando **CTRL-C** ou outros meios.

O loop `while` testa a condição antes de executar o código seguindo `while`. O loop `do ... while` executa o código primeiro e, em seguida, verifica a condição. O loop `do while` é mostrado no código a seguir:

C#

```
int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

Esse loop `do` e o loop `while` anterior produzem a mesma saída.

## Trabalhar com o loop for

O loop `for` é usado normalmente em C#. Experimente esse código:

C#

```
for (int index = 0; index < 10; index++)
{
    Console.WriteLine($"Hello World! The index is {index}");
}
```

Ele faz o mesmo trabalho do loop `while` e do loop `do` que você já usou. A instrução `for` tem três partes que controlam o modo como ela funciona.

A primeira parte é o **inicializador for**: `int index = 0;` declara que `index` é a variável do loop, e define seu valor inicial como `0`.

A parte central é a **condição for**: `index < 10` declara que este loop `for` continuará sendo executado desde que o valor do contador seja inferior a 10.

A parte final é o iterador `for`: `index++` especifica como modificar a variável de loop depois de executar o bloco após a instrução `for`. Aqui, ela especifica que `index` deve ser incrementado com 1 sempre que o bloco `for` executado.

Experimente você mesmo. Experimente cada uma das seguintes variações:

- Altere o inicializador para iniciar em um valor diferente.
- Altere a condição para parar em um valor diferente.

Quando terminar, vamos escrever um código para usar o que você aprendeu.

Há uma outra instrução de loop que não é abordada neste tutorial: a instrução `foreach`. A instrução `foreach` repete sua instrução para cada item em uma sequência de itens. Ela é usada com mais frequência com *coleções*, portanto, será abordada no próximo tutorial.

## Loops aninhados criados

Um loop `while`, `do` ou `for` pode ser aninhado dentro de outro loop para criar uma matriz usando a combinação de cada item no loop externo com cada item no loop interno. Vamos fazer isso para criar um conjunto de pares alfanuméricos para representar linhas e colunas.

Um loop `for` pode gerar as linhas:

```
C#  
  
for (int row = 1; row < 11; row++)  
{  
    Console.WriteLine($"The row is {row}");  
}
```

Outro loop pode gerar as colunas:

```
C#  
  
for (char column = 'a'; column < 'k'; column++)  
{  
    Console.WriteLine($"The column is {column}");  
}
```

Você pode aninhar um loop dentro do outro para formar pares:

```
C#
```

```
for (int row = 1; row < 11; row++)
{
    for (char column = 'a'; column < 'k'; column++)
    {
        Console.WriteLine($"The cell is ({row}, {column})");
    }
}
```

Você pode ver que o loop externo incrementa uma vez para cada execução completa do loop interno. Inverta o aninhamento de linha e da coluna e confira as alterações. Quando terminar, coloque o código desta seção em um método chamado `ExploreLoops()`.

## Combinar branches e loops

Agora que você viu a instrução `if` e as construções de loop na linguagem C#, verifique se você pode escrever o código C# para encontrar a soma de todos os inteiros de 1 a 20 divisíveis por 3. Veja algumas dicas:

- O operador `%` retorna o restante de uma operação de divisão.
- A instrução `if` retorna a condição para ver se um número deve ser parte da soma.
- O loop `for` pode ajudar você a repetir uma série de etapas para todos os números de 1 a 20.

Tente você mesmo. Depois verifique como você fez. Você deve obter 63 como resposta. Veja uma resposta possível [exibindo o código completo no GitHub ↗](#).

Você concluiu o tutorial "branches e loops".

Continue com o tutorial [Matrizes e coleções](#) em seu próprio ambiente de desenvolvimento.

Saiba mais sobre esses conceitos nesses artigos:

- [Instruções de seleção](#)
- [Instruções de iteração](#)

# Saiba como gerenciar coletas de dados usando `List<T>` em C #

Artigo • 10/05/2023

Este tutorial de introdução fornece uma introdução à linguagem C# e os conceitos básicos da classe [List<T>](#).

## Pré-requisitos

Este tutorial espera que você tenha um computador configurado para desenvolvimento local. Consulte [Configurar seu ambiente local](#) para obter instruções de instalação e uma visão geral do desenvolvimento de aplicativos no .NET.

Se você preferir executar o código sem precisar configurar um ambiente local, consulte a [versão interativa no navegador deste tutorial](#).

## Um exemplo de lista básica

Crie um diretório chamado *list-tutorial*. Torne-o o diretório atual e execute `dotnet new console`.

### ⓘ Importante

Os modelos C# para o .NET 6 usam *instruções de nível superior*. Se você já tiver atualizado para o .NET 6, talvez seu aplicativo não corresponda ao código descrito neste artigo. Para obter mais informações, consulte o artigo sobre [Novos modelos C# geram instruções de nível superior](#)

O SDK do .NET 6 também adiciona um conjunto de diretivas `global using` para projetos que usam os seguintes SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Essas diretivas `global using` implícitas incluem os namespaces mais comuns para o tipo de projeto.

Abra *Program.cs* em seu editor favorito e substitua o código existente pelo seguinte:

C#

```
var names = new List<string> { "<name>", "Ana", "Felipe" };
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Substitua `<name>` pelo seu nome. Salve o `Program.cs`. Digite `dotnet run` na janela de console para testá-lo.

Você criou uma lista de cadeias de caracteres, adicionou três nomes a essa lista e imprimiu os nomes em MAIÚSCULAS. Você está usando conceitos que aprendeu em tutoriais anteriores para executar um loop pela lista.

O código para exibir nomes utiliza o recurso de [interpolação de cadeia de caracteres](#). Quando você precede um `string` com o caractere `$`, pode inserir o código C# na declaração da cadeia de caracteres. A cadeia de caracteres real substitui esse código C# pelo valor gerado. Neste exemplo, ela substitui o `{name.ToUpper()}` por cada nome, convertido em letras maiúsculas, pois você chamou o método `ToUpper`.

Vamos continuar explorando.

## Modificar conteúdo da lista

A coleção que você criou usa o tipo `List<T>`. Esse tipo armazena sequências de elementos. Especifique o tipo dos elementos entre os colchetes.

Um aspecto importante desse tipo `List<T>` é que ele pode aumentar ou diminuir, permitindo que você adicione ou remova elementos. Adicione este código ao final do programa:

C#

```
Console.WriteLine();
names.Add("Maria");
names.Add("Bill");
names.Remove("Ana");
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Você adicionou mais dois nomes ao final da lista. Também removeu um. Salve o arquivo e digite `dotnet run` para testá-lo.

O `List<T>` também permite fazer referência a itens individuais por índice. Coloque o índice entre os tokens `[` e `]` após o nome da lista. C# usa 0 para o primeiro índice. Adicione este código diretamente abaixo do código que você acabou de adicionar e teste-o:

```
C#
```

```
Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");
```

Você não pode acessar um índice além do fim da lista. Lembre-se de que os índices começam com 0, portanto, o maior índice válido é uma unidade a menos do que o número de itens na lista. Você pode verificar há quanto tempo a lista está usando a propriedade `Count`. Adicione o código a seguir ao final de seu programa:

```
C#
```

```
Console.WriteLine($"The list has {names.Count} people in it");
```

Salve o arquivo e digite `dotnet run` novamente para ver os resultados.

## Pesquisar e classificar listas

Nossos exemplos usam listas relativamente pequenas, mas seus aplicativos podem criar listas com muitos outros elementos, chegando, às vezes, a milhares. Para localizar elementos nessas coleções maiores, pesquise por itens diferentes na lista. O método `IndexOf` procura um item e retorna o índice do item. Se o item não estiver na lista, `IndexOf` retornará `-1`. Adicione este código à parte inferior de seu programa:

```
C#
```

```
var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

index = names.IndexOf("Not Found");
if (index == -1)
{
```

```
        Console.WriteLine($"When an item is not found, IndexOf returns  
{index}");  
    }  
    else  
    {  
        Console.WriteLine($"The name {names[index]} is at index {index}");  
    }  
}
```

Os itens em sua lista também podem ser classificados. O método [Sort](#) classifica todos os itens na lista na ordem normal (em ordem alfabética para cadeias de caracteres).

Adicione este código à parte inferior de seu programa:

C#

```
names.Sort();  
foreach (var name in names)  
{  
    Console.WriteLine($"Hello {name.ToUpper()}!");  
}
```

Salve o arquivo e digite `dotnet run` para experimentar a versão mais recente.

Antes de iniciar a próxima seção, vamos passar o código atual para um método separado. Isso facilita o começo do trabalho com um exemplo novo. Coloque todo o código que você escreveu em um novo método chamado `WorkWithStrings()`. Chame esse método na parte superior do programa. Quando você terminar, seu código deverá ter a seguinte aparência:

C#

```
WorkWithStrings();  
  
void WorkWithStrings()  
{  
    var names = new List<string> { "<name>", "Ana", "Felipe" };  
    foreach (var name in names)  
    {  
        Console.WriteLine($"Hello {name.ToUpper()}!");  
    }  
  
    Console.WriteLine();  
    names.Add("Maria");  
    names.Add("Bill");  
    names.Remove("Ana");  
    foreach (var name in names)  
    {  
        Console.WriteLine($"Hello {name.ToUpper()}!");  
    }  
}
```

```

Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");

Console.WriteLine($"The list has {names.Count} people in it");

var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

index = names.IndexOf("Not Found");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

names.Sort();
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
}

```

## Listas de outros tipos

Você usou o tipo `string` nas listas até o momento. Vamos fazer `List<T>` usar um tipo diferente. Vamos compilar um conjunto de números.

Adicione o seguinte ao seu programa depois de chamar `WorkWithStrings()`:

C#

```
var fibonacciNumbers = new List<int> {1, 1};
```

Isso cria uma lista de números inteiros e define os primeiros dois inteiros como o valor 1. Estes são os dois primeiros valores de uma *sequência Fibonacci*, uma sequência de

números. Cada número Fibonacci seguinte é encontrado considerando a soma dos dois números anteriores. Adicione este código:

C#

```
var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

fibonacciNumbers.Add(previous + previous2);

foreach (var item in fibonacciNumbers)
{
    Console.WriteLine(item);
}
```

Salve o arquivo e digite `dotnet run` para ver os resultados.

### 💡 Dica

Para se concentrar apenas nesta seção, comente o código que chama `WorkWithStrings();`. Coloque apenas dois caracteres `/` na frente da chamada, desta forma: `// WorkWithStrings();`.

## Desafio

Veja se você consegue combinar alguns dos conceitos desta lição e de lições anteriores. Expanda o que você compilou até o momento com números Fibonacci. Tente escrever o código para gerar os 20 primeiros números na sequência. (Como uma dica, o vigésimo número Fibonacci é 6765.)

## Desafio concluído

Veja um exemplo de solução [analisando o código de exemplo finalizado no GitHub ↗](#).

Com cada iteração do loop, você está pegando os últimos dois inteiros na lista, somando-os e adicionando esse valor à lista. O loop será repetido até que você tenha adicionado 20 itens à lista.

Parabéns, você concluiu o tutorial de lista. Você pode continuar com tutoriais [adicionais](#) em seu próprio ambiente de desenvolvimento.

Saiba mais sobre como trabalhar com o tipo `List` no artigo Noções básicas do .NET em [coleções](#). Você também aprenderá muitos outros tipos de coleção.

# Estrutura geral de um programa em C#

Artigo • 07/04/2023

Os programas C# podem consistir em um ou mais arquivos. Cada arquivo pode conter zero ou mais namespaces. Um namespace pode conter tipos como classes, estruturas, interfaces, enumerações e delegados, além de outros namespaces. Veja a seguir o esqueleto de um programa em C# que contém todos esses elementos.

C#

```
// A skeleton of a C# program
using System;

// Your program starts here:
Console.WriteLine("Hello world!");

namespace YourNamespace
{
    class YourClass
    {

    }

    struct YourStruct
    {

    }

    interface IYourInterface
    {

    }

    delegate int YourDelegate();

    enum YourEnum
    {

    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {

        }
    }
}
```

O exemplo anterior usa *instruções de nível superior* para o ponto de entrada do programa. Esse recurso foi adicionado no C# 9. Antes do C# 9, o ponto de entrada era um método estático chamado `Main`, como mostra o exemplo a seguir:

C#

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {

    }

    struct YourStruct
    {

    }

    interface IYourInterface
    {

    }

    delegate int YourDelegate();

    enum YourEnum
    {

    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {

        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //Your program starts here...
            Console.WriteLine("Hello world!");
        }
    }
}
```

## Seções relacionadas

Você aprenderá sobre esses elementos de programa na seção sobre [tipos](#) do guia de conceitos básicos:

- [Classes](#)
- [Estruturas](#)
- [Namespaces](#)
- [Interfaces](#)

- [Enumerações](#)
- [Representantes](#)

## Especificação da Linguagem C#

Para obter mais informações, veja [Noções básicas](#) na [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# Main() e argumentos de linha de comando

Artigo • 10/05/2023

O método `Main` é o ponto de entrada de um aplicativo C#. (Bibliotecas e serviços não exigem um método `Main` como ponto de entrada). Quando o aplicativo é iniciado, o método `Main` é o primeiro método invocado.

Pode haver apenas um ponto de entrada em um programa C#. Se tiver mais de uma classe que tenha um método `Main`, você deverá compilar seu programa com a opção do compilador `StartupObject` para especificar qual método `Main` será usado como ponto de entrada. Para obter mais informações, consulte [StartupObject \(opções de compilador do C#\)](#).

C#

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments.
        Console.WriteLine(args.Length);
    }
}
```

A partir do C# 9, você pode omitir o método `Main` e gravar instruções C# como se estivessem no método `Main`, como no exemplo a seguir:

C#

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("Hello");
builder.AppendLine("World!");

Console.WriteLine(builder.ToString());
```

Para obter informações sobre como escrever o código do aplicativo com um método de ponto de entrada implícito, consulte [Instruções de nível superior](#).

# Visão geral

- O método `Main` é o ponto de entrada de um programa executável; é onde o controle do programa começa e termina.
- `Main` é declarado dentro de uma classe ou struct. `Main` deve ser `static` e não precisa ser `public`. (No exemplo anterior, ele recebe o acesso padrão de `private`). A classe ou o struct de integração não é necessário para ser estático.
- `Main` pode ter o tipo de retorno `void`, `int`, `Task` ou `Task<int>`.
- Se e somente se `Main` retornar um `Task` ou `Task<int>`, a declaração de `Main` pode incluir o modificador `async`. Isso exclui especificamente um método `async void Main`.
- O método `Main` pode ser declarado com ou sem um parâmetro `string[]` que contém os argumentos de linha de comando. Ao usar o Visual Studio para criar aplicativos do Windows, você pode adicionar o parâmetro manualmente ou usar o método `GetCommandLineArgs()` para obter os argumentos de linha de comando. Os parâmetros são lidos como argumentos de linha de comando indexados por zero. Ao contrário do C e C++, o nome do programa não é tratado como o primeiro argumento de linha de comando na matriz `args`, mas é o primeiro elemento do método `GetCommandLineArgs()`.

A lista a seguir mostra as assinaturas de `Main` válidas:

C#

```
public static void Main() { }
public static int Main() { }
public static void Main(string[] args) { }
public static int Main(string[] args) { }
public static async Task Main() { }
public static async Task<int> Main() { }
public static async Task Main(string[] args) { }
public static async Task<int> Main(string[] args) { }
```

Todos os exemplos anteriores usam o modificador do acessador `public`. Isso é típico, mas não é necessário.

A adição dos tipos de retorno `async`, `Task` e `Task<int>` simplifica o código do programa quando os aplicativos do console precisam iniciar e realizar operações assíncronas `await` no `Main`.

## Valores de retorno de `Main()`

Você pode retornar um `int` do método `Main` ao definir o método de uma das seguintes maneiras:

Código do método <code>Main</code>	Assinatura <code>Main</code>
Nenhum uso de <code>args</code> ou <code>await</code>	<code>static int Main()</code>
Usa <code>args</code> , nenhum uso de <code>await</code>	<code>static int Main(string[] args)</code>
Nenhum uso de <code>args</code> , usa <code>await</code>	<code>static async Task&lt;int&gt; Main()</code>
Usa <code>args</code> e <code>await</code>	<code>static async Task&lt;int&gt; Main(string[] args)</code>

Se o valor retornado de `Main` não for usado, o retorno de `void` ou `Task` permite um código um pouco mais simples.

Código do método <code>Main</code>	Assinatura <code>Main</code>
Nenhum uso de <code>args</code> ou <code>await</code>	<code>static void Main()</code>
Usa <code>args</code> , nenhum uso de <code>await</code>	<code>static void Main(string[] args)</code>
Nenhum uso de <code>args</code> , usa <code>await</code>	<code>static async Task Main()</code>
Usa <code>args</code> e <code>await</code>	<code>static async Task Main(string[] args)</code>

No entanto, o retorno de `int` ou `Task<int>` habilita o programa a comunicar informações de status para outros programas ou scripts, que invocam o arquivo executável.

O exemplo a seguir mostra como o código de saída para o processo pode ser acessado.

Este exemplo usa ferramentas de linha de comando do [.NET Core](#). Se você não estiver familiarizado com as ferramentas de linha de comando do .NET Core, poderá aprender sobre elas neste [artigo de introdução](#).

Crie um novo aplicativo ao executar `dotnet new console`. Modifique o método `Main` em `Program.cs` da seguinte maneira:

```
C#  
  
// Save this program as MainRetValTest.cs.  
class MainRetValTest  
{  
    static int Main()  
    {  
        //...  
        return 0;  
    }  
}
```

```
    }  
}
```

Quando um programa é executado no Windows, qualquer valor retornado da função `Main` é armazenado em uma variável de ambiente. Essa variável de ambiente pode ser recuperada usando `ERRORLEVEL` de um arquivo em lotes ou `$LastExitCode` do PowerShell.

Você pode criar o aplicativo usando o comando `dotnet build` da [CLI do dotnet](#).

Em seguida, crie um script do PowerShell para executar o aplicativo e exibir o resultado. Cole o código a seguir em um arquivo de texto e salve-o como `test.ps1` na pasta que contém o projeto. Execute o script do PowerShell ao digitar `test.ps1` no prompt do PowerShell.

Como o código retorna zero, o arquivo em lotes relatará êxito. No entanto, se você alterar o `MainReturnValTest.cs` para retornar um valor diferente de zero e recompilar o programa, a execução subsequente do script do PowerShell reportará falha.

PowerShell

```
dotnet run  
if ($LastExitCode -eq 0) {  
    Write-Host "Execution succeeded"  
} else  
{  
    Write-Host "Execution Failed"  
}  
Write-Host "Return value = " $LastExitCode
```

Saída

```
Execution succeeded  
Return value = 0
```

## Valores retornados de Async Main

Quando você declara um valor retornado `async` para `Main`, o compilador gera o código clichê para chamar métodos assíncronos em `Main`. Se você não especificar a palavra-chave `async`, precisará escrever esse código por conta própria, conforme mostrado no exemplo a seguir. O código no exemplo garante que o programa seja executado até que a operação assíncrona seja concluída:

C#

```
public static void Main()
{
    AsyncConsoleWork().GetAwaiter().GetResult();
}

private static async Task<int> AsyncConsoleWork()
{
    // Main body here
    return 0;
}
```

Este código clichê pode ser substituído por:

C#

```
static async Task<int> Main(string[] args)
{
    return await AsyncConsoleWork();
}
```

A vantagem de declarar `Main` como `async` é que o compilador sempre gera o código correto.

Quando o ponto de entrada do aplicativo retorna um `Task` ou `Task<int>`, o compilador gera um novo ponto de entrada que chama o método de ponto de entrada declarado no código do aplicativo. Supondo que esse ponto de entrada é chamado `$GeneratedMain`, o compilador gera o código a seguir para esses pontos de entrada:

- `static Task Main()` resulta no compilador emitindo o equivalente a `private static void $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task Main(string[])` resulta no compilador emitindo o equivalente a `private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`
- `static Task<int> Main()` resulta no compilador emitindo o equivalente a `private static int $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task<int> Main(string[])` resulta no compilador emitindo o equivalente a `private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`

### ① Observação

Se os exemplos usassem o modificador `async` no método `Main`, o compilador geraria o mesmo código.

## Argumentos de linha de comando

Você pode enviar argumentos para o método `Main` definindo o método de uma das seguintes maneiras:

Código do método <code>Main</code>	Assinatura <code>Main</code>
Nenhum valor retornado, nenhum uso de <code>await</code>	<code>static void Main(string[] args)</code>
Valor retornado, nenhum uso de <code>await</code>	<code>static int Main(string[] args)</code>
Nenhum valor retornado, usa <code>await</code>	<code>static async Task Main(string[] args)</code>
Valor retornado, usa <code>await</code>	<code>static async Task&lt;int&gt; Main(string[] args)</code>

Se os argumentos não forem usados, você poderá omitir `args` a assinatura do método para um código ligeiramente mais simples:

Código do método <code>Main</code>	Assinatura <code>Main</code>
Nenhum valor retornado, nenhum uso de <code>await</code>	<code>static void Main()</code>
Valor retornado, nenhum uso de <code>await</code>	<code>static int Main()</code>
Nenhum valor retornado, usa <code>await</code>	<code>static async Task Main()</code>
Valor retornado, usa <code>await</code>	<code>static async Task&lt;int&gt; Main()</code>

### Observação

Você também pode usar `Environment.CommandLine` ou `Environment.GetCommandLineArgs` para acessar os argumentos de linha de comando de qualquer ponto em um console ou um aplicativo do Windows Forms. Para habilitar os argumentos de linha de comando na assinatura do método `Main` em um aplicativo do Windows Forms, você deve modificar manualmente a assinatura de `Main`. O código gerado pelo designer do Windows Forms cria um `Main` sem um parâmetro de entrada.

O parâmetro do método `Main` é uma matriz `String` que representa os argumentos de linha de comando. Geralmente você determina se os argumentos existem testando a

propriedade `Length`, por exemplo:

```
C#
```

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

### 💡 Dica

A matriz `args` não pode ser nula. Portanto, é seguro acessar a propriedade `Length` sem verificação de nulos.

Você também pode converter os argumentos de cadeia de caracteres em tipos numéricos, usando a classe `Convert` ou o método `Parse`. Por exemplo, a instrução a seguir converte o `string` em um número `long` usando o método `Parse`:

```
C#
```

```
long num = Int64.Parse(args[0]);
```

Também é possível usar o tipo `long` de C#, que funciona como alias de `Int64`:

```
C#
```

```
long num = long.Parse(args[0]);
```

Você também pode usar o método da classe `Convert`, o `ToInt64`, para fazer a mesma coisa:

```
C#
```

```
long num = Convert.ToInt64(s);
```

Para obter mais informações, consulte [Parse](#) e [Convert](#).

O exemplo a seguir mostra como usar argumentos de linha de comando em um aplicativo de console. O aplicativo recebe um argumento em tempo de execução, converte o argumento em um número inteiro e calcula o fatorial do número. Se nenhum

argumento fornecido, o aplicativo emitirá uma mensagem que explica o uso correto do programa.

Para compilar e executar o aplicativo em um prompt de comando, siga estas etapas:

1. Cole o código a seguir em qualquer editor de texto e, em seguida, salve o arquivo como um arquivo de texto com o nome *Factorial.cs*.

```
C#  
  
public class Functions  
{  
    public static long Factorial(int n)  
    {  
        // Test for invalid input.  
        if ((n < 0) || (n > 20))  
        {  
            return -1;  
        }  
  
        // Calculate the factorial iteratively rather than recursively.  
        long tempResult = 1;  
        for (int i = 1; i <= n; i++)  
        {  
            tempResult *= i;  
        }  
        return tempResult;  
    }  
}  
  
class MainClass  
{  
    static int Main(string[] args)  
    {  
        // Test if input arguments were supplied.  
        if (args.Length == 0)  
        {  
            Console.WriteLine("Please enter a numeric argument.");  
            Console.WriteLine("Usage: Factorial <num>");  
            return 1;  
        }  
  
        // Try to convert the input arguments to numbers. This will  
        throw  
        // an exception if the argument is not a number.  
        // num = int.Parse(args[0]);  
        int num;  
        bool test = int.TryParse(args[0], out num);  
        if (!test)  
        {  
            Console.WriteLine("Please enter a numeric argument.");  
            Console.WriteLine("Usage: Factorial <num>");  
            return 1;  
        }  
    }  
}
```

```
        }

        // Calculate factorial.
        long result = Functions.Factorial(num);

        // Print result.
        if (result == -1)
            Console.WriteLine("Input must be >= 0 and <= 20.");
        else
            Console.WriteLine($"The Factorial of {num} is {result}.");

        return 0;
    }
}

// If 3 is entered on command line, the
// output reads: The factorial of 3 is 6.
```

2. Na tela **Início** ou no menu **Iniciar**, abra uma janela **Prompt de Comando do Desenvolvedor** do Visual Studio e, em seguida, navegue até a pasta que contém o arquivo que você acabou de criar.
3. Digite o seguinte comando para compilar o aplicativo.

```
dotnet build
```

Se seu aplicativo não tiver erros de compilação, um arquivo executável chamado *Factorial.exe* será criado.

4. Digite o seguinte comando para calcular o fatorial de 3:

```
dotnet run -- 3
```

5. O comando produz esta saída: `The factorial of 3 is 6.`

#### ⓘ Observação

Ao executar um aplicativo no Visual Studio, você pode especificar argumentos de linha de comando na [Página de depuração](#), [Designer de Projeto](#).

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- System.Environment
- Como exibir argumentos de linha de comando

# Instruções de nível superior – Programas sem métodos Main

Artigo • 07/04/2023

Do C# 9 em diante, você não precisa incluir explicitamente um método `Main` em projetos de aplicativo de console. Em vez disso, você pode usar o recurso de *instruções de nível superior* para minimizar o volume de código que precisa escrever. Nesse caso, o compilador gera uma classe e um ponto de entrada do método `Main` para o aplicativo.

Aqui está um arquivo `Program.cs` que corresponde a um programa C# completo no C# 10:

```
C#  
  
Console.WriteLine("Hello World!");
```

Instruções de nível superior permitem que você escreva programas simples para utilitários pequenos, como o Azure Functions e o GitHub Actions. Elas também simplificam para novos programadores C# começar a aprender e escrever código.

As seções a seguir explicam as regras sobre o que você pode ou não fazer com instruções de nível superior.

## Apenas um arquivo de nível superior

Cada aplicativo deve ter apenas um ponto de entrada. Cada projeto pode ter apenas um arquivo com instruções de nível superior. Colocar instruções de nível superior em mais de um arquivo de um projeto resulta no seguinte erro do compilador:

CS8802 Somente uma unidade de compilação pode conter instruções de nível superior.

Os projetos podem ter um número indefinido de arquivos de código-fonte adicionais que não têm instruções de nível superior.

## Nenhum outro ponto de entrada

Você pode escrever um método `Main` explicitamente, mas ele não pode funcionar como um ponto de entrada. O compilador emite o seguinte aviso:

CS7022 O ponto de entrada do programa é o código global, ignorando o ponto de entrada "Main()".

Em um projeto com instruções de nível superior, você não pode usar a opção do compilador `-main` para selecionar o ponto de entrada, mesmo que o projeto tenha um ou mais métodos `Main`.

## using diretivas

Se você incluir o uso de diretivas, elas deverão vir primeiro no arquivo, como neste exemplo:

```
C#  
  
using System.Text;  
  
StringBuilder builder = new();  
builder.AppendLine("Hello");  
builder.AppendLine("World!");  
  
Console.WriteLine(builder.ToString());
```

## Namespace global

As instruções de nível superior estão implicitamente no namespace global.

## Namespaces e definições de tipo

Um arquivo com instruções de nível superior também pode conter namespaces e definições de tipo, mas devem vir após as instruções de nível superior. Por exemplo:

```
C#  
  
MyClass.TestMethod();  
MyNamespace.MyClass.MyMethod();  
  
public class MyClass  
{  
    public static void TestMethod()  
    {  
        Console.WriteLine("Hello World!");  
    }  
}
```

```
}
```

```
namespace MyNamespace
```

```
{
```

```
    class MyClass
```

```
    {
```

```
        public static void MyMethod()
```

```
        {
```

```
            Console.WriteLine("Hello World from
```

```
MyNamespace.MyClass.MyMethod!");
```

```
        }
```

```
    }
```

```
}
```

## args

Instruções de nível superior podem fazer referência à variável `args` para acessar quaisquer argumentos de linha de comando que foram inseridos. A variável `args` nunca será nula, mas o `Length` dela será zero se nenhum argumento de linha de comando tiver sido fornecido. Por exemplo:

C#

```
if (args.Length > 0)
```

```
{
```

```
    foreach (var arg in args)
```

```
    {
```

```
        Console.WriteLine($"Argument={arg}");
```

```
    }
```

```
}
```

```
else
```

```
{
```

```
    Console.WriteLine("No arguments");
```

```
}
```

## await

Você pode chamar um método assíncrono usando `await`. Por exemplo:

C#

```
Console.Write("Hello ");
```

```
await Task.Delay(5000);
```

```
Console.WriteLine("World!");
```

# Código de saída do processo

Para retornar um valor `int` quando o aplicativo terminar, use a instrução `return` como faria em um método `Main` que retorna um `int`. Por exemplo:

C#

```
string? s = Console.ReadLine();

int returnValue = int.Parse(s ?? "-1");
return returnValue;
```

## Método de ponto de entrada implícito

O compilador gera um método para servir como o ponto de entrada do programa para um projeto com instruções de nível superior. O nome desse método, na verdade, não é `Main`, é um detalhe de implementação que seu código não pode referenciar diretamente. A assinatura do método depende se as instruções de nível superior contêm a palavra-chave `await` ou a instrução `return`. A tabela a seguir mostra como seria a assinatura do método, usando o nome de método `Main` na tabela para fins de conveniência.

O código de nível superior contém	Assinatura <code>Main</code> implícita
<code>await</code> e <code>return</code>	<code>static async Task&lt;int&gt; Main(string[] args)</code>
<code>await</code>	<code>static async Task Main(string[] args)</code>
<code>return</code>	<code>static int Main(string[] args)</code>
Não <code>await</code> nem <code>return</code>	<code>static void Main(string[] args)</code>

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

[Especificação de recurso – Instruções de nível superior](#)

# O sistema do tipo C#

Artigo • 15/02/2023

C# é uma linguagem fortemente tipada. Todas as variáveis e constantes têm um tipo, assim como cada expressão que é avaliada como um valor. Cada declaração de método especifica um nome, o tipo e a variante (valor, referência ou saída) de cada parâmetro de entrada e do valor retornado. A biblioteca de classes do .NET define tipos numéricos internos e tipos complexos que representam uma grande variedade de constructos. Isso inclui o sistema de arquivos, conexões de rede, coleções e matrizes de objetos e datas. Um programa em C# típico usa tipos da biblioteca de classes e tipos definidos pelo usuário que modelam os conceitos que são específicos para o domínio do problema do programa.

As informações armazenadas em um tipo podem incluir os seguintes itens:

- O espaço de armazenamento que uma variável do tipo requer.
- Os valores mínimo e máximo que ele pode representar.
- Os membros (métodos, campos, eventos e etc.) que ele contém.
- O tipo base do qual ele herda.
- A interface implementada.
- Os tipos de operações que são permitidos.

O compilador usa as informações de tipo para garantir que todas as operações que são realizadas em seu código sejam *fortemente tipadas*. Por exemplo, se você declarar uma variável do tipo `int`, o compilador permitirá que você use a variável nas operações de adição e subtração. Se você tentar executar as mesmas operações em uma variável do tipo `bool`, o compilador gerará um erro, como mostrado no exemplo a seguir:

```
C#  
  
int a = 5;  
int b = a + 2; //OK  
  
bool test = true;  
  
// Error. Operator '+' cannot be applied to operands of type 'int' and  
'bool'.  
int c = a + test;
```

ⓘ Observação

Desenvolvedores de C e C++, observem que, em C#, `bool` não é conversível em `int`.

O compilador insere as informações de tipo no arquivo executável como metadados. O CLR (Common Language Runtime) usa esses metadados em tempo de execução para assegurar mais segurança de tipos ao alocar e recuperar a memória.

## Especificando tipos em declarações de variável

Quando declara uma variável ou constante em um programa, você deve especificar seu tipo ou usar a palavra-chave `var` para permitir que o compilador infira o tipo. O exemplo a seguir mostra algumas declarações de variáveis que usam tipos numéricos internos e tipos complexos definidos pelo usuário:

C#

```
// Declaration only:  
float temperature;  
string name;  
MyClass myClass;  
  
// Declaration with initializers (four examples):  
char firstLetter = 'C';  
var limit = 3;  
int[] source = { 0, 1, 2, 3, 4, 5 };  
var query = from item in source  
            where item <= limit  
            select item;
```

Os tipos de parâmetros de método e valores de retorno são especificados na declaração do método. A assinatura a seguir mostra um método que requer um `int` como um argumento de entrada e retorna uma cadeia de caracteres:

C#

```
public string GetName(int ID)  
{  
    if (ID < names.Length)  
        return names[ID];  
    else  
        return String.Empty;  
}  
private string[] names = { "Spencer", "Sally", "Doug" };
```

Depois de declarar uma variável, não será possível reenviá-la com um novo tipo, nem atribuir um valor incompatível com seu tipo declarado. Por exemplo, você não pode declarar um `int` e, em seguida, atribuir a ele um valor booleano de `true`. No entanto, os valores podem ser convertidos em outros tipos, por exemplo, quando são passados como argumentos de método ou atribuídos a novas variáveis. Uma *conversão de tipo* que não causa a perda de dados é executada automaticamente pelo compilador. Uma conversão que pode causar perda de dados requer um *cast* no código-fonte.

Para obter mais informações, consulte [Conversões Cast e Conversões de Tipo](#).

## Tipos internos

O C# fornece um conjunto padrão de tipos internos. Eles representam números inteiros, valores de ponto flutuante, expressões booleanas, caracteres de texto, valores decimais e outros tipos de dados. Também há tipos `string` e `object` internos. Esses tipos estão disponíveis para uso em qualquer programa em C#. Para obter a lista completa tipos internos, consulte [Tipos internos](#).

## Tipos personalizados

Você usa os constructos `struct`, `class`, `interface`, `enum` e `record` para criar seus próprios tipos personalizados. A biblioteca de classes do .NET em si é uma coleção de tipos personalizados que você pode usar em seus próprios aplicativos. Por padrão, os tipos usados com mais frequência na biblioteca de classes estão disponíveis em qualquer programa em C#. Outros ficam disponíveis somente quando você adiciona explicitamente uma referência de projeto ao assembly que os define. Depois que o compilador tiver uma referência ao assembly, você pode declarar variáveis (e constantes) dos tipos declarados nesse assembly no código-fonte. Para saber mais, confira [Biblioteca de classes do .NET](#).

## O Common Type System

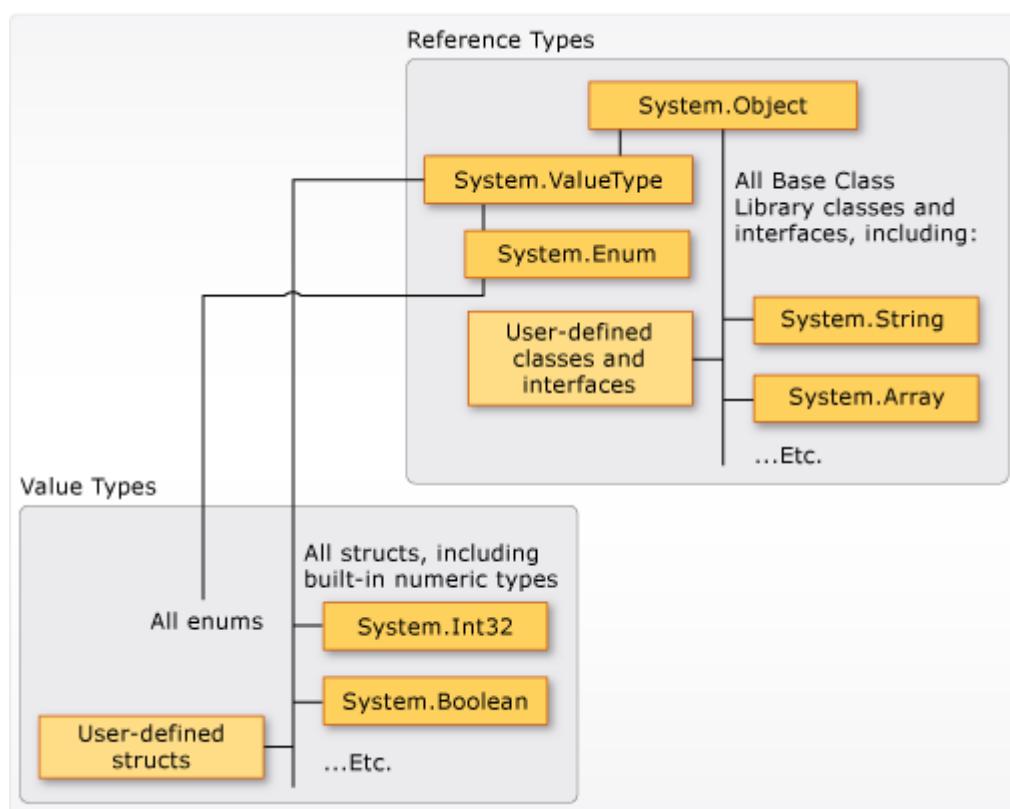
É importante entender os dois pontos fundamentais sobre o sistema de tipos do .NET:

- Ele dá suporte ao conceito de herança. Os tipos podem derivar de outros tipos, chamados *tipos base*. O tipo derivado herda (com algumas restrições) os métodos, as propriedades e outros membros do tipo base. O tipo base, por sua vez, pode derivar de algum outro tipo, nesse caso, o tipo derivado herda os membros de ambos os tipos base na sua hierarquia de herança. Todos os tipos, incluindo tipos numéricos internos, como o `System.Int32` (palavra-chave do C#: `int`), derivam, em

última análise, de um único tipo base, que é o [System.Object](#) (palavra-chave do C#: `object`). Essa hierarquia unificada de tipos é chamada de CTS ([Common Type System](#)). Para obter mais informações sobre herança em C#, consulte [Herança](#).

- Cada tipo no CTS é definido como um *tipo de valor* ou um *tipo de referência*. Esses tipos incluem todos os tipos personalizados na biblioteca de classes do .NET, além de tipos personalizados definidos pelo usuário. Os tipos que você define usando a palavra-chave `struct` são tipos de valor. Todos os tipos numéricos internos são `structs`. Os tipos que você define usando a palavra-chave `class` ou `record` são tipos de referência. Os tipos de referência e os tipos de valor têm diferentes regras de tempo de compilação e comportamento de tempo de execução diferente.

A ilustração a seguir mostra a relação entre tipos de referência e tipos de valor no CTS.



### ⓘ Observação

Você pode ver que os tipos mais usados normalmente são todos organizados no namespace `System`. No entanto, o namespace no qual um tipo está contido não tem relação com a possibilidade de ele ser um tipo de valor ou um tipo de referência.

Classes e structs são duas das construções básicas do Common Type System no .NET. O C# 9 adiciona registros, que são um tipo de classe. Cada um é, essencialmente, uma estrutura de dados que encapsula um conjunto de dados e os comportamentos que são uma unidade lógica. Os dados e comportamentos são os *membros* da classe, `struct` ou

registro. Os membros incluem seus métodos, propriedades, eventos e assim por diante, conforme listado posteriormente neste artigo.

Uma declaração de classe, struct ou registro é como um plano que é usado para criar instâncias ou objetos em tempo de execução. Se você definir uma classe, struct ou registro denominado `Person`, `Person` será o nome do tipo. Se você declarar e inicializar um `p` variável do tipo `Person`, `p` será considerado um objeto ou uma instância de `Person`. Várias instâncias do mesmo tipo `Person` podem ser criadas, e cada instância pode ter valores diferentes em suas propriedades e campos.

Uma classe é um tipo de referência. Quando um objeto do tipo é criado, a variável à qual o objeto é atribuído armazena apenas uma referência na memória. Quando a referência de objeto é atribuída a uma nova variável, a nova variável refere-se ao objeto original. As alterações feitas por meio de uma variável são refletidas na outra variável porque ambas se referem aos mesmos dados.

Um struct é um tipo de valor. Quando um struct é criado, a variável à qual o struct está atribuído contém os dados reais do struct. Quando o struct é atribuído a uma nova variável, ele é copiado. A nova variável e a variável original, portanto, contêm duas cópias separadas dos mesmos dados. As alterações feitas em uma cópia não afetam a outra cópia.

Os tipos de registro podem ser tipos de referência (`record class`) ou tipos de valor (`record struct`).

Em geral, as classes são usadas para modelar um comportamento mais complexo. As classes normalmente armazenam dados que devem ser modificados depois que um objeto de classe é criado. Structs são mais adequados para estruturas de dados pequenas. Os structs normalmente armazenam dados que não se destinam a serem modificados após a criação do struct. Os tipos de registro são estruturas de dados com membros sintetizados de compilador adicionais. Os registros normalmente armazenam dados que não se destinam a serem modificados após a criação do objeto.

## Tipos de valor

Os tipos de valor derivam de [System.ValueType](#), que deriva de [System.Object](#). Os tipos que derivam de [System.ValueType](#) apresentam um comportamento especial no CLR. As variáveis de tipo de valor contêm diretamente seus valores. A memória de um struct é embutida em qualquer contexto em que a variável seja declarada. Não há nenhuma alocação de heap separada ou sobrecarga de coleta de lixo para variáveis do tipo de valor. É possível declarar tipos `record struct` que são tipos de valor e incluir os membros sintetizados para [registros](#).

Há duas categorias de tipos de valor: `struct` e `enum`.

Os tipos numéricos internos são structs e têm campos e métodos que você pode acessar:

C#

```
// constant field on type byte.  
byte b = byte.MaxValue;
```

Mas você declara e atribui valores a eles como se fossem tipos de não agregação simples:

C#

```
byte num = 0xA;  
int i = 5;  
char c = 'Z';
```

Os tipos de valor são *selados*. Não é possível derivar um tipo de qualquer tipo de valor, por exemplo `System.Int32`. Você não pode definir um struct a ser herdado de qualquer classe definida pelo usuário ou struct, porque um struct só pode ser herdado de `System.ValueType`. No entanto, um struct pode implementar uma ou mais interfaces. É possível converter um tipo struct em qualquer tipo de interface que ele implementa. Essa conversão faz com que uma operação de *conversão boxing* encapsule o struct dentro de um objeto de tipo de referência no heap gerenciado. As operações de conversão boxing ocorrem quando você passa um tipo de valor para um método que usa um `System.Object` ou qualquer tipo de interface como parâmetro de entrada. Para obter mais informações, consulte [Conversões boxing e unboxing](#).

Você usa a palavra-chave `struct` para criar seus próprios tipos de valor personalizados. Normalmente, um struct é usado como um contêiner para um pequeno conjunto de variáveis relacionadas, conforme mostrado no exemplo a seguir:

C#

```
public struct Coords  
{  
    public int x, y;  
  
    public Coords(int p1, int p2)  
    {  
        x = p1;  
        y = p2;  
    }  
}
```

Para obter mais informações sobre structs, consulte [Tipos de estrutura](#). Para saber mais sobre os tipos de valor, confira [Tipos de valor](#).

A outra categoria de tipos de valor é `enum`. Uma enum define um conjunto de constantes inteiros nomeadas. Por exemplo, a enumeração `System.IO.FileMode` na biblioteca de classes do .NET contém um conjunto de números inteiros constantes nomeados que especificam como um arquivo deve ser aberto. Ela é definida conforme mostrado no exemplo abaixo:

C#

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

A constante `System.IO.FileMode.Create` tem um valor de 2. No entanto, o nome é muito mais significativo para a leitura do código-fonte por humanos e, por esse motivo, é melhor usar enumerações em vez de números literais constantes. Para obter mais informações, consulte [System.IO.FileMode](#).

Todas as enumerações herdam de `System.Enum`, que herda de `System.ValueType`. Todas as regras que se aplicam a structs também se aplicam a enums. Para obter mais informações sobre enums, consulte [Tipos de enumeração](#).

## Tipos de referência

Um tipo que é definido como `class`, `record`, `delegate`, matriz ou `interface` é um [reference type](#).

Ao declarar uma variável de um [reference type](#), ele contém o valor `null` até que você o atribua com uma instância desse tipo ou crie uma usando o operador `new`. A criação e a atribuição de uma classe são demonstradas no exemplo a seguir:

C#

```
MyClass myClass = new MyClass();
MyClass myClass2 = myClass;
```

Não é possível criar uma instância direta de [interface](#) usando o operador `new`. Em vez disso, crie e atribua uma instância de uma classe que implemente a interface. Considere o seguinte exemplo:

C#

```
MyClass myClass = new MyClass();  
  
// Declare and assign using an existing value.  
IMyInterface myInterface = myClass;  
  
// Or create and assign a value in a single statement.  
IMyInterface myInterface2 = new MyClass();
```

Quando o objeto é criado, a memória é alocada no heap gerenciado. A variável contém apenas uma referência ao local do objeto. Os tipos no heap gerenciado exigem sobrecarga quando são alocados e recuperados. A *coleta de lixo* é a funcionalidade de gerenciamento automático de memória do CLR, que executa a recuperação. No entanto, a coleta de lixo também é altamente otimizada e, na maioria dos cenários, não cria um problema de desempenho. Para obter mais informações sobre a coleta de lixo, consulte [Gerenciamento automático de memória](#).

Todas as matrizes são tipos de referência, mesmo se seus elementos forem tipos de valor. As matrizes derivam implicitamente da classe [System.Array](#). Você declara e usa as matrizes com a sintaxe simplificada fornecida pelo C#, conforme mostrado no exemplo a seguir:

C#

```
// Declare and initialize an array of integers.  
int[] nums = { 1, 2, 3, 4, 5 };  
  
// Access an instance property of System.Array.  
int len = nums.Length;
```

Os tipos de referência dão suporte completo à herança. Quando você cria uma classe, é possível herdar de qualquer outra interface ou classe que não esteja definida como [selada](#). Outras classes podem herdar de sua classe e substituir seus métodos virtuais. Para obter mais informações sobre como criar suas próprias classes, consulte [Classes, structs e registros](#). Para obter mais informações sobre herança e métodos virtuais, consulte [Herança](#).

## Tipos de valores literais

No C#, valores literais recebem um tipo do compilador. Você pode especificar como um literal numérico deve ser digitado anexando uma letra ao final do número. Por exemplo, para especificar que o valor `4.56` deve ser tratado como um `float`, acrescente um "f" ou "F" após o número: `4.56f`. Se nenhuma letra for anexada, o compilador inferirá um tipo para o literal. Para obter mais informações sobre quais tipos podem ser especificados com sufixos de letra, consulte [Tipos numéricos integrais](#) e [Tipos numéricos de ponto flutuante](#).

Como os literais são tipados e todos os tipos derivam basicamente de `System.Object`, você pode escrever e compilar o código como o seguinte:

C#

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

## Tipos genéricos

Um tipo pode ser declarado com um ou mais *parâmetros de tipo* que servem como um espaço reservado para o tipo real (o *tipo concreto*). O código do cliente fornece o tipo concreto quando ele cria uma instância do tipo. Esses tipos são chamados de *tipos genéricos*. Por exemplo, o tipo do .NET `System.Collections.Generic.List<T>` tem um parâmetro de tipo que, por convenção, recebe o nome `T`. Ao criar uma instância do tipo, você pode especificar o tipo dos objetos que a lista conterá, por exemplo, `string`:

C#

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

O uso do parâmetro de tipo possibilita a reutilização da mesma classe para conter qualquer tipo de elemento sem precisar converter cada elemento em [objeto](#). As classes de coleção genéricas são chamadas de *coleções fortemente tipadas* porque o compilador sabe o tipo específico dos elementos da coleção e pode gerar um erro em tempo de compilação se, por exemplo, você tentar adicionar um inteiro ao objeto `stringList` no exemplo anterior. Para obter mais informações, consulte [Genéricos](#).

# Tipos implícitos, tipos anônimos e tipos que permitem valor nulo

Você pode digitar implicitamente uma variável local (mas não os membros de classe) usando a palavra-chave `var`. A variável ainda recebe um tipo em tempo de compilação, mas o tipo é fornecido pelo compilador. Para obter mais informações, consulte [Variáveis locais de tipo implícito](#).

Pode ser inconveniente criar um tipo nomeado para conjuntos simples de valores relacionados que você não pretende armazenar ou transmitir fora dos limites de método. Você pode criar *tipos anônimos* para essa finalidade. Para obter mais informações, consulte [Tipos Anônimos](#).

Os tipos comuns de valor não podem ter um valor `null`. No entanto, você pode criar *tipos de valor anulável* acrescentando uma `?`  após o tipo. Por exemplo, `int?` é um tipo `int` que também pode ter o valor `null`. Os tipos que permitem valor nulo são instâncias do tipo struct genérico `System.Nullable<T>`. Os tipos que permitem valor nulo são especialmente úteis quando você está passando dados entre bancos de dados nos quais os valores numéricos podem ser `null`. Para obter mais informações, consulte [Tipos que permitem valor nulo](#).

## Tipo de tempo de compilação e tipo de tempo de execução

Uma variável pode ter diferentes tipos de tempo de compilação e tempo de execução. O *tipo de tempo de compilação* é o tipo declarado ou inferido da variável no código-fonte. O *tipo de tempo de execução* é o tipo da instância referenciada por essa variável. Geralmente, esses dois tipos são iguais, como no exemplo a seguir:

C#

```
string message = "This is a string of characters";
```

Em outros casos, o tipo de tempo de compilação é diferente, conforme mostrado nos dois exemplos a seguir:

C#

```
object anotherMessage = "This is another string of characters";
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

Em ambos os exemplos acima, o tipo de tempo de execução é um `string`. O tipo de tempo de compilação é `object` na primeira linha e `IEnumerable<char>` na segunda.

Se os dois tipos forem diferentes para uma variável, é importante entender quando o tipo de tempo de compilação e o tipo de tempo de execução se aplicam. O tipo de tempo de compilação determina todas as ações executadas pelo compilador. Essas ações do compilador incluem resolução de chamada de método, resolução de sobrecarga e conversões implícitas e explícitas disponíveis. O tipo de tempo de execução determina todas as ações que são resolvidas em tempo de execução. Essas ações de tempo de execução incluem a expedição de chamadas de método virtual, avaliação das expressões `is` e `switch` e outras APIs de teste de tipo. Para entender melhor como seu código interage com tipos, reconheça qual ação se aplica a qual tipo.

## Seções relacionadas

Para obter mais informações, consulte os seguintes artigos:

- [Tipos internos](#)
- [Tipos de valor](#)
- [Tipos de referência](#)

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# Declarar namespaces para organizar tipos

Artigo • 15/07/2023

Os namespaces são usados intensamente em programações de C# de duas maneiras. Em primeiro lugar, o .NET usa namespaces para organizar suas muitas classes, da seguinte maneira:

C#

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

[System](#) é um namespace e [Console](#) é uma classe nesse namespace. A palavra-chave `using` pode ser usada para que o nome completo não seja necessário, como no exemplo a seguir:

C#

```
using System;
```

C#

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

Para saber mais, confira [Diretiva using](#).

## ⓘ Importante

Os modelos C# para o .NET 6 usam *instruções de nível superior*. Se você já tiver atualizado para o .NET 6, talvez seu aplicativo não corresponda ao código descrito neste artigo. Para obter mais informações, consulte o artigo sobre [Novos modelos C# geram instruções de nível superior](#)

O SDK do .NET 6 também adiciona um conjunto de diretivas *implícitas global* `using` para projetos que usam os seguintes SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Essas diretivas implícitas `global using` incluem os namespaces mais comuns para o tipo de projeto.

Para obter mais informações, consulte o artigo sobre [Diretivas de uso implícito](#)

Em segundo lugar, declarar seus próprios namespaces pode ajudar a controlar o escopo dos nomes de classe e de método em projetos de programação maiores. Use a palavra-chave `namespace` para declarar um namespace, como no exemplo a seguir:

C#

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
                "SampleMethod inside SampleNamespace");
        }
    }
}
```

O nome do namespace deve ser um [nome do identificador](#) válido em C#.

A partir do C# 10, você pode declarar um namespace para todos os tipos definidos nesse arquivo, como mostra o exemplo a seguir:

C#

```
namespace SampleNamespace;

class AnotherSampleClass
{
    public void AnotherSampleMethod()
    {
        System.Console.WriteLine(
            "SampleMethod inside SampleNamespace");
    }
}
```

A vantagem dessa nova sintaxe é que ela é mais simples, economizando espaço horizontal e chaves. Isso facilita a leitura do seu código.

## Visão geral dos namespaces

Os namespaces têm as seguintes propriedades:

- Eles organizam projetos de códigos grandes.
- Eles são delimitados com o uso do operador `.`.
- A diretiva `using` elimina a necessidade de especificar o nome do namespace para cada classe.
- O namespace `global` é o namespace "raiz": `global::System` sempre fará referência ao namespace do .NET `System`.

## Especificação da linguagem C#

Para saber mais, confira a seção [Namespaces](#) da [Especificação da linguagem C#](#).

# Introdução às classes

Artigo • 02/06/2023

## Tipos de referência

Um tipo que é definido como uma `class` é um *tipo de referência*. No tempo de execução, quando você declara uma variável de um tipo de referência, a variável contém o valor `null` até que você crie explicitamente uma instância da classe usando o operador `new` ou atribua a ela um objeto de um tipo compatível que foi criado em outro lugar, conforme mostrado no exemplo a seguir:

C#

```
//Declaring an object of type MyClass.  
MyClass mc = new MyClass();  
  
//Declaring another object of the same type, assigning it the value of the  
//first object.  
MyClass mc2 = mc;
```

Quando o objeto é criado, memória suficiente é alocada no heap gerenciado para o objeto específico, e a variável contém apenas uma referência para o local do objeto. A memória usada por um objeto é recuperada pela funcionalidade de gerenciamento automático de memória do CLR, que é conhecida como *coleta de lixo*. Para obter mais informações sobre a coleta de lixo, consulte [Gerenciamento automático de memória e coleta de lixo](#).

## Declarando classes

As classes são declaradas usando a palavra-chave `class`, seguida por um identificador exclusivo, conforme mostrado no exemplo a seguir:

C#

```
//[access modifier] - [class] - [identifier]  
public class Customer  
{  
    // Fields, properties, methods and events go here...  
}
```

Um modificador de acesso opcional precede a palavra-chave `class`. Como `public` é usado nesse caso, qualquer pessoa pode criar instâncias dessa classe. O nome da classe segue a palavra-chave `class`. O nome da classe deve ser um [nome do identificador](#) válido em C#. O restante da definição é o corpo da classe, em que o comportamento e os dados são definidos. Campos, propriedades, métodos e eventos em uma classe são coletivamente denominados de *membros de classe*.

## Criando objetos

Embora eles sejam usados algumas vezes de maneira intercambiável, uma classe e um objeto são coisas diferentes. Uma classe define um tipo de objeto, mas não é um objeto em si. Um objeto é uma entidade concreta com base em uma classe e, às vezes, é conhecido como uma instância de uma classe.

Os objetos podem ser criados usando a palavra-chave `new` seguida pelo nome da classe, dessa maneira:

```
C#
```

```
Customer object1 = new Customer();
```

Quando uma instância de uma classe é criada, uma referência ao objeto é passada de volta para o programador. No exemplo anterior, `object1` é uma referência a um objeto que é baseado em `Customer`. Esta referência refere-se ao novo objeto, mas não contém os dados de objeto. Na verdade, você pode criar uma referência de objeto sem criar um objeto:

```
C#
```

```
Customer object2;
```

Não recomendamos a criação de referências de objeto, que não faz referência a um objeto, porque tentar acessar um objeto por meio de uma referência desse tipo falhará em tempo de execução. Uma referência pode ser feita para se referir a um objeto, criando um novo objeto ou atribuindo-a a um objeto existente, como abaixo:

```
C#
```

```
Customer object3 = new Customer();
Customer object4 = object3;
```

Esse código cria duas referências de objeto que fazem referência ao mesmo objeto. Portanto, qualquer alteração no objeto feita por meio de `object3` será refletida no usos posteriores de `object4`. Como os objetos que são baseados em classes são referenciados por referência, as classes são conhecidas como tipos de referência.

## Construtores e inicialização

As seções anteriores introduziram a sintaxe para declarar um tipo de classe e criar uma instância desse tipo. Ao criar uma instância de um tipo, você deseja garantir que seus campos e propriedades sejam inicializados para valores úteis. Há várias maneiras de inicializar valores:

- Aceitar valores padrão
- Inicializadores de campo
- Parâmetros do construtor
- Inicializadores de objeto

Cada tipo .NET tem um valor padrão. Normalmente, esse valor é 0 para tipos de número e `null` para todos os tipos de referência. Você pode contar com esse valor padrão quando for razoável em seu aplicativo.

Quando o padrão .NET não é o valor certo, você pode definir um valor inicial usando um *inicializador de campo*:

C#

```
public class Container
{
    // Initialize capacity field to a default value of 10:
    private int _capacity = 10;
}
```

Você pode exigir que os chamadores forneçam um valor inicial definindo um *construtor* responsável por definir esse valor inicial:

C#

```
public class Container
{
    private int _capacity;

    public Container(int capacity) => _capacity = capacity;
}
```

A partir do C# 12, você pode definir um *construtor primário* como parte da declaração de classe:

```
C#  
  
public class Container(int capacity)  
{  
    private int _capacity = capacity;  
}
```

Adicionar parâmetros ao nome da classe define o *construtor primário*. Esses parâmetros estão disponíveis no corpo da classe, que inclui seus membros. Você pode usá-los para inicializar campos ou em qualquer outro lugar em que eles sejam necessários.

Você também pode usar o modificador `required` em uma propriedade e permitir que os chamadores usem um *inicializador de objeto* para definir o valor inicial da propriedade:

```
C#  
  
public class Person  
{  
    public required string LastName { get; set; }  
    public required string FirstName { get; set; }  
}
```

A adição da palavra-chave `required` determina que os chamadores devem definir essas propriedades como parte de uma expressão `new`:

```
C#  
  
var p1 = new Person(); // Error! Required properties not set  
var p2 = new Person() { FirstName = "Grace", LastName = "Hopper" };
```

## Herança de classe

As classes dão suporte completo à *herança*, uma característica fundamental da programação orientada a objetos. Quando você cria uma classe, é possível herdar de qualquer outra classe que não esteja definida como `sealed`. Outras classes podem herdar de sua classe e substituir métodos virtuais de classe. Além disso, você pode implementar uma ou mais interfaces.

A herança é realizada usando uma *derivação*, o que significa que uma classe é declarada usando uma *classe base*, da qual ela herda o comportamento e os dados. Uma classe

base é especificada ao acrescentar dois-pontos e o nome de classe base depois do nome de classe derivada, dessa maneira:

C#

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

Quando uma declaração de classe inclui uma classe base, ela herda todos os membros da classe base, exceto os construtores. Para obter mais informações, consulte [Herança](#).

Uma classe no C# só pode herdar diretamente de uma classe base. No entanto, como uma classe base pode herdar de outra classe, uma classe pode herdar indiretamente várias classes base. Além disso, uma classe pode implementar diretamente uma ou mais interfaces. Para obter mais informações, consulte [Interfaces](#).

Uma classe pode ser declarada como [abstract](#). Uma classe abstrata contém métodos abstratos que têm uma definição de assinatura, mas não têm implementação. As classes abstratas não podem ser instanciadas. Elas só podem ser usadas por meio de classes derivadas que implementam os métodos abstratos. Por outro lado, uma classe [lacrada](#) não permite que outras classes sejam derivadas dela. Para obter mais informações, consulte [Classes e Membros de Classes Abstratos e Lacrados](#).

As definições de classe podem ser divididas entre arquivos de origem diferentes. Para obter mais informações, consulte [Classes parciais e métodos](#).

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# Introdução aos tipos de registro em C#

Artigo • 02/06/2023

Um [registro](#) em C# é uma [classe](#) ou um [struct](#) que fornece sintaxe e comportamento especiais para trabalhar com modelos de dados. O modificador `record` instrui o compilador a sintetizar membros que são úteis para tipos cuja função primária é armazenar dados. Esses membros incluem uma sobrecarga de [ToString\(\)](#) e membros que dão suporte à igualdade de valor.

## Quando usar registros

Considere usar um registro no lugar de uma classe ou struct nos seguintes cenários:

- Você deseja definir um modelo de dados que depende da [igualdade de valor](#).
- Você deseja definir um tipo para o qual os objetos são imutáveis.

## Igualdade de valor

Para registros, a igualdade de valor significa que duas variáveis de um tipo de registro são iguais se os tipos corresponderem e todos os valores de propriedade e de campo corresponderem. Para outros tipos de referência, como classes, igualdade significa [igualdade de referência](#). Ou seja, duas variáveis de um tipo de classe são iguais quando se referem ao mesmo objeto. Métodos e operadores que determinam a igualdade de duas instâncias de registro usam igualdade de valor.

Nem todos os modelos de dados funcionam bem com igualdade de valor. Por exemplo, o [Entity Framework Core](#) depende da igualdade de referência para garantir que ele use apenas uma instância de um tipo de entidade para o que é conceitualmente uma entidade. Por esse motivo, os tipos de registro não são apropriados para uso como tipos de entidade no Entity Framework Core.

## Imutabilidade

Um tipo imutável é aquele que impede que você altere qualquer propriedade ou valor de campo de um objeto após ser instanciado. A imutabilidade pode ser útil quando você precisa que um tipo seja thread-safe ou que um código hash permaneça o mesmo em uma tabela de hash. Os registros fornecem sintaxe concisa para criar e trabalhar com tipos imutáveis.

A imutabilidade não é apropriada para todos os cenários de dados. O [Entity Framework Core](#), por exemplo, não dá suporte à atualização com tipos de entidade imutáveis.

## Como os registros diferem das classes e dos structs

A mesma sintaxe que [declara](#) e [instancia](#) classes ou structs pode ser usada com registros. Basta substituir a palavra-chave `class` por `record`, ou usar `record struct` em vez de `struct`. Da mesma forma, as classes de registro dão suporte à mesma sintaxe para expressar relações de herança. Os registros diferem das classes das seguintes maneiras:

- Você pode usar [parâmetros posicionais](#) em um [construtor primário](#) para criar e instanciar um tipo com propriedades imutáveis.
- Os mesmos métodos e operadores que indicam igualdade de referência ou desigualdade em classes (como `Object.Equals(Object)` e `==`), indicam [igualdade de valor ou desigualdade](#) nos registros.
- Você pode usar uma [expressão with](#) para criar uma cópia de um objeto imutável com novos valores em propriedades selecionadas.
- O método `ToString` de um registro cria uma cadeia de caracteres formatada que mostra o nome do tipo de um objeto e os nomes e valores de todas as propriedades públicas dele.
- Um registro pode [herdar de outro registro](#). Um registro não pode herdar de uma classe, e uma classe não pode herdar de um registro.

Os structs de registro diferem dos structs, pois o compilador sintetiza os métodos de igualdade e `ToString`. O compilador sintetiza o método `Deconstruct` para structs de registro posicional.

O compilador sintetiza uma propriedade public init-only para cada parâmetro de construtor primário em um `record class`. Em um `record struct`, o compilador sintetiza uma propriedade pública de leitura/gravação. O compilador não cria propriedades para parâmetros de construtor primário em tipos `class` e `struct` que não incluem modificador `record`.

## Exemplos

O exemplo a seguir define um registro público que usa parâmetros posicionais para declarar e instanciar um registro. Em seguida, ele imprime o nome do tipo e os valores de propriedade:

C#

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

O seguinte exemplo demonstra a igualdade de valor em registros:

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

O seguinte exemplo demonstra o uso de uma expressão `with` para copiar um objeto imutável e alterar uma das propriedades:

C#

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
}
```

```
// output: Person { FirstName = John, LastName = Davolio, PhoneNumbers =
System.String[] }
Console.WriteLine(person1 == person2); // output: False

person2 = person1 with { PhoneNumbers = new string[1] };
Console.WriteLine(person2);
// output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }
Console.WriteLine(person1 == person2); // output: False

person2 = person1 with { };
Console.WriteLine(person1 == person2); // output: True
}
```

Para obter mais informações, confira [Registros \(referência de C#\)](#).

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# Interfaces – definir comportamento para vários tipos

Artigo • 18/03/2023

Uma interface contém definições para um grupo de funcionalidades relacionadas que uma [class](#) ou um [struct](#) podem implementar. Uma interface pode definir métodos [static](#), que devem ter uma implementação. Uma interface pode definir uma implementação padrão para membros. Uma interface pode não declarar dados de instância, como campos, propriedades implementadas automaticamente ou eventos semelhantes a propriedades.

Usando interfaces, você pode, por exemplo, incluir o comportamento de várias fontes em uma classe. Essa funcionalidade é importante em C# porque a linguagem não dá suporte a várias heranças de classes. Além disso, use uma interface se você deseja simular a herança para structs, pois eles não podem herdar de outro struct ou classe.

Você define uma interface usando a palavra-chave [interface](#), como mostra o exemplo a seguir.

C#

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

O nome da interface deve ser um [nome do identificador](#) válido em C#. Por convenção, os nomes de interface começam com uma letra maiúscula [I](#).

Qualquer classe ou struct que implemente a interface [IEquatable<T>](#) deve conter uma definição para um método [Equals](#) que corresponda à assinatura que a interface especifica. Como resultado, você pode contar com uma classe que implementa [IEquatable<T>](#) para conter um método [Equals](#) com o qual uma instância da classe pode determinar se é igual a outra instância da mesma classe.

A definição de [IEquatable<T>](#) não fornece uma implementação para [Equals](#). Uma classe ou estrutura pode implementar várias interfaces, mas uma classe só pode herdar de uma única classe.

Para obter mais informações sobre classes abstratas, consulte [Classes e membros de classes abstratos e lacrados](#).

As interfaces podem conter métodos, propriedades, eventos, indexadores ou qualquer combinação desses quatro tipos de membros. As interfaces podem conter construtores estáticos, campos, constantes ou operadores. A partir do C# 11, os membros da interface que não são campos podem ser `static abstract`. Uma interface não pode conter campos de instância, construtores de instância ou finalizadores. Os membros da interface são públicos por padrão, e você pode especificar explicitamente modificadores de acessibilidade, como `public`, `protected`, `internal`, `private`, `protected internal` ou `private protected`. Um membro `private` deve ter uma implementação padrão.

Para implementar um membro de interface, o membro correspondente da classe de implementação deve ser público, não estático e ter o mesmo nome e assinatura do membro de interface.

### ⓘ Observação

Quando uma interface declara membros estáticos, um tipo que implementa essa interface também pode declarar membros estáticos com a mesma assinatura. Eles são distintos e identificados exclusivamente pelo tipo que declara o membro. O membro estático declarado em um tipo *não* substitui o membro estático declarado na interface.

Uma classe ou struct que implementa uma interface deve fornecer uma implementação para todos os membros declarados sem uma implementação padrão fornecida pela interface. No entanto, se uma classe base implementa uma interface, qualquer classe que é derivada da classe base herda essa implementação.

O exemplo a seguir mostra uma implementação da interface `IEquatable<T>`. A classe de implementação, `Car`, deverá fornecer uma implementação do método `Equals`.

C#

```
public class Car : IEquatable<Car>
{
    public string? Make { get; set; }
    public string? Model { get; set; }
    public string? Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car? car)
    {
        return (this.Make, this.Model, this.Year) ==
               (car?.Make, car?.Model, car?.Year);
    }
}
```

As propriedades e os indexadores de uma classe podem definir acessadores extras para uma propriedade ou o indexador que é definido em uma interface. Por exemplo, uma interface pode declarar uma propriedade que tem um acessador `get`. A classe que implementa a interface pode declarar a mesma propriedade tanto com um acessador `get` quanto com um `set`. No entanto, se a propriedade ou o indexador usa a implementação explícita, os acessadores devem corresponder. Para obter mais informações sobre a implementação explícita, consulte [Implementação de interface explícita](#) e [Propriedades da interface](#).

Uma interface pode herdar de uma ou mais interfaces. A interface derivada herda os membros de suas interfaces base. Uma classe que implementa uma interface derivada deve implementar todos os membros na interface derivada, incluindo todos os membros das interfaces base da interface derivada. Essa classe pode ser convertida implicitamente para a interface derivada ou para qualquer uma de suas interfaces base. Uma classe pode incluir uma interface várias vezes por meio das classes base que ela herda ou por meio de interfaces que outras interfaces herdam. No entanto, a classe poderá fornecer uma implementação de uma interface apenas uma vez e somente se a classe declarar a interface como parte da definição de classe (`class ClassName : InterfaceName`). Se a interface é herdada porque é herdada de uma classe base que implementa a interface, a classe base fornece a implementação dos membros da interface. No entanto, a classe derivada pode reimplementar qualquer membro de interface virtual em vez de usar a implementação herdada. Quando as interfaces declaram uma implementação padrão de um método, qualquer classe que implemente essa interface herda essa implementação (você precisa converter a instância da classe para o tipo de interface para acessar a implementação padrão no membro da interface).

Uma classe base também pode implementar membros de interface usando membros virtuais. Nesse caso, uma classe derivada pode alterar o comportamento da interface substituindo os membros virtuais. Para obter mais informações sobre membros virtuais, consulte [Polimorfismo](#).

## Resumo de interfaces

Uma interface tem as propriedades a seguir:

- Nas versões do C# anteriores à 8.0, uma interface é como uma classe base abstrata que contém apenas membros abstratos. Qualquer classe ou struct que implemente uma interface deve implementar todos os seus membros.
- A partir do C# 8.0, uma interface pode definir implementações padrão para alguns ou todos os seus membros. Uma classe ou struct que implementa a interface não

precisa implementar membros que tenham implementações padrão. Para obter mais informações, consulte [método de interface padrão](#).

- Uma interface não pode ser instanciada diretamente. Seus membros são implementados por qualquer classe ou struct que implemente a interface.
- Uma classe ou struct pode implementar várias interfaces. Uma classe pode herdar uma classe base e também implementar uma ou mais interfaces.

# Classes e métodos genéricos

Artigo • 15/07/2023

Genéricos apresentam o conceito de parâmetros de tipo ao .NET, que possibilitam a criação de classes e métodos que adiam a especificação de um ou mais tipos até que a classe ou método seja declarado e instanciado pelo código do cliente. Por exemplo, ao usar um parâmetro de tipo genérico `T`, você pode escrever uma única classe que outro código de cliente pode usar sem incorrer o custo ou risco de conversões de runtime ou operações de conversão boxing, conforme mostrado aqui:

C#

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }

}

class TestGenericList
{
    private class ExampleClass { }

    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

As classes e métodos genéricos combinam a capacidade de reutilização, a segurança de tipos e a eficiência de uma maneira que suas contrapartes não genéricas não conseguem. Os genéricos são usados com mais frequência com coleções e com os métodos que operam nelas. O namespace `System.Collections.Generic` contém várias classes de coleção de base genérica. As coleções não-genéricas, como `ArrayList` não são recomendadas e são mantidas para fins de compatibilidade. Para saber mais, confira [Genéricos no .NET](#).

Você também pode criar tipos e métodos genéricos personalizados para fornecer suas próprias soluções e padrões de design generalizados que sejam fortemente tipados e

eficientes. O exemplo de código a seguir mostra uma classe de lista vinculada genérica simples para fins de demonstração. (Na maioria dos casos, você deve usar a classe `List<T>` fornecida pelo .NET em vez de criar sua própria.) O parâmetro de tipo `T` é usado em vários locais em que um tipo concreto normalmente seria usado para indicar o tipo do item armazenado na lista. Ele é usado das seguintes maneiras:

- Como o tipo de um parâmetro de método no método `AddHead`.
- Como o tipo de retorno da propriedade `Data` na classe `Node` aninhada.
- Como o tipo de `data` do membro particular na classe aninhada.

`T` está disponível para a classe aninhada `Node`. Quando `GenericList<T>` é instanciada com um tipo concreto, por exemplo como um `GenericList<int>`, cada ocorrência de `T` será substituída por `int`.

C#

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node? next;
        public Node? Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node? head;

    // constructor
```

```

public GenericList()
{
    head = null;
}

// T as method parameter type:
public void AddHead(T t)
{
    Node n = new Node(t);
    n.Next = head;
    head = n;
}

public IEnumarator<T> GetEnumarator()
{
    Node? current = head;

    while (current != null)
    {
        yield return current.Data;
        current = current.Next;
    }
}
}

```

O exemplo de código a seguir mostra como o código cliente usa a classe `GenericList<T>` genérica para criar uma lista de inteiros. Ao simplesmente alterar o argumento de tipo, o código a seguir poderia facilmente ser modificado para criar listas de cadeias de caracteres ou qualquer outro tipo personalizado:

C#

```

class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}

```

### ⓘ Observação

Os exemplos a seguir se aplicam não apenas aos tipos `class`, mas também aos tipos `interface` e `struct`

## Visão geral de genéricos

- Use tipos genéricos para maximizar a reutilização de código, o desempenho e a segurança de tipo.
- O uso mais comum de genéricos é para criar classes de coleção.
- A biblioteca de classes do .NET contém várias classes de coleção de genéricos no namespace [System.Collections.Generic](#). As coleções genéricas devem ser usadas sempre que possível, em vez de classes como [ArrayList](#) no namespace [System.Collections](#).
- Você pode criar suas próprias interfaces genéricas, classes, métodos, eventos e delegados.
- Classes genéricas podem ser restrinpidas para habilitar o acesso aos métodos em tipos de dados específicos.
- Informações sobre os tipos que são usados em um tipo de dados genérico podem ser obtidas no tempo de execução por meio de reflexão.

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#).

## Confira também

- [System.Collections.Generic](#)
- [Generics in .NET \(Genéricos no .NET\)](#)

# Tipos anônimos

Artigo • 10/05/2023

Os tipos anônimos fornecem um modo conveniente de encapsular um conjunto de propriedades somente leitura em um único objeto sem a necessidade de primeiro definir explicitamente um tipo. O nome do tipo é gerado pelo compilador e não está disponível no nível do código-fonte. O tipo de cada propriedade é inferido pelo compilador.

Crie tipos anônimos usando o operador [new](#) com um inicializador de objeto. Para obter mais informações sobre inicializadores de objeto, consulte [Inicializadores de Objeto e Coleção](#).

O exemplo a seguir mostra um tipo anônimo que é inicializado com duas propriedades chamadas `Amount` e `Message`.

C#

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

Os tipos anônimos são normalmente usados na cláusula [select](#) de uma expressão de consulta para retornar um subconjunto das propriedades de cada objeto na sequência de origem. Para obter mais informações sobre consultas, consulte [LINQ no C#](#).

Os tipos anônimos contêm uma ou mais propriedades públicas somente leitura. Nenhum outro tipo de membros da classe, como métodos ou eventos, é válido. A expressão que é usada para inicializar uma propriedade não pode ser `null`, uma função anônima ou um tipo de ponteiro.

O cenário mais comum é inicializar um tipo anônimo com propriedades de outro tipo. No exemplo a seguir, suponha que existe uma classe com o nome `Product`. A classe `Product` inclui as propriedades `Color` e `Price`, além de outras propriedades que não lhe interessam. A variável `products` é uma coleção de objetos do `Product`. A declaração do tipo anônimo começa com a palavra-chave `new`. A declaração inicializa um novo tipo que usa apenas duas propriedades de `Product`. O uso de tipos anônimos faz com que uma menor quantidade de dados seja retornada na consulta.

Quando você não especifica os nomes de membros no tipo anônimo, o compilador dá aos membros de tipo anônimo o mesmo nome da propriedade que está sendo usada para inicializá-los. Forneça um nome para a propriedade que está sendo inicializada com uma expressão, como mostrado no exemplo anterior. No exemplo a seguir, os nomes das propriedades do tipo anônimo são `Color` e `Price`.

C#

```
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

### 💡 Dica

Você pode usar a regra de estilo .NET [IDE0037](#) para impor se os nomes de membros inferidos ou explícitos são preferenciais.

Também é possível definir um campo por objeto de outro tipo: classe, struct ou até mesmo outro tipo anônimo. Isso é feito usando a variável que contém esse objeto exatamente como no exemplo a seguir, onde dois tipos anônimos são criados usando tipos definidos pelo usuário já instanciados. Em ambos os casos o campo `product` nos tipos anônimos `shipment` e `shipmentWithBonus` serão do tipo `Product` contendo seus valores padrão de cada campo. E o campo `bonus` será do tipo anônimo criado pelo compilador.

C#

```
var product = new Product();
var bonus = new { note = "You won!" };
var shipment = new { address = "Nowhere St.", product };
var shipmentWithBonus = new { address = "Somewhere St.", product, bonus };
```

Normalmente, ao usar um tipo anônimo para inicializar uma variável, a variável é declarada como uma variável local de tipo implícito usando `var`. O nome do tipo não pode ser especificado na declaração da variável, porque apenas o compilador tem acesso ao nome subjacente do tipo anônimo. Para obter mais informações sobre `var`, consulte [Variáveis de local digitadas implicitamente](#).

Você pode criar uma matriz de elementos de tipo anônimo combinando uma variável local de tipo implícito e uma matriz de tipo implícito, como mostrado no exemplo a seguir.

C#

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};
```

Os tipos anônimos são tipos [class](#) que derivam diretamente de [object](#) e que não podem ser convertidos para qualquer tipo, exceto para [object](#). O compilador fornece um nome para cada tipo anônimo, embora o seu aplicativo não possa acessá-lo. Do ponto de vista do Common Language Runtime, um tipo anônimo não é diferente de qualquer outro tipo de referência.

Se dois ou mais inicializadores de objeto anônimos em um assembly especificarem uma sequência de propriedades que estão na mesma ordem e que têm os mesmos nomes e tipos, o compilador tratará os objetos como instâncias do mesmo tipo. Eles compartilham o mesmo tipo de informação gerado pelo compilador.

Tipos anônimos dão suporte a mutações não destrutivas na forma de [expressões with](#). Isso permite que você crie uma nova instância de um tipo anônimo em que uma ou mais propriedades têm novos valores:

C#

```
var apple = new { Item = "apples", Price = 1.35 };
var onSale = apple with { Price = 0.79 };
Console.WriteLine(apple);
Console.WriteLine(onSale);
```

Você não pode declarar que um campo, uma propriedade, um evento ou um tipo de retorno de um método tem um tipo anônimo. Da mesma forma, não pode declarar que um parâmetro formal de um método, propriedade, construtor ou indexador tem um tipo anônimo. Para passar um tipo anônimo ou uma coleção que contenha tipos anônimos como um argumento para um método, você pode declarar o parâmetro como tipo [object](#). No entanto, usar [object](#) para tipos anônimos anula o propósito da tipagem forte. Se você precisa armazenar os resultados da consulta ou passá-los fora do limite do método, considere o uso de uma estrutura ou classe com denominação comum em vez de um tipo anônimo.

Como os métodos [Equals](#) e [GetHashCode](#) em tipos anônimos são definidos em termos dos métodos das propriedades [Equals](#) e [GetHashCode](#), duas instâncias do mesmo tipo

anônimo são iguais somente se todas as suas propriedades forem iguais.

Tipos anônimos substituem o método [ToString](#), concatenando o nome e a saída `ToString` de cada propriedade cercada por chaves.

```
var v = new { Title = "Hello", Age = 24 };

Console.WriteLine(v.ToString()); // "{ Title = Hello, Age = 24 }"
```

# Visão geral de classes, structs e registros em C#

Artigo • 15/02/2023

Em C#, a definição de um tipo — uma classe, um struct ou um registro — é como um blueprint que especifica o que o tipo pode fazer. Um objeto é basicamente um bloco de memória que foi alocado e configurado de acordo com o esquema. Este artigo fornece uma visão geral desses blueprints e os respectivos recursos. O [próximo artigo desta série](#) apresenta objetos.

## Encapsulamento

*Encapsulamento* é chamado, ocasionalmente, de primeiro pilar ou princípio da programação orientada a objeto. Uma classe ou um struct pode especificar qual membro será codificado fora da classe ou do struct. Os métodos e as variáveis que não serão usados fora da classe ou assembly poderão ser ocultados para limitar erros de codificação potenciais ou explorações maliciosas. Para obter mais informações, confira o tutorial de [Programação orientada a objeto](#).

## Membros

Os *membros* de um tipo incluem todos os métodos, campos, constantes, propriedades e eventos. No C#, não existem variáveis globais ou métodos como em algumas das outras linguagens. Mesmo o ponto de entrada de um programa, o método `Main`, deve ser declarado dentro de uma classe ou de um struct (implicitamente quando você usa [instruções de nível superior](#)).

A lista a seguir inclui todos os vários tipos de membros que podem ser declarados em uma classe, um struct ou um registro.

- Campos
- Constantes
- Propriedades
- Métodos
- Construtores
- Eventos
- Finalizadores
- Indexadores
- Operadores

- Tipos aninhados

Para obter mais informações, confira [Membros](#).

## Acessibilidade

Alguns métodos e propriedades devem ser chamados ou acessado pelo código fora da classe ou do struct, também conhecido como *código de cliente*. Outros métodos e propriedades podem ser usados apenas na classe ou struct em si. É importante limitar o acessibilidade do código para que somente o código do cliente desejado possa fazer contato. Você especifica o grau de acessibilidade dos tipos e os respectivos membros ao código do cliente usando os seguintes modificadores de acesso:

- [público](#)
- [protected](#)
- [interno](#)
- [internos protegidos](#)
- [private](#)
- [protegido de forma particular](#).

A acessibilidade padrão é [private](#).

## Herança

Classes (mas não structs) dão suporte ao conceito de herança. Uma classe que deriva de outra classe, chamada *classe base*, contém automaticamente todos os membros públicos, protegidos e internos da classe base, exceto seus construtores e finalizadores.

As classes podem ser declaradas como [abstratas](#), o que significa que um ou mais dos seus métodos não têm nenhuma implementação. Embora as classes abstratas não possam ser instanciadas diretamente, elas servem como classes base para outras classes que fornecem a implementação ausente. As classes também podem ser declaradas como [lacradas](#) para impedir que outras classes herdem delas.

Para obter mais informações, consulte [Herança](#) e [Polimorfismo](#).

## Interfaces

Classes, structs e registros podem implementar várias interfaces. Implementar a partir de uma interface significa que o tipo implementa todos os métodos definidos na interface. Para obter mais informações, consulte [Interfaces](#).

# Tipos genéricos

Classes, structs e registros podem ser definidos com um ou mais parâmetros de tipo. O código do cliente fornece o tipo quando ele cria uma instância do tipo. Por exemplo a classe `List<T>` no namespace `System.Collections.Generic` é definida com um parâmetro de tipo. O código do cliente cria uma instância de um `List<string>` ou `List<int>` para especificar o tipo que a lista conterá. Para obter mais informações, consulte [Genéricos](#).

# Tipos estáticos

Classes (mas não structs ou registros) podem ser declaradas como `static`. Uma classe estática pode conter apenas membros estáticos e não pode ser instanciada com a palavra-chave `new`. Uma cópia da classe é carregada na memória quando o programa é carregado e seus membros são acessados pelo nome da classe. Classes, structs e registros podem conter membros estáticos. Para obter mais informações, consulte [Classes estáticas e membros de classes estáticas](#).

# Tipos aninhados

Uma classe, struct ou registro pode ser aninhado dentro de outra classe, struct ou registro. Para obter mais informações, consulte [Tipos aninhados](#).

# Tipos parciais

Você pode definir parte de uma classe, struct ou método em um arquivo de código e outra parte em um arquivo de código separado. Para obter mais informações, consulte [Classes parciais e métodos](#).

# Inicializadores de objeto

Você pode criar uma instância e inicializar objetos de classe ou struct e coleções de objetos, atribuindo valores às respectivas propriedades. Para mais informações, consulte [Como inicializar objetos usando um inicializador de objeto](#).

# Tipos anônimos

Em situações em que não é conveniente ou necessário criar uma classe nomeada, você usa tipos anônimos. Tipos anônimos são definidos pelos membros de dados nomeados.

Para obter mais informações, consulte [Tipos anônimos](#).

## Métodos de Extensão

Você pode "estender" uma classe sem criar uma classe derivada criando um tipo separado. Esse tipo contém métodos que podem ser chamados como se pertencessem ao tipo original. Para obter mais informações, consulte [Métodos de extensão](#).

## Variáveis Locais Tipadas Implicitamente

Dentro de um método de classe ou struct, você pode usar digitação implícita para instruir o compilador para determinar o tipo de variável no tempo de compilação. Para obter mais informações, confira [var \(referência C#\)](#).

## Registros

O C# 9 introduz o tipo `record`, um tipo de referência que você pode criar em vez de uma classe ou um struct. Os registros são classes com comportamento interno para encapsular dados em tipos imutáveis. O C# 10 introduz o tipo de valor `record struct`. Um registro (`record class` ou `record struct`) fornece os seguintes recursos:

- Sintaxe concisa para criar um tipo de referência com propriedades imutáveis.
- Igualdade de valor. Duas variáveis de um tipo de registro são iguais se tiverem o mesmo tipo e se, para cada campo, os valores em ambos os registros forem iguais. As classes usam igualdade de referência: duas variáveis de um tipo de classe são iguais se elas se referirem ao mesmo objeto.
- Sintaxe concisa para mutação não destrutiva. Uma expressão `with` permite criar uma nova instância de registro que seja uma cópia de uma instância existente, mas com valores de propriedade especificados alterados.
- Formatação interna para exibição. O método `ToString` imprime o nome do tipo de registro e os nomes e valores das propriedades públicas.
- Suporte para hierarquias de herança em classes de registro. Classes de registro dão suporte à herança. Os structs de registro não dão suporte à herança.

Para obter mais informações, confira [Registros](#).

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# Objetos – criar instâncias de tipos

Artigo • 10/05/2023

Uma definição de classe ou struct é como um esquema que especifica o que o tipo pode fazer. Um objeto é basicamente um bloco de memória que foi alocado e configurado de acordo com o esquema. Um programa pode criar vários objetos da mesma classe. Objetos também são chamados de instâncias e podem ser armazenados em uma variável nomeada ou em uma matriz ou coleção. O código de cliente é o código que usa essas variáveis para chamar os métodos e acessar as propriedades públicas do objeto. Em uma linguagem orientada a objetos, como o C#, um programa típico consiste em vários objetos que interagem dinamicamente.

## ⓘ Observação

Tipos estáticos se comportam de modo diferente do que está descrito aqui. Para obter mais informações, consulte [Classes estáticas e membros de classes estáticas](#).

## Instâncias Struct vs. Instâncias de classe

Como as classes são tipos de referência, uma variável de um objeto de classe contém uma referência ao endereço do objeto no heap gerenciado. Se uma segunda variável do mesmo tipo for atribuída à primeira variável, as duas variáveis farão referência ao objeto nesse endereço. Esse ponto é abordado com mais detalhes posteriormente neste artigo.

Instâncias de classes são criadas usando o [new operador](#). No exemplo a seguir, `Person` é o tipo e `person1` e `person2` são instâncias ou objetos desse tipo.

C#

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}

class Program
{
```

```

static void Main()
{
    Person person1 = new Person("Leopold", 6);
    Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name,
person1.Age);

    // Declare new person, assign person1 to it.
    Person person2 = person1;

    // Change the name of person2, and person1 also changes.
    person2.Name = "Molly";
    person2.Age = 16;

    Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name,
person2.Age);
    Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name,
person1.Age);
}
/*
Output:
person1 Name = Leopold Age = 6
person2 Name = Molly Age = 16
person1 Name = Molly Age = 16
*/

```

Como structs são tipos de valor, uma variável de um objeto de struct mantém uma cópia do objeto inteiro. Instâncias de structs também podem ser criadas usando o operador `new`, mas isso não é obrigatório, conforme mostrado no exemplo a seguir:

C#

```

namespace Example;

public struct Person
{
    public string Name;
    public int Age;
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

public class Application
{
    static void Main()
    {
        // Create struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
    }
}

```

```

        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

    // Create new struct object. Note that struct can be initialized
    // without using "new".
    Person p2 = p1;

    // Assign values to p2 members.
    p2.Name = "Spencer";
    p2.Age = 7;
    Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

    // p1 values remain unchanged because p2 is copy.
    Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);
}

/*
Output:
p1 Name = Alex Age = 9
p2 Name = Spencer Age = 7
p1 Name = Alex Age = 9
*/

```

A memória de `p1` e `p2` é alocada na pilha de thread. Essa memória é recuperada em conjunto com o tipo ou método em que ela é declarada. Esse é um dos motivos pelos quais os structs são copiados na atribuição. Por outro lado, a memória alocada a uma instância de classe é recuperada automaticamente (o lixo é coletado) pelo Common Language Runtime quando todas as referências ao objeto tiveram saído do escopo. Não é possível destruir de forma determinista um objeto de classe, como é possível no C++. Para obter mais informações sobre a coleta de lixo no , consulte [Coleta de lixo](#).

#### ① Observação

A alocação e a desalocação de memória no heap gerenciado é altamente otimizada no Common Language Runtime. Na maioria dos casos, não há uma diferença significativa quanto ao custo do desempenho de alocar uma instância da classe no heap em vez de alocar uma instância de struct na pilha.

## Identidade do Objeto vs. Igualdade de Valor

Quando compara dois objetos quanto à igualdade, primeiro você precisa distinguir se quer saber se as duas variáveis representam o mesmo objeto na memória ou se os valores de um ou mais de seus campos são equivalentes. Se quiser comparar valores, você precisa considerar se os objetos são instâncias de tipos de valor (structs) ou tipos de referência (classes, delegados, matrizes).

- Para determinar se duas instâncias de classe se referem ao mesmo local na memória (o que significa que elas têm a mesma *identidade*), use o método `Object.Equals` estático. (`System.Object` é a classe base implícita para todos os tipos de valor e tipos de referência, incluindo classes e structs definidos pelo usuário.)
- Para determinar se os campos de instância em duas instâncias de struct têm os mesmos valores, use o método `ValueType.Equals`. Como todos os structs herdam implicitamente de `System.ValueType`, você chama o método diretamente em seu objeto, conforme mostrado no exemplo a seguir:

C#

```
// Person is defined in the previous example.

//public struct Person
//{
//    public string Name;
//    public int Age;
//    public Person(string name, int age)
//    {
//        Name = name;
//        Age = age;
//    }
//}

Person p1 = new Person("Wallace", 75);
Person p2 = new Person("", 42);
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

A implementação `System.ValueType` de `Equals` usa conversão boxing e reflexão em alguns casos. Para obter informações sobre como fornecer um algoritmo de igualdade eficiente específico ao seu tipo, consulte [Como definir a igualdade de valor para um tipo](#). Os registros são tipos de referência que usam semântica de valor para igualdade.

- Para determinar se os valores dos campos em duas instâncias de classe são iguais, você pode usar o método `Equals` ou o Operador `==`. No entanto, use-os apenas se a classe os tiver substituído ou sobrecarregado para fornecer uma definição personalizada do que "igualdade" significa para objetos desse tipo. A classe também pode implementar a interface `IEquatable<T>` ou a interface `IEqualityComparer<T>`. As duas interfaces fornecem métodos que podem ser

usados para testar a igualdade de valores. Ao criar suas próprias classes que substituem `Equals`, certifique-se de seguir as diretrizes informadas em [Como definir a igualdade de valor para um tipo e Object.Equals\(Object\)](#).

## Seções relacionadas

Para mais informações:

- [Classes](#)
- [Construtores](#)
- [Finalizadores](#)
- [Eventos](#)
- [object](#)
- [Herança](#)
- [class](#)
- [Tipos de estrutura](#)
- [Operador new](#)
- [Common Type System](#)

# Herança – derivar tipos para criar um comportamento mais especializado

Artigo • 07/04/2023

A herança, assim como o encapsulamento e o polimorfismo, é uma das três principais características da programação orientada ao objeto. A herança permite que você crie novas classes que reutilizam, estendem e modificam o comportamento definido em outras classes. A classe cujos membros são herdados é chamada *classe base* e a classe que herda esses membros é chamada *classe derivada*. Uma classe derivada pode ter apenas uma classe base direta. No entanto, a herança é transitiva. Se `ClassC` for derivado de `ClassB` e `ClassB` for derivado de `ClassA`, `ClassC` herdará os membros declarados em `ClassB` e `ClassA`.

## ⓘ Observação

Structs não dão suporte a herança, mas podem implementar interfaces.

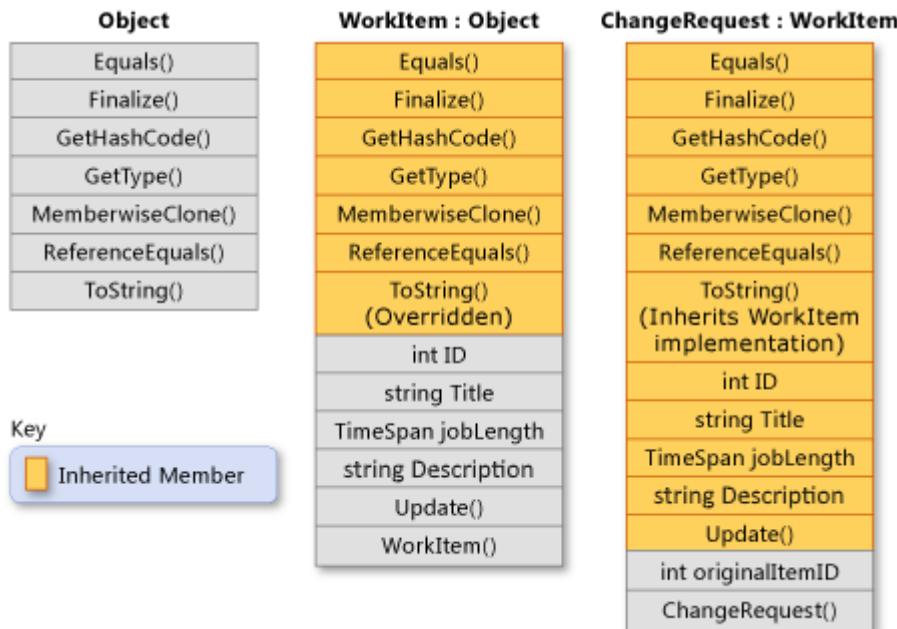
Conceitualmente, uma classe derivada é uma especialização da classe base. Por exemplo, se tiver uma classe base `Animal`, você pode ter uma classe derivada chamada `Mammal` e outra classe derivada chamada `Reptile`. Um `Mammal` é um `Animal` e um `Reptile` é um `Animal`, mas cada classe derivada representa especializações diferentes da classe base.

As declarações de interface podem definir uma implementação padrão para seus membros. Essas implementações são herdadas por interfaces derivadas e por classes que implementam essas interfaces. Para obter mais informações sobre métodos de interface padrão, confira o artigo sobre [interfaces](#).

Quando você define uma classe para derivar de outra classe, a classe derivada obtém implicitamente todos os membros da classe base, exceto por seus construtores e finalizadores. A classe derivada reutiliza o código na classe base sem precisar implementá-lo novamente. Na classe derivada, você pode adicionar mais membros. A classe derivada estende a funcionalidade da classe base.

A ilustração a seguir mostra uma classe `WorkItem` que representa um item de trabalho em um processo comercial. Como todas as classes, ela deriva de `System.Object` e herda todos os seus métodos. `WorkItem` adiciona seis membros próprios. Esses membros incluem um construtor, porque os construtores não são herdados. A classe `ChangeRequest` herda de `WorkItem` e representa um tipo específico de item de trabalho.

`ChangeRequest` adiciona mais dois membros aos membros que herda de `WorkItem` e de `Object`. Ele deve adicionar seu próprio construtor e também adiciona `originalItemID`. A propriedade `originalItemID` permite que a instância `ChangeRequest` seja associada ao `WorkItem` original a que a solicitação de alteração se aplica.



O exemplo a seguir mostra como as relações entre as classes demonstradas na ilustração anterior são expressos em C#. O exemplo também mostra como `WorkItem` substitui o método virtual `Object.ToString` e como a classe `ChangeRequest` herda a implementação de `WorkItem` do método. O primeiro bloco define as classes:

C#

```

// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
    }
}
  
```

```

        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem() => currentID = 0;

    // currentID is a static field. It is incremented each time a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
                         int originalID)
    {
        // The following properties and the GetNexID method are inherited
        // from WorkItem.
    }
}

```

```

        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemID is a member of ChangeRequest, but not
        // of WorkItem.
        this.originalItemID = originalID;
    }
}

```

Este próximo bloco mostra como usar as classes base e derivadas:

C#

```

// Create an instance of WorkItem by using the constructor in the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                             "Fix all bugs in my code branch",
                             new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in
// the derived class that takes four arguments.
ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                         "Add members to the class",
                                         new TimeSpan(4, 0, 0),
                                         1);

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
             new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
   1 - Fix Bugs
   2 - Change the Design of the Base Class
*/

```

## Métodos abstratos e virtuais

Quando uma classe base declara um método como **virtual**, uma classe derivada poderá **override** o método com a própria implementação. Se uma classe base declarar um membro como **abstract**, esse método deve ser substituído em qualquer classe não abstrata que herdar diretamente da classe. Se uma classe derivada for abstrato, ele

herdará membros abstratos sem implementá-los. Membros abstratos e virtuais são a base do polimorfismo, que é a segunda característica principal da programação orientada a objetos. Para obter mais informações, consulte [Polimorfismo](#).

## Classes base abstratas

Você poderá declarar uma classe como [abstrata](#) se quiser impedir a instanciação direta usando o operador [new](#). Uma classe abstrata poderá ser usada somente se uma nova classe for derivada dela. Uma classe abstrata pode conter uma ou mais assinaturas de método que também são declaradas como abstratas. Essas assinaturas especificam os parâmetros e o valor retornado, mas não têm nenhuma implementação (corpo do método). Uma classe abstrata não precisa conter membros abstratos. No entanto, se uma classe contiver um membro abstrato, a própria classe deverá ser declarada como abstrata. Classes derivadas que não são abstratas devem fornecer a implementação para qualquer método abstrato de uma classe base abstrata.

## Interfaces

Uma *interface* é um tipo de referência que define um conjunto de membros. Todas as classes e structs que implementam essa interface devem implementar esse conjunto de membros. Uma interface pode definir uma implementação padrão para qualquer um desses membros. Uma classe pode implementar várias interfaces, mesmo que ela possa derivar de apenas uma classe base direta.

Interfaces são usadas para definir recursos específicos para classes que não têm necessariamente uma relação do tipo "é um". Por exemplo, a interface [System.IEquatable<T>](#) pode ser implementada por qualquer classe ou struct para determinar se dois objetos do tipo são equivalentes (como quer que o tipo defina a equivalência). [IEquatable<T>](#) não implica o mesmo tipo de relação "é um" existente entre uma classe base e uma classe derivada (por exemplo, um `Mammal` é um `Animal`). Para obter mais informações, consulte [Interfaces](#).

## Impedindo derivações adicionais

Uma classe pode impedir que outras classes herdem dela ou de qualquer um de seus membros ao declarar a si mesma ou o membro como [sealed](#).

## Ocultação de membros da classe base pela classe derivada

Uma classe derivada pode ocultar membros da classe base declarando membros com mesmo nome e assinatura. O modificador [new](#) pode ser usado para indicar explicitamente que o membro não pretende ser uma substituição do membro base. O uso de [new](#) não é necessário, mas um aviso do compilador será gerado se [new](#) não for usado. Para obter mais informações, consulte [Controle de versão com as palavras-chave override e new](#) e [Quando usar as palavras-chave override e new](#).

# Polimorfismo

Artigo • 07/04/2023

O polimorfismo costuma ser chamado de o terceiro pilar da programação orientada a objetos, depois do encapsulamento e a herança. O polimorfismo é uma palavra grega que significa "de muitas formas" e tem dois aspectos distintos:

- Em tempo de execução, os objetos de uma classe derivada podem ser tratados como objetos de uma classe base, em locais como parâmetros de método, coleções e matrizes. Quando esse polimorfismo ocorre, o tipo declarado do objeto não é mais idêntico ao seu tipo de tempo de runtime.
- As classes base podem definir e implementar [métodos virtuais](#) e as classes derivadas podem [substituí-los](#), o que significa que elas fornecem sua própria definição e implementação. Em tempo de execução, quando o código do cliente chama o método, o CLR procura o tipo de tempo de execução do objeto e invoca a substituição do método virtual. No código-fonte, você pode chamar um método em uma classe base e fazer com que a versão de uma classe derivada do método seja executada.

Os métodos virtuais permitem que você trabalhe com grupos de objetos relacionados de maneira uniforme. Por exemplo, suponha que você tem um aplicativo de desenho que permite que um usuário crie vários tipos de formas sobre uma superfície de desenho. Você não sabe no tempo de compilação quais tipos específicos de formas o usuário criará. No entanto, o aplicativo precisa manter controle de todos os diferentes tipos de formas que são criados e atualizá-los em resposta às ações do mouse do usuário. Você pode usar o polimorfismo para resolver esse problema em duas etapas básicas:

1. Crie uma hierarquia de classes em que cada classe de forma específica derive de uma classe base comum.
2. Use um método virtual para invocar o método adequado em qualquer classe derivada por meio de uma única chamada para o método da classe base.

Primeiro, crie uma classe base chamada `Shape` e as classes derivadas como `Rectangle`, `Circle` e `Triangle`. Atribua à classe `Shape` um método virtual chamado `Draw` e substitua-o em cada classe derivada para desenhar a forma especial que a classe representa. Crie um objeto `List<Shape>` e adicione um `Circle`, `Triangle` e `Rectangle` a ele.

C#

```

public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}

public class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}

public class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}

```

Para atualizar a superfície de desenho, use um loop `foreach` para iterar na lista e chamar o método `Draw` em cada objeto `Shape` na lista. Mesmo que cada objeto na lista tenha um tipo declarado `Shape`, é o tipo de runtime (a versão de substituição do método em cada classe derivada) que será invocado.

```

// Polymorphism at work #1: a Rectangle, Triangle and Circle
// can all be used wherever a Shape is expected. No cast is
// required because an implicit conversion exists from a derived
// class to its base class.
var shapes = new List<Shape>
{
    new Rectangle(),
    new Triangle(),
    new Circle()
};

// Polymorphism at work #2: the virtual method Draw is
// invoked on each of the derived classes, not the base class.
foreach (var shape in shapes)
{
    shape.Draw();
}
/* Output:
   Drawing a rectangle
   Performing base class drawing tasks
   Drawing a triangle
   Performing base class drawing tasks
   Drawing a circle
   Performing base class drawing tasks
*/

```

Em C#, cada tipo é polimórfico porque todos os tipos, incluindo tipos definidos pelo usuário, herdam de [Object](#).

## Visão geral sobre o polimorfismo

### Membros virtuais

Quando uma classe derivada herda de uma classe base, ela inclui todos os membros da classe base. Todo o comportamento declarado na classe base faz parte da classe derivada. Isso permite que objetos da classe derivada sejam tratados como objetos da classe base. Os modificadores de acesso (`public`, `protected` `private` e assim por diante) determinam se esses membros estão acessíveis a partir da implementação de classe derivada. Os métodos virtuais dão ao designer opções diferentes para o comportamento da classe derivada:

- A classe derivada pode substituir membros virtuais na classe base, definindo o novo comportamento.
- A classe derivada pode herdar o método de classe base mais próximo sem substituí-lo, preservando o comportamento existente, mas permitindo que outras

classes derivadas substituam o método.

- A classe derivada pode definir uma nova implementação não virtual desses membros que ocultam as implementações da classe base.

Uma classe derivada poderá substituir um membro de classe base somente se o membro da classe base tiver sido declarado como [virtual](#) ou [abstrato](#). O membro derivado deve usar a palavra-chave [override](#) para indicar explicitamente que o método destina-se a participar da invocação virtual. O código a seguir mostra um exemplo:

C#

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Os campos não podem ser virtuais, apenas os métodos, as propriedades, os eventos e os indexadores podem ser virtuais. Quando uma classe derivada substitui um membro virtual, esse membro é chamado, mesmo quando uma instância dessa classe está sendo acessada como uma instância da classe base. O código a seguir mostra um exemplo:

C#

```
DerivedClass B = new DerivedClass();
B.DoWork(); // Calls the new method.

BaseClass A = B;
A.DoWork(); // Also calls the new method.
```

Os métodos e propriedades virtuais permitem que classes derivadas estendam uma classe base sem a necessidade de usar a implementação da classe base de um método. Para obter mais informações, consulte [Controle de versão com as palavras-chave override e new](#). Uma interface fornece uma outra maneira de definir um método ou conjunto de métodos cuja implementação é deixada para classes derivadas.

## Ocultar membros da classe base com novos membros

Se quiser que sua classe derivada tenha um membro com o mesmo nome que um membro em uma classe base, você poderá usar a palavra-chave `new` para ocultar o membro da classe base. A palavra-chave `new` é colocada antes do tipo de retorno de um membro de classe que está sendo substituído. O código a seguir mostra um exemplo:

```
C#  
  
public class BaseClass  
{  
    public void DoWork() { WorkField++; }  
    public int WorkField;  
    public int WorkProperty  
    {  
        get { return 0; }  
    }  
}  
  
public class DerivedClass : BaseClass  
{  
    public new void DoWork() { WorkField++; }  
    public new int WorkField;  
    public new int WorkProperty  
    {  
        get { return 0; }  
    }  
}
```

Os membros da classe base oculta podem ser acessados do código do cliente, por meio da seleção da instância da classe derivada em uma instância da classe base. Por exemplo:

```
C#  
  
DerivedClass B = new DerivedClass();  
B.DoWork(); // Calls the new method.  
  
BaseClass A = (BaseClass)B;  
A.DoWork(); // Calls the old method.
```

## Impedir que classes derivadas substituam membros virtuais

Os membros virtuais permanecem virtuais, independentemente de quantas classes foram declaradas entre o membro virtual e a classe que o declarou originalmente. Se a classe A declara um membro virtual, a classe B deriva de A e a classe C deriva de B, a classe C herda o membro virtual e tem a opção de substituí-lo, independentemente de a classe B ter declarado uma substituição para esse membro. O código a seguir mostra um exemplo:

```
C#  
  
public class A  
{  
    public virtual void DoWork() { }  
}  
public class B : A  
{  
    public override void DoWork() { }  
}
```

Uma classe derivada pode interromper a herança virtual, declarando uma substituição como sealed. Isso exige a colocação da palavra-chave sealed antes da palavra-chave override na declaração de membro de classe. O código a seguir mostra um exemplo:

```
C#  
  
public class C : B  
{  
    public sealed override void DoWork() { }  
}
```

No exemplo anterior, o método DoWork não é mais virtual para nenhuma classe derivada de C. Ele ainda é virtual para as instâncias de C, mesmo se elas foram convertidas para o tipo B ou tipo A. Os métodos lacrados podem ser substituídos por classes derivadas usando a palavra-chave new, como mostra o exemplo a seguir:

```
C#  
  
public class D : C  
{  
    public new void DoWork() { }  
}
```

Neste caso, se DoWork é chamado em D usando uma variável do tipo D, o novo DoWork é chamado. Se uma variável do tipo C, B ou A é usada para acessar uma instância de D,

uma chamada de `DoWork` seguirá as regras de herança virtual, encaminhando as chamadas para a implementação de `DoWork` na classe `C`.

## Acessar membros virtuais da classe base de classes derivadas

A classe derivada que substituiu um método ou propriedade ainda pode acessar o método ou propriedade na classe base usando a palavra-chave `base`. O código a seguir mostra um exemplo:

```
C#  
  
public class Base  
{  
    public virtual void DoWork() {/*...*/}  
}  
public class Derived : Base  
{  
    public override void DoWork()  
    {  
        //Perform Derived's work here  
        //...  
        // Call DoWork on base class  
        base.DoWork();  
    }  
}
```

Para obter mais informações, consulte [base](#).

### ⓘ Observação

Recomendamos que os membros virtuais usem `base` para chamar a implementação da classe base do membro em sua própria implementação. Deixar o comportamento da classe base ocorrer permite que a classe derivada se concentre na implementação de comportamento específico para a classe derivada. Se a implementação da classe base não é chamado, cabe à classe derivada tornar seu comportamento compatível com o comportamento da classe base.

# Visão geral dos padrões correspondentes

Artigo • 16/06/2023

A *correspondência de padrões* é uma técnica em que você testa uma expressão para determinar se ela tem determinadas características. A correspondência de padrões C# fornece uma sintaxe mais concisa para testar expressões e tomar medidas quando uma expressão corresponde. A "expressão `is`" dá suporte à correspondência de padrões para testar uma expressão e declarar condicionalmente uma nova variável para o resultado dessa expressão. A "expressão `switch`" permite que você execute ações com base no padrão de primeira correspondência de uma expressão. Essas duas expressões dão suporte a um vocabulário avançado de [padrões](#).

Este artigo fornece uma visão geral dos cenários em que você pode usar a correspondência de padrões. Essas técnicas podem melhorar a legibilidade e a correção do código. Para obter uma discussão completa sobre todos os padrões que você pode aplicar, consulte o artigo sobre [padrões](#) na referência de linguagem.

## Verificações nulas

Um dos cenários mais comuns para correspondência de padrões é garantir que os valores não sejam `null`. Você pode testar e converter um tipo de valor anulável em seu tipo subjacente durante o teste de `null` usando o seguinte exemplo:

```
C#  
  
int? maybe = 12;  
  
if (maybe is int number)  
{  
    Console.WriteLine($"The nullable int 'maybe' has the value {number}");  
}  
else  
{  
    Console.WriteLine("The nullable int 'maybe' doesn't hold a value");  
}
```

O código anterior é um [padrão de declaração](#) para testar o tipo da variável e atribuí-lo a uma nova variável. As regras de idioma tornam essa técnica mais segura do que muitas outras. A variável `number` só é acessível e atribuída na parte verdadeira da cláusula `if`. Se você tentar acessá-la em outro lugar, na cláusula `else` ou após o bloco `if`, o

compilador emitirá um erro. Em segundo lugar, como você não está usando o operador `==`, esse padrão funciona quando um tipo sobrecarrega o operador `==`. Isso o torna uma maneira ideal de verificar os valores de referência nulos, adicionando o padrão `not`:

C#

```
string? message = "This is not the null string";

if (message is not null)
{
    Console.WriteLine(message);
}
```

O exemplo anterior usou um *padrão constante* para comparar a variável com `null`. `not` é um *padrão lógico* que corresponde quando o padrão negado não corresponde.

## Testes de tipo

Outro uso comum para correspondência de padrões é testar uma variável para ver se ela corresponde a um determinado tipo. Por exemplo, o código a seguir testa se uma variável não é nula e implementa a interface `System.Collections.Generic.IList<T>`. Se isso acontecer, ele usará a propriedade `ICollection<T>.Count` nessa lista para localizar o índice do meio. O padrão de declaração não corresponde a um valor `null`, independentemente do tipo de tempo de compilação da variável. O código abaixo protege contra `null` e um tipo que não implementa `IList`.

C#

```
public static T MidPoint<T>(IEnumerable<T> sequence)
{
    if (sequence is IList<T> list)
    {
        return list[list.Count / 2];
    }
    else if (sequence is null)
    {
        throw new ArgumentNullException(nameof(sequence), "Sequence can't be
null.");
    }
    else
    {
        int halfLength = sequence.Count() / 2 - 1;
        if (halfLength < 0) halfLength = 0;
        return sequence.Skip(halfLength).First();
    }
}
```

Os mesmos testes podem ser aplicados em uma expressão `switch` para testar uma variável em vários tipos diferentes. Você pode usar essas informações para criar algoritmos melhores com base no tipo em tempo de execução específico.

## Comparar valores discretos

Você também pode testar uma variável para encontrar uma correspondência em valores específicos. O código a seguir mostra um exemplo em que você testa um valor em relação a todos os valores possíveis declarados em uma enumeração:

C#

```
public State PerformOperation(Operation command) =>
    command switch
    {
        Operation.SystemTest => RunDiagnostics(),
        Operation.Start => StartSystem(),
        Operation.Stop => StopSystem(),
        Operation.Reset => ResetToReady(),
        _ => throw new ArgumentException("Invalid enum value for command",
            nameof(command)),
    };
```

O exemplo anterior demonstra uma expedição de método com base no valor de uma enumeração. O caso `_` final é um *padrão de descarte* que corresponde a todos os valores. Ele identifica quaisquer condições de erro em que o valor não corresponda a um dos valores `enum` definidos. Se você omitir esse braço `switch`, o compilador avisará que você não lidou com todos os valores de entrada possíveis. Em tempo de execução, a expressão `switch` lançará uma exceção se o objeto que está sendo examinado não corresponder a nenhum braço `switch`. Você pode usar constantes numéricas em vez de um conjunto de valores de enumeração. Você também pode usar essa técnica semelhante para valores de cadeia de caracteres constantes que representam os comandos:

C#

```
public State PerformOperation(string command) =>
    command switch
    {
        "SystemTest" => RunDiagnostics(),
        "Start" => StartSystem(),
        "Stop" => StopSystem(),
        "Reset" => ResetToReady(),
        _ => throw new ArgumentException("Invalid string value for command",
```

```
nameof(command)),  
};
```

O exemplo anterior mostra o mesmo algoritmo, mas usa valores de cadeia de caracteres em vez de uma enumeração. Você usaria esse cenário se seu aplicativo respondesse a comandos de texto em vez de um formato de dados regular. A partir do C# 11, você também pode usar um `Span<char>` ou `ReadOnlySpan<char>` para testar valores de cadeia de caracteres constantes, conforme mostrado no exemplo a seguir:

C#

```
public State PerformOperation(ReadOnlySpan<char> command) =>  
    command switch  
    {  
        "SystemTest" => RunDiagnostics(),  
        "Start" => StartSystem(),  
        "Stop" => StopSystem(),  
        "Reset" => ResetToReady(),  
        _ => throw new ArgumentException("Invalid string value for command",  
            nameof(command)),  
    };
```

Em todos esses exemplos, o *padrão de descarte* garante que você identifique todas as entradas. O compilador ajuda você a garantir que todos os valores de entrada possíveis sejam identificados.

## Padrões relacionais

Os *padrões relacionais* podem ser usados para testar como um valor se compara às constantes. Por exemplo, o código a seguir retorna o estado da água com base na temperatura em Fahrenheit:

C#

```
string WaterState(int tempInFahrenheit) =>  
    tempInFahrenheit switch  
    {  
        (> 32) and (< 212) => "liquid",  
        < 32 => "solid",  
        > 212 => "gas",  
        32 => "solid/liquid transition",  
        212 => "liquid / gas transition",  
    };
```

O código anterior também demonstra o `and`*padrão lógico* conjuntivo para verificar se ambos os padrões relacionais correspondem. Você também pode usar um padrão disjuntivo `or` para verificar se um dos padrões corresponde. Os dois padrões relacionais ficam entre parênteses, que podem ser usados em qualquer padrão para maior clareza. Os dois últimos braços switch identificam os casos para o ponto de fusão e o ponto de ebulação. Sem esses dois braços, o compilador avisa que sua lógica não cobre todas as entradas possíveis.

O código anterior também demonstra outro recurso importante que o compilador fornece para expressões de correspondência de padrões: o compilador avisa se você não identifica todos os valores de entrada. O compilador também emitirá um aviso se um braço switch já tiver sido identificado por um braço switch anterior. Isso lhe dá liberdade para refatorar e reordenar as expressões de alternância. Outra maneira de gravar a mesma expressão pode ser:

C#

```
string WaterState2(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        < 32 => "solid",
        32 => "solid/liquid transition",
        < 212 => "liquid",
        212 => "liquid / gas transition",
        _ => "gas",
    };
}
```

A principal lição nesse caso, e em qualquer outra refatoração ou reordenação, é que o compilador valida que todas as entradas foram abordadas.

## Várias entradas

Todos os padrões vistos até agora foram para verificar uma entrada. Você pode gravar padrões que examinam várias propriedades de um objeto. Considere o registro `Order` a seguir:

C#

```
public record Order(int Items, decimal Cost);
```

O tipo de registro posicional anterior declara dois membros em posições explícitas. Primeiro, `Items` aparece e, em seguida, o `Cost` da ordem. Para saber mais, confira [Registros](#).

O código a seguir examina o número de itens e o valor de uma ordem para calcular um preço com desconto:

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        { Items: > 10, Cost: > 1000.00m } => 0.10m,
        { Items: > 5, Cost: > 500.00m } => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't
calculate discount on null order"),
        var someObject => 0m,
    };
}
```

Os dois primeiros braços examinam duas propriedades do `order`. O terceiro examina apenas o custo. O próximo verifica `null` e o final corresponde a qualquer outro valor. Se o tipo `Order` definir um método `Deconstruct` adequado, você poderá omitir os nomes de propriedade do padrão e usar a desconstrução para examinar as propriedades:

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        ( > 10, > 1000.00m ) => 0.10m,
        ( > 5, > 50.00m ) => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't
calculate discount on null order"),
        var someObject => 0m,
    };
}
```

O código anterior demonstra o *padrão posicional* em que as propriedades são desconstruídas para a expressão.

## Padrões de lista

Você pode verificar os elementos em uma lista ou uma matriz usando um *padrão de lista*. Um *padrão de lista* fornece um meio para aplicar um padrão a qualquer elemento de uma sequência. Além disso, você pode aplicar o *padrão de descarte* (`_`) para corresponder a qualquer elemento ou aplicar um *padrão de fatia* para corresponder a zero ou mais elementos.

Os padrões de lista são uma ferramenta valiosa quando os dados não seguem uma estrutura regular. Você pode usar a correspondência de padrões para testar a forma e os valores dos dados em vez de transformá-los em um conjunto de objetos.

Considere o seguinte trecho de um arquivo de texto que contém transações bancárias:

Saída			
04-01-2020,	DEPOSIT,	Initial deposit,	2250.00
04-15-2020,	DEPOSIT,	Refund,	125.65
04-18-2020,	DEPOSIT,	Paycheck,	825.65
04-22-2020,	WITHDRAWAL,	Debit, Groceries,	255.73
05-01-2020,	WITHDRAWAL,	#1102, Rent, apt,	2100.00
05-02-2020,	INTEREST,		0.65
05-07-2020,	WITHDRAWAL,	Debit, Movies,	12.57
04-15-2020,	FEE,		5.55

É um formato CSV, mas algumas das linhas têm mais colunas do que outras. Pior ainda para o processamento, uma coluna do tipo `WITHDRAWAL` tem texto gerado pelo usuário e pode conter uma vírgula no texto. Um *padrão de lista* que inclui o *padrão de descarte*, o *padrão constante* e o *padrão var* para capturar os dados de processos de valor neste formato:

```
C#  
  
decimal balance = 0m;  
foreach (string[] transaction in ReadRecords())  
{  
    balance += transaction switch  
    {  
        [_, "DEPOSIT", _, var amount] => decimal.Parse(amount),  
        [_, "WITHDRAWAL", ..., var amount] => -decimal.Parse(amount),  
        [_, "INTEREST", var amount] => decimal.Parse(amount),  
        [_, "FEE", var fee] => -decimal.Parse(fee),  
        _ => throw new  
            InvalidOperationException($"Record {string.Join(", ", transaction)} is not  
            in the expected format!"),  
    };  
    Console.WriteLine($"Record: {string.Join(", ", transaction)}, New  
balance: {balance:C}");  
}
```

O exemplo anterior usa uma matriz de cadeia de caracteres, em que cada elemento é um campo na linha. As teclas de expressão `switch` no segundo campo, que determina o tipo de transação, e o número de colunas restantes. Cada linha garante que os dados estão no formato correto. O padrão de descarte (`_`) ignora o primeiro campo, com a data da transação. O segundo campo corresponde ao tipo de transação. As

correspondências de elemento restantes pulam para o campo com a quantidade. A partida final usa o padrão *var* para capturar a representação de cadeia de caracteres do valor. A expressão calcula o valor a ser adicionado ou subtraído do saldo.

*Os padrões de lista* permitem que você corresponda na forma de uma sequência de elementos de dados. Você usa os padrões de *descarte* e *fatia* para corresponder ao local dos elementos. Você usa outros padrões para corresponder às características sobre elementos individuais.

Este artigo fez um tour pelos tipos de código que você pode gravar com correspondência de padrões em C#. Os artigos a seguir mostram mais exemplos de uso dos padrões em cenários e o vocabulário completo dos padrões disponíveis para uso.

## Confira também

- Usar padrões correspondentes para evitar a verificação 'is' seguida por uma conversão (regras de estilo IDE0020 e IDE0038)
- Exploração: usar a correspondência de padrões para criar seu comportamento de classe para obter um código melhor
- Tutorial: Usar padrões correspondentes para criar algoritmos controlados por tipo e controlados por dados
- Referência: Correspondência de padrões

# Descartes – conceitos básicos do C#

Artigo • 07/04/2023

Descartes são variáveis de espaço reservado intencionalmente não utilizadas no código do aplicativo. Descartes são equivalentes a variáveis não atribuídas; eles não têm um valor. Um descarte comunica uma intenção para o compilador e outras pessoas que leem seu código: você pretendia ignorar o resultado de uma expressão. Talvez você queira ignorar o resultado de uma expressão, um ou mais membros de uma expressão de tupla, um parâmetro `out` para um método ou o destino de uma expressão de correspondência de padrões.

Os descartes tornam a intenção do seu código clara. Um descarte indica que nosso código nunca usa a variável. Eles aprimoram sua legibilidade e manutenção.

Você indica que uma variável é um descarte atribuindo a ela o sublinhado (`_`) como seu nome. Por exemplo, a chamada de método a seguir retorna uma tupla na qual o primeiro e o segundo valores são descartes. `area` é uma variável declarada anteriormente definida como o terceiro componente retornado por `GetCityInformation`:

```
C#  
(_, _, area) = city.GetCityInformation(cityName);
```

A partir do C# 9.0, você pode usar descartes para especificar parâmetros de entrada não utilizados de uma expressão lambda. Para obter mais informações, consulte a seção [Parâmetros de entrada de uma expressão lambda](#) do artigo [Expressões lambda](#).

Quando `_` é um descarte válido, tentar recuperar seu valor ou usá-lo em uma operação de atribuição gerará o erro do compilador CS0103, "O nome '`_`' não existe no contexto atual". Esse erro ocorre porque não há um valor atribuído a `_`, e pode não haver nem mesmo um local de armazenamento atribuído a ela. Se ela fosse uma variável real, você não poderia descartar mais de um valor, tal como ocorreu no exemplo anterior.

## Desconstrução de objeto e de tupla

Descartes são úteis para trabalhar com tuplas quando seu código de aplicativo usa alguns elementos da tupla, mas ignora outros. Por exemplo, o método `QueryCityDataForYears` a seguir retorna uma tupla de com o nome de uma cidade, sua área, um ano, a população da cidade nesse ano, um segundo ano e população da cidade nesse segundo ano. O exemplo mostra a alteração na população entre esses dois

anos. Entre os dados disponíveis da tupla, não estamos preocupados com a área da cidade e sabemos o nome da cidade e as duas datas em tempo de design. Como resultado, estamos interessados apenas nos dois valores de população armazenados na tupla e podemos lidar com seus valores restantes como descartes.

C#

```
var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");

static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int year2)
{
    int population1 = 0, population2 = 0;
    double area = 0;

    if (name == "New York City")
    {
        area = 468.48;
        if (year1 == 1960)
        {
            population1 = 7781984;
        }
        if (year2 == 2010)
        {
            population2 = 8175133;
        }
        return (name, area, year1, population1, year2, population2);
    }

    return ("", 0, 0, 0, 0, 0);
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Para obter mais informações sobre desconstruir tuplas com descartes, consulte [Desconstruindo tuplas e outros tipos](#).

O método `Deconstruct` de uma classe, estrutura ou interface também permite que você recupere e decomponha um conjunto específico de dados de um objeto. Você poderá usar descartes quando estiver interessado em trabalhar com apenas um subconjunto dos valores desconstruídos. O exemplo a seguir desconstrói um objeto `Person` em quatro cadeias de caracteres (os nomes e sobrenomes, a cidade e o estado), mas descarta o sobrenome e o estado.

C#

```
using System;

namespace Discards
{
    public class Person
    {
        public string FirstName { get; set; }
        public string MiddleName { get; set; }
        public string LastName { get; set; }
        public string City { get; set; }
        public string State { get; set; }

        public Person(string fname, string mname, string lname,
                      string cityName, string stateName)
        {
            FirstName = fname;
            MiddleName = mname;
            LastName = lname;
            City = cityName;
            State = stateName;
        }

        // Return the first and last name.
        public void Deconstruct(out string fname, out string lname)
        {
            fname = FirstName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string mname, out
string lname)
        {
            fname = FirstName;
            mname = MiddleName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string lname,
                           out string city, out string state)
        {
            fname = FirstName;
            lname = LastName;
            city = City;
            state = State;
        }
    }
    class Example
    {
        public static void Main()
        {
            var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

            // Deconstruct the person object.
            var (fName, _, city, _) = p;
```

```
        Console.WriteLine($"Hello {fName} of {city}!");
        // The example displays the following output:
        //      Hello John of Boston!
    }
}
}
```

Para obter mais informações sobre desconstruir tipos definidos pelo usuário com descartes, consulte [Desconstruindo tuplas e outros tipos](#).

## Correspondência de padrões com `switch`

O *padrão de descarte* pode ser usado na correspondência de padrões com a expressão `switch`. Toda expressão, incluindo `null`, sempre corresponde ao padrão de descarte.

O exemplo a seguir define um método `ProvidesFormatInfo` que usa uma expressão `switch` para determinar se um objeto fornece uma implementação de `IFormatProvider` e testa se o objeto é `null`. Ele também usa o padrão de descarte para manipular objetos não nulos de qualquer outro tipo.

C#

```
object?[] objects = { CultureInfo.CurrentCulture,
                      CultureInfo.CurrentCulture.DateTimeFormat,
                      CultureInfo.CurrentCulture.NumberFormat,
                      new ArgumentException(), null };
foreach (var obj in objects)
    ProvidesFormatInfo(obj);

static void ProvidesFormatInfo(object? obj) =>
    Console.WriteLine(obj switch
    {
        IFormatProvider fmt => $"{fmt.GetType()} object",
        null => "A null object reference: Its use could result in a
NullReferenceException",
        _ => "Some object type without format information"
    });
// The example displays the following output:
//      System.Globalization.CultureInfo object
//      System.Globalization.DateTimeFormatInfo object
//      System.Globalization.NumberFormatInfo object
//      Some object type without format information
//      A null object reference: Its use could result in a
NullReferenceException
```

## Chamadas para métodos com parâmetros `out`

Ao chamar o método `Deconstruct` para desconstruir um tipo definido pelo usuário (uma instância de uma classe, estrutura ou interface), você pode descartar os valores de argumentos `out` individuais. Mas você também pode descartar o valor de argumentos `out` ao chamar qualquer método com um parâmetro `out`.

A exemplo a seguir chama o método `DateTime.TryParse(String, out DateTime)` para determinar se a representação de cadeia de caracteres de uma data é válida na cultura atual. Já que o exemplo está preocupado apenas em validar a cadeia de caracteres de data e não em analisá-lo para extrair a data, o argumento `out` para o método é um descarte.

C#

```
string[] dateStrings = {"05/01/2018 14:57:32.8", "2018-05-01 14:57:32.8",
                      "2018-05-01T14:57:32.8375298-04:00", "5/01/2018",
                      "5/01/2018 14:57:32.80 -07:00",
                      "1 May 2018 2:57:32.8 PM", "16-05-2018 1:00:32 PM",
                      "Fri, 15 May 2018 20:10:57 GMT" };
foreach (string dateString in dateStrings)
{
    if (DateTime.TryParse(dateString, out _))
        Console.WriteLine($"'{dateString}': valid");
    else
        Console.WriteLine($"'{dateString}': invalid");
}
// The example displays output like the following:
//      '05/01/2018 14:57:32.8': valid
//      '2018-05-01 14:57:32.8': valid
//      '2018-05-01T14:57:32.8375298-04:00': valid
//      '5/01/2018': valid
//      '5/01/2018 14:57:32.80 -07:00': valid
//      '1 May 2018 2:57:32.8 PM': valid
//      '16-05-2018 1:00:32 PM': invalid
//      'Fri, 15 May 2018 20:10:57 GMT': invalid
```

## Um descarte autônomo

Você pode usar um descarte autônomo para indicar qualquer variável que você opte por ignorar. Um uso típico é usar uma atribuição para garantir que um argumento não seja nulo. O código a seguir usa um descarte para forçar uma atribuição. O lado direito da atribuição usa o [operador de avaliação de nulo](#) para lançar um `System.ArgumentNullException` quando o argumento é `null`. O código não precisa do resultado da atribuição; portanto, ele é descartado. A expressão força uma verificação de nulo. O descarte esclarece sua intenção: o resultado da atribuição não é necessário ou usado.

C#

```
public static void Method(string arg)
{
    _ = arg ?? throw new ArgumentNullException(paramName: nameof(arg),
message: "arg can't be null");

    // Do work with arg.
}
```

O exemplo a seguir usa um descarte autônomo para ignorar o objeto `Task` retornado por uma operação assíncrona. A atribuição da tarefa tem o efeito de suprimir a exceção que a operação gera quando está prestes a ser concluída. Isso deixa clara sua intenção: você deseja descartar `Task` e ignorar todos os erros gerados a partir dessa operação assíncrona.

C#

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    _ = Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
}

// The example displays output like the following:
//      About to launch a task...
//      Completed looping operation...
//      Exiting after 5 second delay
```

Sem atribuir a tarefa a um descarte, o código a seguir gera um aviso do compilador:

C#

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    // CS4014: Because this call is not awaited, execution of the current
method continues before the call is completed.
    // Consider applying the 'await' operator to the result of the call.
    Task.Run(() =>
    {
        var iterations = 0;
```

```

        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
Console.WriteLine("Exiting after 5 second delay");

```

### ⚠ Observação

Se você executar um dos dois exemplos anteriores usando um depurador, o depurador interromperá o programa quando a exceção for lançada. Sem um depurador anexado, a exceção é silenciosamente ignorada em ambos os casos.

`_` também é um identificador válido. Quando usado fora de um contexto com suporte, `_` não é tratado como um descarte, mas como uma variável válida. Se um identificador chamado `_` já está no escopo, o uso de `_` como um descarte autônomo pode resultar em:

- A modificação acidental do valor da variável `_` no escopo atribuindo a ela o valor do descarte pretendido. Por exemplo:

C#

```

private static void ShowValue(int _)
{
    byte[] arr = { 0, 0, 1, 2 };
    _ = BitConverter.ToInt32(arr, 0);
    Console.WriteLine(_);
}
// The example displays the following output:
//      33619968

```

- Um erro do compilador por violação de segurança de tipo. Por exemplo:

C#

```

private static bool RoundTrips(int _)
{
    string value = _.ToString();
    int newValue = 0;
    _ = Int32.TryParse(value, out newValue);
    return _ == newValue;
}
// The example displays the following compiler error:
//      error CS0029: Cannot implicitly convert type 'bool' to 'int'

```

- Erro do compilador CS0136, "Um local ou um parâmetro denominado '\_' não pode ser declarado neste escopo porque esse nome é usado em um escopo delimitador de local para definir um local ou parâmetro." Por exemplo:

```
C#  
  
public void DoSomething(int _)  
{  
    var _ = GetValue(); // Error: cannot declare local _ when one is  
    already in scope  
}  
// The example displays the following compiler error:  
// error CS0136:  
//     A local or parameter named '_' cannot be declared in this  
// scope  
//     because that name is used in an enclosing local scope  
//     to define a local or parameter
```

## Confira também

- [Remover valor de expressão desnecessária \(regra de estilo IDE0058\)](#)
- [Remover atribuição de valor desnecessária \(regra de estilo IDE0059\)](#)
- [Remover parâmetro não usado \(regra de estilo IDE0060\)](#)
- [Desconstruindo tuplas e outros tipos](#)
- [Operador is](#)
- [Expressão switch](#)

# Desconstruindo tuplas e outros tipos

Artigo • 07/04/2023

Uma tupla fornece uma maneira leve de recuperar vários valores de uma chamada de método. Mas depois de recuperar a tupla, você precisa lidar com seus elementos individuais. Trabalhar elemento por elemento é incômodo, conforme mostra o exemplo a seguir. O método `QueryCityData` retorna uma tupla de três e cada um de seus elementos é atribuído a uma variável em uma operação separada.

C#

```
public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

        // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

Recuperar vários valores de propriedade e de campo de um objeto pode ser igualmente complicado: é preciso atribuir um valor de campo ou de propriedade a uma variável, membro por membro.

Você pode recuperar vários elementos de uma tupla ou recuperar vários valores de campo, propriedade e computação de um objeto em uma única operação *de desconstrução*. Para desconstruir uma tupla, você atribui os elementos dela a variáveis individuais. Quando você desconstrói um objeto, você atribui os elementos dela a variáveis individuais.

## Tuplas

O C# conta com suporte interno à desconstrução de tuplas, que permite que você descompacte todos os itens em uma tupla em uma única operação. A sintaxe geral para desconstruir uma tupla é semelhante à sintaxe para definir uma: coloque as variáveis para as quais cada elemento deve ser atribuído entre parênteses no lado esquerdo de uma instrução de atribuição. Por exemplo, a instrução a seguir atribui os elementos de uma tupla de quatro a quatro variáveis separadas:

```
C#
```

```
var (name, address, city, zip) = contact.GetAddressInfo();
```

Há três maneiras de desconstruir uma tupla:

- Você pode declarar explicitamente o tipo de cada campo dentro de parênteses. O exemplo a seguir usa essa abordagem para desconstruir a tupla de três retornada pelo método `QueryCityData`.

```
C#
```

```
public static void Main()
{
    (string city, int population, double area) = QueryCityData("New
    York City");

    // Do something with the data.
}
```

- Você pode usar a palavra-chave `var` de modo que o C# infera o tipo de cada variável. Você coloca a palavra-chave `var` fora dos parênteses. O exemplo a seguir usa a inferência de tipos ao desconstruir a tupla de três retornada pelo método `QueryCityData`.

```
C#
```

```
public static void Main()
{
    var (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

Você também pode usar a palavra-chave `var` individualmente com qualquer uma ou todas as declarações de variável dentro dos parênteses.

```
C#
```

```
public static void Main()
{
    (string city, var population, var area) = QueryCityData("New York
City");

    // Do something with the data.
}
```

Isso é difícil e não é recomendado.

- Por fim, você pode desconstruir a tupla em variáveis que já foram declaradas.

C#

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;
    double area = 144.8;

    (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- A partir do C# 10, você pode misturar declaração de variável e atribuição em uma desconstrução.

C#

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;

    (city, population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

Você não pode especificar um tipo específico fora dos parênteses, mesmo se todos os campos na tupla tiverem o mesmo tipo. Isso gera o erro do compilador CS8136, "O formulário de desconstrução 'var (...)’ não permite um tipo específico para 'var'.".

Você deve atribuir cada elemento da tupla a uma variável. Se você omitir qualquer elemento, o compilador gerará o erro CS8132, "Não é possível desconstruir uma tupla de 'x' elementos em 'y' variáveis".

# Elementos tupla com descartes

Geralmente, ao desconstruir uma tupla, você está interessado nos valores de apenas alguns elementos. Você pode aproveitar o suporte do C# para *descartes*, que são variáveis somente gravação cujos valores você escolheu ignorar. Um descarte é escolhido por um caractere de sublinhado ("\_") em uma atribuição. Você pode descartar tantos valores quantos desejar; todos são representados pelo descarte único, `_`.

O exemplo a seguir ilustra o uso de tuplas com descartes. O método

`QueryCityDataForYears` a seguir retorna uma tupla de seis com o nome de uma cidade, sua área, um ano, a população da cidade nesse ano, um segundo ano e população da cidade nesse segundo ano. O exemplo mostra a alteração na população entre esses dois anos. Entre os dados disponíveis da tupla, não estamos preocupados com a área da cidade e sabemos o nome da cidade e as duas datas em tempo de design. Como resultado, estamos interessados apenas nos dois valores de população armazenados na tupla e podemos lidar com seus valores restantes como descartes.

C#

```
using System;

public class ExampleDiscard
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York
City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 -
pop1:N0}");
    }

    private static (string, double, int, int, int)
QueryCityDataForYears(string name, int year1, int year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
        }
    }
}
```

```
        return (name, area, year1, population1, year2, population2);
    }

    return ("", 0, 0, 0, 0, 0);
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

## Tipos definidos pelo usuário

O C# não oferece suporte interno para desconstruir tipos não tupla diferentes dos [record](#) tipos e [DictionaryEntry](#). No entanto, como o autor de uma classe, um struct ou uma interface, você pode permitir instâncias do tipo a ser desconstruído implementando um ou mais métodos `Deconstruct`. O método retorna void e cada valor a ser desconstruído é indicado por um parâmetro `out` na assinatura do método. Por exemplo, o método `Deconstruct` a seguir de uma classe `Person` retorna o nome, o segundo nome e o sobrenome:

C#

```
public void Deconstruct(out string fname, out string mname, out string lname)
```

Em seguida, você pode desconstruir uma instância da classe `Person` denominada `p` com uma atribuição semelhante à seguinte:

C#

```
var (fName, mName, lName) = p;
```

O exemplo a seguir sobrecarrega o método `Deconstruct` para retornar várias combinações de propriedades de um objeto `Person`. As sobrecargas individuais retornam:

- Um nome e um sobrenome.
- Um nome, nome do meio e sobrenome.
- Um nome, um sobrenome, um nome de cidade e um nome de estado.

C#

```
using System;

public class Person
```

```
{  
    public string FirstName { get; set; }  
    public string MiddleName { get; set; }  
    public string LastName { get; set; }  
    public string City { get; set; }  
    public string State { get; set; }  
  
    public Person(string fname, string mname, string lname,  
                 string cityName, string stateName)  
    {  
        FirstName = fname;  
        MiddleName = mname;  
        LastName = lname;  
        City = cityName;  
        State = stateName;  
    }  
  
    // Return the first and last name.  
    public void Deconstruct(out string fname, out string lname)  
    {  
        fname = FirstName;  
        lname = LastName;  
    }  
  
    public void Deconstruct(out string fname, out string mname, out string  
                           lname)  
    {  
        fname = FirstName;  
        mname = MiddleName;  
        lname = LastName;  
    }  
  
    public void Deconstruct(out string fname, out string lname,  
                           out string city, out string state)  
    {  
        fname = FirstName;  
        lname = LastName;  
        city = City;  
        state = State;  
    }  
}  
  
public class ExampleClassDeconstruction  
{  
    public static void Main()  
    {  
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");  
  
        // Deconstruct the person object.  
        var (fName, lName, city, state) = p;  
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");  
    }  
}  
// The example displays the following output:  
//      Hello John Adams of Boston, MA!
```

Vários métodos `Deconstruct` com o mesmo número de parâmetros são ambíguos. Você deve ter cuidado ao definir métodos `Deconstruct` com diferentes números de parâmetros ou "aridade". Métodos `Deconstruct` com o mesmo número de parâmetros não podem ser distinguidos durante a resolução de sobrecarga.

## Tipo definido pelo usuário com descartes

Assim como você faria com [tuplas](#), você pode usar descartes para ignorar os itens selecionados retornados por um método `Deconstruct`. Cada descarte é definido por uma variável chamada `_`, sendo que uma única operação de desconstrução pode incluir vários descartes.

O exemplo a seguir desconstrói um objeto `Person` em quatro cadeias de caracteres (os nomes e sobrenomes, a cidade e o estado), mas descarta o sobrenome e o estado.

C#

```
// Deconstruct the person object.  
var (fName, _, city, _) = p;  
Console.WriteLine($"Hello {fName} of {city}!");  
// The example displays the following output:  
//      Hello John of Boston!
```

## Métodos de extensão para tipos definidos pelo usuário

Se você não criar uma classe, struct ou interface, você ainda poderá decompor objetos desse tipo implementando um ou mais `Deconstruct` [métodos de extensão](#) para retornar os valores nos quais você estiver interessado.

O exemplo a seguir define dois métodos de extensão `Deconstruct` para a classe `System.Reflection.PropertyInfo`. O primeiro retorna um conjunto de valores que indicam as características da propriedade, incluindo seu tipo, se ela é estática ou instância, se ela é somente leitura e se é indexada. O segundo indica a acessibilidade da propriedade. Já que a acessibilidade dos acessadores get e set pode ser diferente, valores booleanos indicam se a propriedade acessadores get e set separados e, em caso afirmativo, se eles têm a mesma acessibilidade. Se houver apenas um acessador ou ambos os acessadores get e set têm a mesma acessibilidade, a variável `access` indica a acessibilidade da

propriedade como um todo. Caso contrário, a acessibilidade dos acessadores get e set é indicada pelas variáveis `getAccess` e `setAccess`.

C#

```
using System;
using System.Collections.Generic;
using System.Reflection;

public static class ReflectionExtensions
{
    public static void Deconstruct(this PropertyInfo p, out bool isStatic,
                                  out bool isReadOnly, out bool isIndexed,
                                  out Type propertyType)
    {
        var getter = p.GetMethod();

        // Is the property read-only?
        isReadOnly = ! p.CanWrite;

        // Is the property instance or static?
        isStatic = getter.IsStatic;

        // Is the property indexed?
        isIndexed = p.GetIndexParameters().Length > 0;

        // Get the property type.
        propertyType = p.PropertyType;
    }

    public static void Deconstruct(this PropertyInfo p, out bool
hasGetAndSet,
                                out bool sameAccess, out string access,
                                out string getAccess, out string
setAccess)
    {
        hasGetAndSet = sameAccess = false;
        string getAccessTemp = null;
        string setAccessTemp = null;

        MethodInfo getter = null;
        if (p.CanRead)
            getter = p.GetMethod();

        MethodInfo setter = null;
        if (p.CanWrite)
            setter = p.SetMethod();

        if (setter != null && getter != null)
            hasGetAndSet = true;

        if (getter != null)
        {
            if (getter.IsPublic)
```

```

        getAccessTemp = "public";
    else if (getter.IsPrivate)
        getAccessTemp = "private";
    else if (getter.IsAssembly)
        getAccessTemp = "internal";
    else if (getter.IsFamily)
        getAccessTemp = "protected";
    else if (getter.IsFamilyOrAssembly)
        getAccessTemp = "protected internal";
}

if (setter != null)
{
    if (setter.IsPublic)
        setAccessTemp = "public";
    else if (setter.IsPrivate)
        setAccessTemp = "private";
    else if (setter.IsAssembly)
        setAccessTemp = "internal";
    else if (setter.IsFamily)
        setAccessTemp = "protected";
    else if (setter.IsFamilyOrAssembly)
        setAccessTemp = "protected internal";
}

// Are the accessibility of the getter and setter the same?
if (setAccessTemp == getAccessTemp)
{
    sameAccess = true;
    access = getAccessTemp;
    getAccess = setAccess = String.Empty;
}
else
{
    access = null;
    getAccess = getAccessTemp;
    setAccess = setAccessTemp;
}
}

public class ExampleExtension
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"\\nThe {dateType.FullName}.{prop.Name}
property:");
        Console.WriteLine($"    PropertyType: {propType.Name}");
        Console.WriteLine($"    Static:      {isStatic}");
        Console.WriteLine($"    Read-only:   {isRO}");
        Console.WriteLine($"    Indexed:     {isIndexed}");
    }
}

```

```

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",
                                    BindingFlags.Public |
        BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Static);
        var (hasGetAndSet, sameAccess, accessibility, getAccessibility,
setAccessibility) = prop;
        Console.WriteLine($"\\nAccessibility of the {listType.FullName}.

{prop.Name} property: ");

        if (!hasGetAndSet | sameAccess)
        {
            Console.WriteLine(accessibility);
        }
        else
        {
            Console.WriteLine($"\\n    The get accessor: {getAccessibility}");
            Console.WriteLine($"    The set accessor: {setAccessibility}");
        }
    }
}

// The example displays the following output:
//     The System.DateTime.Now property:
//         PropertyType: DateTime
//         Static:      True
//         Read-only:   True
//         Indexed:    False
//
//     Accessibility of the System.Collections.Generic.List`1.Item
property: public

```

## Método de extensão para tipos de sistema

Alguns tipos de sistema fornecem o método `Deconstruct` como uma conveniência. Por exemplo, o tipo `System.Collections.Generic.KeyValuePair<TKey,TValue>` fornece essa funcionalidade. Quando você está iterando em `System.Collections.Generic.Dictionary<TKey,TValue>`, cada elemento é um `KeyValuePair<TKey, TValue>` e pode ser desconstruído. Considere o seguinte exemplo:

C#

```

Dictionary<string, int> snapshotCommitMap =
new(StringComparer.OrdinalIgnoreCase)
{
    ["https://github.com/dotnet/docs"] = 16_465,
    ["https://github.com/dotnet/runtime"] = 114_223,
    ["https://github.com/dotnet/installer"] = 22_436,
    ["https://github.com/dotnet/roslyn"] = 79_484,
    ["https://github.com/dotnet/aspnetcore"] = 48_386
};

```

```
foreach (var (repo, commitCount) in snapshotCommitMap)
{
    Console.WriteLine(
        $"The {repo} repository had {commitCount:N0} commits as of November
10th, 2021.");
}
```

Você pode adicionar um método `Deconstruct` aos tipos de sistema que não têm um. Considere o seguinte método de extensão:

C#

```
public static class NullableExtensions
{
    public static void Deconstruct<T>(
        this T? nullable,
        out bool hasValue,
        out T value) where T : struct
    {
        hasValue = nullable.HasValue;
        value = nullable.GetValueOrDefault();
    }
}
```

Esse método de extensão permite que todos os tipos `Nullable<T>` sejam desconstruídos em uma tupla de `(bool hasValue, T value)`. O exemplo a seguir mostra o código que usa este método de extensão:

C#

```
DateTime? questionableDateTime = default;
var (hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");

questionableDateTime = DateTime.Now;
(hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");

// Example outputs:
// { HasValue = False, Value = 1/1/0001 12:00:00 AM }
// { HasValue = True, Value = 11/10/2021 6:11:45 PM }
```

## Tipos record

Quando você declara um tipo de [registro](#) usando dois ou mais parâmetros posicionais, o compilador cria um método `Deconstruct` com um parâmetro `out` para cada parâmetro posicional na declaração `record`. Para obter mais informações, consulte [Sintaxe posicional para definição de propriedade](#) e [Comportamento de desconstrutor em registros derivados](#).

## Confira também

- [Declaração de variável de desconstrução \(regra de estilo IDE0042\)](#)
- [Descartes](#)
- [Tipos de tupla](#)

# Exceções e manipulação de exceções

Artigo • 15/02/2023

Os recursos de manipulação de exceção da linguagem C# ajudam você a lidar com quaisquer situações excepcionais ou inesperadas que ocorram quando um programa for executado. A manipulação de exceções usa as palavras-chave `try`, `catch` e `finally` para executar ações que talvez não sejam bem-sucedidas, lidar com falhas quando você decidir que é razoável fazê-lo e limpar recursos depois disso. Podem ser geradas exceções pelo CLR (Common Language Runtime), pelo .NET, por quaisquer bibliotecas de terceiros ou pelo código do aplicativo. As exceções são criadas usando a palavra-chave `throw`.

Em muitos casos, uma exceção pode ser lançada não por um método que seu código chamou diretamente, mas por outro método mais abaixo na pilha de chamadas.

Quando isso acontecer, o CLR desenrolará a pilha, em busca de um método com um bloco `catch` para o tipo de exceção específico e executará o primeiro o bloco `catch` desse tipo que encontrar. Se ele não encontrar um bloco `catch` apropriado na pilha de chamadas, ele encerrará o processo e exibirá uma mensagem para o usuário.

Neste exemplo, um método testa a divisão por zero e captura o erro. Sem a manipulação de exceção, esse programa encerraria com um `DivideByZeroException` não resolvido.

C#

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
    }
}
```

```
        catch (DivideByZeroException)
    {
        Console.WriteLine("Attempted divide by zero.");
    }
}
```

## Visão geral sobre exceções

As exceções têm as seguintes propriedades:

- As exceções são tipos que derivam, por fim, de `System.Exception`.
- Use um bloco `try` nas instruções que podem lançar exceções.
- Quando ocorre uma exceção no bloco `try`, o fluxo de controle vai para o primeiro manipulador de exceção associada que está presente em qualquer lugar na pilha de chamadas. No C#, a palavra-chave `catch` é usada para definir um manipulador de exceção.
- Se nenhum manipulador de exceção para uma determinada exceção estiver presente, o programa interromperá a execução com uma mensagem de erro.
- Não capture uma exceção, a menos que você possa manipulá-la e deixar o aplicativo em um estado conhecido. Se você capturar `System.Exception`, relance-o usando a palavra-chave `throw` no final do bloco `catch`.
- Se um bloco `catch` define uma variável de exceção, você pode usá-lo para obter mais informações sobre o tipo de exceção que ocorreu.
- As exceções podem ser geradas explicitamente por um programa usando a palavra-chave `throw`.
- Os objetos de exceção contêm informações detalhadas sobre o erro, como o estado da pilha de chamadas e uma descrição de texto do erro.
- O código em um bloco `finally` será executado mesmo que seja lançada uma exceção. Use um bloco `finally` para liberar recursos, por exemplo, para fechar todos os fluxos ou arquivos que foram abertos no bloco `try`.
- As exceções gerenciadas no .NET são implementadas sobre o mecanismo de manipulação de exceções estruturadas do Win32. Para obter mais informações, consulte [Manipulação de exceções estruturadas \(C/C++\)](#) e [Curso rápido sobre a manipulação de exceções estruturadas do Win32](#).

## Especificação da Linguagem C#

Para obter mais informações, veja [Exceções na Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [SystemException](#)
- [Palavras-chave do C#](#)
- [throw](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)
- [Exceções](#)

# Usar exceções

Artigo • 10/05/2023

No C#, os erros no programa em tempo de execução são propagados pelo programa usando um mecanismo chamado exceções. As exceções são geradas pelo código que encontra um erro e capturadas pelo código que pode corrigir o erro. As exceções podem ser geradas pelo tempo de execução do .NET ou pelo código em um programa. Uma vez que uma exceção é gerada, ela é propagada acima na pilha de chamadas até uma instrução `catch` para a exceção ser encontrada. As exceções não capturadas são tratadas por um manipulador de exceção genérico fornecido pelo sistema que exibe uma caixa de diálogo.

As exceções são representadas por classes derivadas de `Exception`. Essa classe identifica o tipo de exceção e contém propriedades que têm detalhes sobre a exceção. Gerar uma exceção envolve criar uma instância de uma classe derivada de exceção, opcionalmente configurar propriedades da exceção e, em seguida, gerar o objeto usando a palavra-chave `throw`. Por exemplo:

```
C#  
  
class CustomException : Exception  
{  
    public CustomException(string message)  
    {  
    }  
}  
private static void TestThrow()  
{  
    throw new CustomException("Custom exception in TestThrow()");  
}
```

Depois que uma exceção é gerada, o runtime verifica a instrução atual para ver se ela está dentro de um bloco `try`. Se estiver, todos os blocos `catch` associados ao bloco `try` serão verificados para ver se eles podem capturar a exceção. Os blocos `Catch` normalmente especificam os tipos de exceção. Se o tipo do bloco `catch` for do mesmo tipo que a exceção ou uma classe base da exceção, o bloco `catch` poderá manipular o método. Por exemplo:

```
C#  
  
try  
{  
    TestThrow();  
}
```

```
        catch (CustomException ex)
    {
        System.Console.WriteLine(ex.ToString());
}
```

Se a instrução que gera uma exceção não estiver dentro de um bloco `try` ou se o bloco `try` que o contém não tiver um bloco `catch` correspondente, o runtime verificará o método de chamada quanto a uma instrução `try` e blocos `catch`. O runtime continuará acima na pilha de chamada, pesquisando um bloco `catch` compatível. Depois que o bloco `catch` for localizado e executado, o controle será passado para a próxima instrução após aquele bloco `catch`.

Uma instrução `try` pode conter mais de um bloco `catch`. A primeira instrução `catch` que pode manipular a exceção é executado, todas as instruções `catch` posteriores, mesmo se forem compatíveis, são ignoradas. Ordenar blocos `catch` do mais específico (ou mais derivado) para o menos específico. Por exemplo:

C#

```
using System;
using System.IO;

namespace Exceptions
{
    public class CatchOrder
    {
        public static void Main()
        {
            try
            {
                using (var sw = new StreamWriter("./test.txt"))
                {
                    sw.WriteLine("Hello");
                }
            }
            // Put the more specific exceptions first.
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (IOException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```

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

Antes de o bloco `catch` ser executado, o runtime verifica se há blocos `finally`. Os blocos `Finally` permitem que o programador limpe qualquer estado ambíguo que pode ser deixado de um bloco `try` cancelado ou libere quaisquer recursos externos (como identificadores de gráfico, conexões de banco de dados ou fluxos de arquivo) sem esperar o coletor de lixo no runtime finalizar os objetos. Por exemplo:

C#

```
static void TestFinally()
{
    FileStream? file = null;
    //Change the path to something that works on your machine.
    FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise
        // IOException is thrown.
        file?.Close();
    }

    try
    {
        file = fileInfo.OpenWrite();
        Console.WriteLine("OpenWrite() succeeded");
    }
    catch (IOException)
    {
        Console.WriteLine("OpenWrite() failed");
    }
}
```

Se `WriteByte()` gerou uma exceção, o código no segundo bloco `try` que tentar reabrir o arquivo falhará se `file.Close()` não for chamado e o arquivo permanecerá bloqueado. Como os blocos `finally` são executados mesmo se uma exceção for gerada, o bloco `finally` no exemplo anterior permite que o arquivo seja fechado corretamente e ajuda a evitar um erro.

Se não for encontrado nenhum bloco `catch` compatível na pilha de chamadas após uma exceção ser gerada, ocorrerá uma das três coisas:

- Se a exceção estiver em um finalizador, o finalizador será anulado e o finalizador base, se houver, será chamado.
- Se a pilha de chamadas contiver um construtor estático ou um inicializador de campo estático, uma `TypeInitializationException` será gerada, com a exceção original atribuída à propriedade `InnerException` da nova exceção.
- Se o início do thread for atingido, o thread será encerrado.

# Manipulação de exceções (Guia de Programação em C#)

Artigo • 20/03/2023

Um bloco `try` é usado por programadores de C# para partitionar o código que pode ser afetado por uma exceção. Os blocos `catch` associados são usados para tratar qualquer exceção resultante. Um bloco `finally` contém código que será executado de uma exceção ser ou não ser lançada no bloco `try`, como a liberação de recursos que estão alocados no bloco `try`. Um bloco `try` exige um ou mais blocos `catch` associados ou um bloco `finally` ou ambos.

Os exemplos a seguir mostram uma instrução `try-catch`, uma instrução `try-finally` e um instrução `try-catch-finally`.

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

C#

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

C#

```
try
{
    // Code to try goes here.
}
```

```
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

Um bloco `try` sem um bloco `catch` ou `finally` causa um erro do compilador.

## Blocos catch

Um bloco `catch` pode especificar o tipo de exceção a ser capturado. A especificação de tipo é chamada de *filtro de exceção*. O tipo de exceção deve ser derivado de [Exception](#). Em geral, não especifique [Exception](#) como o filtro de exceção, a menos que você saiba como tratar todas as exceções que podem ser lançadas no bloco `try` ou incluiu uma instrução `throw` no final do seu bloco `catch`.

Vários blocos `catch` com filtros de exceção diferentes podem ser encadeados. Os blocos `catch` são avaliados de cima para baixo no seu código, mas somente um bloco `catch` será executado para cada exceção que é lançada. O primeiro bloco `catch` que especifica o tipo exato ou uma classe base da exceção lançada será executado. Se nenhum bloco `catch` especificar uma classe de exceção correspondente, um bloco `catch` que não tem nenhum tipo será selecionado, caso haja algum na instrução. É importante posicionar os blocos `catch` com as classes de exceção mais específicas (ou seja, os mais derivados) primeiro.

Capture exceções quando as seguintes condições forem verdadeiras:

- Você tem uma boa compreensão de porque a exceção seria lançada e pode implementar uma recuperação específica, como solicitar que o usuário insira um novo nome de arquivo, quando você capturar um objeto [FileNotFoundException](#).
- Você pode criar e lançar uma exceção nova e mais específica.

```
C#

int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
```

```
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

- Você deseja tratar parcialmente uma exceção antes de passá-la para mais tratamento. No exemplo a seguir, um bloco `catch` é usado para adicionar uma entrada a um log de erros antes de lançar novamente a exceção.

C#

```
try
{
    // Try to access a resource.
}
catch (UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

Você também pode especificar *filtros de exceção* para adicionar uma expressão booleana a uma cláusula `catch`. Filtros de exceção indicam que uma cláusula `catch` específica corresponde somente quando essa condição é verdadeira. No exemplo a seguir, ambas as cláusulas `catch` usam a mesma classe de exceção, mas uma condição extra é verificada para criar uma mensagem de erro diferente:

C#

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be greater than the array size.", e);
    }
}
```

Um filtro de exceção que sempre retorna `false` pode ser usado para examinar todas as exceções, mas não processá-las. Um uso típico é registrar exceções:

```
C#  
  
public static void Main()  
{  
    try  
    {  
        string? s = null;  
        Console.WriteLine(s.Length);  
    }  
    catch (Exception e) when (LogException(e))  
    {  
    }  
    Console.WriteLine("Exception must have been handled");  
}  
  
private static bool LogException(Exception e)  
{  
    Console.WriteLine($"\\tIn the log routine. Caught {e.GetType()}");  
    Console.WriteLine($"\\tMessage: {e.Message}");  
    return false;  
}
```

O método `LogException` sempre retorna `false`, nenhuma cláusula `catch` que usa esse filtro de exceção corresponde. A cláusula `catch` pode ser geral, usando `System.Exception`, e cláusulas posteriores podem processar classes de exceção mais específicas.

## Blocos Finally

Um bloco `finally` permite que você limpe as ações que são realizadas em um bloco `try`. Se estiver presente, o bloco `finally` será executado por último, depois do bloco `try` e de qualquer bloco `catch` de correspondência. Um bloco `finally` sempre é executado, independentemente de uma exceção ser lançada ou de um bloco `catch` correspondente ao tipo de exceção ser encontrado.

O bloco `finally` pode ser usado para liberar recursos, como fluxos de arquivos, conexões de banco de dados e identificadores de gráficos, sem esperar que o coletor de lixo no runtime finalize os objetos.

No exemplo a seguir, o bloco `finally` é usado para fechar um arquivo está aberto no bloco `try`. Observe que o estado do identificador de arquivo é selecionado antes do arquivo ser fechado. Se o bloco `try` não puder abrir o arquivo, o identificador de

arquivo ainda terá o valor `null` e o bloco `finally` não tentará fechá-lo. Em vez disso, se o arquivo for aberto com êxito no bloco `try`, o bloco `finally` fechará o arquivo aberto.

C#

```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xFF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

## Especificação da Linguagem C#

Para obter mais informações, veja [Exceções](#) e [A declaração try na Especificação da Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)
- [Instrução using](#)

# Criar e lançar exceções

Artigo • 07/04/2023

As exceções são usadas para indicar que ocorreu um erro durante a execução do programa. Objetos de exceção que descrevem um erro são criados e, em seguida, *lançados* com a palavra-chave `throw`. Então, o runtime procura o manipulador de exceção mais compatível.

Os programadores devem lançar exceções quando uma ou mais das seguintes condições forem verdadeiras:

- O método não pode concluir sua funcionalidade definida. Por exemplo, se um parâmetro para um método tem um valor inválido:

C#

```
static void CopyObject(SampleClass original)
{
    _ = original ?? throw new ArgumentException("Parameter cannot be
null", nameof(original));
}
```

- É feita uma chamada inadequada a um objeto, com base no estado do objeto. Um exemplo pode ser a tentativa de gravar em um arquivo somente leitura. Em casos em que um estado do objeto não permita uma operação, lance uma instância de `InvalidOperationException` ou um objeto com base em uma derivação dessa classe. O código a seguir é um exemplo de um método que gera um objeto `InvalidOperationException`:

C#

```
public class ProgramLog
{
    FileStream logFile = null!;
    public void OpenLog(FileInfo fileName, FileMode mode) { }

    public void WriteLog()
    {
        if (!logFile.CanWrite)
        {
            throw new InvalidOperationException("LogFile cannot be
read-only");
        }
        // Else write data to the log and return.
    }
}
```

- Quando um argumento para um método causa uma exceção. Nesse caso, a exceção original deve ser capturada e uma instância de [ArgumentException](#) deve ser criada. A exceção original deve ser passada para o construtor do [ArgumentException](#) como o parâmetro [InnerException](#):

C#

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentException(
            "Parameter index is out of range.", e);
    }
}
```

### ⚠ Observação

O exemplo acima tem fins ilustrativos. A validação de índice por meio de exceções é, na maioria dos casos, não é uma boa prática. As exceções devem ser reservadas para proteger contra condições excepcionais do programa, não para verificação de argumentos, como acima.

As exceções contêm uma propriedade chamada [StackTrace](#). Essa cadeia de caracteres contém o nome dos métodos na pilha de chamadas atual, junto com o nome de arquivo e o número de linha em que a exceção foi lançada para cada método. Um objeto [StackTrace](#) é criado automaticamente pelo CLR (Common Language Runtime) no ponto da instrução `throw`, de modo que as exceções devem ser lançadas do ponto em que o rastreamento de pilha deve começar.

Todas as exceções contêm uma propriedade chamada [Message](#). Essa cadeia de caracteres deve ser definida para explicar o motivo da exceção. As informações que são sensíveis à segurança não devem ser colocadas no texto da mensagem. Além [Message](#), [ArgumentException](#) contém uma propriedade chamada [ParamName](#) que deve ser definida como o nome do argumento que causou a exceção a ser lançada. Em um setter de propriedade, [ParamName](#) deve ser definido como `value`.

Os métodos públicos e protegidos lançam exceções sempre que não puderem concluir suas funções pretendidas. A classe de exceção lançada é a exceção mais específica

disponível que se adapta às condições do erro. Essas exceções devem ser documentadas como parte da funcionalidade de classe e as classes derivadas ou as atualizações da classe original devem manter o mesmo comportamento para compatibilidade com versões anteriores.

## Coisas a serem evitadas ao lançar exceções

A lista a seguir identifica as práticas a serem evitadas ao lançar exceções:

- Não use exceções para alterar o fluxo de um programa como parte da execução normal. Use exceções para relatar e lidar com condições de erro.
- As exceções não devem ser retornadas como um valor retornado ou um parâmetro em vez de serem lançadas.
- Não lance [System.Exception](#), [System.SystemException](#), [System.NullReferenceException](#) ou [System.IndexOutOfRangeException](#) intencionalmente de seu próprio código-fonte.
- Não crie exceções que podem ser lançadas no modo de depuração, mas não no modo de versão. Em vez disso, use o Debug Assert para identificar erros em tempo de execução durante a fase de desenvolvimento.

## Definindo classes de exceção

Os programas podem lançar uma classe de exceção predefinida no namespace [System](#) (exceto quando observado anteriormente) ou criar suas próprias classes de exceção, derivando de [Exception](#). As classes derivadas devem definir pelo menos três construtores: um construtor sem parâmetros, um que define a propriedade de mensagem e um que define as propriedades [Message](#) e [InnerException](#). Por exemplo:

C#

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) :
        base(message, inner) { }
}
```

Adicione novas propriedades à classe de exceção quando os dados que elas fornecem forem úteis para resolver a exceção. Se forem adicionadas novas propriedades à classe

de exceção derivada, `ToString()` deverá ser substituído para retornar as informações adicionadas.

## Especificação da Linguagem C#

Para obter mais informações, veja [Exceções](#) e [A declaração throw](#) na [Especificação da Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Hierarquia de exceções](#)

# Exceções geradas pelo compilador

Artigo • 01/06/2023

Algumas exceções são geradas automaticamente pelo runtime do .NET quando operações básicas falham. Essas exceções e suas condições de erro são listadas na tabela a seguir.

Exceção	Descrição
ArithmeticsException	Uma classe base para exceções que ocorrem durante operações aritméticas, tais como <a href="#">DivideByZeroException</a> e <a href="#">OverflowException</a> .
ArrayTypeMismatchException	Gerada quando uma matriz não pode armazenar determinado elemento porque o tipo real do elemento é incompatível com o tipo real da matriz.
DivideByZeroException	Lançada quando é feita uma tentativa de dividir um valor inteiro por zero.
IndexOutOfRangeException	Lançada quando é feita uma tentativa de indexar uma matriz quando o índice é menor que zero ou fora dos limites da matriz.
InvalidCastException	Gerada quando uma conversão explícita de um tipo base para uma interface ou um tipo derivado falha em tempo de execução.
NullReferenceException	Gerada quando há uma tentativa de fazer referência a um objeto cujo valor é <code>null</code> .
OutOfMemoryException	Lançada quando uma tentativa de alocar memória usando o operador <code>new</code> falha. Essa exceção indica que a memória disponível para o Common Language Runtime está esgotada.
OverflowException	Lançada quando uma operação aritmética em um contexto <code>checked</code> estoura.
StackOverflowException	Lançada quando a pilha de execução acaba tendo muitas chamadas de método pendentes, normalmente indica uma recursão muito profunda ou infinita.
TypeInitializationException	Lançada quando um construtor estático lança uma exceção e não há nenhuma cláusula <code>catch</code> compatível para capturá-la.

## Confira também

- [Instruções para manipulação de exceções](#)

# Regras e convenções de nomenclatura do identificador C#

Artigo • 17/08/2023

Um **identificador** é o nome que você atribui a um tipo (classe, interface, struct, delegado ou enumerado), membro, variável ou namespace.

## Regras de nomenclatura

Os identificadores válidos devem seguir estas regras:

- Os identificadores devem começar com letra ou sublinhado (`_`).
- Os identificadores podem conter caracteres de letra Unicode, caracteres de dígito decimal, caracteres de conexão Unicode, caracteres de combinação Unicode ou caracteres de formatação Unicode. Para obter mais informações sobre as categorias Unicode, consulte o [Banco de dados da categoria Unicode](#).

É possível declarar identificadores que correspondem às palavras-chave em C# usando o prefixo `@` no identificador. O `@` faz parte do nome do identificador. Por exemplo, `@if` declara um identificador chamado `if`. Esses **identificadores textuais** são destinados principalmente para interoperabilidade com os identificadores declarados em outras linguagens.

Para obter uma definição completa de identificadores válidos, confira o [artigo Identificadores na especificação da linguagem C#](#).

## Convenções de nomenclatura

Além das regras, há muitas **convenções de nomenclatura** de identificador usadas em toda as APIs do .NET. Por convenção, os programas C# usam `PascalCase` para nomes de tipo, namespaces e todos os membros públicos. Além disso, a equipe `dotnet/docs` usa as seguintes convenções, adotadas a partir do [estilo de codificação da equipe do .NET runtime](#):

- Os nomes de interface começam com `I` maiúsculo.
- Os tipos de atributo terminam com a palavra `Attribute`.
- Os tipos enumerados usam um substantivo singular para nonflags e um substantivo plural para sinalizadores.

- Os identificadores não devem conter dois caracteres de sublinhado (`_`) consecutivos. Esses nomes estão reservados para identificadores gerados por compilador.
- Use nomes significativos e descritivos para variáveis, métodos e classes.
- Evite usar nomes de letra única, exceto para contadores de loop simples. Confira exceções para obter exemplos de sintaxe anotados na seção a seguir.
- Prefira clareza em vez de brevidade.
- Use PascalCase para nomes de classe e nomes de método.
- Use camelCase para argumentos de método, variáveis locais e campos privados.
- Use PascalCase para nomes de constantes, tanto para constantes de campo quanto para constantes de local.
- Os campos de instância privada começam com um sublinhado (`_`).
- Os campos estáticos começam com `s_`. Observe que esse não é o comportamento padrão do Visual Studio, nem parte das [diretrizes de design do Framework](#), mas é configurável na configuração do editor.
- Evite usar abreviações ou acrônimos em nomes, exceto para abreviações amplamente conhecidas e aceitas.
- Use namespaces significativos e descritivos que seguem a notação de nome de domínio inverso.
- Escolha os nomes de assembly que representam a finalidade primária do assembly.

Os exemplos que descrevem a sintaxe de constructos C# geralmente usam nomes de letra única que correspondem à convenção usada na [especificação da linguagem C#](#):

- Use `s` para structs, `c` para classes.
- Use `M` para métodos.
- Use `v` para variáveis, `p` para parâmetros.
- Use `r` para parâmetros `ref`.

Os nomes de letra única anteriores são permitidos somente na seção de referência de idioma.

Nos exemplos a seguir, as diretrizes relativas aos elementos marcados com `public` são aplicáveis ao trabalhar com elementos `protected` e `protected internal`, todos os quais devem ser visíveis para chamadores externos.

## Pascal case

Use pascal casing ("PascalCasing") ao nomear um tipo `class`, `Interface`, `struct` ou `delegate`.

C#

```
public class DataService
{
}
```

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

C#

```
public struct ValueCoordinate
{
}
```

C#

```
public delegate void DelegateType(string message);
```

Ao nomear um `interface`, use Pascal Case, além de aplicar ao nome um prefixo `I`. Esse prefixo indica claramente aos consumidores que ele é um `interface`.

C#

```
public interface IWorkerQueue
{
}
```

Ao nomear membros `public` de tipos, como campos, propriedades, eventos, use pascal casing. Além disso, use pascal casing para todos os métodos e funções locais.

C#

```
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }
}
```

```
// An event
public event Action EventProcessing;

// Method
public void StartEventProcessing()
{
    // Local function
    static int CountQueueItems() => WorkerQueue.Count;
    // ...
}
}
```

Ao gravar registros posicionais, use Pascal Case para parâmetros, pois são as propriedades públicas do registro.

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

Para obter mais informações sobre registros posicionais, confira [Sintaxe posicional para definição de propriedade](#).

## Camel Case

Use camel casing ("camelCasing") ao nomear campos `private` ou `internal` e dê a eles o prefixo `_`. Use camel casing ao nomear variáveis locais, incluindo instâncias de um tipo delegado.

C#

```
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

### Dica

Ao editar o código do C# que segue essas convenções de nomenclatura em um IDE que permite a conclusão da instrução, digitar `_` mostrará todos os membros no escopo do objeto.

Ao trabalhar com os campos `static` que são `private` ou `internal`, use o prefixo `s_` e, para thread estático, use `t_`.

C#

```
public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

Ao gravar os parâmetros de método, use Camel Case.

C#

```
public T SomeMethod<T>(int someNumber, bool isValid)
{
}
```

Para obter mais informações sobre as convenções de nomenclatura do C#, confira [Estilo de Codificação em C# ↗](#).

## Diretrizes para a nomenclatura de parâmetros de tipo

As diretrizes a seguir se aplicam a parâmetros de tipo em parâmetros de tipo genérico. Esses são os espaços reservados para argumentos em um tipo genérico ou um método genérico. Você pode ler mais sobre [parâmetros](#) de tipo genérico no guia de programação em C#.

- **Nomeie** parâmetros de tipo genérico com nomes descritivos, a menos que um nome com uma única letra seja autoexplicativo e um nome descritivo não agregue valor.

./snippets/coding-conventions

```
public interface ISessionChannel<TSession> { /*...*/ }
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class List<T> { /*...*/ }
```

- **Considere** usar `T` como o nome do parâmetro de tipo para tipos com um parâmetro de tipo de letra única.

./snippets/coding-conventions

```
public int IComparer<T>() { return 0; }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T : struct { /*...*/ }
```

- Insira o prefixo “T” em nomes descritivos de parâmetro de tipo.

```
./snippets/coding-conventions
```

```
public interface ISessionChannel<TSession>
{
    TSession Session { get; }
}
```

- Considere indicar as restrições colocadas em um parâmetro de tipo no nome do parâmetro. Por exemplo, um parâmetro restrito a `ISession` pode ser chamado `TSession`.

A regra de análise de código [CA1715](#) pode ser usada para garantir que os parâmetros de tipo sejam nomeados adequadamente.

## Convenções de nomenclatura extras

- Em exemplos que não incluem o [uso de diretivas](#), use as qualificações do namespace. Se você souber que um namespace é importado por padrão em um projeto, não precisará qualificar totalmente os nomes desse namespace. Os nomes qualificados poderão ser quebrados após um ponto (.) se forem muito longos para uma única linha, conforme mostrado no exemplo a seguir.

```
C#
```

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- Não é necessário alterar os nomes de objetos que foram criados usando as ferramentas de designer do Visual Studio para adequá-los a outras diretrizes.

# Convenções comuns de código C#

Artigo • 17/08/2023

Ter um padrão de código é essencial para manter a legibilidade, a consistência e a colaboração de código em uma equipe de desenvolvimento. Seguir as práticas do setor e as diretrizes estabelecidas ajuda a garantir que o código seja mais fácil de entender, manter e estender. A maioria dos projetos impõe um estilo consistente por meio de convenções de código. Os projetos [dotnet/docs](#) e [dotnet/samples](#) não são exceção. Nesta série de artigos, você aprenderá nossas convenções de codificação e as ferramentas que usamos para aplicá-las. Você pode tomar nossas convenções como estão ou modificá-las para atender às necessidades de sua equipe.

Escolhemos nossas convenções com base nas seguintes metas:

1. *Correção*: nossos exemplos são copiados e colados em seus aplicativos. Esperamos isso, portanto, precisamos criar um código resiliente e correto, mesmo após várias edições.
2. *Ensino*: a finalidade de nossos exemplos é ensinar tudo do .NET e C#. Por esse motivo, não colocamos restrições em nenhum recurso de idioma ou API. Em vez disso, esses exemplos ensinam quando um recurso é uma boa opção.
3. *Consistência*: os leitores esperam uma experiência consistente em nosso conteúdo. Todos os exemplos devem estar em conformidade com o mesmo estilo.
4. *Adoção*: atualizamos agressivamente nossos exemplos para usar novos recursos de linguagem. Essa prática conscientiza os novos recursos e os torna mais familiares para todos os desenvolvedores C#.

## ⓘ Importante

Essas diretrizes são usadas pela Microsoft para desenvolver exemplos e documentação. Elas foram adotados das diretrizes do [.NET runtime, estilo de codificação C#](#) e [compilador C# \(roslyn\)](#). Escolhemos essas diretrizes porque elas foram testadas ao longo de vários anos de desenvolvimento de software livre. Elas ajudaram os membros da comunidade a participar dos projetos de runtime e compilador. Elas devem ser um exemplo de convenções comuns em C# e não uma lista autoritativa (confira [Diretrizes de design de estrutura](#) para isso).

As *metas de ensino* e *adoção* são as razões pelas quais a convenção de codificação de documentos difere das convenções de runtime e compilador. O runtime e o compilador têm métricas de desempenho estritas para caminhos frequentes. Não se pode dizer o mesmo de muitos outros aplicativos. Nossa meta de *ensino* determina que não proibimos nenhuma construção. Em vez disso, os exemplos

mostram quando os constructos devem ser usados. Atualizamos exemplos mais agressivamente do que a maioria dos aplicativos de produção. Nossa meta de adoção determina que mostramos o código que você deve escrever hoje, mesmo quando o código escrito no ano passado não precisa de alterações.

Este artigo explica nossas diretrizes. As diretrizes evoluíram ao longo do tempo e você encontrará exemplos que não seguem nossas diretrizes. Damos as boas-vindas a PRs que trazem esses exemplos para a conformidade, ou problemas que chamam nossa atenção para exemplos que devemos atualizar. Nossas diretrizes são de software livre e damos as boas-vindas a PRs e problemas. No entanto, se o envio alterar essas recomendações, abra um problema para discussão primeiro. Você é bem-vindo a usar nossas diretrizes ou adaptá-las às suas necessidades.

## Ferramentas e analisadores

As ferramentas podem ajudar sua equipe a impor seus padrões. Você pode habilitar qualquer uma das [ferramentas](#) de análise de código para impor as regras que preferir. Você também pode criar uma [editorconfig](#) para que o Visual Studio imponha automaticamente suas diretrizes de estilo. Você pode começar usando [o dotnet/docs](#) para usar nosso estilo como ponto de partida.

Essas ferramentas facilitam a adoção de suas diretrizes preferenciais para sua equipe. O Visual Studio aplica as regras em todos os arquivos `.editorconfig` no escopo para formatar seu código. Você pode usar vários conjuntos de regras para impor padrões corporativos, padrões de equipe e até padrões de projeto granulares.

Todas as ferramentas de análise de código configuradas produzem avisos e diagnósticos quando suas regras são violadas. Você configura as regras que deseja aplicar ao seu projeto. Em seguida, cada build de CI notifica os desenvolvedores quando eles violam qualquer uma das regras.

## Diretrizes de linguagem

As seções a seguir descrevem as práticas que a equipe de documentos do .NET segue para preparar exemplos e exemplos de código. Em geral, siga estas práticas:

- Utilize recursos de linguagem moderna e versões em C# sempre que possível.
- Evite construções de linguagem obsoletas ou desatualizadas.
- Somente capture exceções que possam ser tratadas corretamente; evite capturar exceções genéricas.
- Use tipos de exceção específicos para fornecer mensagens de erro significativas.

- Use consultas LINQ e métodos para manipulação de coleção para melhorar a legibilidade do código.
  - Use programação assíncrona com assíncrono e aguarde operações associadas a E/S.
  - Tenha cuidado com deadlocks e use `Task.ConfigureAwait` quando apropriado.
  - Use as palavras-chave de idioma para tipos de dados em vez dos tipos de runtime. Por exemplo, use `string` em vez de `System.String`, ou `int` em vez de `System.Int32`.
  - Use `int` em vez de tipos não assinados. O uso de `int` é comum em todo o C# e é mais fácil interagir com outras bibliotecas quando você usa `int`. As exceções são para a documentação específica para tipos de dados não assinados.
  - Use `var` somente quando um leitor puder inferir o tipo da expressão. Os leitores exibem nossos exemplos na plataforma de documentos. Eles não têm dicas de focalização ou ferramenta que exibem o tipo de variáveis.
  - Escreva código com clareza e simplicidade em mente.
  - Evite lógica de código excessivamente complexa e complicada.

Há diretrizes mais específicas a seguir.

## Dados de cadeia de caracteres

- Use a [interpolação de cadeia de caracteres](#) para concatenar cadeias de caracteres curtas, como é mostrado no código a seguir.

C#

```
string displayName = $"{nameList[n].LastName},  
{nameList[n].FirstName}";
```

- Para acrescentar cadeias de caracteres em loops, especialmente quando você estiver trabalhando com grandes quantidades de texto, use um objeto `System.Text.StringBuilder`.

C#

# Matrizes

- Use a sintaxe concisa ao inicializar matrizes na linha da declaração. No exemplo a seguir, você não pode usar `var` em vez de `string[]`.

```
C#
```

```
string[] vowels1 = { "a", "e", "i", "o", "u" };
```

- Se você usar uma instanciação explícita, poderá usar `var`.

```
C#
```

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

# Delegados

- Use `Func<>` e `Action<>`, em vez de definir tipos delegados. Em uma classe, defina o método delegado.

```
C#
```

```
Action<string> actionExample1 = x => Console.WriteLine($"x is: {x}");

Action<string, string> actionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

Func<string, int> funcExample1 = x => Convert.ToInt32(x);

Func<int, int, int> funcExample2 = (x, y) => x + y;
```

- Chame o método usando a assinatura definida pelo delegado `Func<>` ou `Action<>`.

```
C#
```

```
actionExample1("string for x");

actionExample2("string for x", "string for y");

Console.WriteLine($"The value is {funcExample1("1")}");

Console.WriteLine($"The sum is {funcExample2(1, 2)}");
```

- Se você criar instâncias de um tipo delegado, use a sintaxe concisa. Em uma classe, defina o tipo delegado e um método que tenha uma assinatura correspondente.

C#

```
public delegate void Del(string message);

public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

- Crie uma instância do tipo delegado e a chame. A declaração a seguir mostra a sintaxe condensada.

C#

```
Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

- A declaração a seguir usa a sintaxe completa.

C#

```
Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

## try-catch Instruções e using no tratamento de exceções

- Use uma instrução **try-catch** para a maioria da manipulação de exceções.

C#

```
static double ComputeDistance(double x1, double y1, double x2, double
y2)
{
    try
    {
        return Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
y2));
    }
    catch (System.ArithmetcException ex)
    {
        Console.WriteLine($"Arithmetc overflow or underflow: {ex}");
        throw;
    }
}
```

- Simplifique o código usando a [instrução using](#) do #C. Se você tiver uma instrução `try-finally` na qual o único código do bloco `finally` é uma chamada para o método [Dispose](#), use, em vez disso, uma instrução `using`.

No exemplo a seguir, a instrução `try-finally` chama apenas `Dispose` no bloco `finally`.

```
C#  
  
Font bodyStyle = new Font("Arial", 10.0f);  
try  
{  
    byte charset = bodyStyle.GdiCharSet;  
}  
finally  
{  
    if (bodyStyle != null)  
    {  
        ((IDisposable)bodyStyle).Dispose();  
    }  
}
```

Você pode fazer a mesma coisa com uma instrução `using`.

```
C#  
  
using (Font arial = new Font("Arial", 10.0f))  
{  
    byte charset2 = arial.GdiCharSet;  
}
```

Use a nova [sintaxe using](#) que não requer chaves:

```
C#  
  
using Font normalStyle = new Font("Arial", 10.0f);  
byte charset3 = normalStyle.GdiCharSet;
```

## Operadores `&&` e `||`

- Use `&&` em vez de `&` e `||` em vez de `|` quando executar comparações, conforme mostrado no exemplo a seguir.

```
C#
```

```
Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

Se o divisor for 0, a segunda cláusula na instrução `if` causará um erro em tempo de execução. Mas o operador `&&` entra em curto-circuito quando a primeira expressão é falsa. Ou seja, ele não avalia a segunda expressão. O operador `&` avalia ambas, resultando em um erro em tempo de execução quando `divisor` é 0.

## Operador `new`

- Use uma das formas concisas de instanciação de objeto, conforme mostrado nas declarações a seguir. O segundo exemplo mostra a sintaxe disponível a partir do C# 9.

C#

```
var firstExample = new ExampleClass();
```

C#

```
ExampleClass instance2 = new();
```

As declarações anteriores são equivalentes à declaração a seguir.

C#

```
ExampleClass secondExample = new ExampleClass();
```

- Use inicializadores de objeto para simplificar a criação de objeto, conforme mostrado no exemplo a seguir.

C#

```
var thirdExample = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };
```

O exemplo a seguir define as mesmas propriedades do exemplo anterior, mas não usa inicializadores.

C#

```
var fourthExample = new ExampleClass();
fourthExample.Name = "Desktop";
fourthExample.ID = 37414;
fourthExample.Location = "Redmond";
fourthExample.Age = 2.3;
```

## Manipulação de eventos

- Use uma expressão lambda para definir um manipulador de eventos que você não precisa remover mais tarde:

C#

```
public Form2()
{
    this.Click += (s, e) =>
    {
        MessageBox.Show(
            ((MouseEventArgs)e).Location.ToString());
    };
}
```

A expressão lambda reduz a definição convencional a seguir.

C#

```
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object? sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

## Membros estáticos

Chame membros **estáticos** usando o nome de classe: *ClassName.StaticMember*. Essa prática torna o código mais legível, tornando o acesso estático limpo. Não qualifique um membro estático definido em uma classe base com o nome de uma classe derivada. Enquanto esse código é compilado, a leitura do código fica incorreta e o código poderá ser danificado no futuro se você adicionar um membro estático com o mesmo nome da classe derivada.

## Consultas LINQ

- Use nomes significativos para variáveis de consulta. O exemplo a seguir usa `seattleCustomers` para os clientes que estão localizados em Seattle.

C#

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Use aliases para se certificar de que os nomes de propriedades de tipos anônimos sejam colocados corretamente em maiúsculas, usando o padrão Pascal-Case.

C#

```
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals
    distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- Renomeie propriedades quando os nomes de propriedades no resultado forem ambíguos. Por exemplo, se a sua consulta retornar um nome de cliente e uma ID de distribuidor, em vez de deixá-los como `Name` e `ID` no resultado, renomeie-os para esclarecer que `Name` é o nome de um cliente, e `ID` é a identificação de um distribuidor.

C#

```
var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals
    distributor.City
    select new { CustomerName = customer.Name, DistributorID =
    distributor.ID };
```

- Usa a digitação implícita na declaração de variáveis de consulta e de intervalo. Essa orientação sobre digitação implícita em consultas LINQ substitui as regras gerais para [variáveis locais de tipo implícito](#). As consultas LINQ geralmente usam projeções que criam tipos anônimos. Outras expressões de consulta criam resultados com tipos genéricos aninhados. Variáveis de tipo implícito geralmente são mais legíveis.

C#

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Alinhe as cláusulas de consulta na cláusula `from`, conforme mostrado nos exemplos anteriores.
- Use as cláusulas `where` antes de outras cláusulas de consulta, para garantir que as cláusulas de consulta posteriores operem no conjunto de dados filtrado e reduzido.

C#

```
var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- Use várias cláusulas `from`, em vez de uma cláusula `join`, para acessar as coleções internas. Por exemplo, cada coleção de objetos `Student` pode conter um conjunto de pontuações no teste. Quando a próxima consulta for executada, ela retorna cada pontuação que seja acima de 90, juntamente com o sobrenome do estudante que recebeu a pontuação.

C#

```
var scoreQuery = from student in students
                  from score in student.Scores!
                  where score > 90
                  select new { Last = student.LastName, score };
```

## Variáveis locais de tipo implícito

- Use a **digitação implícita** para variáveis locais quando o tipo da variável for óbvio do lado direito da atribuição.

C#

```
var message = "This is clearly a string.";  
var currentTemperature = 27;
```

- Não use `var` quando o tipo não for aparente do lado direito da atribuição. Não suponha que o tipo esteja claro partir do nome de um método. Um tipo de variável é considerado claro se for um operador `new`, uma conversão explícita ou uma atribuição a um valor literal.

C#

```
int numberOfIterations = Convert.ToInt32(Console.ReadLine());  
int currentMaximum = ExampleClass.ResultSoFar();
```

- Não use nomes de variáveis para especificar o tipo da variável. Ele pode não estar correto. Em vez disso, use o tipo para especificar o tipo e use o nome da variável para indicar as informações semânticas da variável. O exemplo a seguir deve usar `string` para o tipo e algo como `iterations` indicar o significado das informações lidas do console.

C#

```
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Evite o uso de `var` em vez de `dynamic`. Use `dynamic` quando quiser uma inferência de tipo em tempo de execução. Para obter mais informações, confira [Como usar o tipo dinâmico \(Guia de Programação do C#\)](#).
  - Use a digitação implícita para a variável de loop em loops `for`.

O exemplo a seguir usa digitacão implícita em uma instrução `for`.

C#

```
}
```

```
//Console.WriteLine("tra" + manyPhrases);
```

- Não use a digitação implícita para determinar o tipo da variável em loop nos loops `foreach`. Na maioria dos casos, o tipo de elementos na coleção não é imediatamente evidente. O nome da coleção não deve ser usado apenas para inferir o tipo dos elementos.

O exemplo a seguir usa digitação explícita em uma instrução `foreach`.

```
C#
```

```
foreach (char ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

- use o tipo implícito para as sequências de resultados em consultas LINQ. A seção em [LINQ](#) explica que muitas consultas LINQ resultam em tipos anônimos em que tipos implícitos devem ser usados. Outras consultas resultam em tipos genéricos aninhados em `var` que é mais legível.

#### ① Observação

Tenha cuidado para não alterar accidentalmente um tipo de elemento da coleção iterável. Por exemplo, é fácil alternar de `System.Linq.IQueryable` para `System.Collections.IEnumerable` em uma instrução `foreach`, o que altera a execução de uma consulta.

Alguns de nossos exemplos explicam o *tipo* natural de uma expressão. Esses exemplos devem usar `var` para que o compilador escolha o tipo natural. Embora esses exemplos sejam menos óbvios, o uso de `var` é necessário para o exemplo. O texto deve explicar o comportamento.

## Coloque as diretivas “using” fora da declaração de namespace

Quando uma diretiva `using` está fora de uma declaração de namespace, esse namespace importado é seu nome totalmente qualificado. O nome totalmente qualificado é mais claro. Quando a diretiva `using` está dentro do namespace, ela pode ser relativa a esse namespace ou ao seu nome totalmente qualificado.

C#

```
using Azure;

namespace CoolStuff.AwesomeFeature
{
    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Supondo que haja uma referência (direta ou indireta) à classe `WaitUntil`.

Agora, vamos fazer uma pequena alteração:

C#

```
namespace CoolStuff.AwesomeFeature
{
    using Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

E ela será compilada hoje. E amanhã. Mas, em algum momento da próxima semana, o código anterior (intocado) falha com dois erros:

Console

```
- error CS0246: The type or namespace name 'WaitUntil' could not be found
(are you missing a using directive or an assembly reference?)
```

```
- error CS0103: The name 'WaitUntil' does not exist in the current context
```

Uma das dependências introduziu esta classe em um namespace e termina com `.Azure`:

C#

```
namespace CoolStuff.Azure
{
    public class SecretsManagement
    {
        public string FetchFromKeyVault(string vaultId, string secretId) {
            return null; }
    }
}
```

Uma diretiva `using` colocada dentro de um namespace diferencia contexto e complica a resolução do nome. Neste exemplo, é o primeiro namespace que ela encontra.

- `CoolStuff.AwesomeFeature.Azure`
- `CoolStuff.Azure`
- `Azure`

Adicionar um novo namespace que corresponda a `CoolStuff.Azure` ou

`CoolStuff.AwesomeFeature.Azure` seria combinado antes do namespace `Azure` global.

Você poderia resolver isso adicionando o modificador `global::` à declaração `using`. No entanto, é mais fácil colocar declarações `using` fora do namespace.

C#

```
namespace CoolStuff.AwesomeFeature
{
    using global::Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

## Diretrizes de estilo

Em geral, use o seguinte formato para exemplos de código:

- Use quatro espaços para recuo. Não use guias.
- Alinhe o código consistentemente para melhorar a legibilidade.
- Limite linhas a 65 caracteres para aprimorar a legibilidade de código em documentos, especialmente em telas de dispositivos móveis.
- Divida instruções longas em várias linhas para melhorar a clareza.
- Use o estilo "Allman" para chaves: abra e feche sua própria nova linha. As chaves se alinham com o nível de recuo atual.
- As quebras de linha devem ocorrer antes dos operadores binários, se necessário.

## Estilo de comentário

- Use comentários de linha única (`//`) para breves explicações.
- Evite comentários de várias linhas (`/* */`) para explicações mais longas. Os comentários não são localizados. Em vez disso, explicações mais longas estão no artigo complementar.
- Coloque o comentário em uma linha separada, não no final de uma linha de código.
- Comece o texto do comentário com uma letra maiúscula.
- Termine o texto do comentário com um ponto final.
- Insira um espaço entre o delimitador de comentário (`/`) e o texto do comentário, conforme mostrado no exemplo a seguir.

```
C#
```

```
// The following declaration creates a query. It does not run  
// the query.
```

## Convenções de layout

Um bom layout usa formatação para enfatizar a estrutura do código e para facilitar a leitura de código. Exemplos e amostras Microsoft estão em conformidade com as seguintes convenções:

- Use as configurações padrão do Editor de códigos (recuo inteligente, recuos de quatro caracteres, guias salvas como espaços). Para obter mais informações, consulte [Opções, Editor de Texto, C#, Formatação](#).

- Gravar apenas uma instrução por linha.
- Gravar apenas uma declaração por linha.
- Se as linhas de continuação não forem recuadas automaticamente, recue-as uma parada de tabulação (quatro espaços).
- Adicione pelo menos uma linha em branco entre as definições de método e de propriedade.
- Use parênteses para criar cláusulas em uma expressão aparente, conforme mostrado no código a seguir.

```
C#
```

```
if ((startX > endX) && (startX > previousX))
{
    // Take appropriate action.
}
```

As exceções são quando o exemplo explica o operador ou a precedência de expressão.

## Segurança

Siga as diretrizes em [Diretrizes de codificação segura](#).

# Como exibir argumentos de linha de comando

Artigo • 08/06/2023

Os argumentos fornecidos a um executável na linha de comando são acessíveis em [instruções de nível superior](#) ou por meio de um parâmetro opcional para `Main`. Os argumentos são fornecidos na forma de uma matriz de cadeias de caracteres. Cada elemento da matriz contém um argumento. O espaço em branco entre os argumentos é removido. Por exemplo, considere essas invocações de linha de comando de um executável fictício:

Entrada na linha de comando	Matriz de cadeias de caracteres passada a Main
<code>executável.exe a b c</code>	"a" "b" "c"
<code>executável.exe um dois</code>	"um" "dois"
<code>executável.exe "um dois" três</code>	"one two" "three"

## Observação

Quando estiver executando um aplicativo no Visual Studio, você pode especificar argumentos de linha de comando na [Página de depuração](#), [Designer de Projeto](#).

## Exemplo

Este exemplo exibe os argumentos de linha de comando passados a um aplicativo de linha de comando. A saída mostrada é para a primeira entrada da tabela acima.

C#

```
// The Length property provides the number of array elements.  
Console.WriteLine($"parameter count = {args.Length}");
```

```
for (int i = 0; i < args.Length; i++)
{
    Console.WriteLine($"Arg[{i}] = [{args[i]}]");
}

/* Output (assumes 3 cmd line args):
   parameter count = 3
   Arg[0] = [a]
   Arg[1] = [b]
   Arg[2] = [c]
*/
```

## Confira também

- Visão geral de System.CommandLine
- Tutorial: introdução a System.CommandLine

# Explorar programação orientada a objeto com classes e objetos

Artigo • 02/06/2023

Neste tutorial, você vai compilar um aplicativo de console e ver os recursos básicos orientados a objeto que fazem parte da linguagem C#.

## Pré-requisitos

- Recomendamos o [Visual Studio](#) para Windows ou Mac. Você pode baixar uma versão gratuita na [página de downloads do Visual Studio](#). O Visual Studio inclui o SDK do .NET.
- Você também pode usar o editor do [Visual Studio Code](#). Será preciso instalar o [SDK do .NET](#) mais recente separadamente.
- Se preferir outro editor, você precisará instalar o [SDK do .NET](#) mais recente.

## Criar o aplicativo

Usando uma janela de terminal, crie um diretório chamado *classes*. Você compilará o aplicativo nesse diretório. Altere para esse diretório e digite `dotnet new console` na janela do console. Esse comando cria o aplicativo. Abra *Program.cs*. Ele deverá ser parecido com:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Neste tutorial, você criará novos tipos que representam uma conta bancária. Normalmente, os desenvolvedores definem cada classe em um arquivo de texto diferente. Isso facilita o gerenciamento à medida que o tamanho do programa aumenta. Crie um novo arquivo chamado *BankAccount.cs* no diretório *Classes*.

Esse arquivo conterá a definição de uma **conta bancária**. A programação Orientada a Objetos organiza o código por meio da criação de tipos na forma de **classes**. Essas classes contêm o código que representa uma entidade específica. A classe `BankAccount` representa uma conta bancária. O código implementa operações específicas por meio de métodos e propriedades. Neste tutorial, a conta bancária dá suporte a este comportamento:

1. Ela tem um número com 10 dígitos que identifica exclusivamente a conta bancária.
2. Ela tem uma cadeia de caracteres que armazena o nome ou os nomes dos proprietários.
3. O saldo pode ser recuperado.
4. Ela aceita depósitos.
5. Ela aceita saques.
6. O saldo inicial deve ser positivo.
7. Os saques não podem resultar em um saldo negativo.

## Definir o tipo de conta bancária

Você pode começar criando as noções básicas de uma classe que define esse comportamento. Crie um novo arquivo usando o comando **File>New. Nomeie-o bankAccount.cs**. Adicione o seguinte código ao arquivo *BankAccount.cs*:

C#

```
namespace Classes;

public class BankAccount
{
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

Antes de continuar, vamos dar uma olhada no que você compilou. A declaração `namespace` fornece uma maneira de organizar logicamente seu código. Este tutorial é relativamente pequeno, portanto, você colocará todo o código em um namespace.

`public class BankAccount` define a classe ou o tipo que você está criando. Tudo que vem dentro de `{` e `}` logo após a declaração da classe define o estado e o comportamento da classe. Há cinco **membros** na classe `BankAccount`. Os três primeiros são **propriedades**. Propriedades são elementos de dados que podem ter um código que impõe a validação ou outras regras. Os dois últimos são **métodos**. Os métodos são blocos de código que executam uma única função. A leitura dos nomes de cada um dos

membros deve fornecer informações suficientes para você, ou outro desenvolvedor, entender o que a classe faz.

## Abrir uma nova conta

O primeiro recurso a ser implementado serve para abrir uma conta bancária. Quando um cliente abre uma conta, ele deve fornecer um saldo inicial e informações sobre o proprietário, ou proprietários, dessa conta.

A criação de novo objeto do tipo `BankAccount` significa a definição de um **construtor** que atribui esses valores. Um **construtor** é um membro que tem o mesmo nome da classe. Ele é usado para inicializar objetos desse tipo de classe. Adicione o seguinte construtor ao tipo `BankAccount`. Insira o código a seguir acima da declaração de

`MakeDeposit`:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

O código anterior identifica as propriedades do objeto que está sendo construído, incluindo o qualificador `this`. Esse qualificador geralmente é opcional e omitido. Você também poderia ter escrito:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Owner = name;
    Balance = initialBalance;
}
```

O qualificador `this` só é necessário quando uma variável ou parâmetro locais têm o mesmo nome desse campo ou propriedade. O qualificador `this` é omitido durante todo o restante deste artigo, a menos que seja necessário.

Construtores são chamados quando você cria um objeto usando `new`. Substitua a linha `Console.WriteLine("Hello World!");` no arquivo *Program.cs* pelo seguinte código (substitua `<name>` pelo seu nome):

C#

```
using Classes;

var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner}
with {account.Balance} initial balance.");
```

Vamos executar o que você construiu até agora. Se você estiver usando o Visual Studio, selecione **Iniciar sem depuração** no menu **Depurar**. Se estiver usando uma linha de comando, digite `dotnet run` no diretório em que criou seu projeto.

Você notou que o número da conta está em branco? É hora de corrigir isso. O número da conta deve ser atribuído na construção do objeto. Mas não é responsabilidade do chamador criá-lo. O código da classe `BankAccount` deve saber como atribuir novos números de conta. Uma maneira simples de fazer isso é começar com um número de 10 dígitos. Incremente-o à medida que novas contas são criadas. Por fim, armazene o número da conta atual quando um objeto for construído.

Adicione uma declaração de membro à classe `BankAccount`. Coloque a seguinte linha de código após a chave de abertura `{` no início da classe `BankAccount`:

C#

```
private static int accountNumberSeed = 1234567890;
```

O `accountNumberSeed` é um membro de dados. Ele é `private`, o que significa que ele só pode ser acessado pelo código dentro da classe `BankAccount`. É uma maneira de separar as responsabilidades públicas (como ter um número de conta) da implementação privada (como os números de conta são gerados). Ele também é `static`, o que significa que é compartilhado por todos os objetos `BankAccount`. O valor de uma variável não estática é exclusivo para cada instância do objeto `BankAccount`. Adicione as duas linhas a seguir ao construtor para atribuir o número da conta. Coloque-as depois da linha que diz `this.Balance = initialBalance`:

C#

```
this.Number = accountNumberSeed.ToString();
accountNumberSeed++;
```

Digite `dotnet run` para ver os resultados.

# Criar depósitos e saques

A classe da conta bancária precisa aceitar depósitos e saques para funcionar corretamente. Vamos implementar depósitos e saques criando um diário de todas as transações da conta. Rastrear todas as transações tem algumas vantagens em comparação à simples atualização do saldo em cada transação. O histórico pode ser usado para auditar todas as transações e gerenciar os saldos diários. Calcular o saldo do histórico de todas as transações quando necessário assegura que todos os erros corrigidos em uma única transação serão refletidos corretamente no saldo no próximo cálculo.

Vamos começar criando um novo tipo para representar uma transação. Essa transação é um tipo simples que não tem qualquer responsabilidade. Ele precisa de algumas propriedades. Crie um novo arquivo chamado *Transaction.cs*. Adicione os seguintes códigos a ela:

C#

```
namespace Classes;

public class Transaction
{
    public decimal Amount { get; }
    public DateTime Date { get; }
    public string Notes { get; }

    public Transaction(decimal amount, DateTime date, string note)
    {
        Amount = amount;
        Date = date;
        Notes = note;
    }
}
```

Agora, vamos adicionar um `List<T>` de `Transaction` objetos à classe `BankAccount`. Adicione a seguinte declaração após o construtor no arquivo *BankAccount.cs*:

C#

```
private List<Transaction> allTransactions = new List<Transaction>();
```

Agora, vamos calcular corretamente o `Balance`. O saldo atual pode ser encontrado somando os valores de todas as transações. De acordo com a forma atual do código, você só pode obter o saldo inicial da conta. Portanto, você terá que atualizar a

propriedade `Balance`. Substitua a linha `public decimal Balance { get; } em BankAccount.cs pelo seguinte código:`

```
C#  
  
public decimal Balance  
{  
    get  
    {  
        decimal balance = 0;  
        foreach (var item in allTransactions)  
        {  
            balance += item.Amount;  
        }  
  
        return balance;  
    }  
}
```

Este exemplo mostra um aspecto importante das *propriedades*. Agora, você está calculando o saldo quando outro programador solicita o valor. Seu cálculo enumera todas as transações e fornece a soma como o saldo atual.

Depois, implemente os métodos `MakeDeposit` e `MakeWithdrawal`. Esses métodos vão impor as duas últimas regras: que o saldo inicial deve ser positivo, e que qualquer saque não pode criar um saldo negativo.

Essas regras introduzem o conceito de *exceções*. A forma padrão de indicar que um método não pode concluir seu trabalho com êxito é lançar uma exceção. O tipo de exceção e a mensagem associada a ele descrevem o erro. Aqui, o método `MakeDeposit` lançará uma exceção se o valor do depósito for menor ou igual a 0. O método `MakeWithdrawal` lançará uma exceção se o valor do saque for menor ou igual a 0 ou se a aplicação do saque resultar em um saldo negativo. Adicione o seguinte código depois da declaração da lista `allTransactions`:

```
C#  
  
public void MakeDeposit(decimal amount, DateTime date, string note)  
{  
    if (amount <= 0)  
    {  
        throw new ArgumentException(nameof(amount), "Amount of deposit must be positive");  
    }  
    var deposit = new Transaction(amount, date, note);  
    allTransactions.Add(deposit);  
}
```

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}
```

A instrução `throw` gera uma exceção. A execução do bloco atual é encerrada e o controle transferido para o bloco `catch` da primeira correspondência encontrada na pilha de chamadas. Você adicionará um bloco `catch` para testar esse código um pouco mais tarde.

O construtor deve receber uma alteração para que adicione uma transação inicial, em vez de atualizar o saldo diretamente. Como você já escreveu o método `MakeDeposit`, chame-o de seu construtor. O construtor concluído deve ter esta aparência:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

`DateTime.Now` é uma propriedade que retorna a data e a hora atuais. Teste esse código adicionando alguns depósitos e saques em seu método `Main`, seguindo o código que cria um novo `BankAccount`:

C#

```
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

Em seguida, teste se você está recebendo condições de erro ao tentar criar uma conta com um saldo negativo. Adicione o seguinte código após o código anterior que você acabou de adicionar:

```
C#  
  
// Test that the initial balances must be positive.  
BankAccount invalidAccount;  
try  
{  
    invalidAccount = new BankAccount("invalid", -55);  
}  
catch (ArgumentOutOfRangeException e)  
{  
    Console.WriteLine("Exception caught creating account with negative  
balance");  
    Console.WriteLine(e.ToString());  
    return;  
}
```

Use a [instrução try-catch](#) para marcar um bloco de código que possa gerar exceções e detectar esses erros esperados. Use a mesma técnica a fim de testar o código que gera uma exceção para um saldo negativo. Adicione o seguinte código antes da declaração de `invalidAccount` no seu método `Main`:

```
C#  
  
// Test for a negative balance.  
try  
{  
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");  
}  
catch (InvalidOperationException e)  
{  
    Console.WriteLine("Exception caught trying to overdraw");  
    Console.WriteLine(e.ToString());  
}
```

Salve o arquivo e digite `dotnet run` para testá-lo.

## Desafio – registrar em log todas as transações

Para concluir este tutorial, escreva o método `GetAccountHistory` que cria um `string` para o histórico de transações. Adicione esse método ao tipo `BankAccount`:

```
C#
```

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    decimal balance = 0;
    report.AppendLine("Date\t\tAmount\tBalance\tNote");
    foreach (var item in allTransactions)
    {
        balance += item.Amount;
        report.AppendLine($""
{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");
    }

    return report.ToString();
}
```

O histórico usa a classe [StringBuilder](#) para formatar uma cadeia de caracteres que contém uma linha para cada transação. Você viu o código de formatação da cadeia de caracteres anteriormente nesses tutoriais. Um caractere novo é `\t`. Ele insere uma guia para formatar a saída.

Adicione esta linha para testá-la no *Program.cs*:

C#

```
Console.WriteLine(account.GetAccountHistory());
```

Execute o programa para ver os resultados.

## Próximas etapas

Se você não conseguir avançar, veja a origem deste tutorial [em nosso repositório GitHub](#).

Você pode continuar com o tutorial de [programação orientada a objeto](#).

Saiba mais sobre esses conceitos nestes artigos:

- [Instruções de seleção](#)
- [Instruções de iteração](#)

# Programação orientada a objeto (C#)

Artigo • 08/06/2023

O C# é uma linguagem de programação orientada a objeto. Os quatro princípios básicos da programação orientada a objetos são:

- *Abstração* Modelando os atributos e interações relevantes de entidades como classes para definir uma representação abstrata de um sistema.
- *Encapsulamento* Ocultando o estado interno e a funcionalidade de um objeto e permitindo apenas o acesso por meio de um conjunto público de funções.
- *Herança* Capacidade de criar novas abstrações com base em abstrações existentes.
- *Polimorfismo* Capacidade de implementar propriedades ou métodos herdados de diferentes maneiras em várias abstrações.

No tutorial anterior, [introdução às classes](#) que você viu *abstração* e *encapsulamento*. A classe `BankAccount` forneceu uma abstração para o conceito de conta bancária. Você pode modificar sua implementação sem afetar nenhum dos códigos que usaram a classe `BankAccount`. As classes `BankAccount` e `Transaction` fornecem encapsulamento dos componentes necessários para descrever esses conceitos no código.

Neste tutorial, você estenderá esse aplicativo para usar *herança* e *polimorfismo* para adicionar novos recursos. Você também adicionará recursos à `BankAccount` classe, aproveitando as técnicas de *abstração* e *encapsulamento* que aprendeu no tutorial anterior.

## Criar diferentes tipos de contas

Depois de criar este programa, você recebe solicitações para adicionar recursos a ele. Funciona muito bem na situação em que há apenas um tipo de conta bancária. Ao longo do tempo, as necessidades são alteradas e os tipos de conta relacionados são solicitados:

- Uma conta de ganho de juros que acumula juros no final de cada mês.
- Uma linha de crédito que pode ter saldo negativo, mas quando há saldo, há cobrança de juros a cada mês.
- Uma conta de cartão presente pré-paga que começa com um único depósito, e só pode ser paga. Ela pode ser preenchida novamente uma vez no início de cada mês.

Todas essas contas diferentes são semelhantes à classe `BankAccount` definida no tutorial anterior. Você pode copiar esse código, renomear as classes e fazer modificações. Essa

técnica funcionaria a curto prazo, mas seria mais trabalho ao longo do tempo. Todas as alterações seriam copiadas em todas as classes afetadas.

Em vez disso, você pode criar novos tipos de conta bancária que herdam métodos e dados da classe `BankAccount` criada no tutorial anterior. Essas novas classes podem estender a classe `BankAccount` com o comportamento específico necessário para cada tipo:

C#

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
}

public class GiftCardAccount : BankAccount
{
}
```

Cada uma dessas classes *herda* o comportamento compartilhado de sua *classe base* compartilhada, a classe `BankAccount`. Escreva as implementações para funcionalidades novas e diferentes em cada uma das *classes derivadas*. Essas classes derivadas já têm todo o comportamento definido na classe `BankAccount`.

É uma boa prática criar cada nova classe em um arquivo de origem diferente. No [Visual Studio](#), você pode clicar com o botão direito do mouse no projeto e selecionar *adicionar classe* para adicionar uma nova classe em um novo arquivo. No [Visual Studio Code](#), selecione *Arquivo* e *Novo* para criar um novo arquivo de origem. Em qualquer ferramenta, nomeie o arquivo para corresponder à classe: *InterestEarningAccount.cs*, *LineOfCreditAccount.cs* e *GiftCardAccount.cs*.

Ao criar as classes, conforme mostrado no exemplo anterior, você descobrirá que nenhuma das suas classes derivadas é compilada. Um construtor é responsável por inicializar um objeto. Um construtor de classe derivada deve inicializar a classe derivada e fornecer instruções sobre como inicializar o objeto de classe base incluído na classe derivada. A inicialização adequada normalmente ocorre sem nenhum código extra. A classe `BankAccount` declara um construtor público com a seguinte assinatura:

C#

```
public BankAccount(string name, decimal initialBalance)
```

O compilador não gera um construtor padrão quando você define um construtor por conta própria. Isso significa que cada classe derivada deve chamar explicitamente esse construtor. Você declara um construtor que pode passar argumentos para o construtor de classe base. O código a seguir mostra o construtor para o `InterestEarningAccount`:

C#

```
public InterestEarningAccount(string name, decimal initialBalance) :  
    base(name, initialBalance)  
{  
}
```

Os parâmetros para esse novo construtor correspondem ao tipo de parâmetro e aos nomes do construtor de classe base. Use a sintaxe `: base()` para indicar uma chamada para um construtor de classe base. Algumas classes definem vários construtores e essa sintaxe permite que você escolha qual construtor de classe base você chama. Depois de atualizar os construtores, você pode desenvolver o código para cada uma das classes derivadas. Os requisitos para as novas classes podem ser declarados da seguinte maneira:

- Uma conta de ganho de juros:
  - Obterá um crédito de 2% do saldo final do mês.
- Uma linha de crédito:
  - Pode ter um saldo negativo, mas não ser maior em valor absoluto do que o limite de crédito.
  - Incorrerá em uma cobrança de juros a cada mês em que o saldo de fim de mês não seja 0.
  - Incorrerá em uma taxa em cada saque que ultrapassa o limite de crédito.
- Uma conta de cartão presente:
  - Pode ser preenchido novamente com um valor especificado uma vez por mês, no último dia do mês.

Você pode ver que todos os três tipos de conta têm uma ação que ocorre no final de cada mês. No entanto, cada tipo de conta faz tarefas diferentes. Você usa *polimorfismo* para implementar esse código. Crie um único método `virtual` na classe `BankAccount`:

C#

```
public virtual void PerformMonthEndTransactions() { }
```

O código anterior mostra como você usa a palavra-chave `virtual` para declarar um método na classe base para o qual uma classe derivada pode fornecer uma

implementação diferente. Um método `virtual` é um método em que qualquer classe derivada pode optar por reimplementar. As classes derivadas usam a palavra-chave `override` para definir a nova implementação. Normalmente, você se refere a isso como "substituindo a implementação da classe base". A palavra-chave `virtual` especifica que as classes derivadas podem substituir o comportamento. Você também pode declarar métodos `abstract` em que classes derivadas devem substituir o comportamento. A classe base não fornece uma implementação para um método `abstract`. Em seguida, você precisa definir a implementação para duas das novas classes que você criou. Inicie com um `InterestEarningAccount`:

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        decimal interest = Balance * 0.05m;
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");
    }
}
```

Adicione o seguinte código ao `LineOfCreditAccount`. O código nega o saldo para calcular uma taxa de juros positiva que é retirada da conta:

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        decimal interest = -Balance * 0.07m;
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");
    }
}
```

A classe `GiftCardAccount` precisa de duas alterações para implementar sua funcionalidade de fim de mês. Primeiro, modifique o construtor para incluir um valor opcional a ser adicionado a cada mês:

C#

```
private readonly decimal _monthlyDeposit = 0m;

public GiftCardAccount(string name, decimal initialBalance, decimal
```

```
monthlyDeposit = 0) : base(name, initialBalance)
    => _monthlyDeposit = monthlyDeposit;
```

O construtor fornece um valor padrão para o valor `monthlyDeposit` para que os chamadores possam omitir um `0` sem depósito mensal. Em seguida, substitua o método `PerformMonthEndTransactions` para adicionar o depósito mensal, se ele foi definido como um valor diferente de zero no construtor:

C#

```
public override void PerformMonthEndTransactions()
{
    if (_monthlyDeposit != 0)
    {
        MakeDeposit(_monthlyDeposit, DateTime.Now, "Add monthly deposit");
    }
}
```

A substituição aplica o conjunto de depósito mensal no construtor. Adicione o seguinte código ao método `Main` para testar essas alterações para `GiftCardAccount` e `InterestEarningAccount`:

C#

```
var giftCard = new GiftCardAccount("gift card", 100, 50);
giftCard.MakeWithdrawal(20, DateTime.Now, "get expensive coffee");
giftCard.MakeWithdrawal(50, DateTime.Now, "buy groceries");
giftCard.PerformMonthEndTransactions();
// can make additional deposits:
giftCard.MakeDeposit(27.50m, DateTime.Now, "add some additional spending
money");
Console.WriteLine(giftCard.GetAccountHistory());

var savings = new InterestEarningAccount("savings account", 10000);
savings.MakeDeposit(750, DateTime.Now, "save some money");
savings.MakeDeposit(1250, DateTime.Now, "Add more savings");
savings.MakeWithdrawal(250, DateTime.Now, "Needed to pay monthly bills");
savings.PerformMonthEndTransactions();
Console.WriteLine(savings.GetAccountHistory());
```

Verifique os resultados. Agora, adicione um conjunto semelhante de código de teste para `LineOfCreditAccount`:

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0);
// How much is too much to borrow?
```

```
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly  
advance");  
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");  
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for  
repairs");  
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on  
repairs");  
lineOfCredit.PerformMonthEndTransactions();  
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Ao adicionar o código anterior e executar o programa, você verá algo semelhante ao seguinte erro:

Console

```
Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit  
must be positive (Parameter 'amount')  
at OOProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date,  
String note) in BankAccount.cs:line 42  
at OOProgramming.BankAccount..ctor(String name, Decimal initialBalance)  
in BankAccount.cs:line 31  
at OOProgramming.LineOfCreditAccount..ctor(String name, Decimal  
initialBalance) in LineOfCreditAccount.cs:line 9  
at OOProgramming.Program.Main(String[] args) in Program.cs:line 29
```

### ⓘ Observação

A saída real inclui o caminho completo para a pasta com o projeto. Os nomes das pastas foram omitidos para brevidade. Além disso, dependendo do formato de código, os números de linha podem ser ligeiramente diferentes.

Esse código falha porque `BankAccount` pressupõe que o saldo inicial deve ser maior que 0. Outra suposição feita na classe `BankAccount` é que o saldo não pode ficar negativo. Em vez disso, qualquer retirada que sobrecarrega a conta é rejeitada. Ambas as suposições precisam mudar. A linha de conta de crédito começa em 0 e geralmente terá um saldo negativo. Além disso, se um cliente empresta muito dinheiro, ele incorre em uma taxa. A transação é aceita, só custa mais. A primeira regra pode ser implementada adicionando um argumento opcional ao construtor `BankAccount` que especifica o saldo mínimo. O padrão é 0. A segunda regra requer um mecanismo que permite que classes derivadas modifiquem o algoritmo padrão. De certa forma, a classe base "pergunta" ao tipo derivado o que deve acontecer quando há um cheque especial. O comportamento padrão é rejeitar a transação lançando uma exceção.

Vamos começar adicionando um segundo construtor que inclui um parâmetro `minimumBalance` opcional. Esse novo construtor faz todas as ações feitas pelo construtor existente. Além disso, ele define a propriedade de saldo mínimo. Você pode copiar o corpo do construtor existente, mas isso significa dois locais a serem alterados no futuro. Em vez disso, você pode usar o *encadeamento de construtor* para que um construtor chame outro. O código a seguir mostra os dois construtores e o novo campo adicional:

C#

```
private readonly decimal _minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name,
initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal
minimumBalance)
{
    Number = s_accountNumberSeed.ToString();
    s_accountNumberSeed++;

    Owner = name;
    _minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

O código anterior mostra duas novas técnicas. Primeiro, o campo `minimumBalance` é marcado como `readonly`. Isso significa que o valor não pode ser alterado depois que o objeto é construído. Depois que `BankAccount` é criado, `minimumBalance` não pode alterar. Em segundo lugar, o construtor que usa dois parâmetros `: this(name, initialBalance, 0) { }` como implementação. A expressão `: this()` chama o outro construtor, aquele com três parâmetros. Essa técnica permite que você tenha uma única implementação para inicializar um objeto, embora o código do cliente possa escolher um dos muitos construtores.

Essa implementação chamará `MakeDeposit` somente se o saldo inicial for maior que `0`. Isso preserva a regra de que os depósitos devem ser positivos, mas permite que a conta de crédito abra com um saldo `0`.

Agora que a classe `BankAccount` tem um campo somente leitura para o saldo mínimo, a alteração final é alterar o código físico `0` para `minimumBalance` no método `MakeWithdrawal`:

C#

```
if (Balance - amount < minimumBalance)
```

Depois de estender a classe `BankAccount`, você pode modificar o construtor `LineOfCreditAccount` para chamar o novo construtor base, conforme mostrado no seguinte código:

C#

```
public LineOfCreditAccount(string name, decimal initialBalance, decimal creditLimit) : base(name, initialBalance, -creditLimit)
{
}
```

Observe que o construtor `LineOfCreditAccount` altera o sinal do parâmetro `creditLimit` para que corresponda ao significado do parâmetro `minimumBalance`.

## Regras de cheque especial diferentes

O último recurso a ser adicionado permite que `LineOfCreditAccount` cobre uma taxa por ultrapassar o limite de crédito em vez de recusar a transação.

Uma técnica é definir uma função virtual na qual você implementa o comportamento necessário. A classe `BankAccount` refatora o método `MakeWithdrawal` em dois métodos. O novo método faz a ação especificada quando o saque leva o saldo abaixo do mínimo. O método existente `MakeWithdrawal` tem o seguinte código:

C#

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    if (Balance - amount < _minimumBalance)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}
```

Substitua-o pelo seguinte código:

```
C#  
  
public void MakeWithdrawal(decimal amount, DateTime date, string note)  
{  
    if (amount <= 0)  
    {  
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of  
withdrawal must be positive");  
    }  
    Transaction? overdraftTransaction = CheckWithdrawalLimit(Balance -  
amount < _minimumBalance);  
    Transaction? withdrawal = new(-amount, date, note);  
    _allTransactions.Add(withdrawal);  
    if (overdraftTransaction != null)  
        _allTransactions.Add(overdraftTransaction);  
}  
  
protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)  
{  
    if (isOverdrawn)  
    {  
        throw new InvalidOperationException("Not sufficient funds for this  
withdrawal");  
    }  
    else  
    {  
        return default;  
    }  
}
```

O método adicionado é `protected`, o que significa que ele pode ser chamado apenas de classes derivadas. Essa declaração impede que outros clientes chamem o método. É também `virtual` para que classes derivadas possam alterar o comportamento. O tipo de retorno é `Transaction?`. A anotação `?` indica que o método pode retornar `null`. Adicione a seguinte implementação para `LineOfCreditAccount` cobrar uma taxa quando o limite de saque for excedido:

```
C#  
  
protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>  
    isOverdrawn  
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")  
    : default;
```

A substituição retorna uma transação de taxa quando a conta é sacada. Se o saque não ultrapassar o limite, o método retornará uma transação `null`. Isso indica que não há

nenhuma taxa. Teste essas alterações adicionando o seguinte código ao seu método `Main` na classe `Program`:

```
C#  
  
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);  
// How much is too much to borrow?  
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly  
advance");  
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");  
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for  
repairs");  
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on  
repairs");  
lineOfCredit.PerformMonthEndTransactions();  
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Execute o programa e verifique os resultados.

## Resumo

Se você não conseguir avançar, veja a origem deste tutorial [em nosso repositório GitHub](#).

Este tutorial demonstrou muitas das técnicas usadas na programação Orientada por objeto:

- Você usou *Abstração* quando definiu classes para cada um dos diferentes tipos de conta. Essas classes descreveram o comportamento desse tipo de conta.
- Você usou *Encapsulamento* quando manteve muitos detalhes `private` em cada classe.
- Você usou *Herança* quando aproveitou a implementação já criada na classe para salvar o código `BankAccount`.
- Você usou *Polimorfismo* ao criar métodos `virtual` que as classes derivadas poderiam substituir para criar um comportamento específico para esse tipo de conta.

# Herança em C# e .NET

Artigo • 02/06/2023

Este tutorial apresenta a herança em C#. Herança é um recurso das linguagens de programação orientadas a objeto que permite a definição de uma classe base que, por sua vez, fornece uma funcionalidade específica (dados e comportamento), e a definição de classes derivadas que herdam ou substituem essa funcionalidade.

## Pré-requisitos

- Recomendamos o [Visual Studio](#) para Windows ou Mac. Você pode baixar uma versão gratuita na [página de downloads do Visual Studio](#). O Visual Studio inclui o SDK do .NET.
- Você também pode usar o editor do [Visual Studio Code](#). Será preciso instalar o [SDK do .NET](#) mais recente separadamente.
- Se preferir outro editor, você precisará instalar o [SDK do .NET](#) mais recente.

## Como executar os exemplos

Para criar e executar os exemplos neste tutorial, use o utilitário `dotnet` na linha de comando. Execute estas etapas para cada exemplo:

1. Crie um diretório para armazenar o exemplo.
2. Insira o comando `dotnet new console` no prompt de comando para criar um novo projeto do .NET Core.
3. Copie e cole o código do exemplo em seu editor de código.
4. Insira o comando `dotnet restore` na linha de comando para carregar ou restaurar as dependências do projeto.

Não é necessário executar `dotnet restore`, pois ele é executado implicitamente por todos os comandos que exigem uma restauração, como `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` e `dotnet pack`. Para desabilitar a restauração implícita, use a opção `--no-restore`.

O comando `dotnet restore` ainda é útil em determinados cenários em que realizar uma restauração explícita faz sentido, como [compilações de integração contínua no Azure DevOps Services](#) ou em sistemas de compilação que precisam controlar explicitamente quando a restauração ocorrerá.

Para obter informações sobre como gerenciar feeds do NuGet, confira a [documentação do dotnet restore](#).

5. Insira o comando `dotnet run` para compilar e executar o exemplo.

## Informações: O que é a herança?

*Herança* é um dos atributos fundamentais da programação orientada a objeto. Ela permite que você defina uma classe filha que reutiliza (herda), estende ou modifica o comportamento de uma classe pai. A classe cujos membros são herdados é chamada de *classe base*. A classe que herda os membros da classe base é chamada de *classe derivada*.

C# e .NET oferecem suporte apenas à *herança única*. Ou seja, uma classe pode herdar apenas de uma única classe. No entanto, a herança é transitiva, o que permite que você defina uma hierarquia de herança para um conjunto de tipos. Em outras palavras, o tipo `D` pode herdar do tipo `C`, que herda do tipo `B`, que herda do tipo de classe base `A`. Como a herança é transitiva, os membros do tipo `A` estão disponíveis ao tipo `D`.

Nem todos os membros de uma classe base são herdados por classes derivadas. Os membros a seguir não são herdados:

- **Construtores estáticos**, que inicializam os dados estáticos de uma classe.
- **Construtores de instância**, que você chama para criar uma nova instância da classe. Cada classe deve definir seus próprios construtores.
- **Finalizadores**, que são chamados pelo coletor de lixo do runtime para destruir as instâncias de uma classe.

Enquanto todos os outros membros de uma classe base são herdados por classes derivadas, o fato de serem visíveis ou não depende de sua acessibilidade. A acessibilidade de um membro afeta sua visibilidade para classes derivadas da seguinte maneira:

- Membros **Privados** são visíveis apenas em classes derivadas que estão aninhadas em sua classe base. Caso contrário, eles não são visíveis em classes derivadas. No exemplo a seguir, `A.B` é uma classe aninhada derivada de `A`, e `C` deriva de `A`. O campo particular `A._value` é visível em `A.B`. No entanto, se você remover os comentários do método `C.GetValue` e tentar compilar o exemplo, ele produzirá um erro do compilador CS0122: "'A.\_value' está inacessível em virtude do nível de proteção dele".

```
C#  
  
public class A  
{  
    private int _value = 10;  
  
    public class B : A  
    {  
        public int GetValue()  
        {  
            return _value;  
        }  
    }  
}  
  
public class C : A  
{  
    //    public int GetValue()  
    //    {  
    //        return _value;  
    //    }  
}  
  
public class AccessExample  
{  
    public static void Main(string[] args)  
    {  
        var b = new A.B();  
        Console.WriteLine(b.GetValue());  
    }  
}  
// The example displays the following output:  
//      10
```

- Membros **Protegidos** são visíveis apenas em classes derivadas.
- Membros **Internos** são visíveis apenas em classes derivadas localizadas no mesmo assembly que a classe base. Eles não são visíveis em classes derivadas localizadas em um assembly diferente da classe base.
- Membros **Públicos** são visíveis em classes derivadas e fazem parte da interface pública da classe derivada. Os membros públicos herdados podem ser chamados como se estivessem definidos na classe derivada. No exemplo a seguir, a classe **A** define um método chamado **Method1**, e a classe **B** herda da classe **A**. Depois, o exemplo chama **Method1** como se fosse um método de instância em **B**.

```
C#  
  
public class A  
{
```

```

    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new ();
        b.Method1();
    }
}

```

Classes derivadas também podem *substituir* membros herdados fornecendo uma implementação alternativa. Para poder substituir um membro, o membro na classe base deve ser marcado com a palavra-chave `virtual`. Por padrão, os membros da classe base não são marcados como `virtual` e não podem ser substituídos. A tentativa de substituir um membro não `virtual`, como o seguinte exemplo faz, gera o erro do compilador CS0506: "O <membro> não pode substituir o membro herdado <membro>, pois não está marcado como `virtual`, abstrato ou de substituição."

C#

```

public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}

```

Em alguns casos, uma classe derivada *deve* substituir a implementação da classe base. Membros de classe base marcados com a palavra-chave `abstrato` exigem que as classes derivadas os substituam. A tentativa de compilar o exemplo a seguir gera um erro do

compilador CS0534, a "<classe> não implementa o membro abstrato herdado <membro>", pois a classe `B` não fornece uma implementação para `A.Method1`.

```
C#  
  
public abstract class A  
{  
    public abstract void Method1();  
}  
  
public class B : A // Generates CS0534.  
{  
    public void Method3()  
    {  
        // Do something.  
    }  
}
```

A herança se aplica apenas a classes e interfaces. Outras categorias de tipo (structs, delegados e enumerações) não dão suporte à herança. Devido a essas regras, tentar compilar código como o seguinte exemplo, produz o erro do compilador CS0527: "Tipo 'ValueType' na lista de interfaces não é uma interface". A mensagem de erro indica que, embora você possa definir as interfaces que um struct implementa, não há suporte para a herança.

```
C#  
  
public struct ValueStructure : ValueType // Generates CS0527.  
{  
}
```

## Herança implícita

Apesar dos tipos possíveis de herança por meio de herança única, todos os tipos no sistema de tipo .NET herdam implicitamente de `Object` ou de um tipo derivado dele. A funcionalidade comum de `Object` está disponível para qualquer tipo.

Para ver o que significa herança implícita, vamos definir uma nova classe, `SimpleClass`, que é simplesmente uma definição de classe vazia:

```
C#  
  
public class SimpleClass  
{ }
```

É possível usar reflexão (o que permite inspecionar os metadados de um tipo para obter informações sobre esse tipo) para obter uma lista dos membros que pertencem ao tipo `SimpleClass`. Embora você ainda não tenha definido membros na classe `SimpleClass`, a saída do exemplo indica que, na verdade, ela tem nove membros. Um desses membros é um construtor sem parâmetros (ou padrão) que é fornecido automaticamente para o tipo `SimpleClass` pelo compilador de C#. Os oito são membros do `Object`, o tipo do qual todas as classes e interfaces no sistema do tipo .NET herdam implicitamente.

C#

```
using System.Reflection;

public class SimpleClassExample
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static |
        BindingFlags.Public |
                           BindingFlags.NonPublic |
        BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (MemberInfo member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
                if (method.IsStatic)
                    stat = " Static";
            }
            string output = $"{member.Name} ({member.MemberType}): {access}
{stat}, Declared by {member.DeclaringType}";
            Console.WriteLine(output);
        }
    }
}

// The example displays the following output:
// Type SimpleClass has 9 members:
// ToString (Method): Public, Declared by System.Object
```

```
// Equals (Method): Public, Declared by System.Object
// Equals (Method): Public Static, Declared by System.Object
// ReferenceEquals (Method): Public Static, Declared by System.Object
// GetHashCode (Method): Public, Declared by System.Object
// GetType (Method): Public, Declared by System.Object
// Finalize (Method): Internal, Declared by System.Object
// MemberwiseClone (Method): Internal, Declared by System.Object
// .ctor (Constructor): Public, Declared by SimpleClass
```

A herança implícita da classe `Object` torna esses métodos disponíveis para a classe `SimpleClass`:

- O método `ToString` público, que converte um objeto `SimpleClass` em sua representação de cadeia de caracteres, retorna o nome de tipo totalmente qualificado. Nesse caso, o método `ToString` retorna a cadeia de caracteres "SimpleClass".
- Três métodos de teste de igualdade de dois objetos: o método `Equals(Object)` da instância pública, o método `Equals(Object, Object)` do público estático e o método `ReferenceEquals(Object, Object)` de público estático. Por padrão, esses métodos testam a igualdade de referência; ou seja, para que seja iguais, duas variáveis de objeto devem fazer referência ao mesmo objeto.
- O método público `GetHashCode`, que calcula um valor que permite o uso de uma instância do tipo em conjuntos de hash.
- O método público `GetType`, que retorna um objeto `Type` que representa o tipo `SimpleClass`.
- O método protegido `Finalize`, que é projetado para liberar recursos não gerenciados antes que a memória de um objeto seja reivindicada pelo coletor de lixo.
- O método protegido `MemberwiseClone`, que cria um clone superficial do objeto atual.

Devido à herança implícita, você pode chamar qualquer membro herdado de um objeto `SimpleClass` como se ele fosse, na verdade, um membro definido na classe `SimpleClass`. Por exemplo, o exemplo a seguir chama o método `SimpleClass.ToString`, que `SimpleClass` herda de `Object`.

C#

```
public class EmptyClass
{ }
```

```

public class ClassNameExample
{
    public static void Main()
    {
        EmptyClass sc = new();
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//      EmptyClass

```

A tabela a seguir lista as categorias de tipos que você pode criar em C#, e os tipos de onde eles herdam implicitamente. Cada tipo base disponibiliza um conjunto diferente de membros por meio de herança para tipos derivados implicitamente.

Categoria do tipo	Herda implicitamente de
classe	Object
struct	ValueType, Object
enum	Enum, ValueType, Object
delegado	MulticastDelegate, Delegate, Object

## Herança e um relacionamento "é um(a)"

Normalmente, a herança é usada para expressar um relacionamento "é um(a)" entre uma classe base e uma ou mais classes derivadas, em que as classes derivadas são versões especializadas da classe base; a classe derivada é um tipo de classe base. Por exemplo, a classe `Publication` representa uma publicação de qualquer tipo e as classes `Book` e `Magazine` representam tipos específicos de publicações.

### ⓘ Observação

Uma classe ou struct pode implementar uma ou mais interfaces. Embora a implementação da interface seja apresentada geralmente como uma alternativa para herança única, ou como uma forma de usar a herança com structs, ela tem como objetivo expressar um relacionamento diferente (um relacionamento "pode fazer") entre uma interface e seu tipo de implementação em comparação com a herança. Uma interface define um subconjunto de funcionalidades (como a capacidade de testar a igualdade, comparar ou classificar objetos ou dar suporte à formatação e análise sensível à cultura) que disponibiliza para seus tipos de implementação.

Observe que "é um(a)" também expressa o relacionamento entre um tipo e uma instanciação específica desse tipo. No exemplo a seguir, `Automobile` é uma classe que tem três propriedades somente leitura exclusivas: `Make`, o fabricante do automóvel; `Model`, o tipo de automóvel; e `Year`, o ano de fabricação. A classe `Automobile` também tem um construtor cujos argumentos são atribuídos aos valores de propriedade. Ela também substitui o método `Object.ToString` para produzir uma cadeia de caracteres que identifica exclusivamente a instância `Automobile` em vez da classe `Automobile`.

C#

```
public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
            throw new ArgumentNullException(nameof(make), "The make cannot
be null.");
        else if (string.IsNullOrWhiteSpace(make))
            throw new ArgumentException("make cannot be an empty string or
have space characters only.");
        Make = make;

        if (model == null)
            throw new ArgumentNullException(nameof(model), "The model cannot
be null.");
        else if (string.IsNullOrWhiteSpace(model))
            throw new ArgumentException("model cannot be an empty string or
have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
            throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

Nesse caso, você não deve depender da herança para representar marcas e modelos de carro específicos. Por exemplo, você não precisa definir um tipo `Packard` para representar automóveis fabricados pela empresa Packard Motor Car. Nesse caso, é

possível representá-los criando um objeto `Automobile` com os valores apropriados passados ao construtor de classe, como no exemplo a seguir.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}

// The example displays the following output:
//      1948 Packard Custom Eight
```

Um relacionamento é-um(a) baseado na herança é mais bem aplicado a uma classe base e em classes derivadas que adicionam outros membros à classe base, ou que exigem funcionalidades adicionais não incluídas na classe base.

## Criação da classe base e das classes derivadas

Vamos examinar o processo de criação de uma classe base e de suas classes derivadas. Nesta seção, você definirá uma classe base, `Publication`, que representa uma publicação de qualquer natureza, como um livro, uma revista, um jornal, um diário, um artigo etc. Você também definirá uma classe `Book` que deriva de `Publication`. É possível estender facilmente o exemplo para definir outras classes derivadas, como `Magazine`, `Journal`, `Newspaper` e `Article`.

### A classe base de Publicação

Em projetar a classe `Publication`, você precisa tomar várias decisões de design:

- Quais membros devem ser incluídos na classe base `Publication` e se os membros de `Publication` fornecem implementações de método, ou se `Publication` é uma classe base abstrata que funciona como um modelo para suas classes derivadas.

Nesse caso, a classe `Publication` fornecerá implementações de método. A seção [Criação de classes base abstratas e de suas classes derivadas](#) contém um exemplo que usa uma classe base abstrata para definir os métodos que as classes derivadas

devem substituir. As classes derivadas são livres para fornecer qualquer implementação adequada ao tipo derivado.

A capacidade de reutilizar o código (ou seja, várias classes derivadas compartilham a declaração e a implementação dos métodos de classe base, e não é necessário substituí-las) é uma vantagem das classes base não abstratas. Portanto, você deverá adicionar membros à `Publication` se o código precisar ser compartilhado por um ou mais tipos `Publication` especializados. Se você não conseguir fornecer implementações da classe base de forma eficiente, será necessário fornecer implementações de membro praticamente idênticas em classes derivadas em vez de uma única implementação na classe base. A necessidade de manter o código duplicado em vários locais é uma possível fonte de bugs.

Para maximizar a reutilização do código e criar uma hierarquia de herança lógica e intuitiva, inclua na classe `Publication` apenas dos dados e a funcionalidade comuns a todas as publicações ou à maioria delas. Depois, as classes derivadas implementam os membros exclusivos a determinados tipos de publicação que eles representam.

- O quanto devemos ampliar a hierarquia de classe. Você deseja desenvolver uma hierarquia de três ou mais classes, em vez de simplesmente uma classe base e uma ou mais classes derivadas? Por exemplo, `Publication` poderia ser uma classe base de `Periodical`, que por sua vez é uma classe base de `Magazine`, `Journal` e `Newspaper`.

Para o seu exemplo, você usará a hierarquia pequena de uma classe `Publication` e uma única classe derivada, a `Book`. É possível ampliar o exemplo facilmente para criar várias classes adicionais derivadas de `Publication`, como `Magazine` e `Article`.

- Se faz sentido instanciar a classe base. Se não fizer, você deverá aplicar a palavra-chave `abstract` à classe. Caso contrário, a instância da classe `Publication` poderá ser criada chamando seu construtor de classe. Se for feita uma tentativa de criar uma instância de classe marcada com a palavra-chave `abstract` por uma chamada direta ao construtor dela, o compilador C# vai gerar o erro CS0144, "Impossível criar uma instância da interface ou da classe abstrata". Se for feita uma tentativa de instanciar a classe usando reflexão, o método de reflexão vai gerar o erro `MemberAccessException`.

Por padrão, uma classe base pode ser instanciada chamando seu construtor de classe. Você não precisa definir um construtor de classe explicitamente. Se não houver um presente no código-fonte da classe base, o compilador de C# fornecerá automaticamente um construtor (sem parâmetros) padrão.

No seu exemplo, você marcará a classe `Publication` como `abstract`, para que não seja possível criar uma instância dela. Uma classe `abstract` sem nenhum método `abstract` indica que essa classe representa um conceito abstrato que é compartilhado entre várias classes concretas (como `Book`, `Journal`).

- Se as classes derivadas precisam herdar a implementação de membros específicos da classe base, se elas têm a opção de substituir a implementação da classe base ou se precisam fornecer uma implementação. Use a palavra-chave `abstract` para forçar as classes derivadas a fornecer uma implementação. Use a palavra-chave `virtual` para permitir que as classes derivadas substituam um método de classe base. Por padrão, os métodos definidos na classe base *não* são substituíveis.

A classe `Publication` não tem nenhum método `abstract`, mas a classe em si é `abstract`.

- Se uma classe derivada representa a classe final na hierarquia de herança e não pode ser usada como uma classe base para outras classes derivadas. Por padrão, qualquer classe pode servir como classe base. Você pode aplicar a palavra-chave `sealed` para indicar que uma classe não pode funcionar como uma classe base para quaisquer classes adicionais. A tentativa de derivar de uma classe selada gerou o erro do compilador CS0509, "Não é possível derivar do tipo selado <typeName>."

No seu exemplo, você marcará a classe derivada como `sealed`.

O exemplo a seguir mostra o código-fonte para a classe `Publication`, bem como uma enumeração `PublicationType` que é retornada pela propriedade `Publication.PublicationType`. Além dos membros herdados de `Object`, a classe `Publication` define os seguintes membros exclusivos e substituições de membro:

C#

```
public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
    private bool _published = false;
    private DateTime _datePublished;
    private int _totalPages;

    public Publication(string title, string publisher, PublicationType type)
    {
        if (string.IsNullOrWhiteSpace(publisher))
            throw new ArgumentException("The publisher is required.");
        Publisher = publisher;
    }

    public string Title { get; set; }
    public string Publisher { get; set; }
    public DateTime DatePublished { get; set; }
    public int TotalPages { get; set; }
}
```

```

        if (string.IsNullOrWhiteSpace(title))
            throw new ArgumentException("The title is required.");
        Title = title;

        Type = type;
    }

    public string Publisher { get; }

    public string Title { get; }

    public PublicationType Type { get; }

    public string? CopyrightName { get; private set; }

    public int CopyrightDate { get; private set; }

    public int Pages
    {
        get { return _totalPages; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException(nameof(value), "The
number of pages cannot be zero or negative.");
            _totalPages = value;
        }
    }

    public string GetPublicationDate()
    {
        if (!_published)
            return "NYP";
        else
            return _datePublished.ToString("d");
    }

    public void Publish(DateTime datePublished)
    {
        _published = true;
        _datePublished = datePublished;
    }

    public void Copyright(string copyrightName, int copyrightDate)
    {
        if (string.IsNullOrWhiteSpace(copyrightName))
            throw new ArgumentException("The name of the copyright holder is
required.");
        CopyrightName = copyrightName;

        int currentYear = DateTime.Now.Year;
        if (copyrightDate < currentYear - 10 || copyrightDate > currentYear
+ 2)
            throw new ArgumentOutOfRangeException($"The copyright year must

```

```
be between {currentYear - 10} and {currentYear + 1}");  
    CopyrightDate = copyrightDate;  
}  
  
public override string ToString() => Title;  
}
```

- Um construtor

Como a classe `Publication` é `abstract`, sua instância não pode ser criada diretamente no código, com no exemplo a seguir:

C#

```
var publication = new Publication("Tiddlywinks for Experts", "Fun and  
Games",  
                                  PublicationType.Book);
```

No entanto, o construtor de instância pode ser chamado diretamente dos construtores de classe derivada, como mostra o código-fonte para a classe `Book`.

- Duas propriedades relacionadas à publicação

`Title` é uma propriedade `String` somente leitura cujo valor é fornecido pela chamada do construtor `Publication`.

`Pages` é uma propriedade `Int32` de leitura-gravação que indica o número total de páginas da publicação. O valor é armazenado em um campo privado chamado `totalPages`. O lançamento deve ser de um número positivo ou de um `ArgumentOutOfRangeException`.

- Membros relacionados ao publicador

Duas propriedades somente leitura, `Publisher` e `Type`. Originalmente, os valores eram fornecidos pela chamada para o construtor da classe `Publication`.

- Membros relacionados à publicação

Dois métodos, `Publish` e `GetPublicationDate`, definem e retornam a data de publicação. O método `Publish` define um sinalizador particular `published` como `true` quando é chamado e atribui a data passada para ele como argumento ao campo particular `datePublished`. O método `GetPublicationDate` retorna a cadeia de caracteres "NYP" se o sinalizador `published` for `false`, e o valor do campo `datePublished` for `true`.

- Membros relacionados a direitos autorais

O método `Copyright` usa o nome do proprietário dos direitos autorais e o ano dos direitos autorais como argumentos e os atribui às propriedades `CopyrightName` e `CopyrightDate`.

- Uma substituição do método `ToString`

Se um tipo não substituir o método `Object.ToString`, ele retornará o nome totalmente qualificado do tipo, que é de pouca utilidade na diferenciação de uma instância para outra. A classe `Publication` substitui `Object.ToString` para retornar o valor da propriedade `Title`.

A figura a seguir ilustra o relacionamento entre a classe base `Publication` e sua classe herdada implicitamente `Object`.

<b>Object</b>	<b>Publication</b>
Equals(Object)	Equals(Object)
Equals(Object, Object)	Equals(Object, Object)
Finalize()	Finalize()
GetHashCode()	GetHashCode()
GetType()	GetType()
MemberwiseClone()	MemberwiseClone()
ReferenceEquals()	ReferenceEquals()
ToString()	ToString()
#ctor()	#ctor(String, String, PublicationType)
	PublicationType
	Publisher
	Title
	CopyrightDate
	CopyrightName
	Pages
	Copyright()
	GetPublicationDate()
	Publish()

**Key**

Unique member	
Inherited member	
Overridden member	

## A classe `Book`

A classe `Book` representa um livro como tipo especializado de publicação. O exemplo a seguir mostra o código-fonte para a classe `Book`.

C#

```
using System;

public sealed class Book : Publication
{
    public Book(string title, string author, string publisher) :
        this(title, string.Empty, author, publisher)
    { }

    public Book(string title, string isbn, string author, string publisher)
        : base(title, publisher, PublicationType.Book)
    {
        // isbn argument must be a 10- or 13-character numeric string
        // without "-" characters.
        // We could also determine whether the ISBN is valid by comparing
        // its checksum digit
        // with a computed checksum.
        //
        if (!string.IsNullOrEmpty(isbn))
        {
            // Determine if ISBN length is correct.
            if (!(isbn.Length == 10 || isbn.Length == 13))
                throw new ArgumentException("The ISBN must be a 10- or 13-
character numeric string.");
            if (!ulong.TryParse(isbn, out _))
                throw new ArgumentException("The ISBN can consist of numeric
characters only.");
        }
        ISBN = isbn;

        Author = author;
    }

    public string ISBN { get; }

    public string Author { get; }

    public decimal Price { get; private set; }

    // A three-digit ISO currency symbol.
    public string? Currency { get; private set; }

    // Returns the old price, and sets a new price.
    public decimal SetPrice(decimal price, string currency)
    {
        if (price < 0)
            throw new ArgumentOutOfRangeException(nameof(price), "The price
cannot be negative.");
        decimal oldValue = Price;
        Price = price;

        if (currency.Length != 3)
            throw new ArgumentException("The ISO currency symbol is a 3-
```

```

        character string.");
        Currency = currency;

        return oldValue;
    }

    public override bool Equals(object? obj)
    {
        if (obj is not Book book)
            return false;
        else
            return ISBN == book.ISBN;
    }

    public override int GetHashCode() => ISBN.GetHashCode();

    public override string ToString() => $"{string.IsNullOrEmpty(Author) ?
"" : Author + ", "}{Title}";
}

```

Além dos membros herdados de `Publication`, a classe `Book` define os seguintes membros exclusivos e substituições de membro:

- Dois construtores

Os dois construtores `Book` compartilham três parâmetros comuns. Dois, *title* e *publisher*, correspondem aos parâmetros do construtor `Publication`. O terceiro é *author*, que é armazenado em uma propriedade pública `Author` imutável. Um construtor inclui um parâmetro *isbn*, que é armazenado na propriedade automática `ISBN`.

O primeiro construtor usa a palavra-chave `this` para chamar o outro construtor. O encadeamento do construtor é um padrão comum na definição de construtores. Construtores com menos parâmetros fornecem valores padrão ao chamar o construtor com o maior número de parâmetros.

O segundo construtor usa a palavra-chave `base` para passar o título e o nome do publicador para o construtor da classe base. Se você não fizer uma chamada explícita para um construtor de classe base em seu código-fonte, o compilador de C# fornecerá automaticamente uma chamada para a classe base padrão ou para o construtor sem parâmetros.

- Uma propriedade `ISBN` somente leitura, que retorna o ISBN (International Standard Book Number) do objeto `Book`, um número exclusivo com 10 ou 13 dígitos. O ISBN é fornecido como um argumento para um dos construtores `Book`.

O ISBN é armazenado em um campo de suporte particular, gerado automaticamente pelo compilador.

- Uma propriedade `Author` somente leitura. O nome do autor é fornecido como um argumento para os dois construtores `Book` e é armazenado na propriedade.
- Duas propriedades somente leitura relacionadas ao preço, `Price` e `Currency`. Seus valores são fornecidos como argumentos em uma chamada do método `SetPrice`. A propriedade `Currency` é o símbolo de moeda ISO de três dígitos (por exemplo, USD para dólar americano). Símbolos de moeda ISO podem ser obtidos da propriedade `ISOCurrencySymbol`. Ambas as propriedades são somente leitura externamente, mas podem ser definidas por código na classe `Book`.
- Um método `SetPrice`, que define os valores das propriedades `Price` e `Currency`. Esses valores são retornados por essas mesmas propriedades.
- Substitui o método `ToString` (herdado de `Publication`) e os métodos `Object.Equals(Object)` e `GetHashCode` (herdados de `Object`).

A menos que seja substituído, o método `Object.Equals(Object)` testa a igualdade de referência. Ou seja, duas variáveis de objeto são consideradas iguais se fizerem referência ao mesmo objeto. Na classe `Book`, por outro lado, dois objetos `Book` devem ser iguais quando têm o mesmo ISBN.

Ao substituir o método `Object.Equals(Object)`, substitua também o método `GetHashCode`, que retorna um valor usado pelo runtime para armazenar itens em coleções de hash para uma recuperação eficiente. O código de hash deve retornar um valor que é consistente com o teste de igualdade. Como você substituiu `Object.Equals(Object)` para retornar `true`, se as propriedades de ISBN de dois objetos `Book` forem iguais, retorne o código hash computado chamando o método `GetHashCode` da cadeia de caracteres retornada pela propriedade `ISBN`.

A figura a seguir ilustra o relacionamento entre a classe `Book` e `Publication`, sua classe base.

## Publication

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, PublicationType)
PublicationType
Publisher
Title
CopyrightDate
CopyrightName
Pages
Copyright()
GetPublicationDate()
Publish()

## Book

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, String)
#ctor(String, String, String, String)
PublicationType
Publisher
Author
Title
CopyrightDate
CopyrightName
ISBN
Pages
Price
Currency
Copyright()
GetPublicationDate()
Publish()
SetPrice()

### Key

Unique member	
Inherited member	
Overridden member	

Agora você pode criar a instância de um objeto `Book`, invocar seus membros exclusivos e herdados e passá-lo como um argumento a um método que espera um parâmetro do tipo `Publication` ou do tipo `Book`, como mostra o exemplo a seguir.

C#

```
public class ClassExample
{
    public static void Main()
    {
        var book = new Book("The Tempest", "0971655819", "Shakespeare,
William",
```

```

                "Public Domain Press");
ShowPublicationInfo(book);
book.Publish(new DateTime(2016, 8, 18));
ShowPublicationInfo(book);

    var book2 = new Book("The Tempest", "Classic Works Press",
"Shakespeare, William");
    Console.WriteLine($"{book.Title} and {book2.Title} are the same
publication: " +
        $"{{((Publication)book).Equals(book2)}");
}

public static void ShowPublicationInfo(Publication pub)
{
    string pubDate = pub.GetPublicationDate();
    Console.WriteLine($"{pub.Title}, " +
        $"{(pubDate == "NYP" ? "Not Yet Published" : "published on
" + pubDate):d} by {pub.Publisher}");
}
// The example displays the following output:
//      The Tempest, Not Yet Published by Public Domain Press
//      The Tempest, published on 8/18/2016 by Public Domain Press
//      The Tempest and The Tempest are the same publication: False

```

## Criando classes base abstratas e suas classes derivadas

No exemplo anterior, você definiu uma classe base que forneceu uma implementação de diversos métodos para permitir que classes derivadas compartilhem código. Em muitos casos, no entanto, não espera-se que a classe base forneça uma implementação. Nesse caso, a classe base é uma *classe abstrata* que declara *métodos abstratos*. Ela funciona como um modelo que define os membros que cada classe derivada precisa implementar. Normalmente em uma classe base abstrata, a implementação de cada tipo derivado é exclusiva para esse tipo. Você marcou a classe com a palavra-chave `abstract` porque não fazia sentido criar uma instância de um objeto `Publication`, embora a classe fornecesse implementações de funcionalidades comuns para publicações.

Por exemplo, cada forma geométrica bidimensional fechada inclui duas propriedades: área, a extensão interna da forma; e perímetro, ou a distância entre as bordas da forma. A maneira com a qual essas propriedades são calculadas, no entanto, depende completamente da forma específica. A fórmula para calcular o perímetro (ou a circunferência) de um círculo, por exemplo, é diferente do quadrado. A classe `Shape` é uma classe `abstract` com métodos `abstract`. Isso indica que as classes derivadas

compartilham a mesma funcionalidade, mas essas classes derivadas implementam essa funcionalidade de forma diferente.

O exemplo a seguir define uma classe base abstrata denominada `Shape` que define duas propriedades: `Area` e `Perimeter`. Além da classe ser marcada com a palavra-chave `abstract`, cada membro da instância também é marcado com a palavra-chave `abstract`. Nesse caso, o `Shape` também substitui o método `Object.ToString` para retornar o nome do tipo, em vez de seu nome totalmente qualificado. E define dois membros estáticos, `GetArea` e `GetPerimeter`, que permitem a recuperação fácil da área e do perímetro de uma instância de qualquer classe derivada. Quando você passa uma instância de uma classe derivada para um desses métodos, o runtime chama a substituição do método da classe derivada.

C#

```
public abstract class Shape
{
    public abstract double Area { get; }

    public abstract double Perimeter { get; }

    public override string ToString() => GetType().Name;

    public static double GetArea(Shape shape) => shape.Area;

    public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

Em seguida, você pode derivar algumas classes de `Shape` que representam formas específicas. O exemplo a seguir define três classes, `Square`, `Rectangle` e `Circle`. Cada uma usa uma fórmula exclusiva para essa forma específica para calcular a área e o perímetro. Algumas das classes derivadas também definem propriedades, como `Rectangle.Diagonal` e `Circle.Diameter`, que são exclusivas para a forma que representam.

C#

```
using System;

public class Square : Shape
{
    public Square(double length)
    {
        Side = length;
    }

    public double Side { get; }
```

```

        public override double Area => Math.Pow(Side, 2);

        public override double Perimeter => Side * 4;

        public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
    }

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; }

    public double Width { get; }

    public override double Area => Length * Width;

    public override double Perimeter => 2 * Length + 2 * Width;

    public bool IsSquare() => Length == Width;

    public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) +
Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2),
2);

    public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

    // Define a circumference, since it's the more familiar term.
    public double Circumference => Perimeter;

    public double Radius { get; }

    public double Diameter => Radius * 2;
}

```

O exemplo a seguir usa objetos derivados de `Shape`. Ele cria uma matriz de objetos derivados de `Shape` e chama os métodos estáticos da classe `Shape`, que retorna valores de propriedade `Shape`. O runtime recupera os valores das propriedades substituídas dos

tipos derivados. O exemplo também converte cada objeto `Shape` na matriz ao seu tipo derivado e, se a conversão for bem-sucedida, recupera as propriedades dessa subclasse específica de `Shape`.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                           new Circle(3) };
        foreach (Shape shape in shapes)
        {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                              $"perimeter, {Shape.GetPerimeter(shape)}");
            if (shape is Rectangle rect)
            {
                Console.WriteLine($"    Is Square: {rect.IsSquare()}, "
Diagonal: {rect.Diagonal}");
                continue;
            }
            if (shape is Square sq)
            {
                Console.WriteLine($"    Diagonal: {sq.Diagonal}");
                continue;
            }
        }
    }
    // The example displays the following output:
    //      Rectangle: area, 120; perimeter, 44
    //      Is Square: False, Diagonal: 15.62
    //      Square: area, 25; perimeter, 20
    //      Diagonal: 7.07
    //      Circle: area, 28.27; perimeter, 18.85
}
```

# Como converter com segurança usando a correspondência de padrões e os operadores `is` e `as`

Artigo • 07/04/2023

Como os objetos são polimórficos, é possível que uma variável de tipo de classe base tenha um [tipo](#) derivado. Para acessar os métodos de instância do tipo derivado, é necessário [converter](#) o valor de volta no tipo derivado. No entanto, uma conversão cria o risco de lançar um [InvalidCastException](#). O C# fornece instruções de [correspondência de padrões](#) que executarão uma conversão condicionalmente somente quando ela tiver êxito. O C# também oferece os operadores `is` e `as` para testar se um valor é de um determinado tipo.

O exemplo a seguir mostra como usar a instrução `is` de correspondência de padrões:

C#

```
var g = new Giraffe();
var a = new Animal();
FeedMammals(g);
FeedMammals(a);
// Output:
// Eating.
// Animal is not a Mammal

SuperNova sn = new SuperNova();
TestForMammals(g);
TestForMammals(sn);

static void FeedMammals(Animal a)
{
    if (a is Mammal m)
    {
        m.Eat();
    }
    else
    {
        // variable 'm' is not in scope here, and can't be used.
        Console.WriteLine($"{a.GetType().Name} is not a Mammal");
    }
}

static void TestForMammals(object o)
{
    // You also can use the as operator and test for null
    // before referencing the variable.
    var m = o as Mammal;
```

```

    if (m != null)
    {
        Console.WriteLine(m.ToString());
    }
    else
    {
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
    }
}
// Output:
// I am an animal.
// SuperNova is not a Mammal

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

```

O exemplo anterior demonstra alguns recursos da sintaxe de correspondência de padrões. A instrução `if (a is Mammal m)` combina o teste com uma atribuição de inicialização. A atribuição ocorre apenas quando o teste é bem-sucedido. A variável `m` está somente no escopo na instrução `if` inserida em ela foi atribuída. Não é possível acessar `m` posteriormente no mesmo método. O exemplo anterior também mostra como usar o `as operador` para converter um objeto em um tipo especificado.

Também é possível usar a mesma sintaxe para testar se um tipo que permite valor nulo tem um valor, como mostra o código de exemplo a seguir:

C#

```

int i = 5;
PatternMatchingNullable(i);

int? j = null;
PatternMatchingNullable(j);

double d = 9.78654;
PatternMatchingNullable(d);

PatternMatchingSwitch(i);
PatternMatchingSwitch(j);
PatternMatchingSwitch(d);

```

```

static void PatternMatchingNullable(ValueType? val)
{
    if (val is int j) // Nullable types are not allowed in patterns
    {
        Console.WriteLine(j);
    }
    else if (val is null) // If val is a nullable type with no value, this
    expression is true
    {
        Console.WriteLine("val is a nullable type with the null value");
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}

static void PatternMatchingSwitch(ValueType? val)
{
    switch (val)
    {
        case int number:
            Console.WriteLine(number);
            break;
        case long number:
            Console.WriteLine(number);
            break;
        case decimal number:
            Console.WriteLine(number);
            break;
        case float number:
            Console.WriteLine(number);
            break;
        case double number:
            Console.WriteLine(number);
            break;
        case null:
            Console.WriteLine("val is a nullable type with the null value");
            break;
        default:
            Console.WriteLine("Could not convert " + val.ToString());
            break;
    }
}

```

O exemplo anterior demonstra outros recursos da correspondência de padrões a serem usados com conversões. É possível testar se uma variável tem padrão nulo verificando especificamente o valor `null`. Quando o valor de runtime da variável é `null`, uma instrução `is` que verifica um tipo retorna sempre `false`. A instrução `is` da correspondência de padrões não permite um tipo de valor anulável, como `int?` ou

`Nullable<int>`, mas é possível testar qualquer outro tipo de valor. Os padrões `is` do exemplo anterior não se limitam aos tipos que permitem valor nulo. Você também pode usar esses padrões para testar se uma variável de um tipo de referência tem um valor ou é `null`.

O exemplo anterior também mostra como usar o padrão de tipo em uma expressão `switch` em que a variável pode ser um de muitos tipos diferentes.

Se você quiser testar se uma variável é um determinado tipo sem atribuição a uma nova variável, poderá usar os operadores `is` e `as` para tipos de referência e tipos que permitem valor nulo. O seguinte código mostra como usar as instruções `is` e `as` que faziam parte da linguagem C# antes que a correspondência de padrões fosse introduzida para testar se uma variável é de um determinado tipo:

C#

```
// Use the is operator to verify the type.
// before performing a cast.
Giraffe g = new();
UseIsOperator(g);

// Use the as operator and test for null
// before referencing the variable.
UseAsOperator(g);

// Use pattern matching to test for null
// before referencing the variable
UsePatternMatchingIs(g);

// Use the as operator to test
// an incompatible type.
SuperNova sn = new();
UseAsOperator(sn);

// Use the as operator with a value type.
// Note the implicit conversion to int? in
// the method body.
int i = 5;
UseAsWithNullable(i);

double d = 9.78654;
UseAsWithNullable(d);

static void UseIsOperator(Animal a)
{
    if (a is Mammal)
    {
        Mammal m = (Mammal)a;
        m.Eat();
    }
}
```

```

}

static void UsePatternMatchingIs(Animal a)
{
    if (a is Mammal m)
    {
        m.Eat();
    }
}

static void UseAsOperator(object o)
{
    Mammal? m = o as Mammal;
    if (m is not null)
    {
        Console.WriteLine(m.ToString());
    }
    else
    {
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
    }
}

static void UseAsWithNullable(System.ValueType val)
{
    int? j = val as int?;
    if (j is not null)
    {
        Console.WriteLine(j);
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}

class Animal
{
    public void Eat() => Console.WriteLine("Eating.");
    public override string ToString() => "I am an animal.";
}

class Mammal : Animal { }

class Giraffe : Mammal { }

class SuperNova { }

```

Como você pode ver na comparação desse código com o de correspondência de padrões, a sintaxe de correspondência de padrões oferece recursos mais robustos combinando o teste e a atribuição em uma única instrução. Use a sintaxe de correspondência de padrões sempre que possível.

# Tutorial: Usar padrões correspondentes para criar algoritmos controlados por tipo e controlados por dados

Artigo • 10/05/2023

É possível escrever uma funcionalidade que se comporte como se você tivesse estendido tipos que poderiam estar em outras bibliotecas. Outro uso dos padrões é criar a funcionalidade de que seu aplicativo precisa, mas que não é um recurso fundamental do tipo que está sendo estendido.

Neste tutorial, você aprenderá como:

- ✓ Reconhecer situações em que a correspondência de padrões deverá ser usada.
- ✓ Usar expressões de correspondência de padrões para implementar o comportamento com base em tipos e valores de propriedade.
- ✓ Combinar a correspondência de padrões com outras técnicas para criar algoritmos completos.

## Pré-requisitos

- Recomendamos o [Visual Studio](#) para Windows ou Mac. Você pode baixar uma versão gratuita na [página de downloads do Visual Studio](#). O Visual Studio inclui o SDK do .NET.
- Você também pode usar o editor do [Visual Studio Code](#). Será preciso instalar o [SDK do .NET](#) mais recente separadamente.
- Se preferir outro editor, você precisará instalar o [SDK do .NET](#) mais recente.

Este tutorial pressupõe que você esteja familiarizado com o C# e .NET, incluindo o Visual Studio ou a CLI do .NET.

## Cenários para a correspondência de padrões

O desenvolvimento moderno geralmente inclui a integração de dados de várias fontes e a apresentação de informações e insights de dados em um único aplicativo coeso. Você e sua equipe não terão controle ou acesso a todos os tipos que representam os dados de entrada.

O design orientado a objeto clássico exigiria a criação de tipos em seu aplicativo que representassem cada tipo de dados das várias fontes de dados. O aplicativo, então,

trabalharia com esses novos tipos, criaria hierarquias de herança, métodos virtuais e implementaria abstrações. Essas técnicas funcionam e, às vezes, são as melhores ferramentas. Outras vezes, é possível escrever menos código. Você pode escrever um código mais claro usando técnicas que separam os dados das operações que manipulam esses dados.

Neste tutorial, você vai criar e explorar um aplicativo que usa dados recebidos de várias fontes externas para um único cenário. Você verá como a **correspondência de padrões** fornece uma maneira eficiente para consumir e processar esses dados de maneiras que não eram parte do sistema original.

Imagine, por exemplo, uma área metropolitana principal que está implantando pedágios e preços diferenciados em horário de pico para gerenciar o tráfego. Você escreve um aplicativo que calcula o pedágio de um veículo com base em seu tipo. Posteriormente, as melhorias vão incorporar preços com base no número de ocupantes do veículo. Outros aprimoramentos vão adicionar o preço com base na hora e no dia da semana.

Com base nessa breve descrição, você pode ter elaborado rapidamente uma hierarquia de objetos para modelar esse sistema. No entanto, seus dados são provenientes de várias fontes, como outros sistemas de gerenciamento de registro do veículo. Esses sistemas fornecem classes diferentes para modelar aqueles dados, e você não tem um modelo de objeto único o qual seja possível usar. Neste tutorial, você usará essas classes simplificadas para criar o modelo para os dados do veículo, a partir desses sistemas externos, conforme mostrado no código a seguir:

```
C#  
  
namespace ConsumerVehicleRegistration  
{  
    public class Car  
    {  
        public int Passengers { get; set; }  
    }  
}  
  
namespace CommercialRegistration  
{  
    public class DeliveryTruck  
    {  
        public int GrossWeightClass { get; set; }  
    }  
}  
  
namespace LiveryRegistration  
{  
    public class Taxi  
    {  
        public int Occupants { get; set; }  
    }  
}
```

```
        public int Fares { get; set; }

    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}
```

Faça o download do código inicial no repositório [dotnet/samples](#) do GitHub. É possível ver que as classes de veículos são de sistemas diferentes, e estão em namespaces diferentes. Nenhuma base comum de classe, além da `System.Object` pode ser aproveitada.

## Designs de correspondência de padrões

O cenário usado neste tutorial destaca os tipos de problemas que a correspondência de padrões pode resolver de forma adequada:

- Os objetos com os quais você precisa trabalhar não estão em uma hierarquia de objetos que corresponde aos seus objetivos. É possível que você esteja trabalhando com classes que fazem parte dos sistemas não relacionados.
- A funcionalidade que você está adicionando não faz parte da abstração central dessas classes. A tarifa paga por um veículo *muda* de acordo com diferentes tipos de veículos, mas o pedágio não é uma função principal do veículo.

Quando a *forma* dos dados e as *operações* nos dados não são descritas em conjunto, o recurso de correspondência padrões no C# facilita o trabalho.

## Implementar os cálculos básicos de pedágio

O cálculo mais básico do pedágio dependerá apenas do tipo do veículo:

- Um `Car` será R\$2,00.
- Um `Taxi` será R\$3,50.
- Um `Bus` será R\$5,00.
- Um `DeliveryTruck` será R\$10,00.

Crie uma nova classe `TollCalculator` e implemente a correspondência de padrões no tipo de veículo para obter a quantidade do pedágio. O código a seguir mostra a implementação inicial do `TollCalculator`.

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace Calculators;

public class TollCalculator
{
    public decimal CalculateToll(object vehicle) =>
        vehicle switch
    {
        Car c           => 2.00m,
        Taxi t          => 3.50m,
        Bus b           => 5.00m,
        DeliveryTruck t => 10.00m,
        { }              => throw new ArgumentException(message: "Not a known
vehicle type", paramName: nameof(vehicle)),
        null            => throw new ArgumentNullException(nameof(vehicle))
    };
}
```

O código anterior usa uma expressão `switch` (não igual a uma instrução `switch`) que testa o padrão de `instrução`. A expressão `switch` inicia-se com a variável, `vehicle` no código anterior, seguida pela palavra-chave `switch`. Em seguida, estão os braços `switch` dentro de chaves. A expressão `switch` faz outros refinamentos na sintaxe que circunda a instrução `switch`. A palavra-chave `case` é omitida, e o resultado de cada braço é uma expressão. Os dois últimos braços apresentam um novo recurso de linguagem. O caso `{ }` corresponde a qualquer objeto não nulo que não correspondia a um braço anterior. Este braço captura qualquer tipo incorreto passado para esse método. O caso `{ }` precisa seguir os casos para cada tipo de veículo. Se a ordem for revertida, o caso `{ }` terá precedência. Por fim, o padrão constante `null` detecta quando `null` é passado para esse método. O padrão `null` pode ser o último, porque os outros padrões correspondem apenas a um objeto não nulo do tipo correto.

Você pode testar esse código usando o seguinte código no `Program.cs`:

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

using toll_calculator;
```

```

var tollCalc = new TollCalculator();

var car = new Car();
var taxi = new Taxi();
var bus = new Bus();
var truck = new DeliveryTruck();

Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
Console.WriteLine($"The toll for a truck is
{tollCalc.CalculateToll(truck)}");

try
{
    tollCalc.CalculateToll("this will fail");
}
catch (ArgumentException e)
{
    Console.WriteLine("Caught an argument exception when using the wrong
type");
}
try
{
    tollCalc.CalculateToll(null!);
}
catch (ArgumentNullException e)
{
    Console.WriteLine("Caught an argument exception when using null");
}

```

Esse código está incluído no projeto inicial, mas é comentado. Remova os comentários e você pode testar o que escreveu.

Você está começando a ver como os padrões podem ajudar a criar algoritmos em que o código e os dados estão separados. A expressão `switch` testa o tipo e produz valores diferentes com base nos resultados. Mas isso é somente o começo.

## Adicionar preços de acordo com a ocupação do veículo

A autoridade de pedágio deseja incentivar que os veículos viagem com a capacidade máxima de pessoas. Eles decidiram cobrar valores mais altos quando os veículos tiverem poucos passageiros e oferecer redução da tarifa para veículos com a capacidade total ocupada:

- Os carros e táxis com nenhum passageiro pagam uma taxa adicional de R\$ 0,50.
- Os carros e táxis com dois passageiros obtêm um desconto de R\$ 0,50.

- Os carros e táxis com três ou mais passageiros obtêm um desconto de R\$ 1,00.
- Os ônibus com menos de 50% da capacidade completa pagam uma taxa adicional de R\$ 2,00.
- Os ônibus com 90% da capacidade de passageiros completa, ganham um desconto de R\$ 1,00.

Essas regras podem ser implementadas usando o [padrão de propriedade](#) na mesma expressão switch. Um padrão de propriedade compara um valor de propriedade com um valor constante. O padrão de propriedade examina as propriedades do objeto depois que o tipo foi determinado. O único caso de um `Car` se expande para quatro casos diferentes:

```
C#  
  
vehicle switch  
{  
    Car {Passengers: 0} => 2.00m + 0.50m,  
    Car {Passengers: 1} => 2.0m,  
    Car {Passengers: 2} => 2.0m - 0.50m,  
    Car                 => 2.00m - 1.0m,  
  
    // ...  
};
```

Os três primeiros casos testam o tipo como um `Car`, em seguida, verificam o valor da propriedade `Passengers`. Se ambos corresponderem, essa expressão é avaliada e retornada.

Você também expande os casos para táxis de maneira semelhante:

```
C#  
  
vehicle switch  
{  
    // ...  
  
    Taxi {Fares: 0}  => 3.50m + 1.00m,  
    Taxi {Fares: 1}  => 3.50m,  
    Taxi {Fares: 2}  => 3.50m - 0.50m,  
    Taxi             => 3.50m - 1.00m,  
  
    // ...  
};
```

Em seguida, implemente as regras de ocupação expandindo os casos para os ônibus, conforme mostrado no exemplo a seguir:

C#

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
    2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
    1.00m,
    Bus => 5.00m,

    // ...
};
```

A autoridade de pedágio não está preocupada com o número de passageiros nos caminhões de carga. Em vez disso, ela ajusta a quantidade de pedágios com base na classe de peso dos caminhões da seguinte maneira:

- Os caminhões mais de 5000 quilos pagam uma taxa adicional de R\$ 5,00.
- Os caminhões leves abaixo de 3.000 lb recebem um desconto de US\$ 2,00.

Essa regra é implementada com o código a seguir:

C#

```
vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,
};
```

O código anterior mostra a cláusula `when` de um braço `switch`. Você usa a cláusula `when` para testar as condições, com exceção da igualdade, em uma propriedade. Ao terminar, o método será muito semelhante ao seguinte código:

C#

```
vehicle switch
{
    Car {Passengers: 0}      => 2.00m + 0.50m,
    Car {Passengers: 1}      => 2.0m,
    Car {Passengers: 2}      => 2.0m - 0.50m,
    Car                      => 2.00m - 1.0m,

    Taxi {Fares: 0}  => 3.50m + 1.00m,
```

```

    Taxi {Fares: 1} => 3.50m,
    Taxi {Fares: 2} => 3.50m - 0.50m,
    Taxi           => 3.50m - 1.00m,

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,

    { }      => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
    null     => throw new ArgumentNullException(nameof(vehicle))
};

```

Muitos desses braços switch são exemplos de **padrões recursivos**. Por exemplo, `Car { Passengers: 1}` mostra um padrão constante dentro de um padrão de propriedade.

É possível fazer esse código menos repetitivo, usando switches aninhados. O `Car` e `Taxi` têm quatro braços diferentes nos exemplos anteriores. Em ambos os casos, você pode criar um padrão de declaração que alimenta um padrão constante. Essa técnica é mostrada no código a seguir:

C#

```

public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },
        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },
        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -

```

```

1.00m,
Bus b => 5.00m,

DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
DeliveryTruck t => 10.00m,

{} => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
null => throw new ArgumentNullException(nameof(vehicle))
};

```

Na amostra anterior, o uso de uma expressão recursiva significa não repetir os braços `Car` e `Taxi` contendo braços filho que testam o valor da propriedade. Essa técnica não é usada para os braços `Bus` e `DeliveryTruck` porque esses estão testando intervalos da propriedade, e não valores discretos.

## Adicionar preço de horário de pico

Para um último recurso, a autoridade de pedágio quer adicionar um preço os horários de pico. Durante os horários de pico da manhã e do final da tarde, os pedágios serão dobrados. Essa regra afetará apenas o tráfego em uma direção: entrada para a cidade, no período da manhã, e de saída da cidade, no período da tarde. Em outros períodos durante o dia útil, os pedágios aumentam 50%. Nos períodos da noite e madrugada e de manhã cedo, as tarifas são 25% mais baratas. Durante o fim de semana, a taxa é normal, independentemente da hora. Você pode usar uma série de instruções `if` e `else` para expressar isso usando o seguinte código:

C#

```

public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{
    if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
        (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
    {
        return 1.0m;
    }
    else
    {
        int hour = timeOfToll.Hour;
        if (hour < 6)
        {
            return 0.75m;
        }
        else if (hour < 10)
        {
            if (inbound)

```

```

    {
        return 2.0m;
    }
    else
    {
        return 1.0m;
    }
}
else if (hour < 16)
{
    return 1.5m;
}
else if (hour < 20)
{
    if (inbound)
    {
        return 1.0m;
    }
    else
    {
        return 2.0m;
    }
}
else // Overnight
{
    return 0.75m;
}
}
}

```

O código anterior funciona corretamente, mas não é legível. Você precisa encadear todos os casos de entrada e as instruções aninhadas `if` para raciocinar sobre o código. Em vez disso, você usará a correspondência de padrões para esse recurso, mas poderá integrá-lo a outras técnicas. É possível criar uma única expressão de correspondência de padrões que leva em conta todas as combinações de direção, dia da semana e hora. O resultado seria uma expressão complicada. Seria difícil de ler e entender. O que dificulta garantir a exatidão. Em vez disso, combine esses métodos para criar uma tupla de valores que descreve de forma concisa todos os estados. Em seguida, use a correspondência de padrões para calcular um multiplicador para o pedágio. A tupla contém três condições distintas:

- O dia é um dia da semana ou do fim de semana.
- A faixa de tempo é quando o pedágio é coletado.
- A direção é para a cidade ou da cidade

A tabela a seguir mostra as combinações de valores de entrada e multiplicador de preços para os horários de pico:

<b>Dia</b>	<b>Hora</b>	<b>Direção</b>	<b>Premium</b>
Weekday	horário de pico da manhã	entrada	x 2,00
Weekday	horário de pico da manhã	saída	x 1,00
Weekday	hora do dia	entrada	x 1,50
Weekday	hora do dia	saída	x 1,50
Weekday	horário de pico do fim da tarde	entrada	x 1,00
Weekday	horário de pico do fim da tarde	saída	x 2,00
Weekday	noite e madrugada	entrada	x 0,75
Weekday	noite e madrugada	saída	x 0,75
Fim de Semana	horário de pico da manhã	entrada	x 1,00
Fim de Semana	horário de pico da manhã	saída	x 1,00
Fim de Semana	hora do dia	entrada	x 1,00
Fim de Semana	hora do dia	saída	x 1,00
Fim de Semana	horário de pico do fim da tarde	entrada	x 1,00
Fim de Semana	horário de pico do fim da tarde	saída	x 1,00
Fim de Semana	noite e madrugada	entrada	x 1,00
Fim de Semana	noite e madrugada	saída	x 1,00

Há 16 combinações diferentes das três variáveis. Ao combinar algumas das condições, você simplificará a expressão switch.

O sistema que coleta os pedágios usa uma estrutura [DateTime](#) para a hora em que o pedágio foi cobrado. Construa métodos de membro que criam as variáveis da tabela anterior. A seguinte função usa como correspondência de padrões a expressão switch para expressar se um [DateTime](#) representa um fim de semana ou um dia útil:

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday     => true,
        DayOfWeek.Tuesday    => true,
        DayOfWeek.Wednesday  => true,
        DayOfWeek.Thursday   => true,
```

```
        DayOfWeek.Friday    => true,
        DayOfWeek.Saturday  => false,
        DayOfWeek.Sunday    => false
    };
}
```

Esse método está correto, mas é redundante. Para simplificar, faça conforme mostrado no código a seguir:

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday   => false,
        _                  => true
    };
}
```

Depois, adicione uma função semelhante para categorizar o tempo nos blocos:

C#

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.OVERNIGHT,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _          => TimeBand.EveningRush,
    };
}
```

Adicione um `enum` privado para converter cada intervalo de tempo em um valor discreto. Em seguida, o método `GetTimeBand` usa [padrões relacionais](#) e [padrões or conjuntivos](#), ambos adicionados no C# 9.0. Um padrão relacional permite testar um valor numérico usando `<`, `>`, `<=` ou `>=`. O padrão `or` testa se uma expressão corresponde a um ou mais padrões. Você também pode usar um padrão `and` para garantir que uma expressão corresponda a dois padrões distintos e um padrão `not` para testar se uma expressão não corresponde a um padrão.

Depois de criar esses métodos, é possível usar outra expressão `switch` com o padrão de `tupla` para calcular o preço premium. Você pode construir uma expressão `switch` com todos os 16 braços:

C#

```
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
{
    (true, TimeBand.MorningRush, true) => 2.00m,
    (true, TimeBand.MorningRush, false) => 1.00m,
    (true, TimeBand.Daytime, true) => 1.50m,
    (true, TimeBand.Daytime, false) => 1.50m,
    (true, TimeBand.EveningRush, true) => 1.00m,
    (true, TimeBand.EveningRush, false) => 2.00m,
    (true, TimeBand.OVERNIGHT, true) => 0.75m,
    (true, TimeBand.OVERNIGHT, false) => 0.75m,
    (false, TimeBand.MorningRush, true) => 1.00m,
    (false, TimeBand.MorningRush, false) => 1.00m,
    (false, TimeBand.Daytime, true) => 1.00m,
    (false, TimeBand.Daytime, false) => 1.00m,
    (false, TimeBand.EveningRush, true) => 1.00m,
    (false, TimeBand.EveningRush, false) => 1.00m,
    (false, TimeBand.OVERNIGHT, true) => 1.00m,
    (false, TimeBand.OVERNIGHT, false) => 1.00m,
};
```

O código acima funciona, mas pode ser simplificado. Todas as oito combinações para o fim de semana têm o mesmo pedágio. É possível substituir todas as oito pela seguinte linha:

C#

```
(false, _, _) => 1.0m,
```

Tanto o tráfego de entrada quanto o de saída têm o mesmo multiplicador durante o dia e a noite, nos dias úteis. Esses quatro braços `switch` podem ser substituídos por estas duas linhas:

C#

```
(true, TimeBand.OVERNIGHT, _) => 0.75m,
(true, TimeBand.DAYTIME, _)   => 1.5m,
```

O código deverá ser semelhante ao seguinte após essas duas alterações:

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true) => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime, _) => 1.50m,
        (true, TimeBand.EveningRush, true) => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight, _) => 0.75m,
        (false, _, _) => 1.00m,
    };
}
```

Por fim, você pode remover os dois horários de pico em que é pago o preço normal. Quando remover essas braços, substitua o `false` por um descarte (`_`) no braço `switch` final. O método concluído será o seguinte:

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.Overnight, _) => 0.75m,
        (true, TimeBand.Daytime, _) => 1.5m,
        (true, TimeBand.MorningRush, true) => 2.0m,
        (true, TimeBand.EveningRush, false) => 2.0m,
        _ => 1.0m,
    };
}
```

Este exemplo destaca uma das vantagens da correspondência de padrões: os branches de padrões são avaliados na ordem. Se você os reorganizar para que um branch anterior trate um dos casos posteriores, o compilador emitirá um aviso sobre o código inacessível. Essas regras de linguagem tornam as simplificações anteriores mais fáceis com a certeza de que o código não foi alterado.

A correspondência de padrões torna alguns tipos de código mais legíveis e oferece uma alternativa às técnicas orientadas a objeto quando não é possível adicionar o código às classes. A nuvem está fazendo com que os dados e a funcionalidade existam separadamente. A *forma* dos dados e as *operações* nela não são necessariamente descritas juntas. Neste tutorial, você utilizou os dados existentes de maneiras completamente diferentes de sua função original. A correspondência de padrões proporcionou a capacidade de escrever a funcionalidade que substituiu esses tipos, ainda que não tenha sido possível estendê-los.

## Próximas etapas

Baixe o código concluído no repositório [dotnet/samples](#) do GitHub. Explore os padrões por conta própria e adicione essa técnica em suas atividades regulares de codificação. Aprender essas técnicas lhe oferece outra maneira de abordar problemas e criar novas funcionalidades.

## Confira também

- [Padrões](#)
- [Expressão switch](#)

# Como manipular uma exceção usando try/catch

Artigo • 07/04/2023

A finalidade de um bloco `try-catch` é capturar e manipular uma exceção gerada pelo código de trabalho. Algumas exceções podem ser manipuladas em um bloco `catch` e o problema pode ser resolvido sem que a exceção seja gerada novamente. No entanto, com mais frequência, a única coisa que você pode fazer é certificar-se de que a exceção apropriada seja gerada.

## Exemplo

Neste exemplo, `IndexOutOfRangeException` não é a exceção mais apropriada: `ArgumentOutOfRangeException` faz mais sentido para o método porque o erro é causado pelo argumento `index` passado pelo chamador.

```
C#  
  
static int GetInt(int[] array, int index)  
{  
    try  
    {  
        return array[index];  
    }  
    catch (IndexOutOfRangeException e) // CS0168  
    {  
        Console.WriteLine(e.Message);  
        // Set IndexOutOfRangeException to the new exception's  
        InnerException.  
        throw new ArgumentException("index parameter is out of  
        range.", e);  
    }  
}
```

## Comentários

O código que causa uma exceção fica dentro do bloco `try`. Uma instrução `catch` é adicionada imediatamente após para lidar com `IndexOutOfRangeException`, caso ocorra. O bloco `catch` manipula o `IndexOutOfRangeException` e gera o `ArgumentException` mais apropriado. Para fornecer ao chamador tantas informações quanto possível, considere especificar a exceção original como o

[InnerException](#) da nova exceção. Uma vez que a propriedade [InnerException](#) é somente leitura, você precisa atribuí-la no construtor da nova exceção.

# Como executar código de limpeza usando finally

Artigo • 14/03/2023

O propósito de uma instrução `finally` é garantir que a limpeza necessária de objetos, normalmente objetos que estão mantendo recursos externos, ocorra imediatamente, mesmo que uma exceção seja lançada. Um exemplo dessa limpeza é chamar `Close` em um `FileStream` imediatamente após o uso, em vez de esperar que o objeto passe pela coleta de lixo feita pelo Common Language Runtime, da seguinte maneira:

C#

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

## Exemplo

Para transformar o código anterior em uma instrução `try-catch-finally`, o código de limpeza é separado do código funcional, da seguinte maneira.

C#

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

```
    }
    finally
    {
        file?.Close();
    }
}
```

Como uma exceção pode ocorrer a qualquer momento no bloco `try` antes da chamada `OpenWrite()` ou a própria chamada `OpenWrite()` poderia falhar, não há garantia de que o arquivo esteja aberto quanto tentarmos fechá-lo. O bloco `finally` adiciona uma verificação para checar se o objeto `FileStream` não é `null` antes de chamar o método `Close`. Sem a verificação de `null`, o bloco `finally` poderia gerar uma `NullReferenceException` própria, mas a geração de exceções em blocos `finally` deve ser evitada, se possível.

Uma conexão de banco de dados é outra boa candidata a ser fechada em um bloco `finally`. Como o número de conexões permitidas para um servidor de banco de dados é, às vezes, limitado, você deve fechar conexões de banco de dados assim que possível. Se uma exceção for gerada antes que você possa fechar a conexão, usar o bloco `finally` será melhor do que aguardar a coleta de lixo.

## Confira também

- [Instrução using](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)

# Novidades do C# 11

Artigo • 20/03/2023

Os seguintes recursos foram adicionados em C# 11:

- Literais brutos de cadeia de caracteres
- Suporte matemático genérico
- Atributos genéricos
- Cadeia de caracteres UTF-8 literais
- Linhas novas em expressões de interpolação de cadeia de caracteres
- Padrões de lista
- Tipos de locais de arquivos
- Membros necessários
- Structs de padrão automático
- Correspondência de padrão `Span<char>` em uma constante `string`
- Escopo estendido `nameof`
- `IntPtr` numérico
- Campos `ref` e `scoped ref`
- Conversão aprimorada do grupo de métodos para delegado
- Ciclo de aviso 7

Baixe a versão mais recente do [Visual Studio 2022](#). Você também pode experimentar todos esses recursos com o SDK do .NET 7, que pode ser baixado na página de [downloads do .NET](#).

## ⓘ Observação

Estamos interessados em seus comentários sobre esses recursos. Se você encontrar problemas com qualquer um desses novos recursos, crie um [problema](#) no repositório [dotnet/roslyn](#).

## Atributos genéricos

Você pode declarar uma [classe genérica](#) cuja classe base é `System.Attribute`. Este recurso fornece uma sintaxe mais conveniente para atributos que exigem um parâmetro `System.Type`. Anteriormente, você precisaria criar um atributo com um `Type` como parâmetro de construtor:

```
// Before C# 11:  
public class TypeAttribute : Attribute  
{  
    public TypeAttribute(Type t) => ParamType = t;  
  
    public Type ParamType { get; }  
}
```

E para aplicar o atributo, você usa o operador `typeof`:

C#

```
[TypeAttribute(typeof(string))]  
public string Method() => default;
```

Usando esse novo recurso, você pode criar um atributo genérico em vez disso:

C#

```
// C# 11 feature:  
public class GenericAttribute<T> : Attribute { }
```

Em seguida, especifique o tipo de parâmetro para usar o atributo:

C#

```
[GenericAttribute<string>()]  
public string Method() => default;
```

Você deve fornecer todos os parâmetros de tipo ao aplicar o atributo. Em outras palavras, o tipo genérico deve ser **totalmente construído**. No exemplo acima, os parênteses vazios (( e )) podem ser omitidos, pois o atributo não tem argumentos.

C#

```
public class GenericType<T>  
{  
    [GenericAttribute<T>()] // Not allowed! generic attributes must be fully  
    // constructed types.  
    public string Method() => default;  
}
```

Os argumentos de tipo devem atender às mesmas restrições do operador `typeof`. Tipos que exigem anotações de metadados não são permitidos. Por exemplo, os seguintes tipos não são permitidos como o parâmetro de tipo:

- `dynamic`
- `string?` (ou qualquer tipo de referência anulável)
- `(int X, int Y)` (ou qualquer outro tipo de tupla usando a sintaxe de tupla C#).

Esses tipos não são representados diretamente em metadados. Elas incluem anotações que descrevem o tipo. Em todos os casos, você pode usar o tipo subjacente em vez disso:

- `object` para `dynamic`.
- `string` em vez de `string?`.
- `ValueTuple<int, int>` em vez de `(int X, int Y)`.

## Suporte matemático genérico

Há vários recursos de linguagem que permitem suporte matemático genérico:

- Membros `static virtual` em interfaces
- operadores verificados definidos pelo usuário
- operadores de deslocamento flexíveis
- operador de deslocamento para a direita sem sinal

Você pode adicionar membros `static abstract` ou `static virtual` em interfaces para definir interfaces que incluem operadores sobrecarregados, outros membros estáticos e propriedades estáticas. O cenário principal para esse recurso é usar operadores matemáticos em tipos genéricos. Por exemplo, você pode implementar a interface `System.IAdditionOperators<TSelf, TOther, TResult>` em um tipo que implementa o `operator +`. Outras interfaces definem outras operações matemáticas ou valores bem definidos. Você pode aprender sobre a nova sintaxe no artigo sobre [interfaces](#).

Interfaces que incluem métodos `static virtual` normalmente são [interfaces genéricas](#). Além disso, a maioria declarará uma restrição de que o parâmetro de tipo [implemente a interface declarada](#).

Você pode saber mais e experimentar o recurso no tutorial [Explorar membros da interface abstrata estática](#) ou na postagem no blog sobre [Versão prévia do recurso no .NET 6 – matemática genérica ↗](#).

A matemática genérica criou outros requisitos na linguagem.

- *Operador de deslocamento para a direita sem sinal*: antes do C# 11, para forçar um deslocamento para a direita sem sinal, você precisaria converter qualquer tipo inteiro com sinal em um tipo sem sinal, executar a mudança e, em seguida,

converter o resultado de volta para um tipo com sinal. A partir do C# 11, você pode usar o `>>>`, o [operador de deslocamento sem sinal](#).

- *Requisitos de operador de deslocamento para a direita sem sinal:* o C# 11 remove o requisito de que o segundo operando deve ser um `int` ou implicitamente conversível para `int`. Essa alteração permite que os tipos que implementam interfaces matemáticas genéricas sejam usados nesses locais.
- *operadores marcados e desmarcados definidos pelo usuário:* os desenvolvedores agora podem definir operadores aritméticos `checked` e `unchecked`. O compilador gera chamadas para a variante correta com base no contexto atual. Você pode ler mais sobre operadores `checked` no artigo sobre [operadores aritméticos](#).

## Numérico `IntPtr` e `UIntPtr`

Os tipos `nint` e `nuint` agora têm como alias [System.IntPtr](#) e [System.UIntPtr](#), respectivamente.

## Novas linhas em interpolações de cadeia de caracteres

O texto dentro dos caracteres `{` e `}` para uma interpolação de cadeia de caracteres agora pode abranger várias linhas. O texto entre os marcadores `{` e `}` é analisado como C#. Qualquer C# legal, inclusive novas linhas, é permitido. Esse recurso facilita a leitura de interpolações de cadeia de caracteres que usam expressões C# mais longas, como expressões `switch` de padrões correspondentes ou consultas LINQ.

Você pode saber mais sobre o recurso de novas linhas no artigo de [interpolações de cadeia de caracteres](#) na referência de linguagem.

## Padrões de lista

*Padrões de lista* estendem a correspondência de padrões para corresponder a sequências de elementos em uma lista ou em uma matriz. Por exemplo, `sequence is [1, 2, 3]` é `true` quando a `sequence` é um matriz ou uma lista de três inteiros (1, 2 e 3). Você pode fazer a correspondência de elementos usando qualquer padrão, inclusive uma constante, tipo, propriedade e padrões relacionais. O padrão de descarte (`_`) corresponde a qualquer elemento único e o novo *padrão de intervalo* (...) corresponde a qualquer sequência de zero ou mais elementos.

Você pode saber mais detalhes sobre padrões de lista no artigo de [padrões correspondentes](#) na referência de linguagem.

## Conversão aprimorada do grupo de métodos para delegado

O padrão C# em [conversões de grupo de métodos](#) agora inclui o seguinte item:

- A conversão é permitida (mas não é necessária) para usar uma instância delegada existente que já contenha essas referências.

As versões anteriores do padrão proibiam o compilador de reutilizar o objeto delegado criado para uma conversão de grupo de métodos. O compilador C# 11 armazena em cache o objeto delegado criado de uma conversão de grupo de métodos e reutiliza esse único objeto delegado. Esse recurso foi disponibilizado pela primeira vez no Visual Studio 2022 versão 17.2 como uma versão prévia do recurso e no .NET 7 Versão Prévia 2.

## Literais de cadeia de caracteres bruta

*Literais de cadeia de caracteres bruta* são um novo formato para literais de cadeia de caracteres. Literais de cadeia de caracteres bruta podem conter texto arbitrário, incluindo espaços em branco, novas linhas, aspas inseridas e outros caracteres especiais sem a necessidade de sequências de escape. Um literal de cadeia de caracteres bruta começa com pelo menos três caracteres de aspas duplas (""""). Ele termina com o mesmo número de caracteres de aspas duplas. Normalmente, um literal de cadeia de caracteres bruta usa três aspas duplas em uma única linha para iniciar a cadeia de caracteres e três aspas duplas em uma linha separada para terminar a cadeia de caracteres. As novas linhas que seguem as aspas de abertura e precedem as aspas de fechamento não estão incluídas no conteúdo final:

C#

```
string longMessage = """
    This is a long message.
    It has several lines.
        Some are indented
            more than others.
    Some should start at the first column.
    Some have "quoted text" in them.
""";
```

Qualquer espaço em branco à esquerda das aspas duplas de fechamento será removido do literal da cadeia de caracteres. Os literais da cadeia de caracteres bruta podem ser combinados com a interpolação de cadeia de caracteres para incluir chaves no texto de saída. Vários caracteres `$` indicam quantas chaves consecutivas iniciam e terminam a interpolação:

C#

```
var location = $$"""
    You are at {{Longitude}}, {{Latitude}}}
""";
```

O exemplo anterior especifica que duas chaves iniciam e terminam uma interpolação. A terceira chave de abertura e de fechamento repetidas são incluídas na cadeia de caracteres de saída.

Você pode saber mais sobre literais de cadeia de caracteres bruta no artigo sobre [cadeias de caracteres no guia de programação](#) e nos artigos de referência de linguagem sobre [literais de cadeia de caracteres](#) e [cadeias de caracteres interpoladas](#).

## Structs de padrão automático

O compilador C# 11 garante que todos os campos de um tipo `struct` sejam inicializados para seu valor padrão como parte da execução de um construtor. Essa alteração significa que qualquer campo ou propriedade automática não inicializados por um construtor são inicializados automaticamente pelo compilador. Os structs em que o construtor não atribui definitivamente todos os campos agora são compilados, e todos os campos não inicializados explicitamente são definidos para o valor padrão. Você pode ler mais sobre como essa alteração afeta a inicialização de structs no artigo sobre [structs](#).

## Correspondência de padrão `Span<char>` e `ReadOnlySpan<char>` em uma constante `string`

Você conseguiu testar se um `string` tinha um valor constante específico usando padrões correspondentes para várias versões. Agora você pode usar a mesma lógica de padrões correspondentes com variáveis que são `Span<char>` ou `ReadOnlySpan<char>`.

## Escopo `nameof` estendido

Nomes de parâmetros de tipo e nomes de parâmetro agora estão no escopo quando usados em uma expressão `nameof` em uma [declaração de atributo](#) nesse método. Essa funcionalidade significa que você pode usar o operador `nameof` para especificar o nome de um parâmetro de método em um atributo na declaração de método ou parâmetro. Esse recurso costuma ser útil para adicionar atributos para [análise anulável](#).

## Cadeia de caracteres UTF-8 literais

Você pode especificar o sufixo `u8` em um literal de cadeia de caracteres para especificar a codificação de caracteres UTF-8. Se o aplicativo precisar de cadeias de caracteres UTF-8, para constantes de cadeia de caracteres HTTP ou protocolos de texto semelhantes, você pode usar esse recurso para simplificar a criação de cadeias de caracteres UTF-8.

Você pode saber mais sobre literais de cadeia de caracteres UTF-8 na seção literal de cadeia de caracteres do artigo sobre [tipos de referência internos](#).

## Membros necessários

Você pode adicionar o [modificador required](#) a propriedades e campos para impor construtores e chamadores para inicializar esses valores. O [System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute](#) pode ser adicionado aos construtores para informar ao compilador que um construtor inicializa *todos* os membros necessários.

Para obter mais informações sobre os membros necessários, consulte a seção [somente init](#) do artigo de propriedades.

## Campos `ref` e variáveis `ref scoped`

Você pode declarar campos `ref` dentro de uma classe [ref struct](#). Isso dá suporte a tipos como [System.Span<T>](#) sem atributos especiais ou tipos internos ocultos.

Você pode adicionar o modificador [scoped](#) a qualquer declaração `ref`. Isso limita o escopo para o qual a referência pode escapar.

## Tipos de locais de arquivo

A partir do C# 11, você pode usar o modificador de acesso `file` para criar um tipo cuja visibilidade está no escopo do arquivo de origem no qual ele é declarado. Esse recurso ajuda os autores do gerador de origem a evitar colisões de nomenclatura. Você pode

saber mais sobre esse recurso no artigo sobre [tipos com escopo de arquivo](#) na referência de linguagem.

## Confira também

- [Novidades do .NET 7](#)

# Novidades do C# 10

Artigo • 10/05/2023

O C# 10 adiciona os seguintes recursos e aprimoramentos à linguagem C#:

- Estruturas de registro
- Aprimoramentos de tipos de estrutura
- Manipuladores de cadeia de caracteres interpolada
- Diretivas global using
- Declaração de namespace com escopo de arquivo
- Padrões de propriedade estendida
- Aprimoramentos nas expressões lambda
- Permissão de cadeias de caracteres interpoladas const
- Tipos de registro podem selar ToString()
- Atribuição definida aprimorada
- Permissão de atribuição e declaração na mesma desconstrução
- Permissão do atributo AsyncMethodBuilder em métodos
- Atributo CallerArgumentExpression
- Pragma #line aprimorado
- Ciclo de aviso 6

O C# 10 tem suporte no .NET 6. Para obter mais informações, confira [Controle de versão da linguagem C#](#).

Você pode baixar o SDK mais recente do .NET 6 na [página de downloads do .NET](#). Você também pode baixar o [Visual Studio 2022](#), que inclui o SDK do .NET 6.

## ⓘ Observação

Estamos interessados em seus comentários sobre esses recursos. Se você encontrar problemas com qualquer um desses novos recursos, crie um [problema](#) no repositório [dotnet/roslyn](#).

## Structs de registro

Você pode declarar registros de tipo de valor usando as declarações `record struct` ou `readonly record struct`. Agora você pode esclarecer que `record` é um tipo de referência com a declaração `record class`.

# Aprimoramentos de tipos de estrutura

O C# 10 apresenta as seguintes melhorias relacionadas aos tipos de estrutura:

- Você pode declarar um construtor sem parâmetros de instância em um tipo de estrutura e inicializar um campo ou propriedade de instância na declaração. Para obter mais informações, consulte a seção [Inicialização de struct e valores padrão](#) do artigo [Tipos de estrutura](#).
- Um operando à esquerda da expressão `with` pode ser de qualquer tipo de estrutura ou um tipo anônimo (referência).

## Manipuladores de cadeia de caracteres interpolada

Você pode criar um tipo que compila a cadeia de caracteres resultante de uma [expressão de cadeia de caracteres interpolada](#). As bibliotecas .NET usam esse recurso em muitas APIs. Você pode criar um [seguindo este tutorial](#).

## Diretivas using globais

Você pode adicionar o modificador `global` a qualquer [diretiva de uso](#) para instruir o compilador ao qual a diretiva se aplica a todos os arquivos de origem na compilação. Normalmente, são todos os arquivos de origem em um projeto.

## Declaração de namespace com escopo de arquivo

Você pode usar uma nova forma da declaração `namespace` para declarar que todas as seguintes declarações são membros do namespace declarado:

C#

```
namespace MyNamespace;
```

Essa nova sintaxe salva espaço horizontal e vertical para declarações `namespace`.

## Padrões de propriedade estendida

A partir do C# 10, você pode referenciar propriedades aninhadas ou campos dentro de um padrão de propriedade. Por exemplo, um padrão do formulário

```
C#  
{ Prop1.Prop2: pattern }
```

é válido em C# 10 e posterior e equivalente a

```
C#  
{ Prop1: { Prop2: pattern } }
```

válido em C# 8.0 e posterior.

Para obter mais informações, confira a nota de proposta de recursos de [Padrões de propriedade estendida](#). Para obter mais informações sobre um padrão de propriedade, confira a seção [Padrão de propriedade](#) do artigo [Padrões](#).

## Aprimoramentos da expressão lambda

O C# 10 inclui muitas melhorias na forma como as expressões lambda são tratadas:

- As expressões Lambda podem ter um [tipo natural](#), em que o compilador pode inferir um tipo delegado da expressão lambda ou do grupo de métodos.
- As expressões Lambda podem declarar um [tipo de retorno](#) quando o compilador não pode inferi-lo.
- Os [atributos](#) podem ser aplicados a expressões lambda.

Esses recursos tornam as expressões lambda mais semelhantes a métodos e funções locais. Elas facilitam o uso de expressões lambda sem declarar uma variável de um tipo delegado e funcionam mais perfeitamente com as novas APIs mínimas de ASP.NET Core.

## Cadeias de caracteres interpoladas constantes

No C# 10, as cadeias de caracteres `const` poderão ser inicializadas usando a [interpolação de cadeia de caracteres](#) se todos os espaços reservados forem cadeias de caracteres constantes. A interpolação de cadeia de caracteres pode criar cadeias de caracteres constantes mais legíveis à medida que elas são criadas e usadas no aplicativo. As expressões de espaço reservado não podem ser constantes numéricas porque essas constantes são convertidas em cadeias de caracteres em tempo de execução. A cultura

atual pode afetar a representação de cadeia de caracteres. Saiba mais na referência de linguagem sobre expressões [const](#).

## Tipos de registro podem selar `ToString`

No C# 10, você pode adicionar o modificador `sealed` ao substituir `ToString` em um tipo de registro. Selar o método `ToString` impede que o compilador sintetize um método `ToString` para qualquer tipo de registro derivado. Um `sealed ToString` garante que todos os tipos de registro derivados usem o método `ToString` definido em um tipo de registro base comum. Você pode saber mais sobre esse recurso no artigo sobre [registros](#).

## Atribuição e declaração na mesma desconstrução

Essa alteração remove uma restrição de versões anteriores do C#. Anteriormente, uma desconstrução poderia atribuir todos os valores a variáveis existentes ou inicializar variáveis recém-declaradas:

```
C#  
  
// Initialization:  
(int x, int y) = point;  
  
// assignment:  
int x1 = 0;  
int y1 = 0;  
(x1, y1) = point;
```

O C# 10 remove essa restrição:

```
C#  
  
int x = 0;  
(x, int y) = point;
```

## Atribuição definitiva aprimorada

Antes do C# 10, havia muitos cenários em que a atribuição definitiva e a análise de estado nulo produziam avisos que eram falsos positivos. Geralmente, elas envolviam comparações com constantes booleanas, acesso a uma variável somente nas instruções

`true` ou `false` em uma instrução `if` e expressões de associação nulas. Esses exemplos geraram avisos em versões anteriores do C#, mas não no C# 10:

C#

```
string representation = "N/A";
if ((c != null && c.GetDependentValue(out object obj)) == true)
{
    representation = obj.ToString(); // undesired error
}

// Or, using ?.
if (c?.GetDependentValue(out object obj) == true)
{
    representation = obj.ToString(); // undesired error
}

// Or, using ??
if (c?.GetDependentValue(out object obj) ?? false)
{
    representation = obj.ToString(); // undesired error
}
```

O principal impacto dessa melhoria é que os avisos para atribuição definitiva e análise de estado nulo são mais precisos.

## Permitir atributo `AsyncMethodBuilder` em métodos

No C# 10 e posterior, você pode especificar um construtor de métodos assíncrono diferente para um método único, além de especificar o tipo de construtor de métodos para todos os métodos que retornam um determinado tipo de tarefa. Um construtor de métodos assíncrono personalizado permite cenários avançados de ajuste de desempenho em que um determinado método pode se beneficiar de um construtor personalizado.

Para saber mais, confira a seção sobre [AsyncMethodBuilder](#) no artigo que aborda atributos lidos pelo compilador.

## Diagnóstico de atributo `CallerArgumentExpression`

Você pode usar o parâmetro [System.Runtime.CompilerServices.CallerArgumentExpressionAttribute](#) para especificar

um parâmetro que o compilador substitui pela representação de texto de outro argumento. Esse recurso permite que as bibliotecas criem diagnósticos mais específicos. O código a seguir testa uma condição. Se a condição for falsa, a mensagem de exceção conterá a representação de texto do argumento passado para `condition`:

C#

```
public static void Validate(bool condition,
[CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new InvalidOperationException($"Argument failed validation:
<{message}>");
    }
}
```

Você pode saber mais sobre esse recurso no artigo sobre [Atributos de informações do chamador](#) na seção de referência de linguagem.

## Pragma de #line avançado

O C# 10 dá suporte a um novo formato para o pragma `#line`. Você provavelmente não usará o novo formato, mas verá os respectivos. Os aprimoramentos permitem uma saída mais refinada em DSLs (linguagens específicas do domínio), como Razor. O mecanismo Razor usa essas melhorias para aprimorar a experiência de depuração. Você encontrará depuradores que podem realçar a fonte Razor com mais precisão. Para saber mais sobre a nova sintaxe, confira o artigo sobre [Diretivas de pré-processador](#) na referência de linguagem. Você também pode ler a [especificação de recurso](#) para exemplos baseados em Razor.

# Novidades do C# 9.0

Artigo • 10/05/2023

O C# 9.0 adiciona os seguintes recursos e aprimoramentos à linguagem C#:

- [Registros](#)
- [Setters somente init](#)
- [Instruções de nível superior](#)
- [Melhorias na correspondência de padrões](#)
- [Desempenho e interoperabilidade](#)
  - [Inteiros de tamanho nativo](#)
  - [Ponteiros de função](#)
  - [Suprimir a emissão do sinalizador localsinit](#)
- [Ajustar e concluir recursos](#)
  - [Expressões com tipo de destino new](#)
  - [static funções anônimas](#)
  - [Expressão condicional com tipo de destino](#)
  - [Tipos de retorno covariantes](#)
  - [Suporte à extensão GetEnumerator para loops foreach](#)
  - [Parâmetros discard de lambda](#)
  - [Atributos em funções locais](#)
- [Suporte para geradores de código](#)
  - [Inicializadores de módulo](#)
  - [Novos recursos para métodos parciais](#)
- [Ciclo de aviso 5](#)

O C# 9.0 tem suporte no [.NET 5](#). Para obter mais informações, consulte [Controle de versão da linguagem C#](#).

Você pode baixar o SDK mais recente do .NET na [página de downloads do .NET](#).

## ⓘ Observação

Estamos interessados em seus comentários sobre esses recursos. Se você encontrar problemas com qualquer um desses novos recursos, crie um [problema](#) no repositório [dotnet/roslyn](#).

## Tipos de registro

O C# 9.0 introduz tipos de *registro*. Você usa a palavra-chave `record` para definir um tipo de referência que fornece funcionalidade interna para encapsular dados. Você pode criar tipos de registro com propriedades imutáveis usando parâmetros posicionais ou sintaxe de propriedade padrão:

```
C#
```

```
public record Person(string FirstName, string LastName);
```

```
C#
```

```
public record Person
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
};
```

Você também pode criar tipos de registros com propriedades e campos mutáveis:

```
C#
```

```
public record Person
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
};
```

Embora os registros possam ser mutáveis, eles se destinam principalmente a dar suporte a modelos de dados imutáveis. O tipo de registro oferece os seguintes recursos:

- Sintaxe concisa para criar um tipo de referência com propriedades imutáveis
- Comportamento útil para um tipo de referência centrado em dados:
  - Igualdade de valor
  - Sintaxe concisa para mutação não destrutiva
  - Formatação interna para exibição
- Suporte para hierarquias de herança

Você pode usar [tipos de estrutura](#) para criar tipos centrados em dados que fornecem igualdade de valor e pouco ou nenhum comportamento. Mas para modelos de dados relativamente grandes, os tipos de estrutura têm algumas desvantagens:

- Eles não dão suporte à herança.
- Eles são menos eficientes em determinar a igualdade de valor. Para tipos de valor, o método `ValueType.Equals` usa reflexão para localizar todos os campos. Para

registros, o compilador gera o método `Equals`. Na prática, a implementação da igualdade de valor nos registros é mensuravelmente mais rápida.

- Eles usam mais memória em alguns cenários, pois cada instância tem uma cópia completa de todos os dados. Os tipos de registro são [tipos de referência](#), portanto, uma instância de registro contém apenas uma referência aos dados.

## Sintaxe posicional para definição de propriedade

Você pode usar parâmetros posicionais para declarar propriedades de um registro e inicializar os valores de propriedade ao criar uma instância:

C#

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

Quando você usa a sintaxe posicional para definição de propriedade, o compilador cria:

- Uma propriedade pública `init-only` implementada automaticamente para cada parâmetro posicional fornecido na declaração de registro. Uma propriedade `init-only` só pode ser definida no construtor ou usando um inicializador de propriedade.
- Um construtor primário cujos parâmetros correspondem aos parâmetros posicionais na declaração de registro.
- Um método `Deconstruct` com um parâmetro `out` para cada parâmetro posicional fornecido na declaração de registro.

Para obter mais informações, consulte [Sintaxe posicional](#) no artigo de referência de linguagem C# sobre registros.

## Imutabilidade

Um tipo de registro não é necessariamente imutável. Você pode declarar propriedades com acessadores `set` e campos que não são `readonly`. Mas, embora os registros possam ser mutáveis, eles facilitam a criação de modelos de dados imutáveis. As propriedades criadas usando a sintaxe posicional são imutáveis.

A imutabilidade pode ser útil quando você deseja que um tipo centrado em dados seja thread-safe ou um código hash permaneça o mesmo em uma tabela de hash. Ela pode evitar bugs que ocorrem quando você passa um argumento por referência a um método e o método altera inesperadamente o valor do argumento.

Os recursos exclusivos para tipos de registro são implementados por métodos sintetizados pelo compilador e nenhum desses métodos compromete a imutabilidade por meio da modificação do estado do objeto.

## Igualdade de valor

A igualdade de valor significa que duas variáveis de um tipo de registro são iguais se os tipos corresponderem e todos os valores de propriedade e campo corresponderem. Para outros tipos de referência, igualdade significa identidade. Ou seja, duas variáveis de um tipo de referência são iguais se referirem ao mesmo objeto.

O exemplo a seguir ilustra a igualdade de valores dos tipos de registro:

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

Em tipos `class`, você pode substituir manualmente métodos e operadores de igualdade para obter igualdade de valor, mas desenvolver e testar esse código seria demorado e propenso a erros. Ter essa funcionalidade interna impede que bugs resultantes do esquecimento atualizem o código de substituição personalizado quando propriedades ou campos forem adicionados ou alterados.

Para obter mais informações, consulte [Sintaxe posicional](#) no artigo de referência de linguagem C# sobre registros.

## Mutação não destrutiva

Se você precisar modificar propriedades imutáveis de uma instância de registro, poderá usar uma expressão `with` para obter *mutação não estruturativa*. Uma expressão `with` faz uma nova instância de registro que é uma cópia de uma instância de registro existente, com propriedades e campos especificados modificados. Use a sintaxe do [inicializador de objetos](#) para especificar os valores a serem alterados, conforme mostrado no exemplo a seguir:

C#

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers =
    // System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers =
    // System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers =
    // System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

Para obter mais informações, consulte a [mutação não estruturativa](#) no artigo de referência da linguagem C# sobre registros.

## Formatação interna para exibição

Os tipos de registro têm um método `ToString` gerado pelo compilador que exibe os nomes e valores de propriedades e campos públicos. O método `ToString` retorna uma cadeia de caracteres do seguinte formato:

```
<nome do tipo de registro> { <nome da propriedade> = <valor>, <nome da propriedade> = <valor>, ...}
```

Para tipos de referência, o nome do tipo do objeto ao qual a propriedade se refere é exibido em vez do valor da propriedade. No exemplo a seguir, a matriz é um tipo de referência, portanto `System.String[]` é exibida em vez dos valores reais do elemento de matriz:

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[] }
```

Para obter mais informações, consulte [Sintaxe posicional](#) no artigo de referência de linguagem C# sobre registros.

## Herança

Um registro pode herdar de outro registro. No entanto, um registro não pode herdar de uma classe, e uma classe não pode herdar de um registro.

O exemplo a seguir ilustra a herança com sintaxe de propriedade posicional:

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Para que duas variáveis de registro sejam iguais, o tipo de tempo de execução deve ser igual. Os tipos das variáveis de conteúdo podem ser diferentes. Isso é ilustrado no exemplo de código a seguir:

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
```

```

    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}

```

No exemplo, todas as instâncias têm as mesmas propriedades e os mesmos valores de propriedade. Mas `student == teacher` retorna `False`, embora ambas sejam variáveis de tipo `Person`. E `student == student2` retorna `True`, embora uma seja uma variável `Person` e outra seja uma variável `Student`.

Todas as propriedades públicas e campos de tipos derivados e base estão incluídos na saída `ToString`, conforme mostrado no exemplo a seguir:

C#

```

public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}

```

Para obter mais informações, consulte [Herança](#) no artigo de referência de linguagem C# sobre registros.

## Setters somente init

Os *setters init* apenas fornecem sintaxe consistente para inicializar membros de um objeto. Inicializadores de propriedade deixam claro qual valor está definindo qual propriedade. A desvantagem é que essas propriedades devem ser configuráveis. A partir do C# 9.0, você pode criar acessadores `init` m vez de `set` para propriedades e indexadores. Os chamadores podem usar a sintaxe do inicializador de propriedades para definir esses valores em expressões de criação, mas essas propriedades são lidas somente depois que a construção for concluída. Os setters init apenas fornecem uma

janela para alterar o estado. Essa janela fecha quando a fase de construção termina. A fase de construção termina efetivamente após toda a inicialização, incluindo inicializadores de propriedades e com expressões concluídas.

Você pode declarar setters somente `init` em qualquer tipo que você escrever. Por exemplo, o struct a seguir define uma estrutura de observação meteorológica:

```
C#  
  
public struct WeatherObservation  
{  
    public DateTime RecordedAt { get; init; }  
    public decimal TemperatureInCelsius { get; init; }  
    public decimal PressureInMillibars { get; init; }  
  
    public override string ToString() =>  
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +  
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars}  
pressure";  
}
```

Os chamadores podem usar a sintaxe do inicializador de propriedades para definir os valores, preservando ainda a imutabilidade:

```
C#  
  
var now = new WeatherObservation  
{  
    RecordedAt = DateTime.Now,  
    TemperatureInCelsius = 20,  
    PressureInMillibars = 998.0m  
};
```

Uma tentativa de alterar uma observação após a inicialização resulta em um erro do compilador:

```
C#  
  
// Error! CS8852.  
now.TemperatureInCelsius = 18;
```

Somente setters `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 apenas setters `init`. Esses setters são usados em expressões `with`. Você pode declarar setters somente `init` para qualquer `class`, `struct` ou `record` que você define.

Para obter mais informações, confira [init \(Referência C#\)](#).

## Instruções de nível superior

*Instruções de nível superior* removem a cerimônia desnecessária de muitos aplicativos. Considere o programa canônico "Olá, Mundo!":

```
C#  
  
using System;  
  
namespace HelloWorld  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Hello World!");  
        }  
    }  
}
```

Só há uma linha de código que faz qualquer coisa. Com instruções de nível superior, você pode substituir todo esse código clichê pela diretiva `using` e pela única linha que faz o trabalho:

```
C#  
  
using System;  
  
Console.WriteLine("Hello World!");
```

Se você quisesse um programa de uma linha, poderia remover a diretiva `using` e usar o nome de tipo totalmente qualificado:

```
C#  
  
System.Console.WriteLine("Hello World!");
```

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 método `Main`.

De certa forma, você pode pensar que um arquivo contém as instruções que normalmente estariam no método `Main` de uma classe `Program`.

Um dos usos mais comuns para esse recurso é a criação de materiais didáticos. Os desenvolvedores iniciantes do C# podem escrever o "Olá, Mundo!" canônico em uma ou duas linhas de código. Nenhuma cerimônia extra é necessária. No entanto, os desenvolvedores experientes também encontrarão muitos usos para esse recurso. Instruções de nível superior permitem uma experiência semelhante a script para experimentação semelhante ao que os notebooks Jupyter fornecem. Instruções de nível superior são ótimas para pequenos programas de console e utilitários. [Azure Functions](#) é um caso de uso ideal para instruções de nível superior.

Mais importante, as instruções de nível superior não limitam o escopo ou a complexidade do aplicativo. Essas instruções podem acessar ou usar qualquer classe .NET. Eles também não limitam o uso de argumentos de linha de comando nem retornam valores. Instruções de nível superior podem acessar uma matriz de cadeias de caracteres nomeadas `args`. Se as instruções de nível superior retornarem um valor inteiro, esse valor se tornará o código de retorno inteiro de um método sintetizado `Main`. As instruções de nível superior podem conter expressões assíncronas. Nesse caso, o ponto de entrada sintetizado retorna um `Task` ou `Task<int>`.

Para obter mais informações, consulte [instruções de nível superior](#) no Guia de Programação em C#.

## Melhorias na correspondência de padrões

O C# 9 inclui novas melhorias de correspondência de padrões:

- **Padrões de tipo** correspondem a um objeto correspondente a um tipo específico
- **Padrões entre parênteses** impõem ou enfatizam a precedência de combinações de padrões
- **Padrões conjuntivos and** exigem que ambos os padrões correspondam
- **Padrões conjuntivos or** exigem que ambos os padrões correspondam
- **Padrões conjuntivos not** exigem que ambos os padrões correspondam
- **Os padrões relacionais** exigem que a entrada seja menor que, maior que, menor ou igual ou maior que ou igual a uma determinada constante.

Esses padrões enriquecem a sintaxe para padrões. Considere estes exemplos:

C#

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

Com parênteses opcionais para deixar claro que `and` tem precedência maior que `or`:

C#

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

Um dos usos mais comuns é uma nova sintaxe para uma verificação nula:

C#

```
if (e is not null)
{
    // ...
}
```

Qualquer um desses padrões pode ser usado em qualquer contexto em que os padrões são permitidos: expressões de padrão `is`, expressões `switch`, padrões aninhados e o padrão de uma instrução `switch` do rótulo `case`.

Para obter mais informações, confira [Padrões \(referência de C#\)](#).

Para obter mais informações, consulte as seções [padrões relacionais](#) e [padrões lógicos](#) do artigo [Padrões](#).

## Desempenho e interoperabilidade

Três novos recursos melhoram o suporte para bibliotecas nativas de interoperabilidade e de baixo nível que exigem alto desempenho: inteiros de tamanho nativo, ponteiros de função e omitindo o sinalizador `localsinit`.

Inteiros de tamanho nativo, `nint` e `nuint`, são tipos inteiros. Eles são expressos pelos tipos subjacentes `System.IntPtr` e `System.UIntPtr`. O compilador apresenta conversões e operações adicionais para esses tipos como ints nativos. Inteiros de tamanho nativo definem propriedades para `.MaxValue` ou `.MinValue`. Esses valores não podem ser expressos como constantes de tempo de compilação porque dependem do tamanho nativo de um inteiro no computador de destino. Esses valores são lidos somente em tempo de execução. Você pode usar valores constantes no intervalo `nint [int.MinValue .. int.MaxValue]`. Você pode usar valores constantes no intervalo `nuint [uint.MinValue ..`

`uint.MaxValue`]. O compilador executa a dobra constante para todos os operadores unários e binários usando os tipos [System.Int32](#) e [System.UInt32](#). Se o resultado não se encaixar em 32 bits, a operação será executada em tempo de execução e não será considerada uma constante. Inteiros de tamanho nativo podem aumentar o desempenho em cenários em que a matemática inteiro é usada extensivamente e precisa ter o desempenho mais rápido possível. Para obter mais informações, confira os tipos [nint](#) e [nuint](#).

Os ponteiros de função fornecem uma sintaxe fácil para acessar os opcodes de IL `ldftn` e `calli`. Você pode declarar ponteiros de função usando uma nova sintaxe `delegate*`. Um tipo `delegate*` é um tipo de ponteiro. Invocar o tipo `delegate*` usa `calli`, em contraste com um delegado que usa `callvirt` no método `Invoke()`. Sintaticamente, as invocações são idênticas. A invocação do ponteiro de função usa a convenção de chamada `managed`. 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*`. Para obter mais informações, consulte [Códigos não seguros e tipos de ponteiro](#).

Por fim, você pode adicionar o [System.Runtime.CompilerServices.SkipLocalsInitAttribute](#) para instruir o compilador a não emitir o sinalizador `localsinit`. Esse sinalizador instrui o CLR a inicializar zero todas as variáveis locais. O sinalizador `localsinit` tem sido o comportamento padrão para C# desde 1.0. No entanto, a inicialização extra zero pode ter impacto mensurável no desempenho em alguns cenários. Em particular, quando você usa o `stackalloc`. Nesses casos, você pode adicionar o atributo [SkipLocalsInitAttribute](#). 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 métodos `abstract`; ele afeta o código gerado para a implementação. Para obter mais informações, consulte [atributo SkipLocalsInit](#).

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

## Ajustar e concluir recursos

Muitos dos outros recursos ajudam você a escrever código com mais eficiência. No C# 9.0, você pode omitir o tipo em uma [newexpressão](#) quando o tipo do objeto criado já for conhecido. O uso mais comum está em declarações de campo:

```
C#
```

```
private List<WeatherObservation> _observations = new();
```

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

```
C#
```

```
public WeatherForecast ForecastFor(DateTime forecastDate,  
WeatherForecastOptions options)
```

Você pode chamá-lo da seguinte maneira:

```
C#
```

```
var forecast = station.ForecastFor(DateTime.Now.AddDays(2), new());
```

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

```
C#
```

```
WeatherStation station = new() { Location = "Seattle, WA" };
```

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

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 outra, mas podem ter conversões implícitas em um tipo de destino. Você provavelmente não observará essa alteração. O que você observará é que algumas expressões condicionais que anteriormente exigiam conversões ou não seriam compiladas agora apenas funcionam.

A partir do C# 9.0, você pode adicionar o modificador `static` a [expressões lambda](#) ou [métodos anônimos](#). Expressões lambda estáticas são análogas às funções locais `static`: um método lambda estático ou anônimo não pode capturar variáveis locais ou o estado da instância. O modificador `static` impede a captura acidental de outras variáveis.

Tipos de retorno covariantes fornecem flexibilidade para os tipos de retorno de métodos de [substituição](#). Um método de substituição pode retornar um tipo derivado

do tipo de retorno do método base substituído. Isso pode ser útil para registros e para outros tipos que dão suporte a clones virtuais ou métodos de fábrica.

Além disso, o [loop foreach](#) reconhecerá e usará um método `GetEnumerator` de extensão que, de outra forma, satisfaz o padrão `foreach`. Essa alteração significa que `foreach` é consistente com outras construções baseadas em padrão, como o padrão assíncrono e a desconstrução baseada em padrão. Na prática, essa alteração significa que você pode adicionar suporte `foreach` a qualquer tipo. Você deve limitar seu uso ao enumerar um objeto faz sentido em seu design.

Em seguida, você pode usar descartes como parâmetros para expressões lambda. Essa conveniência permite que você evite nomear o argumento e o compilador pode evitar usá-lo. Você usa o `_` para qualquer argumento. Para obter mais informações, consulte a seção [Parâmetros de entrada de uma expressão lambda](#) do artigo [Expressões lambda](#).

Por fim, agora você pode aplicar atributos a [funções locais](#). Por exemplo, você pode aplicar [anotações de atributo anuláveis](#) a funções locais.

## Suporte para geradores de código

Dois recursos finais dão suporte a geradores de código C#. Geradores de código C# são um componente que você pode escrever que é semelhante a um analisador roslyn ou 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 para atributos ou outras convenções.

Um gerador de código lê atributos ou outros elementos de código usando as APIs de análise Roslyn. Com base nessa informação, ele adiciona um novo código à compilação. 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 extensões à [\*sintaxe parcial do método\*](#) e [\*inicializadores de módulo\*](#). Primeiro, as alterações em métodos parciais. Antes do C# 9.0, os métodos parciais são `private`, mas não podem especificar um modificador de acesso, ter um retorno `void` e não podem ter parâmetros `out`. Essas restrições significaram 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 exige que as declarações de método parcial 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 modificador de acesso `private`, as novas regras regem esse método parcial. Para obter mais informações, consulte [o método parcial \(Referência de C#\)](#).

O segundo novo recurso para geradores de código são *inicializadores de módulo*. Inicializadores de módulo são métodos que têm o atributo `ModuleInitializerAttribute` anexado a eles. Esses métodos serão chamados pelo runtime antes de qualquer outro acesso de campo ou invocação de método dentro de todo o módulo. Um método inicializador de módulo:

- Deve ser estático
- Deve ser sem parâmetros
- Deve retornar nulo
- Não deve ser um método genérico
- Não deve estar contido em uma classe genérica
- Deve ser acessível por meio do módulo que o contém

Esse último ponto de marcador efetivamente significa que o método e sua classe de contenção devem ser internos ou públicos. O método não pode ser uma função local. Para obter mais informações, consulte o [ModuleInitializeratributo](#).

# O histórico da linguagem C#

Artigo • 09/03/2023

Este artigo fornece um histórico de cada versão principal da linguagem C#. A equipe C# continua a inovar e a adicionar novos recursos. Os status detalhados do recurso de linguagem, incluindo os recursos considerados para as versões futuras, podem ser encontrados [no repositório dotnet/roslyn](#) no GitHub.

## ⓘ Importante

A linguagem C# depende de tipos e métodos nos quais a especificação C# é definida como uma *biblioteca padrão* para alguns dos recursos. A plataforma .NET fornece esses tipos e métodos em alguns pacotes. Um exemplo é o processamento de exceção. Cada instrução ou expressão `throw` é verificada para garantir que o objeto que está sendo gerado é derivado de [Exception](#). Da mesma forma, cada `catch` é verificado para garantir que o tipo que está sendo capturado é derivado de [Exception](#). Cada versão pode adicionar novos requisitos. Para usar os recursos de linguagem mais recentes em ambientes mais antigos, talvez seja necessário instalar bibliotecas específicas. Essas dependências estão documentadas na página de cada versão específica. Saiba mais sobre as [relações entre linguagem e biblioteca](#) para obter informações sobre essa dependência.

## C# versão 11

Lançado em novembro de 2022

Os seguintes recursos foram adicionados em C# 11:

- [Literais brutos de cadeia de caracteres](#)
- [Suporte matemático genérico](#)
- [Atributos genéricos](#)
- [Cadeia de caracteres UTF-8 literais](#)
- [Linhas novas em expressões de interpolação de cadeia de caracteres](#)
- [Padrões de lista](#)
- [Tipos de locais de arquivos](#)
- [Membros necessários](#)
- [Structs de padrão automático](#)
- [Correspondência de padrão `Span<char>` em uma constante `string`](#)
- [Escopo estendido `nameof`](#)

- [IntPtr numérico](#)
- [Campos ref e scoped ref](#)
- [Conversão aprimorada do grupo de métodos para delegado](#)
- [Ciclo de aviso 7](#)

O C# 11 apresenta *matemática genérica* e vários recursos que dão suporte a essa meta. Você pode escrever algoritmos numéricos uma vez para todos os tipos de número. Há mais recursos para facilitar o trabalho com os tipos de `struct`, como membros necessários e structs de padrão automático. O trabalho com cadeias de caracteres fica mais fácil com literais de cadeia de caracteres brutais, nova linha em interpolações de cadeia de caracteres e literais de cadeia de caracteres UTF-8. Recursos como os tipos de locais de arquivo permitem que os geradores de origem sejam mais simples. Por fim, os padrões de lista adicionam mais suporte para correspondência de padrões.

## C# versão 10

*Lançado em novembro de 2021*

C# 10 adiciona os seguintes recursos e aprimoramentos à linguagem C#:

- [Estruturas de registro](#)
- [Aprimoramentos de tipos de estrutura](#)
- [Manipuladores de cadeia de caracteres interpolada](#)
- [Diretivas global using](#)
- [Declaração de namespace com escopo de arquivo](#)
- [Padrões de propriedade estendida](#)
- [Aprimoramentos nas expressões lambda](#)
- [Permissão de cadeias de caracteres interpoladas const](#)
- [Tipos de registro podem selar ToString\(\)](#)
- [Atribuição definida aprimorada](#)
- [Permissão de atribuição e declaração na mesma desconstrução](#)
- [Permissão do atributo AsyncMethodBuilder em métodos](#)
- [Atributo CallerArgumentExpression](#)
- [Pragma #line aprimorado](#)

Mais recursos estavam disponíveis no modo de *visualização*. Para usar esses recursos, você deve definir `<LangVersion>` como `Preview` em seu projeto:

- [Atributos genéricos](#), posteriormente neste artigo.
- [membros abstratos estáticos em interfaces](#)

C# 10 continua trabalhando em temas de remoção de cerimônia, separação de dados de algoritmos e aprimoramento do desempenho para o Runtime do .NET.

Muitos dos recursos significam que você digitará menos código para expressar os mesmos conceitos. *Structs de registro* sintetizam muitos dos mesmos métodos que as *classes de registro*. Structs e tipos anônimos dão suporte a expressões. *Diretivas de uso global* e *declarações de namespace com escopo de arquivo* significam expressar dependências e organização de namespace com mais clareza. *Melhorias lambda* facilitam a declaração de expressões lambda onde são usadas. Novos padrões de propriedade e melhorias de desconstrução criam um código mais conciso.

Os novos manipuladores de cadeia de caracteres interpolados e o comportamento `AsyncMethodBuilder` podem melhorar o desempenho. Esses recursos de linguagem foram aproveitados no .NET Runtime para obter aprimoramentos de desempenho no .NET 6.

C# 10 também marca uma mudança para a cadência anual de versões do .NET. Como nem todos os recursos podem ser concluídos em um período anual, você pode experimentar alguns recursos de "versão prévia" no C# 10. Os *atributos genéricos* e os *membros abstratos estáticos em interfaces* podem ser usados, mas eles são versões prévias do recurso e podem ser alterados antes da versão final.

## C# versão 9

*Lançado em novembro de 2020*

C# 9 foi lançada com o .NET 5. É a versão de linguagem padrão para qualquer assembly direcionado à versão do .NET 5. Ela contém os seguintes recursos novos e aprimorados:

- [Registros](#)
- [Setters somente init](#)
- [Instruções de nível superior](#)
- [Melhorias na correspondência de padrões](#)
- [Desempenho e interoperabilidade](#)
  - [Inteiros de tamanho nativo](#)
  - [Ponteiros de função](#)
  - [Suprimir a emissão do sinalizador localsinit](#)
- [Ajustar e concluir recursos](#)
  - [Expressões com tipo de destino new](#)
  - [static funções anônimas](#)
  - [Expressão condicional com tipo de destino](#)
  - [Tipos de retorno covariantes](#)

- [Suporte à extensão GetEnumerator para loops foreach](#)
- [Parâmetros discard de lambda](#)
- [Atributos em funções locais](#)
- [Suporte para geradores de código](#)
  - [Inicializadores de módulo](#)
  - [Novos recursos para métodos parciais](#)

C# 9 continua três dos temas de versões anteriores: remoção da cerimônia, separação dos dados de algoritmos e fornecimento de mais padrões em mais lugares.

[Instruções de nível superior](#) significam que seu programa principal é mais simples de ler. Há menos necessidade de cerimônia: um namespace, uma classe `Program` e `static void Main()` são todos desnecessários.

A introdução de [records](#) fornece uma sintaxe concisa para tipos de referência que seguem semântica de valor para manter a igualdade. Você usará esses tipos para definir contêineres de dados que normalmente definem o comportamento mínimo. [Os setters somente init](#) fornecem o recurso de mutação não destrutiva (expressões `with`) nos registros. C# 9 também adiciona [tipos de retorno covariantes](#) para que os registros derivados possam substituir métodos virtuais e retornar um tipo derivado do tipo de retorno do método base.

Os recursos de [padrões correspondentes](#) foram expandidos de várias maneiras. Agora, os tipos numéricos dão suporte aos *padrões de intervalo*. Os padrões podem ser combinados usando padrões `and`, `or` e `not`. É possível adicionar parênteses para esclarecer padrões mais complexos.

Outro conjunto de recursos dá suporte à computação de alto desempenho em C#:

- Os tipos `nint` e `nuint` modelam os tipos inteiros de tamanho nativo na CPU de destino.
- Os [ponteiros de função](#) fornecem funcionalidade semelhante a delegado, evitando as alocações necessárias para criar um objeto delegado.
- A instrução `localsinit` pode ser omitida para salvar instruções.

Outro conjunto de aprimoramentos dá suporte a cenários em que os *geradores de código* adicionam funcionalidade:

- [Inicializadores de módulo](#) são métodos que o runtime chama quando um assembly é carregado.
- [Métodos parciais](#) dão suporte a novos modificadores acessíveis e tipos de retorno não nulos. Nesses casos, uma implementação deve ser fornecida.

C# 9 adiciona muitos outros pequenos recursos que melhoram a produtividade do desenvolvedor, escrevendo e lendo código:

- Expressões `new` de tipo de destino
- Funções anônimas de `static`
- Expressão condicional com tipo de destino
- Suporte a `GetEnumerator()` de extensão para loops `foreach`
- Expressões Lambda podem declarar parâmetros de descarte
- Atributos podem ser aplicados a funções locais

A versão C# 9 continua o trabalho para manter C# uma linguagem de programação moderna e de uso geral. Os recursos continuam a dar suporte a cargas de trabalho e tipos de aplicativo modernos.

## C# versão 8.0

*Lançado em setembro de 2019*

C# 8.0 é a primeira grande versão em C# que tem como destino especificamente o .NET Core. Alguns recursos dependem de novos recursos de CLR, outros, de tipos de biblioteca adicionados somente no .NET Core. C# 8.0 adiciona os seguintes recursos e aprimoramentos à linguagem C#:

- [Membros somente leitura](#)
- [Métodos de interface padrão](#)
- [Aprimoramentos de correspondência de padrões:](#)
  - Expressões switch
  - Padrões da propriedade
  - Padrões de tupla
  - Padrões posicionais
- [Declarações using](#)
- [Funções locais estáticas](#)
- [Estruturas ref descartáveis](#)
- [Tipos de referência anuláveis](#)
- [Fluxos assíncronos](#)
- [Índices e intervalos](#)
- [Atribuição de coalescência nula](#)
- [Tipos construídos não gerenciados](#)
- [Stackalloc em expressões aninhadas](#)
- [Aprimoramento de cadeias de caracteres verbatim interpoladas](#)

Os membros de interface padrão exigem aprimoramentos na CLR. Esses recursos foram adicionados na CLR para .NET Core 3.0. Intervalos e índices, além de fluxos assíncronos, exigem novos tipos nas bibliotecas do .NET Core 3.0. Tipos de referência anuláveis, implementados no compilador, são muito mais úteis quando bibliotecas são anotadas para fornecer informações semânticas sobre o estado nulo de argumentos e valores retornados. Essas anotações estão sendo adicionadas nas bibliotecas do .NET Core.

## C# versão 7.3

*Lançado em maio de 2018*

Há dois temas principais para a versão C# 7.3. Um tema fornece recursos que permitem que o código seguro tenha o mesmo desempenho que o código não seguro. O segundo tema fornece melhorias incrementais aos recursos existentes. Novas opções do compilador também foram adicionadas a essa versão.

Os novos recursos a seguir são compatíveis com o tema de melhor desempenho para código seguro:

- Você pode acessar campos fixos sem fixação.
- Você pode reatribuir variáveis locais `ref`.
- Você pode usar inicializadores em matrizes `stackalloc`.
- Você pode usar instruções `fixed` com qualquer tipo compatível com um padrão.
- Você pode usar mais restrições genéricas.

Os seguintes recursos e aprimoramentos foram feitos nos recursos existentes:

- Você pode testar `==` e `!=` com tipos de tupla.
- Você pode usar variáveis de expressão em mais locais.
- Você pode anexar atributos ao campo de suporte de propriedades autoimplementadas.
- A resolução de métodos quando os argumentos se diferenciam por `in` foi aprimorada.
- A resolução de sobrecarga agora tem menos casos ambíguos.

As novas opções do compilador são:

- `-publicsign` para habilitar a assinatura de Software de código aberto (OSS) de assemblies.
- `-pathmap` para fornecer um mapeamento para diretórios de origem.

## C# versão 7.2

Lançado em novembro de 2017

C# 7.2 adicionou vários recursos de linguagem menores:

- Inicializadores em matrizes `stackalloc`.
- Uso de instruções `fixed` com qualquer tipo compatível com um padrão.
- Acesso a campos fixos sem fixação.
- Reatribuição de variáveis locais `ref`.
- Declaração de tipos `readonly struct`, para indicar que uma struct é imutável e deve ser passada como um parâmetro `in` para seus métodos de membro.
- Adição do modificador `in` em parâmetros, para especificar que um argumento seja passado por referência, mas não modificado pelo método chamado.
- Uso do modificador `ref readonly` nos retornos de método, para indicar que um método retorna seu valor por referência, mas não permite gravações nesse objeto.
- Declaração de tipos `ref struct`, para indicar que um tipo de struct acessa a memória gerenciada diretamente e deve sempre ser alocado por pilha.
- Uso de restrições genéricas adicionais.
- **Argumentos nomeados que não estejam à direita**
  - Os argumentos nomeados podem ser seguidos por argumentos posicionais.
- Sublinhados à esquerda em literais numéricos
  - Agora os literais numéricos podem ter sublinhados à esquerda, antes dos dígitos impressos.
- **Modificador de acesso `private protected`**
  - O modificador de acesso `private protected` permite o acesso a classes derivadas no mesmo assembly.
- Expressões `ref` condicionais
  - O resultado de uma expressão condicional (`? :`) agora já pode ser uma referência.

## C# versão 7.1

Lançado em agosto de 2017

C# começou a lançar *versões de ponto* com C# 7.1. Essa versão adiciona a configuração de [seleção de versão da linguagem](#), três novos recursos de linguagem e um novo comportamento do compilador.

Os novos recursos de linguagem nesta versão são:

- **asyncMain método**
  - O ponto de entrada para um aplicativo pode ter o modificador `async`.

- [Expressões literais default](#)
  - Use expressões literais padrão em expressões de valor padrão quando o tipo de destino pode ser inferido.
- Nomes de elementos de tupla inferidos
  - Em muitos casos, os nomes dos elementos de tupla podem ser inferidos com base na inicialização da tupla.
- Restrições em parâmetros de tipo genérico
  - Você pode usar expressões de correspondência de padrão em variáveis cujo tipo é um parâmetro de tipo genérico.

Por fim, o compilador traz duas opções [-refout](#) e [-refonly](#), que controlam a geração de assembly de referência

## C# versão 7.0

*Lançado em março de 2017*

A versão 7.0 de C# foi lançada com o Visual Studio 2017. Esta versão tem algumas coisas interessantes e evolutivas na mesma direção que o C# 6.0. Aqui estão alguns dos novos recursos:

- Variáveis out
- [Tuplas e desconstrução](#)
- [Correspondência de padrões](#)
- [Funções locais](#)
- [Membros aptos para expressão expandidos](#)
- [Ref locals](#)
- [Retornos de referências](#)

Outros recursos incluíam:

- [Descartes](#)
- Literais binários e os separadores de dígito
- [Expressões throw](#)

Todas essas funcionalidades oferecem novos recursos para desenvolvedores e a oportunidade de escrever um código mais limpo do que nunca. Um ponto alto é a condensação da declaração de variáveis a serem usadas com a palavra-chave `out` e a permissão de vários valores retornados por meio de tupla. Agora o .NET Core tem qualquer sistema operacional como destino e tem a visão firme na nuvem e na portabilidade. Essas novas funcionalidades certamente ocupam a mente e o tempo dos designers da linguagem, além de levarem a novos recursos.

# C# versão 6.0

*Lançado em julho de 2015*

A versão 6.0, lançada com o Visual Studio 2015, trouxe muitos recursos menores que tornaram a programação em C# mais produtiva. Eis algumas delas:

- [Importações estáticas](#)
- [Filtros de exceção](#)
- [Inicializadores de propriedade automática](#)
- [Membros aptos para expressão](#)
- [Propagador nulo](#)
- [Interpolação de cadeia de caracteres](#)
- [Operador nameof](#)

Outros novos recursos incluem:

- Inicializadores de índice
- Await em blocos catch/finally
- Valores padrão para propriedades somente getter

Cada um desses recursos é interessante em seus próprios méritos. Mas, se você os observar em conjunto, verá um padrão interessante. Nesta versão, o C# começou a eliminar o clichê de linguagem para tornar o código mais conciso e legível. Portanto, para os fãs de código simples e conciso, essa versão da linguagem foi um grande benefício.

Fizeram ainda outra coisa com esta versão, embora não seja um recurso de linguagem tradicional em si. Lançaram [Roslyn, o compilador como um serviço](#). Agora o compilador de C# é escrito em C#, e você pode usar o compilador como parte de seus esforços de programação.

# C# versão 5.0

*Lançado em agosto de 2012*

A versão 5.0 de C#, lançada com o Visual Studio 2012, era uma versão focada da linguagem. Quase todo o esforço para essa versão foi dedicado a outro conceito inovador de linguagem: os modelos `async` e `await` para programação assíncrona. Aqui está a lista dos recursos principais:

- [Membros assíncronos](#)
- [Atributos de informações do chamador](#)

- [Code Project: Caller Info Attributes in C# 5.0](#) (Code Project: Atributos de informações do chamador em C# 5.0)

O atributo de informações do chamador permite facilmente recuperar informações sobre o contexto no qual você está executando sem recorrer a uma infinidade de código de reflexão clichê. Ele tem muitos usos em diagnóstico e tarefas de registro em log.

Mas `async` e `await` são as verdadeiras estrelas dessa versão. Quando esses recursos foram lançados em 2012, o C# virou o jogo novamente, implantando a assincronia na linguagem como uma participante da maior importância. Se você já teve que lidar com operações de longa execução e a implementação de redes de retorno de chamada, você provavelmente adorou esse recurso de linguagem.

## C# versão 4.0

*Lançado em abril de 2010*

A versão 4.0 de C#, lançada com o Visual Studio 2010, enfrentou dificuldades para sobreviver ao status inovador da versão 3.0. Esta versão introduziu alguns novos recursos interessantes:

- [Associação dinâmica](#)
- [Argumentos opcionais/nomeados](#)
- [Genérico covariante e contravariante](#)
- [Tipos de interoperabilidade inseridos](#)

Tipos de interoperabilidade inseridos facilitaram os problemas de implantação da criação de assemblies de interoperabilidade COM para seu aplicativo. A contravariância e a covariância genérica oferecem maior capacidade para usar genéricos, mas eles são um tanto acadêmicos e provavelmente mais apreciados por autores de estruturas e bibliotecas. Os parâmetros nomeados e opcionais permitem eliminar várias sobrecargas de método e oferecem conveniência. Mas nenhum desses recursos é exatamente uma alteração de paradigma.

O recurso principal foi a introdução da palavra-chave `dynamic`. A palavra-chave `dynamic` introduziu na versão 4.0 do C# a capacidade de substituir o compilador na tipagem em tempo de compilação. Com o uso da palavra-chave dinâmica, você pode criar constructos semelhantes a linguagens dinamicamente tipadas, como JavaScript. Você pode criar um `dynamic x = "a string"` e, em seguida, adicionar seis a ela, deixando que o runtime decida o que acontece em seguida.

Associação dinâmica tem potencial de erros, mas também grande eficiência na linguagem.

## C# versão 3.0

*Lançado em novembro de 2007*

O C# versão 3.0 chegou no final de 2007, juntamente com o Visual Studio 2008, porém o pacote completo de recursos de linguagem veio, na verdade, com o C# versão 3.5. Esta versão foi o marco de uma alteração significativa no crescimento do C#. Ela estabeleceu o C# como uma linguagem de programação realmente formidável. Vamos dar uma olhada em alguns recursos importantes nesta versão:

- [Propriedades autoimplementadas](#)
- [Tipos anônimos](#)
- [Expressões de consulta](#)
- [Expressões lambda](#)
- [Árvores de expressão](#)
- [Métodos de extensão](#)
- [Variáveis locais de tipo implícito](#)
- [Métodos parciais](#)
- [Inicializadores de objeto e de coleção](#)

Numa retrospectiva, muitos desses recursos parecerem inevitáveis e inseparáveis. Todos eles se encaixam estrategicamente. Achavam que o melhor recurso dessa versão de C# era a expressão de consulta, também conhecida como LINQ (consulta integrada à linguagem).

Uma exibição mais detalhada analisa árvores de expressão, expressões lambda e tipos anônimos como a base na qual o LINQ é construído. Mas, de uma forma ou de outra, o C# 3.0 apresentou um conceito revolucionário. O C# 3.0 começou a definir as bases para transformar o C# em uma linguagem híbrida orientada a objeto e funcional.

Especificamente, agora você pode escrever consultas declarativas no estilo SQL para executar operações em coleções, entre outras coisas. Em vez de escrever um loop `for` para calcular a média de uma lista de inteiros, agora você pode fazer isso simplesmente como `list.Average()`. A combinação de expressões de consulta e métodos de extensão tornou uma lista de inteiros muito mais inteligente.

## C# versão 2.0

*Lançado em novembro de 2005*

Vamos dar uma olhada em alguns recursos principais do C# 2.0, lançado em 2005, junto com o Visual Studio 2005:

- [Genéricos](#)
- [Tipos parciais](#)
- [Métodos anônimos](#)
- [Tipos de valor anuláveis](#)
- [Iteradores](#)
- [Covariância e contravariância](#)

Outros recursos do C# 2.0 adicionaram funcionalidades a recursos existentes:

- Acessibilidade separada getter/setter
- Conversões de grupo de método (delegados)
- Classes estáticas
- Inferência de delegado

Embora C# possa ter começado como uma linguagem OO (orientada a objeto) genérica, a versão 2.0 do C# tratou de mudar isso rapidamente. Com genéricos, tipos e métodos podem operar em um tipo arbitrário enquanto ainda mantêm a segurança de tipos. Por exemplo, ter um `List<T>` permite que você tenha `List<string>` ou `List<int>` e execute operações fortemente tipadas nessas cadeias de caracteres ou inteiros enquanto itera neles. Usar genéricos é melhor do que criar um `ListInt` que deriva de `ArrayList` ou converter de `Object` para cada operação.

A versão 2.0 do C# trouxe iteradores. Em resumo, os iteradores permitem que você examine todos os itens em um `List` (ou outros tipos Enumeráveis) com um loop `foreach`. Ter iteradores como uma parte importante da linguagem aprimorou drasticamente a legibilidade da linguagem e a capacidade das pessoas de raciocinar a respeito do código.

E ainda assim, o C# continuava na tentativa de alcançar o mesmo nível do Java. O Java já tinha liberado versões que incluíam genéricos e iteradores. Mas isso seria alterado logo, pois as linguagens continuaram a evoluir separadamente.

## C# versão 1.2

*Lançado em abril de 2003*

C# versão 1.2 fornecida com o Visual Studio .NET 2003. Ele continha algumas pequenas melhorias na linguagem. Muito notável é que, a partir desta versão, o código gerado em

um loop `foreach` chamou `Dispose`, em um `IEnumerator`, quando o `IEnumerator` implementou `IDisposable`.

## C# versão 1.0

*Lançado em janeiro de 2002*

Ao olhar para trás, a versão 1.0 de C#, lançada com o Visual Studio .NET 2002, ficou muito parecida com o Java. Como [parte de suas metas de design declaradas para ECMA](#), ela procurou ser uma "linguagem simples, moderna e orientada a objetos para uso geral". Na época, parecer com Java significava que havia atingido essas metas de design iniciais.

Mas agora, se examinar novamente a C# 1.0, você poderá se sentir um pouco confuso. Carecia das funcionalidades assíncronas internas e algumas das funcionalidades relacionadas a genéricos que você nem valoriza. Na verdade, ela não tinha nada relacionado a genéricos. E a [LINQ](#)? Ainda não estava disponível. Essas adições levariam alguns anos para sair.

A versão 1.0 do C# parecia ter poucos recursos, em comparação com os dias de hoje. Você teria que escrever código um tanto detalhado. Mas, ainda assim, você poderia iniciar em algum lugar. A versão 1.0 do C# era uma alternativa viável ao Java na plataforma Windows.

Os principais recursos do C# 1.0 incluíam:

- [Classes](#)
- [Estruturas](#)
- [Interfaces](#)
- [Eventos](#)
- [Propriedades](#)
- [Representantes](#)
- [Operadores e expressões](#)
- [Instruções](#)
- [Atributos](#)

Artigo [originalmente publicado no blog NDepend](#), cortesia de Erik Dietrich e Patrick Smacchia.

# Saiba mais sobre alterações interruptivas no compilador C#

Artigo • 10/05/2023

Você pode localizar alterações significativas desde a versão do C# 10 [aqui](#).

A equipe da [Roslyn](#) mantém uma lista de alterações interruptivas nos compiladores do C# e do Visual Basic. Você pode encontrar informações sobre essas alterações nesses links no repositório GitHub:

- [Alterações interruptivas em Roslyn no C# 10.0/.NET 6](#)
- [Alterações interruptivas em Roslyn após .NET 5](#)
- [Alterações interruptivas em VS2019 versão 16.8 introduzidas com o .NET 5 e o C# 9.0](#)
- [Alterações interruptivas no VS2019 Atualização 1 e posteriores em comparação com o VS2019](#)
- [Alterações interruptivas desde o VS2017 \(C# 7\)](#)
- [Alterações interruptivas no Roslyn 3.0 \(VS2019\) do Roslyn 2.\\* \(VS2017\)](#)
- [Alterações interruptivas no Roslyn 2.0 \(VS2017\) do Roslyn 1. \\* \(VS2015\) e do compilador nativo C# \(VS2013 e anterior\).](#)
- [Alterações interruptivas no Roslyn 1.0 \(VS2015\) do compilador nativo C# \(VS2013 e anterior\).](#)
- [Alteração de versão Unicode no C# 6](#)

# O histórico da linguagem C#

Artigo • 09/03/2023

Este artigo fornece um histórico de cada versão principal da linguagem C#. A equipe C# continua a inovar e a adicionar novos recursos. Os status detalhados do recurso de linguagem, incluindo os recursos considerados para as versões futuras, podem ser encontrados [no repositório dotnet/roslyn](#) no GitHub.

## ⓘ Importante

A linguagem C# depende de tipos e métodos nos quais a especificação C# é definida como uma *biblioteca padrão* para alguns dos recursos. A plataforma .NET fornece esses tipos e métodos em alguns pacotes. Um exemplo é o processamento de exceção. Cada instrução ou expressão `throw` é verificada para garantir que o objeto que está sendo gerado é derivado de [Exception](#). Da mesma forma, cada `catch` é verificado para garantir que o tipo que está sendo capturado é derivado de [Exception](#). Cada versão pode adicionar novos requisitos. Para usar os recursos de linguagem mais recentes em ambientes mais antigos, talvez seja necessário instalar bibliotecas específicas. Essas dependências estão documentadas na página de cada versão específica. Saiba mais sobre as [relações entre linguagem e biblioteca](#) para obter informações sobre essa dependência.

## C# versão 11

Lançado em novembro de 2022

Os seguintes recursos foram adicionados em C# 11:

- [Literais brutos de cadeia de caracteres](#)
- [Suporte matemático genérico](#)
- [Atributos genéricos](#)
- [Cadeia de caracteres UTF-8 literais](#)
- [Linhas novas em expressões de interpolação de cadeia de caracteres](#)
- [Padrões de lista](#)
- [Tipos de locais de arquivos](#)
- [Membros necessários](#)
- [Structs de padrão automático](#)
- [Correspondência de padrão `Span<char>` em uma constante `string`](#)
- [Escopo estendido `nameof`](#)

- [IntPtr numérico](#)
- [Campos ref e scoped ref](#)
- [Conversão aprimorada do grupo de métodos para delegado](#)
- [Ciclo de aviso 7](#)

O C# 11 apresenta *matemática genérica* e vários recursos que dão suporte a essa meta. Você pode escrever algoritmos numéricos uma vez para todos os tipos de número. Há mais recursos para facilitar o trabalho com os tipos de `struct`, como membros necessários e structs de padrão automático. O trabalho com cadeias de caracteres fica mais fácil com literais de cadeia de caracteres brutais, nova linha em interpolações de cadeia de caracteres e literais de cadeia de caracteres UTF-8. Recursos como os tipos de locais de arquivo permitem que os geradores de origem sejam mais simples. Por fim, os padrões de lista adicionam mais suporte para correspondência de padrões.

## C# versão 10

*Lançado em novembro de 2021*

C# 10 adiciona os seguintes recursos e aprimoramentos à linguagem C#:

- [Estruturas de registro](#)
- [Aprimoramentos de tipos de estrutura](#)
- [Manipuladores de cadeia de caracteres interpolada](#)
- [Diretivas global using](#)
- [Declaração de namespace com escopo de arquivo](#)
- [Padrões de propriedade estendida](#)
- [Aprimoramentos nas expressões lambda](#)
- [Permissão de cadeias de caracteres interpoladas const](#)
- [Tipos de registro podem selar ToString\(\)](#)
- [Atribuição definida aprimorada](#)
- [Permissão de atribuição e declaração na mesma desconstrução](#)
- [Permissão do atributo AsyncMethodBuilder em métodos](#)
- [Atributo CallerArgumentExpression](#)
- [Pragma #line aprimorado](#)

Mais recursos estavam disponíveis no modo de *visualização*. Para usar esses recursos, você deve definir `<LangVersion>` como `Preview` em seu projeto:

- [Atributos genéricos](#), posteriormente neste artigo.
- [membros abstratos estáticos em interfaces](#)

C# 10 continua trabalhando em temas de remoção de cerimônia, separação de dados de algoritmos e aprimoramento do desempenho para o Runtime do .NET.

Muitos dos recursos significam que você digitará menos código para expressar os mesmos conceitos. *Structs de registro* sintetizam muitos dos mesmos métodos que as *classes de registro*. Structs e tipos anônimos dão suporte a expressões. *Diretivas de uso global* e *declarações de namespace com escopo de arquivo* significam expressar dependências e organização de namespace com mais clareza. *Melhorias lambda* facilitam a declaração de expressões lambda onde são usadas. Novos padrões de propriedade e melhorias de desconstrução criam um código mais conciso.

Os novos manipuladores de cadeia de caracteres interpolados e o comportamento `AsyncMethodBuilder` podem melhorar o desempenho. Esses recursos de linguagem foram aproveitados no .NET Runtime para obter aprimoramentos de desempenho no .NET 6.

C# 10 também marca uma mudança para a cadência anual de versões do .NET. Como nem todos os recursos podem ser concluídos em um período anual, você pode experimentar alguns recursos de "versão prévia" no C# 10. Os *atributos genéricos* e os *membros abstratos estáticos em interfaces* podem ser usados, mas eles são versões prévias do recurso e podem ser alterados antes da versão final.

## C# versão 9

*Lançado em novembro de 2020*

C# 9 foi lançada com o .NET 5. É a versão de linguagem padrão para qualquer assembly direcionado à versão do .NET 5. Ela contém os seguintes recursos novos e aprimorados:

- [Registros](#)
- [Setters somente init](#)
- [Instruções de nível superior](#)
- [Melhorias na correspondência de padrões](#)
- [Desempenho e interoperabilidade](#)
  - [Inteiros de tamanho nativo](#)
  - [Ponteiros de função](#)
  - [Suprimir a emissão do sinalizador localsinit](#)
- [Ajustar e concluir recursos](#)
  - [Expressões com tipo de destino new](#)
  - [static funções anônimas](#)
  - [Expressão condicional com tipo de destino](#)
  - [Tipos de retorno covariantes](#)

- [Suporte à extensão GetEnumerator para loops foreach](#)
- [Parâmetros discard de lambda](#)
- [Atributos em funções locais](#)
- [Suporte para geradores de código](#)
  - [Inicializadores de módulo](#)
  - [Novos recursos para métodos parciais](#)

C# 9 continua três dos temas de versões anteriores: remoção da cerimônia, separação dos dados de algoritmos e fornecimento de mais padrões em mais lugares.

[Instruções de nível superior](#) significam que seu programa principal é mais simples de ler. Há menos necessidade de cerimônia: um namespace, uma classe `Program` e `static void Main()` são todos desnecessários.

A introdução de [records](#) fornece uma sintaxe concisa para tipos de referência que seguem semântica de valor para manter a igualdade. Você usará esses tipos para definir contêineres de dados que normalmente definem o comportamento mínimo. [Os setters somente init](#) fornecem o recurso de mutação não destrutiva (expressões `with`) nos registros. C# 9 também adiciona [tipos de retorno covariantes](#) para que os registros derivados possam substituir métodos virtuais e retornar um tipo derivado do tipo de retorno do método base.

Os recursos de [padrões correspondentes](#) foram expandidos de várias maneiras. Agora, os tipos numéricos dão suporte aos *padrões de intervalo*. Os padrões podem ser combinados usando padrões `and`, `or` e `not`. É possível adicionar parênteses para esclarecer padrões mais complexos.

Outro conjunto de recursos dá suporte à computação de alto desempenho em C#:

- Os tipos `nint` e `nuint` modelam os tipos inteiros de tamanho nativo na CPU de destino.
- Os [ponteiros de função](#) fornecem funcionalidade semelhante a delegado, evitando as alocações necessárias para criar um objeto delegado.
- A instrução `localsinit` pode ser omitida para salvar instruções.

Outro conjunto de aprimoramentos dá suporte a cenários em que os *geradores de código* adicionam funcionalidade:

- [Inicializadores de módulo](#) são métodos que o runtime chama quando um assembly é carregado.
- [Métodos parciais](#) dão suporte a novos modificadores acessíveis e tipos de retorno não nulos. Nesses casos, uma implementação deve ser fornecida.

C# 9 adiciona muitos outros pequenos recursos que melhoram a produtividade do desenvolvedor, escrevendo e lendo código:

- Expressões `new` de tipo de destino
- Funções anônimas de `static`
- Expressão condicional com tipo de destino
- Suporte a `GetEnumerator()` de extensão para loops `foreach`
- Expressões Lambda podem declarar parâmetros de descarte
- Atributos podem ser aplicados a funções locais

A versão C# 9 continua o trabalho para manter C# uma linguagem de programação moderna e de uso geral. Os recursos continuam a dar suporte a cargas de trabalho e tipos de aplicativo modernos.

## C# versão 8.0

*Lançado em setembro de 2019*

C# 8.0 é a primeira grande versão em C# que tem como destino especificamente o .NET Core. Alguns recursos dependem de novos recursos de CLR, outros, de tipos de biblioteca adicionados somente no .NET Core. C# 8.0 adiciona os seguintes recursos e aprimoramentos à linguagem C#:

- [Membros somente leitura](#)
- [Métodos de interface padrão](#)
- [Aprimoramentos de correspondência de padrões:](#)
  - Expressões switch
  - Padrões da propriedade
  - Padrões de tupla
  - Padrões posicionais
- [Declarações using](#)
- [Funções locais estáticas](#)
- [Estruturas ref descartáveis](#)
- [Tipos de referência anuláveis](#)
- [Fluxos assíncronos](#)
- [Índices e intervalos](#)
- [Atribuição de coalescência nula](#)
- [Tipos construídos não gerenciados](#)
- [Stackalloc em expressões aninhadas](#)
- [Aprimoramento de cadeias de caracteres verbatim interpoladas](#)

Os membros de interface padrão exigem aprimoramentos na CLR. Esses recursos foram adicionados na CLR para .NET Core 3.0. Intervalos e índices, além de fluxos assíncronos, exigem novos tipos nas bibliotecas do .NET Core 3.0. Tipos de referência anuláveis, implementados no compilador, são muito mais úteis quando bibliotecas são anotadas para fornecer informações semânticas sobre o estado nulo de argumentos e valores retornados. Essas anotações estão sendo adicionadas nas bibliotecas do .NET Core.

## C# versão 7.3

*Lançado em maio de 2018*

Há dois temas principais para a versão C# 7.3. Um tema fornece recursos que permitem que o código seguro tenha o mesmo desempenho que o código não seguro. O segundo tema fornece melhorias incrementais aos recursos existentes. Novas opções do compilador também foram adicionadas a essa versão.

Os novos recursos a seguir são compatíveis com o tema de melhor desempenho para código seguro:

- Você pode acessar campos fixos sem fixação.
- Você pode reatribuir variáveis locais `ref`.
- Você pode usar inicializadores em matrizes `stackalloc`.
- Você pode usar instruções `fixed` com qualquer tipo compatível com um padrão.
- Você pode usar mais restrições genéricas.

Os seguintes recursos e aprimoramentos foram feitos nos recursos existentes:

- Você pode testar `==` e `!=` com tipos de tupla.
- Você pode usar variáveis de expressão em mais locais.
- Você pode anexar atributos ao campo de suporte de propriedades autoimplementadas.
- A resolução de métodos quando os argumentos se diferenciam por `in` foi aprimorada.
- A resolução de sobrecarga agora tem menos casos ambíguos.

As novas opções do compilador são:

- `-publicsign` para habilitar a assinatura de Software de código aberto (OSS) de assemblies.
- `-pathmap` para fornecer um mapeamento para diretórios de origem.

## C# versão 7.2

Lançado em novembro de 2017

C# 7.2 adicionou vários recursos de linguagem menores:

- Inicializadores em matrizes `stackalloc`.
- Uso de instruções `fixed` com qualquer tipo compatível com um padrão.
- Acesso a campos fixos sem fixação.
- Reatribuição de variáveis locais `ref`.
- Declaração de tipos `readonly struct`, para indicar que uma struct é imutável e deve ser passada como um parâmetro `in` para seus métodos de membro.
- Adição do modificador `in` em parâmetros, para especificar que um argumento seja passado por referência, mas não modificado pelo método chamado.
- Uso do modificador `ref readonly` nos retornos de método, para indicar que um método retorna seu valor por referência, mas não permite gravações nesse objeto.
- Declaração de tipos `ref struct`, para indicar que um tipo de struct acessa a memória gerenciada diretamente e deve sempre ser alocado por pilha.
- Uso de restrições genéricas adicionais.
- **Argumentos nomeados que não estejam à direita**
  - Os argumentos nomeados podem ser seguidos por argumentos posicionais.
- Sublinhados à esquerda em literais numéricos
  - Agora os literais numéricos podem ter sublinhados à esquerda, antes dos dígitos impressos.
- **Modificador de acesso `private protected`**
  - O modificador de acesso `private protected` permite o acesso a classes derivadas no mesmo assembly.
- Expressões `ref` condicionais
  - O resultado de uma expressão condicional (`?:`) agora já pode ser uma referência.

## C# versão 7.1

Lançado em agosto de 2017

C# começou a lançar *versões de ponto* com C# 7.1. Essa versão adiciona a configuração de [seleção de versão da linguagem](#), três novos recursos de linguagem e um novo comportamento do compilador.

Os novos recursos de linguagem nesta versão são:

- **asyncMain método**
  - O ponto de entrada para um aplicativo pode ter o modificador `async`.

- [Expressões literais default](#)
  - Use expressões literais padrão em expressões de valor padrão quando o tipo de destino pode ser inferido.
- Nomes de elementos de tupla inferidos
  - Em muitos casos, os nomes dos elementos de tupla podem ser inferidos com base na inicialização da tupla.
- Restrições em parâmetros de tipo genérico
  - Você pode usar expressões de correspondência de padrão em variáveis cujo tipo é um parâmetro de tipo genérico.

Por fim, o compilador traz duas opções [-refout](#) e [-refonly](#), que controlam a geração de assembly de referência

## C# versão 7.0

*Lançado em março de 2017*

A versão 7.0 de C# foi lançada com o Visual Studio 2017. Esta versão tem algumas coisas interessantes e evolutivas na mesma direção que o C# 6.0. Aqui estão alguns dos novos recursos:

- Variáveis out
- [Tuplas e desconstrução](#)
- [Correspondência de padrões](#)
- [Funções locais](#)
- [Membros aptos para expressão expandidos](#)
- [Ref locals](#)
- [Retornos de referências](#)

Outros recursos incluíam:

- [Descartes](#)
- Literais binários e os separadores de dígito
- [Expressões throw](#)

Todas essas funcionalidades oferecem novos recursos para desenvolvedores e a oportunidade de escrever um código mais limpo do que nunca. Um ponto alto é a condensação da declaração de variáveis a serem usadas com a palavra-chave `out` e a permissão de vários valores retornados por meio de tupla. Agora o .NET Core tem qualquer sistema operacional como destino e tem a visão firme na nuvem e na portabilidade. Essas novas funcionalidades certamente ocupam a mente e o tempo dos designers da linguagem, além de levarem a novos recursos.

# C# versão 6.0

*Lançado em julho de 2015*

A versão 6.0, lançada com o Visual Studio 2015, trouxe muitos recursos menores que tornaram a programação em C# mais produtiva. Eis algumas delas:

- [Importações estáticas](#)
- [Filtros de exceção](#)
- [Inicializadores de propriedade automática](#)
- [Membros aptos para expressão](#)
- [Propagador nulo](#)
- [Interpolação de cadeia de caracteres](#)
- [Operador nameof](#)

Outros novos recursos incluem:

- Inicializadores de índice
- Await em blocos catch/finally
- Valores padrão para propriedades somente getter

Cada um desses recursos é interessante em seus próprios méritos. Mas, se você os observar em conjunto, verá um padrão interessante. Nesta versão, o C# começou a eliminar o clichê de linguagem para tornar o código mais conciso e legível. Portanto, para os fãs de código simples e conciso, essa versão da linguagem foi um grande benefício.

Fizeram ainda outra coisa com esta versão, embora não seja um recurso de linguagem tradicional em si. Lançaram [Roslyn, o compilador como um serviço](#). Agora o compilador de C# é escrito em C#, e você pode usar o compilador como parte de seus esforços de programação.

# C# versão 5.0

*Lançado em agosto de 2012*

A versão 5.0 de C#, lançada com o Visual Studio 2012, era uma versão focada da linguagem. Quase todo o esforço para essa versão foi dedicado a outro conceito inovador de linguagem: os modelos `async` e `await` para programação assíncrona. Aqui está a lista dos recursos principais:

- [Membros assíncronos](#)
- [Atributos de informações do chamador](#)

- [Code Project: Caller Info Attributes in C# 5.0](#) (Code Project: Atributos de informações do chamador em C# 5.0)

O atributo de informações do chamador permite facilmente recuperar informações sobre o contexto no qual você está executando sem recorrer a uma infinidade de código de reflexão clichê. Ele tem muitos usos em diagnóstico e tarefas de registro em log.

Mas `async` e `await` são as verdadeiras estrelas dessa versão. Quando esses recursos foram lançados em 2012, o C# virou o jogo novamente, implantando a assincronia na linguagem como uma participante da maior importância. Se você já teve que lidar com operações de longa execução e a implementação de redes de retorno de chamada, você provavelmente adorou esse recurso de linguagem.

## C# versão 4.0

*Lançado em abril de 2010*

A versão 4.0 de C#, lançada com o Visual Studio 2010, enfrentou dificuldades para sobreviver ao status inovador da versão 3.0. Esta versão introduziu alguns novos recursos interessantes:

- [Associação dinâmica](#)
- [Argumentos opcionais/nomeados](#)
- [Genérico covariante e contravariante](#)
- [Tipos de interoperabilidade inseridos](#)

Tipos de interoperabilidade inseridos facilitaram os problemas de implantação da criação de assemblies de interoperabilidade COM para seu aplicativo. A contravariância e a covariância genérica oferecem maior capacidade para usar genéricos, mas eles são um tanto acadêmicos e provavelmente mais apreciados por autores de estruturas e bibliotecas. Os parâmetros nomeados e opcionais permitem eliminar várias sobrecargas de método e oferecem conveniência. Mas nenhum desses recursos é exatamente uma alteração de paradigma.

O recurso principal foi a introdução da palavra-chave `dynamic`. A palavra-chave `dynamic` introduziu na versão 4.0 do C# a capacidade de substituir o compilador na tipagem em tempo de compilação. Com o uso da palavra-chave dinâmica, você pode criar constructos semelhantes a linguagens dinamicamente tipadas, como JavaScript. Você pode criar um `dynamic x = "a string"` e, em seguida, adicionar seis a ela, deixando que o runtime decida o que acontece em seguida.

Associação dinâmica tem potencial de erros, mas também grande eficiência na linguagem.

## C# versão 3.0

*Lançado em novembro de 2007*

O C# versão 3.0 chegou no final de 2007, juntamente com o Visual Studio 2008, porém o pacote completo de recursos de linguagem veio, na verdade, com o C# versão 3.5. Esta versão foi o marco de uma alteração significativa no crescimento do C#. Ela estabeleceu o C# como uma linguagem de programação realmente formidável. Vamos dar uma olhada em alguns recursos importantes nesta versão:

- [Propriedades autoimplementadas](#)
- [Tipos anônimos](#)
- [Expressões de consulta](#)
- [Expressões lambda](#)
- [Árvores de expressão](#)
- [Métodos de extensão](#)
- [Variáveis locais de tipo implícito](#)
- [Métodos parciais](#)
- [Inicializadores de objeto e de coleção](#)

Numa retrospectiva, muitos desses recursos parecerem inevitáveis e inseparáveis. Todos eles se encaixam estrategicamente. Achavam que o melhor recurso dessa versão de C# era a expressão de consulta, também conhecida como LINQ (consulta integrada à linguagem).

Uma exibição mais detalhada analisa árvores de expressão, expressões lambda e tipos anônimos como a base na qual o LINQ é construído. Mas, de uma forma ou de outra, o C# 3.0 apresentou um conceito revolucionário. O C# 3.0 começou a definir as bases para transformar o C# em uma linguagem híbrida orientada a objeto e funcional.

Especificamente, agora você pode escrever consultas declarativas no estilo SQL para executar operações em coleções, entre outras coisas. Em vez de escrever um loop `for` para calcular a média de uma lista de inteiros, agora você pode fazer isso simplesmente como `list.Average()`. A combinação de expressões de consulta e métodos de extensão tornou uma lista de inteiros muito mais inteligente.

## C# versão 2.0

*Lançado em novembro de 2005*

Vamos dar uma olhada em alguns recursos principais do C# 2.0, lançado em 2005, junto com o Visual Studio 2005:

- [Genéricos](#)
- [Tipos parciais](#)
- [Métodos anônimos](#)
- [Tipos de valor anuláveis](#)
- [Iteradores](#)
- [Covariância e contravariância](#)

Outros recursos do C# 2.0 adicionaram funcionalidades a recursos existentes:

- Acessibilidade separada getter/setter
- Conversões de grupo de método (delegados)
- Classes estáticas
- Inferência de delegado

Embora C# possa ter começado como uma linguagem OO (orientada a objeto) genérica, a versão 2.0 do C# tratou de mudar isso rapidamente. Com genéricos, tipos e métodos podem operar em um tipo arbitrário enquanto ainda mantêm a segurança de tipos. Por exemplo, ter um `List<T>` permite que você tenha `List<string>` ou `List<int>` e execute operações fortemente tipadas nessas cadeias de caracteres ou inteiros enquanto itera neles. Usar genéricos é melhor do que criar um `ListInt` que deriva de `ArrayList` ou converter de `Object` para cada operação.

A versão 2.0 do C# trouxe iteradores. Em resumo, os iteradores permitem que você examine todos os itens em um `List` (ou outros tipos Enumeráveis) com um loop `foreach`. Ter iteradores como uma parte importante da linguagem aprimorou drasticamente a legibilidade da linguagem e a capacidade das pessoas de raciocinar a respeito do código.

E ainda assim, o C# continuava na tentativa de alcançar o mesmo nível do Java. O Java já tinha liberado versões que incluíam genéricos e iteradores. Mas isso seria alterado logo, pois as linguagens continuaram a evoluir separadamente.

## C# versão 1.2

*Lançado em abril de 2003*

C# versão 1.2 fornecida com o Visual Studio .NET 2003. Ele continha algumas pequenas melhorias na linguagem. Muito notável é que, a partir desta versão, o código gerado em

um loop `foreach` chamou `Dispose`, em um `IEnumerator`, quando o `IEnumerator` implementou `IDisposable`.

## C# versão 1.0

*Lançado em janeiro de 2002*

Ao olhar para trás, a versão 1.0 de C#, lançada com o Visual Studio .NET 2002, ficou muito parecida com o Java. Como [parte de suas metas de design declaradas para ECMA](#), ela procurou ser uma "linguagem simples, moderna e orientada a objetos para uso geral". Na época, parecer com Java significava que havia atingido essas metas de design iniciais.

Mas agora, se examinar novamente a C# 1.0, você poderá se sentir um pouco confuso. Carecia das funcionalidades assíncronas internas e algumas das funcionalidades relacionadas a genéricos que você nem valoriza. Na verdade, ela não tinha nada relacionado a genéricos. E a [LINQ](#)? Ainda não estava disponível. Essas adições levariam alguns anos para sair.

A versão 1.0 do C# parecia ter poucos recursos, em comparação com os dias de hoje. Você teria que escrever código um tanto detalhado. Mas, ainda assim, você poderia iniciar em algum lugar. A versão 1.0 do C# era uma alternativa viável ao Java na plataforma Windows.

Os principais recursos do C# 1.0 incluíam:

- [Classes](#)
- [Estruturas](#)
- [Interfaces](#)
- [Eventos](#)
- [Propriedades](#)
- [Representantes](#)
- [Operadores e expressões](#)
- [Instruções](#)
- [Atributos](#)

Artigo [originalmente publicado no blog NDepend](#), cortesia de Erik Dietrich e Patrick Smacchia.

# Relações entre os recursos de linguagem e os tipos de bibliotecas

Artigo • 10/05/2023

A definição de linguagem C# exige que uma biblioteca padrão tenha determinados tipos e determinados membros acessíveis nesses tipos. O compilador gera o código que usa esses tipos e membros necessários para muitos recursos de linguagem diferentes. Quando necessário, há pacotes NuGet que contêm os tipos necessários para as versões mais recentes da linguagem ao escrever um código para ambientes em que esses tipos ou membros ainda não foram implantados.

Essa dependência da funcionalidade da biblioteca padrão faz parte da linguagem C# desde sua primeira versão. Nessa versão, os exemplos incluíam:

- [Exception](#) – usado para todas as exceções geradas pelo compilador.
- [String](#) – tipo `string` C# é um sinônimo de [String](#).
- [Int32](#) – sinônimo de `int`.

A primeira versão era simples: o compilador e a biblioteca padrão eram fornecidos juntos e havia somente uma versão de cada um.

As versões posteriores do C# ocasionalmente adicionaram novos tipos ou membros às dependências. Os exemplos incluem: [INotifyCompletion](#), [CallerFilePathAttribute](#) e [CallerMemberNameAttribute](#). O C# 7.0 continua isso adicionando uma dependência de [ValueTuple](#) para implementar o recurso de linguagem de [tuplas](#).

A equipe de design de linguagem trabalha para minimizar a área de superfície dos tipos e membros necessários em uma biblioteca padrão em conformidade. Essa meta é equilibrada em relação a um design limpo no qual novos recursos de biblioteca são incorporados diretamente na linguagem. Haverá novos recursos em versões futuras do C# que exigem novos tipos e membros em uma biblioteca padrão. É importante entender como gerenciar essas dependências em seu trabalho.

## Gerenciando as dependências

As ferramentas do compilador C# agora são desacopladas do ciclo de liberação das bibliotecas .NET em plataformas com suporte. Na verdade, bibliotecas .NET diferentes têm ciclos de liberação diferentes: o .NET Framework no Windows é liberado como um Windows Update, o .NET Core é fornecido em um agendamento separado e as versões

do Xamarin das atualizações de biblioteca são fornecidas com as ferramentas do Xamarin para cada plataforma de destino.

Na maior parte do tempo, você não perceberá essas alterações. No entanto, quando estiver trabalhando com uma versão mais recente da linguagem que exige recursos que ainda não estão nas bibliotecas .NET nessa plataforma, você referenciará os pacotes NuGet para fornecer esses novos tipos. Conforme as plataformas para as quais seu aplicativo dá suporte forem atualizadas com as novas instalações de estrutura, você poderá remover a referência extra.

Essa separação significa que você pode usar os novos recursos de linguagem até mesmo quando direcionar computadores que podem não ter a estrutura correspondente.

# Considerações sobre versão e atualização para os desenvolvedores de C#

Artigo • 30/06/2023

A compatibilidade é uma meta importante quando novos recursos são adicionados à linguagem C#. Em quase todos os casos, o código existente pode ser recompilado com uma nova versão do compilador sem nenhum problema. A equipe de runtime do .NET também tem uma meta para garantir a compatibilidade das bibliotecas atualizadas. Em quase todos os casos, quando seu aplicativo é iniciado de um runtime atualizado com bibliotecas atualizadas, o comportamento é exatamente o mesmo que nas versões anteriores.

A versão de linguagem usada para compilar seu aplicativo normalmente corresponde ao TFM (moniker da estrutura de destino) de runtime referenciado em seu projeto. Para obter mais informações sobre como alterar a versão de linguagem padrão, confira o artigo intitulado [configurar sua versão de linguagem](#). Esse comportamento padrão garante a compatibilidade máxima.

Quando *alterações interruptivas* são introduzidas, elas são classificadas como:

- *Alteração interruptiva binária*: uma alteração interruptiva binária causa um comportamento diferente, incluindo possivelmente falha, em seu aplicativo ou biblioteca quando iniciado usando um novo runtime. Você deve recompilar seu aplicativo para incorporar essas alterações. O binário existente não funcionará corretamente.
- *Alteração interruptiva de origem*: uma alteração interruptiva de origem altera o significado do código-fonte. Você precisa fazer edições de código-fonte antes de compilar seu aplicativo com a versão mais recente do idioma. O binário existente será executado corretamente com o host e o runtime mais recentes. Observe que, para a sintaxe da linguagem, uma *alteração interruptiva de origem* também é uma *alteração comportamental*, conforme definido nas [alterações interruptivas do runtime](#).

Quando uma alteração interruptiva binária afeta seu aplicativo, você precisa recompilar seu aplicativo, mas não precisa editar nenhum código-fonte. Quando uma alteração interruptiva de origem afeta seu aplicativo, o binário existente ainda é executado corretamente em ambientes com o runtime e as bibliotecas atualizados. No entanto, você deve fazer alterações de origem para recompilar com a nova versão do idioma e o

runtime. Se uma alteração for interruptiva de origem e interruptiva binária, você deverá recompilar seu aplicativo com a versão mais recente e fazer atualizações de origem.

Devido à meta de evitar alterações interruptivas da equipe de linguagem C# e da equipe de runtime, atualizar seu aplicativo normalmente é uma questão de atualizar o TFM e recompilar o aplicativo. No entanto, para bibliotecas distribuídas publicamente, você deve avaliar cuidadosamente sua política quanto a TFM com suporte e versões de linguagem com suporte. Você poderá estar criando uma nova biblioteca com recursos encontrados na versão mais recente, e precisará garantir que os aplicativos criados com versões anteriores do compilador possam usá-la. Ou você pode estar atualizando uma biblioteca existente e muitos dos seus usuários podem não ter versões atualizadas ainda.

## Como introduzir alterações interruptivas em suas bibliotecas

Ao adotar novos recursos de linguagem na API pública da biblioteca, você deve avaliar se a adoção do recurso introduz uma alteração interruptiva binária ou de origem para os usuários da biblioteca. Todas as alterações na implementação interna que não aparecem nas interfaces `public` ou `protected` são compatíveis.

### ⓘ Observação

Se você usar o `System.Runtime.CompilerServices.InternalsVisibleToAttribute` para habilitar tipos para ver membros internos, os membros internos poderão introduzir alterações interruptivas.

Uma *alteração interruptiva binária* exige que os usuários recompilem o código para usar a nova versão. Por exemplo, considere este método público:

C#

```
public double CalculateSquare(double value) => value * value;
```

Se você adicionar o modificador `in` ao método, essa será uma alteração interruptiva binária:

C#

```
public double CalculateSquare(in double value) => value * value;
```

Os usuários devem recompilar qualquer aplicativo que use o método `CalculateSquare` para que a nova biblioteca funcione corretamente.

Uma *alteração interruptiva de origem* exige que os usuários alterem o código antes de recompilarem. Por exemplo, pense neste tipo:

C#

```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName) => (FirstName,
LastName) = (firstName, lastName);

    // other details omitted
}
```

Em uma versão mais recente, você gostaria de aproveitar os membros sintetizados gerados para tipos `record`. Você faz as seguintes alterações:

C#

```
public record class Person(string FirstName, string LastName);
```

A alteração anterior requer alterações para qualquer tipo derivado de `Person`. Todas essas declarações devem adicionar o modificador `record` às declarações.

## Impacto de alterações interruptivas

Ao adicionar uma *alteração interruptiva binária* à biblioteca, você força todos os projetos que usam sua biblioteca a recompilar. No entanto, nenhum código-fonte nesses projetos precisa ser alterado. Como resultado, o impacto da alteração interruptiva é razoavelmente pequeno para cada projeto.

Ao fazer uma *alteração interruptiva de origem* em sua biblioteca, você exige que todos os projetos façam alterações de origem para usar sua nova biblioteca. Se a alteração necessária exigir novos recursos de linguagem, você força esses projetos a atualizar para a mesma versão de idioma e TFM que você está usando agora. Você exigiu mais trabalho para seus usuários e, possivelmente, forçou-os a atualizar também.

O impacto de qualquer alteração interruptiva que você faz depende do número de projetos que têm uma dependência em sua biblioteca. Se sua biblioteca for usada

internamente por alguns aplicativos, você poderá reagir a todas alterações interruptivas em todos os projetos afetados. No entanto, se sua biblioteca for baixada publicamente, você deverá avaliar o impacto potencial e considerar alternativas:

- Você pode adicionar novas APIs que são APIs existentes paralelas.
- Você pode considerar builds paralelos para diferentes TFM's.
- Você pode considerar a multiplataforma.

# Tutorial: Explorar o recurso C# 11 – membros virtuais estáticos em interfaces

Artigo • 09/05/2023

O C# 11 e o .NET 7 incluem *membros virtuais estáticos em interfaces*. Esse recurso permite definir interfaces que incluem [operadores sobrecarregados](#) ou outros membros estáticos. Depois de definir interfaces com membros estáticos, você pode usar essas interfaces como [restrições](#) para criar tipos genéricos que usam operadores ou outros métodos estáticos. Mesmo que você não crie interfaces com operadores sobrecarregados, provavelmente se beneficiará desse recurso e das classes matemáticas genéricas habilitadas pela atualização da linguagem.

Neste tutorial, você aprenderá como:

- ✓ Definir interfaces com membros estáticos.
- ✓ Use interfaces para definir classes que implementam interfaces com operadores definidos.
- ✓ Crie algoritmos genéricos que dependem de métodos de interface estática.

## Pré-requisitos

Você precisará configurar seu computador para executar o .NET 7, que dá suporte ao C# 11. O compilador C# 11 está disponível a partir do [Visual Studio 2022, versão 17.3](#) ou do [.NET 7 SDK](#).

## Métodos de interface abstrato estáticos

Vamos começar com um exemplo. O método a seguir retorna o ponto médio de dois números `double`:

C#

```
public static double MidPoint(double left, double right) =>
    (left + right) / (2.0);
```

A mesma lógica funcionaria para qualquer tipo numérico: `int`, `short`, `long`, `float`, `decimal` ou qualquer tipo que represente um número. Você precisa ter uma

maneira de usar os operadores `+` e `/`, e definir um valor para `2`. Você pode usar a interface `System.Numerics.INumber<TSelf>` para escrever o método anterior como o seguinte método genérico:

C#

```
public static T MidPoint<T>(T left, T right)
    where T : INumber<T> => (left + right) / T.CreateChecked(2); // note:
    the addition of left and right may overflow here; it's just for
    demonstration purposes
```

Qualquer tipo que implemente a interface `INumber<TSelf>` deve incluir uma definição para `operator +` e `operator /`. O denominador é definido por `T.CreateChecked(2)` para criar o valor `2` para qualquer tipo numérico, o que força o denominador a ser do mesmo tipo que os dois parâmetros. `INumberBase<TSelf>.CreateChecked<TOther>(TOther)` cria uma instância do tipo do valor especificado e gera um `OverflowException` se o valor ficar fora do intervalo representável. (Essa implementação tem o potencial de estouro se `left` e `right` são valores grandes o suficiente. Há algoritmos alternativos que podem evitar esse possível problema.)

Você define membros abstratos estáticos em uma interface usando a sintaxe familiar: você adiciona os modificadores `static` e `abstract` a qualquer membro estático que não forneça uma implementação. O exemplo a seguir define uma interface `IGetNext<T>` que pode ser aplicada a qualquer tipo que substitua `operator ++`:

C#

```
public interface IGetNext<T> where T : IGetNext<T>
{
    static abstract T operator ++(T other);
}
```

A restrição de que o argumento de tipo, `T` implementa `IGetNext<T>`, garante que a assinatura do operador inclua o tipo que contém ou seu argumento de tipo. Muitos operadores impõem que seus parâmetros devem corresponder ao tipo ou ser o parâmetro de tipo restrito para implementar o tipo que contém. Sem essa restrição, o operador `++` não pôde ser definido na interface `IGetNext<T>`.

Você pode criar uma estrutura que cria uma cadeia de caracteres 'A' em que cada incremento adiciona outro caractere à cadeia de caracteres usando o seguinte código:

C#

```
public struct RepeatSequence : IGetNext<RepeatSequence>
{
    private const char Ch = 'A';
    public string Text = new string(Ch, 1);

    public RepeatSequence() {}

    public static RepeatSequence operator ++(RepeatSequence other)
        => other with { Text = other.Text + Ch };

    public override string ToString() => Text;
}
```

De modo mais geral, você pode criar qualquer algoritmo em que queira definir `++` para significar "produzir o próximo valor desse tipo". O uso dessa interface produz código e resultados claros:

C#

```
var str = new RepeatSequence();

for (int i = 0; i < 10; i++)
    Console.WriteLine(str++);
```

O código anterior gerencia a saída a seguir:

PowerShell

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
AAAAAAA
AAAAAA
AAAAAA
```

Este pequeno exemplo demonstra a motivação para esse recurso. Você pode usar sintaxe natural para operadores, valores constantes e outras operações estáticas. Você pode explorar essas técnicas ao criar vários tipos que dependem de membros estáticos, incluindo operadores sobrecarregados. Defina as interfaces que correspondem aos recursos de seus tipos e declare o suporte desses tipos para a nova interface.

## Matemática genérica

O cenário motivador para permitir métodos estáticos, incluindo operadores, em interfaces é dar suporte a algoritmos [matemáticos genéricos](#). A biblioteca de classes base do .NET 7 contém definições de interface para muitos operadores aritméticos e interfaces derivadas que combinam muitos operadores aritméticos em uma interface `INumber<T>`. Vamos aplicar esses tipos para criar um registro `Point<T>` que possa usar qualquer tipo numérico para `T`. O ponto pode ser movido por alguns `xoffset` e `yoffset` usando o operador `+`.

Comece criando um novo aplicativo console usando `dotnet new` ou o Visual Studio. Defina a versão da linguagem C# como "versão prévia", o que habilita recursos de visualização do C# 11. Adicione o seguinte elemento ao arquivo `csproj` dentro de um elemento `<PropertyGroup>`:

XML

```
<LangVersion>preview</LangVersion>
```

### ⓘ Observação

Esse elemento não pode ser definido usando a interface do usuário do Visual Studio. Você precisa editar o arquivo de projeto diretamente.

A interface pública para o `Translation<T>` e `Point<T>` deve se parecer com o seguinte código:

C#

```
// Note: Not complete. This won't compile yet.
public record Translation<T>(T XOffset, T YOffset);

public record Point<T>(T X, T Y)
{
    public static Point<T> operator +(Point<T> left, Translation<T> right);
}
```

Você usa o tipo `record` para os tipos `Translation<T>` e `Point<T>`: ambos armazenam dois valores e eles representam o armazenamento de dados em vez de um comportamento sofisticado. A implementação de `operator +` seria semelhante ao seguinte código:

C#

```
public static Point<T> operator +(Point<T> left, Translation<T> right) =>
    left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset };
```

Para que o código anterior seja compilado, você precisará declarar que `T` dá suporte à interface `IAdditionOperators<TSelf, TOther, TResult>`. Essa interface inclui o método estático `operator +`. Ele declara três parâmetros de tipo: um para o operando à esquerda, um para o operando à direita e outro para o resultado. Alguns tipos implementam `+` para diferentes tipos de operando e de resultado. Adicione uma declaração de que o argumento `T` de tipo implementa `IAdditionOperators<T, T, T>`:

C#

```
public record Point<T>(T X, T Y) where T : IAdditionOperators<T, T, T>
```

Depois de adicionar essa restrição, sua classe `Point<T>` poderá usar o `+` para seu operador de adição. Adicione a mesma restrição à declaração `Translation<T>`:

C#

```
public record Translation<T>(T XOffset, T YOffset) where T :
    IAdditionOperators<T, T, T>;
```

A restrição `IAdditionOperators<T, T, T>` impede que um desenvolvedor que usa sua classe crie um `Translation` usando um tipo que não atenda à restrição para a adição a um ponto. Você adicionou as restrições necessárias ao parâmetro de tipo para `Translation<T>` e `Point<T>`, portanto, esse código funciona. Você pode testar adicionando código como o seguinte acima das declarações de `Translation` e `Point` em seu arquivo `Program.cs`:

C#

```
var pt = new Point<int>(3, 4);

var translate = new Translation<int>(5, 10);

var final = pt + translate;

Console.WriteLine(pt);
Console.WriteLine(translate);
Console.WriteLine(final);
```

Você pode tornar esse código mais reutilizável declarando que esses tipos implementam as interfaces aritméticas apropriadas. A primeira alteração a ser executada

é declarar que `Point<T, T>` implementa a interface `IAdditionOperators<Point<T>, Translation<T>, Point<T>>`. O tipo `Point` usa diferentes tipos para operandos e o resultado. O tipo `Point` já implementa um `operator +` com essa assinatura, portanto, adicionar a interface à declaração é tudo o que você precisa:

C#

```
public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>, Translation<T>, Point<T>>
    where T : IAdditionOperators<T, T, T>
```

Por fim, ao executar a adição, é útil ter uma propriedade que defina o valor de identidade aditivo para esse tipo. Há uma nova interface para esse recurso: `IAdditiveIdentity<TSelf, TResult>`. Uma tradução de `{0, 0}` é a identidade aditiva: o ponto resultante é o mesmo que o operando à esquerda. A interface `IAdditiveIdentity<TSelf, TResult>` define uma propriedade `readonly AdditiveIdentity`, que retorna o valor de identidade. A `Translation<T>` precisa de algumas alterações para implementar essa interface:

C#

```
using System.Numerics;

public record Translation<T>(T XOffset, T YOffset) :
    IAdditiveIdentity<Translation<T>, Translation<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Translation<T> AdditiveIdentity =>
        new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
            T.AdditiveIdentity);
}
```

Há algumas alterações aqui, então vamos analisá-las uma por uma. Primeiro, você declara que o tipo `Translation` implementa a interface `IAdditiveIdentity`:

C#

```
public record Translation<T>(T XOffset, T YOffset) :
    IAdditiveIdentity<Translation<T>, Translation<T>>
```

Em seguida, você pode tentar implementar o membro da interface, conforme mostrado no seguinte código:

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: 0, YOffset: 0);
```

O código anterior não será compilado, pois `0` depende do tipo. A resposta: use `IAdditiveIdentity<T>.AdditiveIdentity` para `0`. Essa alteração significa que suas restrições agora devem incluir essa `T` implementa `IAdditiveIdentity<T>`. Isso resulta na seguinte implementação:

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
    T.AdditiveIdentity);
```

Agora que você adicionou essa restrição em `Translation<T>`, precisa adicionar a mesma restrição para `Point<T>`:

C#

```
using System.Numerics;

public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>,
    Translation<T>, Point<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Point<T> operator +(Point<T> left, Translation<T> right)
=>
    left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset
    };
}
```

Este exemplo deu uma olhada em como as interfaces para composição matemática genérica. Você aprendeu a:

- ✓ Escrever um método que dependia da interface `INumber<T>` para que esse método pudesse ser usado com qualquer tipo numérico.
- ✓ Criar um tipo que depende das interfaces de adição para implementar um tipo que dá suporte apenas a uma operação matemática. Que o tipo declara seu suporte para essas mesmas interfaces para que ele possa ser composto de outras maneiras. Os algoritmos são escritos usando a sintaxe mais natural dos operadores matemáticos.

Experimente esses recursos e registre comentários. Você pode usar o item de menu *Enviar Comentários* no Visual Studio ou criar um novo [problema](#) no repositório roslyn

no GitHub. Crie algoritmos genéricos que funcionam com qualquer tipo numérico. Criar algoritmos usando essas interfaces em que o argumento de tipo só pode implementar um subconjunto de recursos semelhantes a números. Mesmo que você não crie novas interfaces que usam essas funcionalidades, pode experimentar usá-las em seus algoritmos.

## Confira também

- [Matemática genérica](#)

# Criar tipos de registro

Artigo • 02/06/2023

C# 9 introduz *registros*, um novo tipo de referência que você pode criar em vez de classes ou structs. C# 10 adiciona *structs de registro* para que você possa definir registros como tipos de valor. Os registros são distintos das classes em que os tipos de registro usam *igualdade baseada em valor*. Duas variáveis de um tipo de registro são iguais se as definições de tipo de registro forem idênticas e, se para cada campo, os valores em ambos os registros forem iguais. Duas variáveis de um tipo de classe são iguais se os objetos referenciados forem do mesmo tipo de classe e as variáveis se referirem ao mesmo objeto. A igualdade baseada em valor implica outros recursos que você provavelmente desejará em tipos de registro. O compilador gera muitos desses membros quando você declara um `record` em vez de um `class`. O compilador gera esses mesmos métodos para tipos `record struct`.

Neste tutorial, você aprenderá como:

- ✓ Decida se você deve declarar um `class` ou um `record`.
- ✓ Declare tipos de registro e tipos de registro posicionais.
- ✓ Substitua seus métodos por métodos gerados pelo compilador em registros.

## Pré-requisitos

Você precisará configurar seu computador para executar o .NET 6 ou posterior, incluindo o compilador C# 10 ou posterior. O compilador C# 10 está disponível a partir do [Visual Studio 2022](#) ou do [.NET 6 SDK](#).

## Características dos registros

Você define um *registro* declarando um tipo com a `record` palavra-chave, em vez da `class` palavra-chave ou `struct`. Opcionalmente, você pode declarar um `record class` para esclarecer que ele é um tipo de referência. Um registro é um tipo de referência e segue a semântica de igualdade baseada em valor. Você pode definir um `record struct` para criar um registro que seja um tipo de valor. Para impor semântica de valor, o compilador gera vários métodos para o tipo de registro (para tipos `record class` e `record struct`):

- Uma substituição de [Object.Equals\(Object\)](#).
- Um método virtual `Equals` cujo parâmetro é o tipo de registro.

- Uma substituição de `Object.GetHashCode()`.
- Métodos para `operator ==` e `operator !=`.
- Tipos de registro implementam `System.IEquatable<T>`.

Os registros também fornecem uma substituição de `Object.ToString()`. O compilador sintetiza métodos para exibir registros usando `Object.ToString()`. Você explorará esses membros enquanto escreve o código para este tutorial. Os registros aceitam as expressões `with` para habilitar a mutação não destrutiva de registros.

Você também pode declarar *registros posicionais* usando uma sintaxe mais concisa. O compilador sintetiza mais métodos para você ao declarar registros posicionais:

- Um construtor primário cujos parâmetros correspondem aos parâmetros posicionais na declaração de registro.
- Propriedades públicas para cada parâmetro de um construtor primário. Essas propriedades são *somente init* para tipos `record class` e tipos `readonly record struct`. Para tipos `record struct`, elas são *leitura-gravação*.
- Um método `Deconstruct` para extrair propriedades do registro.

## Compilar dados de temperatura

Dados e estatísticas estão entre os cenários em que você deseja usar registros. Para este tutorial, você criará um aplicativo que computa *dias de grau* para usos diferentes. Os *dias de grau* são uma medida de calor (ou falta de calor) durante um período de dias, semanas ou meses. Os dias de grau acompanham e prevêem o uso de energia. Dias mais quentes significam mais ar condicionado, e dias mais frios significam mais uso de fornos. Os dias de grau ajudam a gerenciar as populações vegetais e a correlacionar-se com o crescimento das plantas à medida que as estações mudam. Os dias de grau ajudam a acompanhar as migrações de animais para espécies que viajam para corresponder ao clima.

A fórmula baseia-se na temperatura média em um determinado dia e em uma temperatura de linha de base. Para calcular dias de grau ao longo do tempo, você precisará da temperatura alta e baixa todos os dias por um período de tempo. Vamos começar criando um novo aplicativo. Crie um novo aplicativo de console Crie um novo tipo de registro em um novo arquivo chamado "DailyTemperature.cs":

C#

```
public readonly record struct DailyTemperature(double HighTemp, double LowTemp);
```

O código anterior define um *registro posicional*. O registro `DailyTemperature` é um `readonly record struct`, porque você não pretende herdar dele, e deve ser imutável. As propriedades `HighTemp` e `LowTemp` são *apenas propriedades init*, o que significa que elas podem ser definidas no construtor ou usando um inicializador de propriedade. Se você quiser que os parâmetros posicionais sejam leitura-gravação, declare `record struct` em vez de `readonly record struct`. O tipo `DailyTemperature` também tem um *construtor primário* que tem dois parâmetros que correspondem às duas propriedades. Você usa o construtor primário para inicializar um registro `DailyTemperature`. O código a seguir cria e inicializa vários registros `DailyTemperature`. O primeiro usa parâmetros nomeados para esclarecer `HighTemp` e `LowTemp`. Os inicializadores restantes usam parâmetros posicionais para inicializar `HighTemp` e `LowTemp`:

C#

```
private static DailyTemperature[] data = new DailyTemperature[]
{
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
};
```

Você pode adicionar suas próprias propriedades ou métodos a registros, incluindo registros posicionais. Você precisará calcular a temperatura média para cada dia. Você pode adicionar essa propriedade ao registro `DailyTemperature`:

C#

```
public readonly record struct DailyTemperature(double HighTemp, double
LowTemp)
{
```

```
    public double Mean => (HighTemp + LowTemp) / 2.0;  
}
```

Vamos garantir que você possa usar esses dados. Adicione o código a seguir ao método `Main`:

C#

```
foreach (var item in data)  
    Console.WriteLine(item);
```

Execute seu aplicativo e você verá uma saída semelhante à exibição a seguir (várias linhas removidas para espaço):

CLI do .NET

```
DailyTemperature { HighTemp = 57, LowTemp = 30, Mean = 43.5 }  
DailyTemperature { HighTemp = 60, LowTemp = 35, Mean = 47.5 }  
  
DailyTemperature { HighTemp = 80, LowTemp = 60, Mean = 70 }  
DailyTemperature { HighTemp = 85, LowTemp = 66, Mean = 75.5 }
```

O código anterior mostra a saída da substituição de `ToString` sintetizado pelo compilador. Se preferir texto diferente, você poderá escrever sua própria versão que `ToString` impede que o compilador sintetize uma versão para você.

## Computar dias de grau

Para computar dias de grau, você faz a diferença de uma temperatura de linha de base e da temperatura média em um determinado dia. Para medir o calor ao longo do tempo, descarte qualquer dia em que a temperatura média esteja abaixo da linha de base. Para medir o frio ao longo do tempo, descarte qualquer dia em que a temperatura média esteja acima da linha de base. Por exemplo, os EUA usam 65F como base para dias de aquecimento e grau de resfriamento. Essa é a temperatura em que nenhum aquecimento ou resfriamento é necessário. Se um dia tiver uma temperatura média de 70F, esse dia será de cinco dias de resfriamento e zero dias de grau de aquecimento. Por outro lado, se a temperatura média for 55F, esse dia será de 10 dias de aquecimento e 0 dias de resfriamento.

Você pode expressar essas fórmulas como uma pequena hierarquia de tipos de registro: um tipo de dia de grau abstrato e dois tipos concretos para dias de grau de aquecimento e dias de grau de resfriamento. Esses tipos também podem ser registros

posicionais. Eles obtêm uma temperatura de linha de base e uma sequência de registros de temperatura diária como argumentos para o construtor primário:

```
C#  
  
public abstract record DegreeDays(double BaseTemperature,  
IEnumerable<DailyTemperature> TempRecords);  
  
public sealed record HeatingDegreeDays(double BaseTemperature,  
IEnumerable<DailyTemperature> TempRecords)  
    : DegreeDays(BaseTemperature, TempRecords)  
{  
    public double DegreeDays => TempRecords.Where(s => s.Mean <  
BaseTemperature).Sum(s => BaseTemperature - s.Mean);  
}  
  
public sealed record CoolingDegreeDays(double BaseTemperature,  
IEnumerable<DailyTemperature> TempRecords)  
    : DegreeDays(BaseTemperature, TempRecords)  
{  
    public double DegreeDays => TempRecords.Where(s => s.Mean >  
BaseTemperature).Sum(s => s.Mean - BaseTemperature);  
}
```

O registro abstrato `DegreeDays` é a classe base compartilhada para os registros `HeatingDegreeDays` e `CoolingDegreeDays`. As declarações de construtor primário nos registros derivados mostram como gerenciar a inicialização do registro base. Seu registro derivado declara parâmetros para todos os parâmetros no construtor primário do registro base. O registro base declara e inicializa essas propriedades. O registro derivado não os oculta, mas apenas cria e inicializa propriedades para parâmetros que não são declarados em seu registro base. Neste exemplo, os registros derivados não adicionam novos parâmetros de construtor primário. Teste seu código adicionando o seguinte código ao seu método `Main`:

```
C#  
  
var heatingDegreeDays = new HeatingDegreeDays(65, data);  
Console.WriteLine(heatingDegreeDays);  
  
var coolingDegreeDays = new CoolingDegreeDays(65, data);  
Console.WriteLine(coolingDegreeDays);
```

Você obterá uma saída como a seguinte exibição:

CLI do .NET

```
HeatingDegreeDays { BaseTemperature = 65, TempRecords =
record_types.DailyTemperature[], DegreeDays = 85 }
CoolingDegreeDays { BaseTemperature = 65, TempRecords =
record_types.DailyTemperature[], DegreeDays = 71.5 }
```

## Definir métodos sintetizados pelo compilador

Seu código calcula o número correto de dias de grau de aquecimento e resfriamento durante esse período de tempo. Mas este exemplo mostra por que talvez você queira substituir alguns dos métodos sintetizados para registros. Você pode declarar sua própria versão de qualquer um dos métodos sintetizados pelo compilador em um tipo de registro, exceto o método `clone`. O método `clone` tem um nome gerado pelo compilador e você não pode fornecer uma implementação diferente. Esses métodos sintetizados incluem um construtor de cópia, os membros da interface

`System.IEquatable<T>`, os testes de igualdade e desigualdade, e `GetHashCode()`. Para essa finalidade, você sintetizará `PrintMembers`. Você também pode declarar sua própria `ToString`, mas `PrintMembers` fornece uma opção melhor para cenários de herança. Para fornecer sua própria versão de um método sintetizado, a assinatura deve corresponder ao método sintetizado.

O elemento `TempRecords` na saída do console não é útil. Ele exibe o tipo, mas nada mais. Você pode alterar esse comportamento fornecendo sua própria implementação do método sintetizado `PrintMembers`. A assinatura depende de modificadores aplicados à declaração `record`:

- Se um tipo de registro for `sealed`, ou um `record struct`, a assinatura será `private bool PrintMembers(StringBuilder builder);`
- Se um tipo de registro não `sealed` for e derivar de `object` (ou seja, ele não declara um registro base), a assinatura será `protected virtual bool PrintMembers(StringBuilder builder);`
- Se um tipo de registro não for `sealed` e derivar de outro registro, a assinatura será `protected override bool PrintMembers(StringBuilder builder);`

Essas regras são mais fáceis de compreender por meio da compreensão da finalidade de `PrintMembers`. `PrintMembers` adiciona informações sobre cada propriedade em um tipo de registro a uma cadeia de caracteres. O contrato requer registros base para adicionar seus membros à exibição e pressupõe que os membros derivados adicionarão seus membros. Cada tipo de registro sintetiza uma substituição `ToString` semelhante ao exemplo a seguir para `HeatingDegreeDays`:

C#

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("HeatingDegreeDays");
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

Você declara um método `PrintMembers` no registro `DegreeDays` que não imprime o tipo da coleção:

C#

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

A assinatura declara um método `virtual protected` para corresponder à versão do compilador. Não se preocupe se você tiver errado os acessadores; a linguagem impõe a assinatura correta. Se você esquecer os modificadores corretos para qualquer método sintetizado, o compilador emitirá avisos ou erros que o ajudarão a obter a assinatura certa.

No C# 10 e posterior, você pode declarar o método `ToString` como `sealed` em um tipo de registro. Isso impede que registros derivados forneçam uma nova implementação. Os registros derivados ainda conterão a substituição `PrintMembers`. Você vedaria `ToString` se não quisesse que ele exibisse o tipo de runtime do registro. No exemplo anterior, você perderia as informações sobre onde o registro estava medindo dias de aquecimento ou grau de resfriamento.

## Mutação não destrutiva

Os membros sintetizados em uma classe de registro posicional não modificam o estado do registro. A meta é que você possa criar registros imutáveis com mais facilidade. Lembre-se de declarar um `readonly record struct` para criar um `struct` de registro

imutável. Examine novamente as declarações anteriores para `HeatingDegreeDays` e `CoolingDegreeDays`. Os membros adicionados executam cálculos nos valores do registro, mas não modificam o estado. Os registros posicionais facilitam a criação de tipos de referência imutáveis.

Criar tipos de referência imutáveis significa que você deseja usar mutação não destrutiva. Você cria novas instâncias de registro semelhantes às instâncias de registro existentes usando [withexpressões](#). Essas expressões são uma construção de cópia com atribuições adicionais que modificam a cópia. O resultado é uma nova instância de registro em que cada propriedade foi copiada do registro existente e, opcionalmente, modificada. O registro original não foi alterado.

Vamos adicionar alguns recursos ao programa que demonstram expressões `with`. Primeiro, vamos criar um novo registro para calcular dias de grau crescente usando os mesmos dados. *Dias de grau crescente* normalmente usam 41F como a linha de base e mede as temperaturas acima da linha de base. Para usar os mesmos dados, você pode criar um novo registro semelhante ao `coolingDegreeDays`, mas com uma temperatura base diferente:

C#

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

Você pode comparar o número de graus calculados com os números gerados com uma temperatura de linha de base mais alta. Lembre-se de que os registros são *tipos de referência* e essas cópias são cópias rasas. A matriz para os dados não é copiada, mas ambos os registros se referem aos mesmos dados. Esse fato é uma vantagem em um outro cenário. Para dias de grau crescente, é útil controlar o total dos cinco dias anteriores. Você pode criar novos registros com dados de origem diferentes usando expressões `with`. O código a seguir cria uma coleção desses acúmulos e exibe os valores:

C#

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..
(start + rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
```

```
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

Você também pode usar expressões `with` para criar cópias de registros. Não especifique nenhuma propriedade entre as chaves para a expressão `with`. Isso significa criar uma cópia e não alterar nenhuma propriedade:

C#

```
var growingDegreeDaysCopy = growingDegreeDays with { };
```

Execute o aplicativo concluído para ver os resultados.

## Resumo

Este tutorial mostrou vários aspectos dos registros. Os registros fornecem sintaxe concisa para tipos em que o uso fundamental é armazenar dados. Para classes orientadas a objeto, o uso fundamental é definir responsabilidades. Este tutorial se concentrou em *registros posicionais*, em que você pode usar uma sintaxe concisa para declarar as propriedades de um registro. O compilador sintetiza vários membros do registro para copiar e comparar registros. Você pode adicionar outros membros necessários para seus tipos de registro. Você pode criar tipos de registro imutáveis sabendo que nenhum dos membros gerados pelo compilador alteraria o estado. E as expressões `with` facilitam o suporte a mutações não destrutivas.

Os registros adicionam outra maneira de definir tipos. Você usa definições `class` para criar hierarquias orientadas a objeto que se concentram nas responsabilidades e no comportamento dos objetos. Você cria `struct` tipos para estruturas de dados que armazenam dados e são pequenos o suficiente para copiar com eficiência. Você cria tipos `record` quando deseja igualdade e comparação baseadas em valor, não deseja copiar valores e deseja usar variáveis de referência. Você cria tipos `record struct` quando deseja os recursos de registros para um tipo pequeno o suficiente para copiar com eficiência.

Você pode saber mais sobre registros no [artigo de referência de linguagem C# para o tipo de registro](#), a [especificação de tipo de registro proposto](#) e a [especificação do struct de registro](#).

# Tutorial: Explorar ideias usando instruções de nível superior para criar código conforme você aprende

Artigo • 10/05/2023

Neste tutorial, você aprenderá como:

- ✓ Conheça as regras que regem o uso de instruções de nível superior.
- ✓ Use instruções de nível superior para explorar algoritmos.
- ✓ Refatore explorações em componentes reutilizáveis.

## Pré-requisitos

Você precisará configurar seu computador para executar o .NET 6, que inclui o compilador C# 10. O compilador C# 10 está disponível a partir do [Visual Studio 2022](#) ou do [.NET 6 SDK](#).

Este tutorial pressupõe que você esteja familiarizado com o C# e .NET, incluindo o Visual Studio ou a CLI do .NET.

## Comece a explorar

As instruções de nível superior permitem evitar a cerimônia extra necessária colocando o ponto de entrada do programa em um método estático em uma classe. O ponto de partida típico para um novo aplicativo de console se parece com o seguinte código:

C#

```
using System;

namespace Application
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

O código anterior é o resultado da execução do comando `dotnet new console` e da criação de um novo aplicativo de console. Essas 11 linhas contêm apenas uma linha de código executável. Você pode simplificar esse programa com o novo recurso de instruções de nível superior. Isso permite que você remova todas, exceto duas das linhas deste programa:

```
C#
```

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

## ⓘ Importante

Os modelos C# para o .NET 6 usam *instruções de nível superior*. Se você já tiver atualizado para o .NET 6, talvez seu aplicativo não corresponda ao código descrito neste artigo. Para obter mais informações, consulte o artigo sobre [Novos modelos C# geram instruções de nível superior](#)

O SDK do .NET 6 também adiciona um conjunto de diretivas *implícitas global using* para projetos que usam os seguintes SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Essas diretivas *implícitas global using* incluem os namespaces mais comuns para o tipo de projeto.

Esse recurso simplifica o que é necessário para começar a explorar novas ideias. Você pode usar instruções de nível superior em cenários de script ou para explorar. Assim que o básico estiver funcionando, você pode começar a refatorar o código e criar métodos, classes ou outros assemblies para componentes reutilizáveis criados. As instruções de nível superior habilitam experimentações rápidas e tutoriais iniciantes. Elas também fornecem um caminho tranquilo da experimentação a programas completos.

As instruções de nível superior são executadas na ordem em que aparecem no arquivo. As instruções de nível superior só podem ser usadas em um arquivo de origem em seu aplicativo. O compilador gera um erro se você usá-los em mais de um arquivo.

# Criar um computador de resposta mágico do .NET

Neste tutorial, vamos criar um aplicativo de console que responda a uma pergunta "sim" ou "não" com uma resposta aleatória. Você criará a funcionalidade passo a passo. Você pode se concentrar em sua tarefa em vez da cerimônia necessária para a estrutura de um programa típico. Em seguida, quando estiver satisfeito com a funcionalidade, você poderá refatorar o aplicativo conforme julgar adequado.

Um bom ponto de partida é gravar a pergunta de volta para o console. Você pode começar gravando o seguinte código:

```
C#  
  
Console.WriteLine(args);
```

Você não declara uma variável `args`. Para o arquivo de origem único que contém suas instruções de nível superior, o compilador reconhece `args` como significando os argumentos de linha de comando. O tipo de argumento é um `string[]`, como em todos os programas C#.

O código pode ser testado executando o comando `dotnet run` a seguir:

```
CLI do .NET  
  
dotnet run -- Should I use top level statements in all my programs?
```

Os argumentos após a linha de comando `--` são passados para o programa. Você pode ver o tipo da variável `args`, porque é isso que é impresso no console:

```
Console  
  
System.String[]
```

Para gravar a pergunta no console, você precisará enumerar os argumentos e separá-los com um espaço. Substitua a chamada `WriteLine` pelo código a seguir:

```
C#  
  
Console.WriteLine();  
foreach(var s in args)  
{  
    Console.Write(s);
```

```
        Console.Write(' ');
    }
Console.WriteLine();
```

Agora, ao executar o programa, ele exibirá corretamente a pergunta como uma cadeia de argumentos.

## Responder com uma resposta aleatória

Após ecoar a pergunta, você pode adicionar o código para gerar a resposta aleatória. Comece adicionando uma matriz de respostas possíveis:

C#

```
string[] answers =
{
    "It is certain.",           "Reply hazy, try again.",      "Don't count on
it.",
    "It is decidedly so.",     "Ask again later.",            "My reply is no.",
    "Without a doubt.",        "Better not tell you now.",   "My sources say
no.",
    "Yes - definitely.",       "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",     "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};
```

Essa matriz tem dez respostas afirmativas, cinco não confirmadas e cinco negativas. Em seguida, adicione o seguinte código para gerar e exibir uma resposta aleatória da matriz:

C#

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Você pode executar o aplicativo novamente para ver os resultados. Você deve ver algo semelhante à seguinte saída:

CLI do .NET

```
dotnet run -- Should I use top level statements in all my programs?
```

```
Should I use top level statements in all my programs?  
Better not tell you now.
```

Esse código responde às perguntas, mas vamos adicionar mais um recurso. Você gostaria que seu aplicativo de perguntas simulasse o pensamento sobre a resposta. Você pode fazer isso adicionando um pouco de animação do ASCII e pausando enquanto trabalha. Adicione o seguinte código após a linha que ecoa a pergunta:

```
C#
```

```
for (int i = 0; i < 20; i++)  
{  
    Console.Write(" | -");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("/ \\");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("- |");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("\\ /");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
}  
Console.WriteLine();
```

Também será necessário adicionar uma instrução `using` à parte superior do arquivo de origem:

```
C#
```

```
using System.Threading.Tasks;
```

As instruções `using` devem aparecer antes de qualquer outra instrução no arquivo. Caso contrário, é um erro do compilador. Você pode executar o programa novamente e ver a animação. Isso torna uma experiência melhor. Escolha um comprimento do atraso que corresponda ao seu gosto.

O código anterior cria um conjunto de linhas de rotação separadas por um espaço. A adição da palavra-chave `await` instrui o compilador a gerar o ponto de entrada do programa como um método que tem o modificador `async` e retorna um `System.Threading.Tasks.Task`. Esse programa não retorna um valor, portanto, o ponto de

entrada do programa retorna um `Task`. Se o programa retornar um valor inteiro, você adicionará uma instrução `return` ao final de suas instruções de nível superior. Essa instrução `return` especificaria o valor inteiro a ser retornado. Se suas instruções de nível superior incluírem uma expressão `await`, o tipo de retorno se tornará `System.Threading.Tasks.Task<TResult>`.

## Refatoração para o futuro

O programa deverá ser semelhante ao seguinte código:

```
C#  
  
Console.WriteLine();  
foreach(var s in args)  
{  
    Console.Write(s);  
    Console.Write(' ');  
}  
Console.WriteLine();  
  
for (int i = 0; i < 20; i++)  
{  
    Console.Write("| -");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("/ \\");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("- |");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
    Console.Write("\\ /");  
    await Task.Delay(50);  
    Console.Write("\b\b\b");  
}  
Console.WriteLine();  
  
string[] answers =  
{  
    "It is certain.",           "Reply hazy, try again.",      "Don't count on  
it.",  
    "It is decidedly so.",     "Ask again later.",          "My reply is no.",  
    "Without a doubt.",        "Better not tell you now.",   "My sources say  
no.",  
    "Yes - definitely.",       "Cannot predict now.",        "Outlook not so  
good.",  
    "You may rely on it.",     "Concentrate and ask again.", "Very doubtful.",  
    "As I see it, yes.",  
    "Most likely.",  
    "Outlook good.",  
    "Yes."  
}
```

```
    "Signs point to yes.",  
};  
  
var index = new Random().Next(answers.Length - 1);  
Console.WriteLine(answers[index]);
```

O código anterior é razoável. Funciona. Mas não é reutilizável. Agora que o aplicativo está funcionando, é hora de retirar as partes reutilizáveis.

Um candidato é o código que exibe a animação de espera. Esse snippet pode se tornar um método:

Você pode começar criando uma função local em seu arquivo. Substitua a animação atual pelo seguinte:

```
C#  
  
await ShowConsoleAnimation();  
  
static async Task ShowConsoleAnimation()  
{  
    for (int i = 0; i < 20; i++)  
    {  
        Console.Write("| -");  
        await Task.Delay(50);  
        Console.Write("\b\b\b");  
        Console.Write("/ \\");  
        await Task.Delay(50);  
        Console.Write("\b\b\b");  
        Console.Write("- |");  
        await Task.Delay(50);  
        Console.Write("\b\b\b");  
        Console.Write("\\ /");  
        await Task.Delay(50);  
        Console.Write("\b\b\b");  
    }  
    Console.WriteLine();  
}
```

O código anterior cria uma função local no método principal. Ainda não é reutilizável. Então, extraia esse código em uma classe. Crie um novo arquivo chamado *utilities.cs* e adicione o seguinte código:

```
C#  
  
namespace MyNamespace  
{  
    public static class Utilities  
    {  
        public static async Task ShowConsoleAnimation()
```

```

    {
        for (int i = 0; i < 20; i++)
        {
            Console.Write(" | -");
            await Task.Delay(50);
            Console.Write("\b\b\b");
            Console.Write("/ \\");
            await Task.Delay(50);
            Console.Write("\b\b\b");
            Console.Write("- |");
            await Task.Delay(50);
            Console.Write("\b\b\b");
            Console.Write("\\ /");
            await Task.Delay(50);
            Console.Write("\b\b\b");
        }
        Console.WriteLine();
    }
}

```

Um arquivo com instruções de nível superior também pode conter namespaces e tipos no final do arquivo, após as instruções de nível superior. Porém, para este tutorial, você coloca o método de animação em um arquivo separado para torná-lo mais facilmente reutilizável.

Por fim, é possível limpar o código de animação para remover algumas duplicações:

```
C#
foreach (string s in new[] { "| -", "/ \\", "- |", "\\ /", })
{
    Console.Write(s);
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
```

Agora você tem um aplicativo completo e refatorou as partes reutilizáveis para uso posterior. Você pode chamar o novo método utilitário de suas instruções de nível superior, conforme exibido abaixo na versão final do programa principal:

```
C#
using MyNamespace;

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' '');
```

```
}

Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
{
    "It is certain.",           "Reply hazy, try again.",      "Don't count on
it.",
    "It is decidedly so.",     "Ask again later.",            "My reply is no.",
    "Without a doubt.",        "Better not tell you now.",   "My sources say
no.",
    "Yes - definitely.",       "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",     "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

O exemplo anterior adiciona a chamada `Utilities.ShowConsoleAnimation` e uma instrução adicional `using`.

## Resumo

As instruções de nível superior facilitam a criação de programas simples para uso para explorar novos algoritmos. É possível experimentar algoritmos tentando diferentes snippets de código. Após verificar o que funciona, é possível refatorar o código para ser mais sustentável.

As instruções de nível superior simplificam programas baseados em aplicativos de console. Elas incluem funções do Azure, ações do GitHub e outros utilitários pequenos. Para obter mais informações, consulte [Instruções de nível superior \(Guia de Programação em C#\)](#).

# Usar a correspondência de padrões para criar o comportamento de classe para um código melhor

Artigo • 09/05/2023

Os recursos de correspondência de padrões em C# fornecem sintaxe para expressar os algoritmos. Você pode usar essas técnicas para implementar o comportamento nas classes. Você pode combinar o design de classe orientada a objeto com uma implementação orientada a dados para fornecer código conciso durante a modelagem de objetos do mundo real.

Neste tutorial, você aprenderá como:

- ✓ Expressar as classes orientadas a objeto usando padrões de dados.
- ✓ Implementar esses padrões usando os recursos de correspondência de padrões do C#.
- ✓ Aproveitar o diagnóstico do compilador para validar a implementação.

## Pré-requisitos

Você precisará configurar o computador para executar o .NET 5, incluindo o compilador C# 9. O compilador C# 9 está disponível a partir do [Visual Studio 2019, versão 16.8](#) ou do [.NET 5 SDK](#).

## Criar uma simulação de um bloqueio de canal

Neste tutorial, você criará uma classe C# que simula uma [eclusa de canal](#).

Resumidamente, uma eclusa de canal é um dispositivo que levanta e abaixa os barcos à medida que viajam entre dois trechos de água em diferentes níveis. Um bloqueio tem dois portões e algum mecanismo para alterar o nível da água.

Em sua operação normal, um barco entra em um dos portões enquanto o nível da água na eclusa coincide com o nível da água do lado em que o barco entra. Uma vez na eclusa, o nível da água é alterado para corresponder ao nível da água no qual o barco sairá da eclusa. Quando o nível da água corresponder a esse lado, o portão do lado de saída se abre. Medidas de segurança garantem que um operador não possa criar uma situação perigosa no canal. O nível da água só pode ser alterado quando ambos os portões são fechados. No máximo, um portão pode ser aberto. Para abrir um portão, o

nível da água na clausa deve corresponder ao nível da água fora do portão que está sendo aberto.

Você pode criar uma classe C# para modelar esse comportamento. Uma classe `CanalLock` daria suporte a comandos para abrir ou fechar um dos portões. Ela teria outros comandos para subir ou descer a água. A classe também deve dar suporte a propriedades para ler o estado atual dos portões e do nível da água. Os métodos implementam as medidas de segurança.

## Definir um classe

Você criará um aplicativo de console para testar a classe `CanalLock`. Crie um novo projeto de console para o .NET 5 usando o Visual Studio ou a CLI do .NET. Depois, adicione uma nova classe e nomeie-a `CanalLock`. Em seguida, crie a API pública, mas deixe os métodos não implementados:

C#

```
public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } =
    WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }
}
```

```
    public override string ToString() =>
        $"The lower gate is {{(LowWaterGateOpen ? "Open" : "Closed")}}. " +
        $"The upper gate is {{(HighWaterGateOpen ? "Open" : "Closed")}}. " +
        $"The water level is {CanalLockWaterLevel}.";
```

O código anterior inicializa o objeto para que ambos os portões sejam fechados e o nível da água seja baixo. Em seguida, escreva o seguinte código de teste no método `Main` para orientar você ao criar uma primeira implementação da classe:

C#

```
// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate: {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate: {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate: {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate: {canalGate}");
```

Em seguida, adicione uma primeira implementação de cada método na classe `CanalLock`. O código a seguir implementa os métodos da classe sem preocupação com

as regras de segurança. Você adicionará testes de segurança mais tarde:

```
C#  
  
// Change the upper gate.  
public void SetHighGate(bool open)  
{  
    HighWaterGateOpen = open;  
}  
  
// Change the lower gate.  
public void SetLowGate(bool open)  
{  
    LowWaterGateOpen = open;  
}  
  
// Change water level.  
public void SetWaterLevel(WaterLevel newLevel)  
{  
    CanalLockWaterLevel = newLevel;  
}
```

Os testes que você escreveu até agora são aprovados. Você implementou as noções básicas. Agora, escreva um teste para a primeira condição de falha. No final dos testes anteriores, ambos os portões são fechados, e o nível da água é definido como baixo. Adicione um teste para tentar abrir o portão superior:

```
C#  
  
Console.WriteLine("=====");  
Console.WriteLine("      Test invalid commands");  
// Open "wrong" gate (2 tests)  
try  
{  
    canalGate = new CanalLock();  
    canalGate.SetHighGate(open: true);  
}  
catch (InvalidOperationException)  
{  
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is low.");  
}  
Console.WriteLine($"Try to open upper gate: {canalGate}");
```

Esse teste não deu certo porque o portão foi aberto. Como uma primeira implementação, você pode corrigí-la com o seguinte código:

```
C#
```

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    if (open && (CanalLockWaterLevel == WaterLevel.High))
        HighWaterGateOpen = true;
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))
        throw new InvalidOperationException("Cannot open high gate when the
water is low");
}
```

Os testes são aprovados. Mas, à medida que mais testes são adicionados, você adicionará mais e mais cláusulas `if` e testará propriedades diferentes. Em breve, esses métodos ficarão muito complicados à medida que você adicionar mais condicionais.

## Implementar os comandos com padrões

Uma outra maneira melhor é usar *padrões* para determinar se o objeto está em um estado válido para executar um comando. Você poderá expressar se um comando for permitido como uma função de três variáveis: o estado do portão, o nível da água e a nova configuração:

Nova configuração	Estado do portão	Nível da Água	Result
Fechado	Fechado	Alto	Fechado
Fechado	Fechado	Baixo	Fechado
Fechado	Abrir	Alto	Fechadas
<del>Fechadas</del>	<del>Abrir</del>	<del>Baixo</del>	<del>Fechadas</del>
Abrir	Fechadas	Alto	Abrir
Abrir	Fechadas	Baixo	Fechado (Erro)
Abrir	Abrir	Alto	Abrir
<del>Abrir</del>	<del>Abrir</del>	<del>Baixo</del>	<del>Fechado (Erro)</del>

As quartas e últimas linhas da tabela atingiram o texto porque são inválidas. O código que você está adicionando agora deve garantir que o portão de água alta nunca seja aberto quando a água estiver baixa. Esses estados podem ser codificados como uma expressão única de comutador (lembre-se de que `false` indica "Fechado"):

```

HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};

```

Experimente esta versão. Os testes passam, validando o código. A tabela completa mostra as possíveis combinações de entradas e resultados. Isso significa que você e outros desenvolvedores podem examinar rapidamente a tabela e ver que todas as entradas possíveis foram abrangidas. Ainda mais fácil, o compilador também pode ajudar. Depois de adicionar o código anterior, você pode ver que o compilador gera um aviso: CS8524 indica que a expressão de comutador não abrange todas as entradas possíveis. O motivo desse aviso é que uma das entradas é um tipo `enum`. O compilador interpreta "todas as entradas possíveis" como todas as entradas do tipo subjacente, normalmente um `int`. Essa expressão `switch` verifica apenas os valores declarados no `enum`. Para remover o aviso, você pode adicionar um padrão de descarte catch-all para o último braço da expressão. Essa condição gera uma exceção, pois indica uma entrada inválida:

C#

```
_ => throw new InvalidOperationException("Invalid internal state"),
```

O braço de comutador anterior deve ser o último na expressão `switch` porque corresponde a todas as entradas. Experimente movendo-o anteriormente na ordem. Isso causa um erro do compilador CS8510 para código inacessível em um padrão. A estrutura natural de expressões de comutador permite que o compilador gere erros e avisos para possíveis erros. A "rede de segurança" do compilador facilita a criação de código correto em menos iterações e a liberdade de combinar armas com curingas. O compilador emitirá erros se a combinação resultar em braços inacessíveis que você não esperava e avisos se você remover um braço necessário.

A primeira alteração é combinar todos os braços em que o comando deve fechar o portão; isso é sempre permitido. Adicione o seguinte código como o primeiro braço na expressão de comutador:

```
C#
```

```
(false, _, _) => false,
```

Depois de adicionar o braço de comutador anterior, você receberá quatro erros do compilador, um em cada um dos braços em que o comando é `false`. Esses braços já estão cobertos por aquele recém-adicionado. Você pode remover essas quatro linhas com segurança. Você pretendia que este novo braço do interruptor substituisse essas condições.

Em seguida, você pode simplificar os quatro braços em que o comando é abrir o portão. Em ambos os casos em que o nível da água é alto, o portão pode ser aberto. (Em um deles, ele já está aberto.) Um caso em que o nível da água é baixo gera uma exceção e o outro não deve acontecer. Deve ser seguro lançar a mesma exceção se o bloqueio de água já estiver em um estado inválido. Você pode fazer as seguintes simplificações para esses braços:

```
C#
```

```
(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot
open high gate when the water is low"),
_ => throw new InvalidOperationException("Invalid internal state"),
```

Execute os testes novamente e eles serão aprovados. Esta é a versão final do método `SetHighGate`:

```
C#
```

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel)
    switch
    {
        (false, _, _)          => false,
        (true, _,     WaterLevel.High) => true,
        (true, false, WaterLevel.Low)  => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
        _                      => throw new
InvalidOperationException("Invalid internal state"),
    };
}
```

## Implementar padrões por conta própria

Agora que você já viu a técnica, preencha os métodos `SetLowGate` e `SetWaterLevel` por conta própria. Comece adicionando o seguinte código para testar operações inválidas nesses métodos:

C#

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water
is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower
gate is open.");
}
Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetHighGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.Low);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't lower water when the high
gate is open.");
}
Console.WriteLine($"Try to lower water with high gate open: {canalGate}");
```

Execute o aplicativo novamente. Você pode ver que os novos testes falham e o bloqueio do canal entra em um estado inválido. Tente implementar os métodos restantes por conta própria. O método para definir a porta inferior deve ser semelhante àquele para definir o portão superior. O método que altera o nível da água tem verificações diferentes, mas deve seguir uma estrutura semelhante. Você pode achar útil usar o mesmo processo para o método que define o nível da água. Comece com todas as quatro entradas: o estado de ambos os portões, o estado atual do nível da água e o novo nível de água solicitado. A expressão de comutador deve começar com:

C#

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
{
    // elided
};
```

Você terá 16 braços de comutador totais para preencher. Em seguida, teste e simplifique.

Você fez métodos como este?

C#

```
// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new
InvalidOperationException("Cannot lower water when the high gate is open"),
    };
}
```

```
(WaterLevel.High, WaterLevel.Low, true, false) => throw new  
InvalidOperationException("Cannot raise water when the low gate is open"),  
        _ => throw new InvalidOperationException("Invalid internal state"),  
    );  
}
```

Os testes devem ser aprovados e o bloqueio do canal deve operar com segurança.

## Resumo

Neste tutorial, você aprendeu a usar a correspondência de padrões para verificar o estado interno de um objeto antes de aplicar quaisquer alterações a ele. Você pode verificar combinações de propriedades. Depois de criar tabelas para qualquer uma dessas transições, você testa o código e, em seguida, simplifica a legibilidade e a manutenção. Essas refatorações iniciais podem sugerir refatorações adicionais que validam o estado interno ou gerenciam outras alterações de API. Este tutorial combinou classes e objetos com uma abordagem mais orientada a dados e baseada em padrões para implementar essas classes.

# Tutorial: atualizar interfaces com métodos de interface padrão

Artigo • 21/03/2023

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 vários clientes.

Neste tutorial, você aprenderá como:

- ✓ Estender interfaces com segurança, adicionando métodos com implementações.
- ✓ Criar implementações parametrizadas para oferecer maior flexibilidade.
- ✓ Permitir que os implementadores forneçam uma implementação mais específica na forma de uma substituição.

## Pré-requisitos

Você precisa configurar o computador para executar o .NET, incluindo o compilador C#. O compilador C# está disponível com o [Visual Studio 2022](#) ou o [SDK do .NET](#).

## Visão geral do cenário

Este tutorial começa com a versão 1 de uma biblioteca de relacionamento com o cliente. Você pode obter o aplicativo de iniciante em nosso [repositório de exemplos no GitHub](#). A empresa que criou essa biblioteca pretendia que clientes com aplicativos existentes adotassem a biblioteca. Eles forneciam definições de interface mínimas para os usuários da biblioteca implementarem. Aqui está a definição de interface para um cliente:

C#

```
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 representava um pedido:

C#

```
public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

Dessas interfaces, a equipe poderia criar uma biblioteca para os usuários criarem uma experiência melhor para os clientes. A meta era criar um relacionamento mais profundo com os clientes existentes e melhorar o relacionamento com clientes novos.

Agora é hora de atualizar a biblioteca para a próxima versão. Um dos recursos solicitados habilitará um desconto de fidelidade para os clientes que tiverem muitos pedidos. Esse novo desconto de fidelidade é aplicado sempre que um cliente faz um pedido. O desconto específico é uma propriedade de cada cliente. Cada implementação de `ICustomer` pode definir regras diferentes para o desconto de fidelidade.

A forma mais natural de adicionar essa funcionalidade é melhorar a interface `ICustomer` com um método para aplicar qualquer desconto de fidelidade. Essa sugestão de design causou preocupação entre desenvolvedores experientes: "As interfaces são imutáveis depois de serem lançadas. Não realize uma alteração interruptiva". *Implementações de interface padrão* para atualizar interfaces. Os autores de biblioteca podem adicionar novos membros à interface e fornecer uma implementação padrão para esses membros.

Implementações de interface padrão permitem que os desenvolvedores atualizem uma interface, permitindo que qualquer implementador substitua essa implementação. Os usuários da biblioteca podem aceitar a implementação padrão como uma alteração da falha. Se as regras de negócio forem diferentes, elas poderão substituir a implementação.

## Atualizar com métodos de interface padrão

A equipe concordou na implementação padrão mais provável: um desconto de fidelidade para os clientes.

A atualização deve fornecer a funcionalidade para definir duas propriedades: o número de pedidos necessários para ser qualificado para o desconto e o percentual de desconto. Esses recursos o tornam um cenário ideal para métodos de interface padrão. Você pode adicionar um método à interface `ICustomer` e fornecer a implementação

mais provável. Todas as implementações novas e existentes podem usar a implementação padrão ou fornecer as suas próprias.

Primeiro, adicione o novo método à interface, incluindo o corpo do método:

C#

```
// Version 1:  
public decimal ComputeLoyaltyDiscount()  
{  
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);  
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))  
    {  
        return 0.10m;  
    }  
    return 0;  
}
```

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

C#

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5,  
31))  
{  
    Reminders =  
    {  
        { new DateTime(2010, 08, 12), "child'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()}");
```

Observe a seguinte parte do teste:

C#

```
// Check the discount:  
ICustomer theCustomer = c;
```

```
Console.WriteLine($"Current discount:  
{theCustomer.ComputeLoyaltyDiscount()}");
```

Essa conversão de `SampleCustomer` em `ICustomer` é necessária. A classe `SampleCustomer` não precisa fornecer uma implementação de `ComputeLoyaltyDiscount`; isso é fornecido pela interface `ICustomer`. No entanto, a classe `SampleCustomer` não herda membros de suas interfaces. Essa regra não foi alterada. Para chamar qualquer método declarado e implementado na interface, a variável deve ser o tipo da interface: `ICustomer` neste exemplo.

## Fornecer parametrização

A implementação padrão é muito restritiva. Muitos consumidores desse sistema podem escolher limites diferentes para o número de compras, um período diferente de associação ou um percentual de desconto diferente. Você pode fornecer uma experiência de atualização melhor para mais clientes, fornecendo uma maneira de definir esses parâmetros. Vamos adicionar um método estático que defina esses três parâmetros, controlando a implementação padrão:

C#

```
// Version 2:  
public static void SetLoyaltyThresholds(  
    TimeSpan ago,  
    int minimumOrders = 10,  
    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 = 10;  
private static decimal discountPercent = 0.10m;  
  
public decimal ComputeLoyaltyDiscount()  
{  
    DateTime start = DateTime.Now - length;  
  
    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))  
    {  
        return discountPercent;  
    }  
    return 0;  
}
```

Há muitos novos recursos de linguagem mostrados naquele pequeno fragmento de código. Agora as interfaces podem incluir membros estáticos, incluindo campos e métodos. Modificadores de acesso diferentes também estão habilitados. Os outros campos são privados, mas o novo método é público. Qualquer dos modificadores são permitidos em membros de interface.

Aplicativos que usam a fórmula geral para calcular o desconto de fidelidade, mas que usam 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 um "agradecimento ao cliente" que recompensa qualquer cliente com mais de um mês de associação:

C#

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");
```

## Estender a implementação padrão

O código que você adicionou até agora forneceu uma implementação conveniente para esses cenários em que os usuários querem algo semelhante à implementação padrão ou para fornecer um conjunto de regras não relacionado. Como um último recurso, vamos refatorar o código um pouco para permitir cenários em que os usuários queiram criar com base na implementação padrão.

Considere uma startup que deseja atrair novos clientes. Eles oferecem um desconto de 50% no primeiro pedido de um novo cliente. Por outro lado, clientes existentes obtêm o desconto padrão. O autor da biblioteca precisa mover a implementação padrão para um método `protected static`, de modo que qualquer classe que implemente essa interface possa reutilizar o código em sua implementação. A implementação padrão do membro da interface também chama esse método compartilhado:

C#

```
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":

C#

```
public decimal ComputeLoyaltyDiscount()  
{  
    if (PreviousOrders.Any() == false)  
        return 0.50m;  
    else  
        return ICustomer.DefaultLoyaltyDiscount(this);  
}
```

Você pode ver todo o código concluído no nosso [repositório de amostras no GitHub](#). Você pode obter o aplicativo de iniciante em nosso [repositório de exemplos no GitHub](#).

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

# Tutorial: Funcionalidade mix ao criar classes usando interfaces com métodos de interface padrão

Artigo • 21/03/2023

Você pode definir uma implementação ao declarar um membro de uma interface. Essa funcionalidade fornece novos recursos em que você pode definir implementações padrão para recursos declarados em interfaces. As classes podem escolher quando substituir a funcionalidade, quando usar a funcionalidade padrão e quando não declarar suporte a recursos discretos.

Neste tutorial, você aprenderá como:

- ✓ Crie interfaces com implementações que descrevem recursos discretos.
- ✓ Crie classes que usam as implementações padrão.
- ✓ Crie classes que substituam algumas ou todas as implementações padrão.

## Pré-requisitos

Você precisa configurar o computador para executar o .NET, incluindo o compilador C#. O compilador C# está disponível com o [Visual Studio 2022](#) ou o [SDK do .NET](#).

## Limitações de métodos de extensão

Uma maneira de implementar o comportamento que aparece como parte de uma interface é definindo [métodos de extensão](#) que fornecem o comportamento padrão. As interfaces declaram um conjunto mínimo de membros, fornecendo uma área de superfície maior para qualquer classe que implemente essa interface. Por exemplo, os métodos de extensão em [Enumerable](#) fornecem a implementação para qualquer sequência ser a fonte de uma consulta LINQ.

Os métodos de extensão são resolvidos no tempo de compilação, usando o tipo declarado da variável. As classes que implementam a interface podem fornecer uma implementação melhor para qualquer método de extensão. As declarações de variável devem corresponder ao tipo de implementação para permitir que o compilador escolha essa implementação. Quando o tipo de tempo de compilação corresponder à interface, as chamadas de método são resolvidas para o método de extensão. Outra preocupação com os métodos de extensão é que eles podem ser acessados pelo mesmo local que a

classe contendo os métodos de extensão. As classes não poderão declarar se devem ou não fornecer recursos declarados nos métodos de extensão.

Você pode declarar as implementações padrão como métodos de interface. Em seguida, cada classe usa automaticamente a implementação padrão. Qualquer classe que possa fornecer uma implementação melhor pode substituir a definição do método de interface por um algoritmo melhor. De certa forma, essa técnica é semelhante à forma de utilização dos [métodos de extensão](#).

Neste artigo, você aprenderá como as implementações de interface padrão habilitam novos cenários.

## Criar o aplicativo

Considere um aplicativo de automação residencial. Você provavelmente tem muitos tipos de luzes e indicadores que podem ser usados em toda a casa. Cada luz deve dar suporte a APIs para ativá-las, desativá-las e relatar o estado atual. Algumas luzes e indicadores podem dar suporte a outros recursos, como:

- Ativar a luz e desligá-la após um temporizador.
- Piscar a luz por um período de tempo.

Alguns desses recursos estendidos podem ser emulados em dispositivos que dão suporte ao conjunto mínimo. Isso indica o fornecimento de uma implementação padrão. Para os dispositivos com mais funcionalidades internas, o software do dispositivo usaria os recursos nativos. Para outras luzes, eles podem optar por implementar a interface e usar a implementação padrão.

Os membros de interface padrão são uma solução melhor para esse cenário do que os métodos de extensão. Os autores de classe podem controlar quais interfaces eles escolhem implementar. Essas interfaces escolhidas estão disponíveis como métodos. Além disso, como os métodos de interface padrão são virtuais por padrão, a expedição de método sempre escolhe a implementação na classe.

Vamos criar o código para demonstrar essas diferenças.

## Criar interfaces

Comece criando a interface que define o comportamento de todas as luzes:

C#

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

Uma luminária suspensa básica pode implementar essa interface, conforme mostrado no seguinte código:

C#

```
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

Neste tutorial, o código não conduz os dispositivos IoT, mas emula essas atividades gravando mensagens no console. É possível explorar o código sem automatizar sua casa.

Em seguida, vamos definir a interface para uma luz que pode ser desativada automaticamente após um tempo limite:

C#

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

Você pode adicionar uma implementação básica à luz suspensa, mas uma solução melhor é modificar essa definição de interface para fornecer uma implementação padrão `virtual`:

C#

```
public interface ITimerLight : ILight
{
    public async Task TurnOnFor(int duration)
    {
```

```
        Console.WriteLine("Using the default interface method for the  
ITimerLight.TurnOnFor.");  
        SwitchOn();  
        await Task.Delay(duration);  
        SwitchOff();  
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");  
    }  
}
```

A classe `OverheadLight` pode implementar a função de temporizador declarando suporte à interface:

C#

```
public class OverheadLight : ITimerLight { }
```

Um tipo de luz diferente pode dar suporte a um protocolo mais sofisticado. Ele pode fornecer sua própria implementação para `TurnOnFor`, conforme mostrado no seguinte código:

C#

```
public class HalogenLight : ITimerLight  
{  
    private enum HalogenLightState  
    {  
        Off,  
        On,  
        TimerModeOn  
    }  
  
    private HalogenLightState state;  
    public void SwitchOn() => state = HalogenLightState.On;  
    public void SwitchOff() => state = HalogenLightState.Off;  
    public bool IsOn() => state != HalogenLightState.Off;  
    public async Task TurnOnFor(int duration)  
    {  
        Console.WriteLine("Halogen light starting timer function.");  
        state = HalogenLightState.TimerModeOn;  
        await Task.Delay(duration);  
        state = HalogenLightState.Off;  
        Console.WriteLine("Halogen light finished custom timer function");  
    }  
  
    public override string ToString() => $"The light is {state}";  
}
```

Ao contrário da substituição de métodos de classe virtual, a declaração de `TurnOnFor` na classe `HalogenLight` não usa a palavra-chave `override`.

# Recursos combinados

As vantagens dos métodos de interface padrão ficam mais claras à medida que você introduz recursos mais avançados. O uso de interfaces permite combinar recursos.

Também permite que cada autor de classe escolha entre uma implementação padrão ou uma implementação personalizada. Vamos adicionar uma interface com uma implementação padrão para uma luz piscando:

C#

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for
IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
        {
            SwitchOn();
            await Task.Delay(duration);
            SwitchOff();
            await Task.Delay(duration);
        }
        Console.WriteLine("Done with the default interface method for
IBlinkingLight.Blink.");
    }
}
```

A implementação padrão permite que qualquer luz pisque. A luz suspensa pode adicionar recursos de temporizador e piscar usando a implementação padrão:

C#

```
public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

Um novo tipo de luz `LEDLight`, dá suporte à função de temporizador e à função de piscar diretamente. Esse estilo de luz implementa as interfaces `ITimerLight` e `IBlinkingLight` e substitui o método `Blink`:

C#

```
public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink function.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

Um `ExtraFancyLight` pode dar suporte diretamente a funções de piscar e temporizador:

C#

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink
function.");
    }
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer
function");
    }

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

O `HalogenLight` criado anteriormente não dá suporte a piscar. Portanto, não adicione `IBlinkingLight` à lista de interfaces com suporte.

# Detectar os tipos de luz usando a correspondência de padrões

Em seguida, vamos gravar um código de teste. Você pode usar o recurso de [correspondência de padrões](#) do C# para determinar as funcionalidades de uma luz examinando quais interfaces ele dá suporte. O método a seguir exercita as funcionalidades com suporte de cada luz:

C#

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"\\tAfter switching on, the light is {(light.IsOn() ? "on" : "off")}");
    light.SwitchOff();
    Console.WriteLine($"\\tAfter switching off, the light is {(light.IsOn() ? "on" : "off")}");

    if (light is ITimerLight timer)
    {
        Console.WriteLine("\\tTesting timer function");
        await timer.TurnOnFor(1000);
        Console.WriteLine("\\tTimer function completed");
    }
    else
    {
        Console.WriteLine("\\tTimer function not supported.");
    }

    if (light is IBlinkingLight blinker)
    {
        Console.WriteLine("\\tTesting blinking function");
        await blinker.Blink(500, 5);
        Console.WriteLine("\\tBlink function completed");
    }
    else
    {
        Console.WriteLine("\\tBlink function not supported.");
    }
}
```

O código a seguir no método `Main` cria cada tipo de luz na sequência e testa essa luz:

C#

```
static async Task Main(string[] args)
{
```

```

Console.WriteLine("Testing the overhead light");
var overhead = new OverheadLight();
await TestLightCapabilities(overhead);
Console.WriteLine();

Console.WriteLine("Testing the halogen light");
var halogen = new HalogenLight();
await TestLightCapabilities(halogen);
Console.WriteLine();

Console.WriteLine("Testing the LED light");
var led = new LEDLight();
await TestLightCapabilities(led);
Console.WriteLine();

Console.WriteLine("Testing the fancy light");
var fancy = new ExtraFancyLight();
await TestLightCapabilities(fancy);
Console.WriteLine();
}

```

## Como o compilador determina a melhor implementação

Esse cenário mostra uma interface base sem implementações. A adição de um método à interface `ILight`, introduz novas complexidades. As regras de linguagem que regem os métodos de interface padrão minimizam o efeito nas classes concretas que implementam várias interfaces derivadas. Vamos aprimorar a interface original com um novo método para demonstrar como isso altera seu uso. Cada luz do indicador pode relatar seu status de energia como um valor enumerado:

C#

```

public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}

```

A implementação padrão pressupõe que não há energia:

C#

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

Essas alterações são compiladas de forma limpa, embora `ExtraFancyLight` declare suporte para a interface `ILight` e as interfaces derivadas `ITimerLight` e `IBlinkingLight`. Há apenas uma implementação "mais próxima" declarada na interface `ILight`. Qualquer classe que declarasse uma substituição se tornaria a única implementação "mais próxima". Você viu exemplos nas classes anteriores que substituíram os membros de outras interfaces derivadas.

Evite substituir o mesmo método em várias interfaces derivadas. Isso cria uma chamada de método ambíguo sempre que uma classe implementa ambas as interfaces derivadas. O compilador não pode escolher um único método, logo, um erro é gerado. Por exemplo, se `IBlinkingLight` e `ITimerLight` implementaram uma substituição de `PowerStatus`, `OverheadLight` precisaria fornecer uma substituição mais específica. Caso contrário, o compilador não poderá escolher entre as implementações nas duas interfaces derivadas. Normalmente, você pode evitar essa situação mantendo as definições de interface pequenas e focadas em um recurso. Nesse cenário, cada funcionalidade de uma luz é a própria interface, e várias interfaces são herdadas apenas por classes.

Este exemplo mostra um cenário em que você pode definir recursos discretos que podem ser misturados em classes. Você declara qualquer conjunto de funcionalidades com suporte declarando a quais interfaces uma classe dá suporte. O uso de métodos de interface padrão virtual permite que as classes usem ou definam uma implementação diferente para um ou todos os métodos de interface. Essa funcionalidade de linguagem fornece novas maneiras de modelar os sistemas do mundo real sendo criados. Os métodos de interface padrão fornecem uma maneira mais clara de expressar classes relacionadas que podem combinar diferentes recursos usando implementações virtuais deles.

# Índices e intervalos

Artigo • 09/05/2023

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

Neste tutorial, você aprenderá como:

- ✓ Use a sintaxe para intervalos em uma sequência.
- ✓ Defina implicitamente uma classe [Range](#).
- ✓ Compreenda as decisões de design para o início e o fim de cada sequência.
- ✓ Conheça cenários para os tipos [Index](#) e [Range](#).

## Suporte a idioma para intervalos e índices

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

Este suporte à linguagem depende de dois tipos novos e dois operadores novos:

- [System.Index](#) representa um índice em uma sequência.
- O índice do operador final `^`, que especifica que um índice é relativo ao final da sequência.
- [System.Range](#) representa um subintervalo de uma sequência.
- O operador de intervalo `..`, que especifica o início e o final de um intervalo como seus operandos.

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

C#

```
string[] words = new string[]
{
    // index from start      index from end
    "The",      // 0            ^9
    "quick",    // 1            ^8
    "brown",    // 2            ^7
    "fox",      // 3            ^6
    "jumps",    // 4            ^5
    "over",     // 5            ^4
    "the",      // 6            ^3
```

```
"lazy",      // 7          ^2
"dog",       // 8          ^1
};           // 9 (or words.Length) ^0
```

Você pode recuperar a última palavra com o índice `^1`. Adicione o código a seguir abaixo da inicialização:

C#

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

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

O código a seguir cria um subintervalo com as palavras "quick", "brown" e "fox". Ele inclui `words[1]` até `words[3]`. O elemento `words[4]` não está no intervalo.

C#

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.Write($"< {word} >");
Console.WriteLine();
```

O código a seguir retorna o intervalo com "lazy" e "dog". Ele inclui `words[^2]` e `words[^1]`. O índice final `words[^0]` não está incluído. Adicione o seguinte código também:

C#

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.Write($"< {word} >");
Console.WriteLine();
```

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

C#

```
string[] allWords = words[..]; // contains "The" through "dog".
string[] firstPhrase = words[..4]; // contains "The" through "fox"
string[] lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
foreach (var word in allWords)
```

```

        Console.WriteLine($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
foreach (var word in lastPhrase)
    Console.WriteLine($"< {word} >");
Console.WriteLine();

```

Você também pode declarar intervalos ou índices como variáveis. A variável então pode ser usada dentro dos caracteres [ e ]:

C#

```

Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
string[] text = words[phrase];
foreach (var word in text)
    Console.WriteLine($"< {word} >");
Console.WriteLine();

```

O exemplo a seguir mostra muitos dos motivos para essas escolhas. Modifique x, y e z para tentar combinações diferentes. Quando você testar, use valores em que x é menor que y e y é menor que z para as combinações válidas. Adicione o seguinte código a um novo método. Tente usar combinações diferentes:

C#

```

int[] numbers = Enumerable.Range(0, 100).ToArray();
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length - x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\\tnumbers[x..y] is {x_y[0]} through {x_y[^1]},\nnumbers[y..z] is {y_z[0]} through {y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\\tnumbers[x..^x] starts with {x_x[0]} and ends with\n{x_x[^1]}");

```

```

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means
numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\\t{start_x[0]}..{start_x[^1]} is the same as
{zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..
{z_zero[^1]}");

```

Não apenas matrizes dão suporte a índices e intervalos. Você também pode usar índices e intervalos com [cadeia de caracteres](#), [Span<T>](#) ou [ReadOnlySpan<T>](#).

## Conversões de expressão de operador de intervalo implícito

Ao usar a sintaxe de expressão do operador de intervalo, o compilador converte implicitamente os valores inicial e final em um [Index](#) e a partir deles cria uma instância [Range](#). O código a seguir mostra um exemplo de conversão implícita da sintaxe de expressão do operador de intervalo e sua alternativa explícita correspondente:

C#

```

Range implicitRange = 3..^5;

Range explicitRange = new(
    start: new Index(value: 3, fromEnd: false),
    end: new Index(value: 5, fromEnd: true));

if (implicitRange.Equals(explicitRange))
{
    Console.WriteLine(
        $"The implicit range '{implicitRange}' equals the explicit range
'{explicitRange}'");
}
// Sample output:
//      The implicit range '3..^5' equals the explicit range '3..^5'

```

### ⓘ Importante

Conversões implícitas de [Int32](#) para [Index](#) lançam uma [ArgumentOutOfRangeException](#) quando o valor é negativo. Da mesma forma, o construtor [Index](#) lança uma [ArgumentOutOfRangeException](#) quando o parâmetro [value](#) é negativo.

# Suporte de tipo para índices e intervalos

Índices e intervalos fornecem sintaxe clara e concisa para acessar um único elemento ou um intervalo de elementos em uma sequência. Uma expressão de índice normalmente retorna o tipo dos elementos de uma sequência. Uma expressão de intervalo normalmente retorna o mesmo tipo de sequência que a sequência de origem.

Qualquer tipo que forneça um [indexador](#) com um parâmetro [Index](#) ou [Range](#) dá suporte explicitamente a índices ou intervalos, respectivamente. Um indexador que usa um único parâmetro [Range](#) pode retornar um tipo de sequência diferente, como [System.Span<T>](#).

## ⓘ Importante

O desempenho do código usando o operador de intervalo depende do tipo do operando da sequência.

A complexidade temporal do operador de intervalo depende do tipo de sequência. Por exemplo, se a sequência for uma [string](#) ou uma matriz, o resultado será uma cópia da seção especificada da entrada. Portanto, a complexidade de tempo será  $O(N)$  (em que  $N$  é o comprimento do intervalo). Por outro lado, se for um [System.Span<T>](#) ou um [System.Memory<T>](#), o resultado fará referência ao mesmo repositório de backup, o que significa que não há cópia e a operação é  $O(1)$ .

Além da complexidade de tempo, isso causa alocações e cópias extras, afetando o desempenho. No código sensível ao desempenho, considere usar [Span<T>](#) ou [Memory<T>](#) como o tipo de sequência, já que o operador de intervalo não aloca para eles.

Um tipo é [contável](#) se tiver uma propriedade nomeada [Length](#) ou [Count](#) com um getter acessível e um tipo de retorno [int](#). Um tipo contável que não dá suporte explicitamente a índices ou intervalos pode fornecer um suporte implícito para eles. Para obter mais informações, consulte as seções de [suporte a índice implícito](#) e [suporte a intervalo implícito](#) da [nota de proposta de recurso](#). Intervalos que usam suporte de intervalo implícito retornam o mesmo tipo de sequência da sequência de origem.

Por exemplo, os seguintes tipos .NET dão suporte a índices e intervalos: [String](#), [Span<T>](#) e [ReadOnlySpan<T>](#). O [List<T>](#) suporta índices, mas não dá suporte a intervalos.

[Array](#) tem um comportamento com mais nuances. Matrizes de dimensão única dão suporte a índices e intervalos. Matrizes multidimensionais não dão suporte a

indexadores ou intervalos. O indexador de uma matriz multidimensional tem vários parâmetros, não um único parâmetro. Matrizes denteadas, também conhecidas como uma matriz de matrizes, dão suporte a intervalos e indexadores. O exemplo a seguir mostra como iterar uma subseção retangular de uma matriz denteada. É iterada a seção no centro, excluindo-se as três primeiras e últimas linhas e as duas primeiras e últimas colunas de cada linha selecionada:

C#

```
var jagged = new int[10][]
{
    new int[10] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
    new int[10] { 10,11,12,13,14,15,16,17,18,19 },
    new int[10] { 20,21,22,23,24,25,26,27,28,29 },
    new int[10] { 30,31,32,33,34,35,36,37,38,39 },
    new int[10] { 40,41,42,43,44,45,46,47,48,49 },
    new int[10] { 50,51,52,53,54,55,56,57,58,59 },
    new int[10] { 60,61,62,63,64,65,66,67,68,69 },
    new int[10] { 70,71,72,73,74,75,76,77,78,79 },
    new int[10] { 80,81,82,83,84,85,86,87,88,89 },
    new int[10] { 90,91,92,93,94,95,96,97,98,99 },
};

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
    Console.WriteLine();
}
```

Em todos os casos, o operador de intervalo para [Array](#) aloca uma matriz para armazenar os elementos retornados.

## Cenários para intervalos e índices

Geralmente, você usará intervalos e índices quando quiser analisar uma parte de uma sequência maior. A nova sintaxe é mais clara porque lê exatamente qual parte da sequência está envolvida. A função local `MovingAverage` usa um `Range` como seu argumento. O método então enumera apenas esse intervalo ao calcular o mínimo, máximo e média. Experimente o seguinte código em seu projeto:

C#

```

int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}: \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^start..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}: \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range
range) =>
(
    subSequence[range].Min(),
    subSequence[range].Max(),
    subSequence[range].Average()
);

int[] Sequence(int count) =>
    Enumerable.Range(0, count).Select(x => (int)(Math.Sqrt(x) *
100)).ToArray();

```

## Uma observação sobre índices e matrizes de intervalo

Ao tirar um intervalo de uma matriz, o resultado é uma matriz copiada da matriz inicial, em vez de referenciada. Modificar valores na matriz resultante não altera os valores na matriz inicial.

Por exemplo:

C#

```

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

var firstThreeItems = arrayOfFiveItems[..3]; // contains 1,2,3
firstThreeItems[0] = 11; // now contains 11,2,3

Console.WriteLine(string.Join(", ", firstThreeItems));
Console.WriteLine(string.Join(", ", arrayOfFiveItems));

```

```
// output:  
// 11,2,3  
// 1,2,3,4,5
```

# Tutorial: Expressar sua intenção de design mais claramente com tipos de referência que permitem valor nulo e tipos que não permitem valor nulo

Artigo • 10/05/2023

Tipos de referência que permitem valor nulo complementam os tipos de referência da mesma maneira que os tipos de valor que permitem valor nulo complementam os tipos de valor. Para declarar que uma variável é um **tipo de referência que permite valor nulo**, anexe um `?`  ao tipo. Por exemplo, `string?` representa uma `string` que permite valor nulo. Você pode usar esses novos tipos para expressar mais claramente sua intenção de design: algumas variáveis *devem sempre ter um valor*, outras *podem ter um valor ausente*.

Neste tutorial, você aprenderá como:

- ✓ Incorporar tipos de referência que permitem valores nulos e tipos de referência que não permitem valores nulos aos designs
- ✓ Habilitar verificações de tipo de referência que permitem valor nulo em todo o código.
- ✓ Gravar código em locais onde o compilador imponha essas decisões de design.
- ✓ Usar o recurso de referência que permite valor nulo em seus próprios designs

## Pré-requisitos

Você precisará configurar o computador para executar o .NET, incluindo o compilador C#. O compilador C# está disponível com o [Visual Studio 2022](#) ou o [SDK do .NET](#).

Este tutorial pressupõe que você esteja familiarizado com o C# e .NET, incluindo o Visual Studio ou a CLI do .NET.

## Incorporar tipos de referência que permitem valor nulo aos designs

Neste tutorial, você criará uma biblioteca para modelar a executar uma pesquisa. O código usa tipos de referência que permitem valores nulos e tipos de referência que não permitem valores nulos para representar os conceitos do mundo real. As perguntas da

pesquisa nunca podem ser um valor nulo. Um entrevistado pode optar por não responder a uma pergunta. As respostas podem ser `null` nesse caso.

O código que você gravará para este exemplo expressa essa intenção e o compilador a aplica.

## Criar o aplicativo e habilitar os tipos de referência que permitem valor nulo

Crie um novo aplicativo de console no Visual Studio ou na linha de comando usando `dotnet new console`. Dê o nome `NullableIntroduction` ao aplicativo. Depois de criar o aplicativo, você precisará especificar que todo o projeto é compilado em um **contexto de anotação anulável** habilitado. Abra o arquivo `.csproj` e adicione um elemento `Nullable` ao elemento `PropertyGroup`. Defina seu valor como `enable`. Você precisa aceitar o recurso de **tipos de referência que permitem valor nulo** em projetos anteriores ao C# 11. Isso porque, quando o recurso é ativado, as declarações de variáveis de referência existentes tornam-se **tipos de referência que não permitem valor nulo**. Embora essa decisão auxilie na localização de problemas em que o código existente pode não ter verificações de valores nulos adequadas, ela pode não refletir com precisão a intenção original do design:

XML

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

Antes do .NET 6, novos projetos não incluem o elemento `Nullable`. Do .NET 6 em diante, os novos projetos incluem o elemento `<Nullable>enable</Nullable>` no arquivo de projeto.

## Criar os tipos para o aplicativo

Este aplicativo de pesquisa requer a criação de várias classes:

- Uma classe que modela a lista de perguntas.
- Uma classe que modela uma lista de pessoas contatadas para a pesquisa.
- Uma classe que modela as respostas de uma pessoa que participou da pesquisa.

Esses tipos usarão os tipos de referência que permitem valor nulo e tipos de referência que não permitem valor nulo para expressar quais membros são obrigatórios e quais são opcionais. Os tipos de referência que permitem valor nulo informam claramente essa intenção de design:

- As perguntas que fazem parte da pesquisa nunca podem ser valores nulos: não faz sentido fazer uma pergunta vazia.
- Os entrevistados nunca poderão ser nulos. Convém controlar as pessoas contatadas, mesmo os entrevistados que se recusaram a participar.
- Qualquer resposta a uma pergunta pode ser um valor nulo. Os entrevistados podem se recusar a responder a algumas ou a todas as perguntas.

Se já tiver programado em C#, pode estar tão acostumado a fazer referência a tipos que permitem valores `null` que poderá ter perdido outras oportunidades de declarar instâncias não anuláveis:

- O conjunto de perguntas não deve permitir um valor nulo.
- O conjunto de entrevistados não deve permitir um valor nulo.

Ao escrever o código, você verá que um tipo de referência não anulável como o padrão para referências evita erros comuns que poderiam levar a `NullReferenceExceptions`. Uma das lições retirada deste tutorial é que você tomou decisões sobre quais variáveis poderiam ou não ser `null`. O idioma não forneceu sintaxe para expressar essas decisões. Agora ele já fornece.

O aplicativo que você criará executa as seguintes etapas:

1. Cria uma pesquisa e adiciona perguntas a ela.
2. Cria um conjunto pseudo aleatório de entrevistados para a pesquisa.
3. Entre em contato com os entrevistados até que o tamanho da pesquisa preenchida atinja o número da meta.
4. Grava estatísticas importantes nas respostas da pesquisa.

## Criar a pesquisa com tipos de referência anuláveis e não anuláveis

O primeiro código gravado criará a pesquisa. Você escreverá classes para modelar uma pergunta da pesquisa e uma execução da pesquisa. A pesquisa tem três tipos de perguntas, diferenciadas pelo formato da resposta: respostas do tipo Sim/Não, respostas com números e respostas com texto. Crie uma classe `public SurveyQuestion`:

C#

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
```

```
    }  
}
```

O compilador interpreta cada declaração de variável de tipo de referência como um tipo de referência **não anulável** para o código em um contexto de anotação anulável habilitado. Para ver seu primeiro aviso, adicione propriedades ao texto da pergunta e tipo de pergunta, conforme mostrado no código a seguir:

C#

```
namespace NullableIntroduction  
{  
    public enum QuestionType  
    {  
        YesNo,  
        Number,  
        Text  
    }  
  
    public class SurveyQuestion  
    {  
        public string QuestionText { get; }  
        public QuestionType TypeOfQuestion { get; }  
    }  
}
```

Como você não inicializou `QuestionText`, o compilador emitirá um aviso informando que uma propriedade que não permite valor nulo não foi inicializada. Seu design exige que o texto da pergunta não seja um valor nulo, portanto, você inclui um construtor para inicializá-lo e o valor `QuestionType` também. A definição da classe concluída se parece com o código a seguir:

C#

```
namespace NullableIntroduction;  
  
public enum QuestionType  
{  
    YesNo,  
    Number,  
    Text  
}  
  
public class SurveyQuestion  
{  
    public string QuestionText { get; }  
    public QuestionType TypeOfQuestion { get; }  
  
    public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
```

```
        (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);  
    }
```

A adição do construtor removerá o aviso. O argumento do construtor também é um tipo de referência que não permite valor nulo, portanto, o compilador não emite avisos.

Em seguida, crie uma classe `public` chamada `SurveyRun`. Esta classe contém uma lista de métodos e objetos `SurveyQuestion` para adicionar perguntas à pesquisa, conforme mostrado no código a seguir:

C#

```
using System.Collections.Generic;  
  
namespace NullableIntroduction  
{  
    public class SurveyRun  
    {  
        private List<SurveyQuestion> surveyQuestions = new  
List<SurveyQuestion>();  
  
        public void AddQuestion(QuestionType type, string question) =>  
            AddQuestion(new SurveyQuestion(type, question));  
        public void AddQuestion(SurveyQuestion surveyQuestion) =>  
            surveyQuestions.Add(surveyQuestion);  
    }  
}
```

Como foi feito anteriormente, você deve inicializar o objeto de lista com um valor não nulo ou o compilador emitirá um aviso. Não há verificações de valores nulos na segunda sobrecarga de `AddQuestion`, pois elas são desnecessárias: você declarou que a variável não permite valor nulo. Seu valor não pode ser `null`.

Alterne para `Program.cs` em seu editor e substitua o conteúdo de `Main` pelas seguintes linhas de código:

C#

```
var surveyRun = new SurveyRun();  
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a  
NullReferenceException?");  
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many  
times (to the nearest 100) has that happened?"));  
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Como o projeto inteiro está em um contexto de anotação anulável habilitado, você receberá avisos quando passar `null` para qualquer método que espera um tipo de

referência não anulável. Experimente adicionar a seguinte linha a `Main`:

```
C#
```

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

## Criar entrevistados e obter respostas para a pesquisa

Em seguida, grave o código que gerará respostas para a pesquisa. Esse processo envolve várias tarefas pequenas:

1. Criar um método para gerar objetos dos entrevistados. Eles representam pessoas solicitadas a preencher a pesquisa.
2. Criar lógica para simular a realização de perguntas para um pesquisado e coletar respostas ou perceber que um pesquisado não respondeu.
3. Repetir até que entrevistados suficientes tenham respondido à pesquisa.

Será necessária uma classe para representar uma resposta da pesquisa. Adicione-a agora. Habilitar o suporte para tipos que permitem valor nulo. Adicione uma propriedade `Id` e um construtor para inicializá-la, conforme mostrado no código a seguir:

```
C#
```

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

Em seguida, adicione um método `static` para criar novos participantes ao gerar uma ID aleatória:

```
C#
```

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new
SurveyResponse(randomGenerator.Next());
```

A principal responsabilidade dessa classe é gerar as respostas de um participante para as perguntas da pesquisa. Essa responsabilidade conta com algumas etapas:

1. Peça para participar da pesquisa. Se a pessoa não consentir, retorne uma resposta de ausente (ou de valor nulo).
  2. Faça as perguntas e registre a resposta. As respostas também pode ser ausentes (ou de valor nulo).

Adicione o seguinte código à classe SurveyResponse:

C#

```

        case 2:
            return "Green";
        case 3:
            return "Blue";
    }
    return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
}
}

```

O armazenamento das respostas da pesquisa é um `Dictionary<int, string>?`, indicando que ele pode ser um valor nulo. Você está usando o novo recurso de idioma para declarar sua intenção de design, tanto para o compilador quanto para qualquer pessoa que leia seu código posteriormente. Se, em algum momento, você desreferenciar `surveyResponses` sem primeiro verificar o valor de `null`, será gerado um aviso do compilador. Você não receberá um aviso no método `AnswerSurvey` porque o compilador pode determinar que a variável `surveyResponses` foi definida como um valor não nulo acima.

O uso de `null` para respostas ausentes destaca um ponto importante para trabalhar com tipos de referência anuláveis: seu objetivo não é remover todos os valores `null` de seu programa. Em vez disso, sua meta é garantir que o código escrito expresse a intenção do seu design. Os valores ausentes representam um conceito que precisa ser expresso em seu código. O valor `null` é uma forma clara de expressar esses valores ausentes. Tentar remover todos os valores `null` leva somente à definição de alguma outra maneira de expressar esses valores ausentes sem `null`.

Em seguida, é necessário gravar o método `PerformSurvey` na classe `SurveyRun`. Adicione o seguinte código à classe `SurveyRun`:

C#

```

private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberofRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberofRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}

```

Aqui, novamente, sua opção por uma `List<SurveyResponse>?` que permite valor nulo indica que a resposta pode ser um valor nulo. Isso indica que a pesquisa ainda não foi entregue a nenhum pesquisado. Observe que os entrevistados são adicionados até que um suficiente de pessoas tiver consentido.

A última etapa para executar a pesquisa é adicionar uma chamada para executar a pesquisa no final do método `Main`:

```
C#
```

```
surveyRun.PerformSurvey(50);
```

## Examinar as respostas da pesquisa

A última etapa é exibir os resultados da pesquisa. Você adicionará código a várias classes gravadas. Este código demonstra o valor da distinção dos tipos de referência que permitem valor nulo e tipos de referência que não permitem valor nulo. Comece adicionando os dois membros com corpo de expressão à classe `SurveyResponse`:

```
C#
```

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index)
?? "No answer";
```

Como `surveyResponses` é um tipo de referência anulável, verificações de nulo são necessárias antes de desreferenciá-la. O método `Answer` retorna uma cadeia de caracteres não anulável, portanto, precisamos cobrir o caso de ausência de resposta usando o operador de avaliação de nulo.

Em seguida, adicione esses três membros com corpo de expressão à classe `SurveyRun`:

```
C#
```

```
public IEnumerable<SurveyResponse> AllParticipants => (respondents ??
Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

O membro `AllParticipants` deve levar em conta que a variável `respondents` pode ser um valor nulo, mas o valor de retorno não pode ser nulo. Se você alterar essa expressão removendo `??` e a sequência vazia que se segue, o compilador avisará que o método

poderá retornar `null` e sua assinatura de retorno retornará um tipo que não permite valor nulo.

Por fim, adicione o seguinte loop à parte inferior do método `Main`:

```
C#
```

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"{surveyRun.GetQuestion(i).QuestionText} :
{answer}");
        }
    }
    else
    {
        Console.WriteLine("\tNo responses");
    }
}
```

Você não precisa de verificações de `null` neste código porque criou as interfaces subjacentes para que todas elas retornem tipos de referência que não permitem valor nulo.

## Obter o código

Obtenha o código do tutorial concluído em nosso repositório de [amostras](#) na pasta `csharp/NullableIntroduction`.

Experimente alterar as declarações de tipo entre os tipos de referência que permitem valor nulo e tipos de referência que não permitem valor nulo. Veja como isso gera avisos diferentes para garantir que um `null` não será acidentalmente cancelado.

## Próximas etapas

Saiba como usar o tipo de referência anulável ao trabalhar com o Entity Framework:

[Conceitos Básicos do Entity Framework Core: trabalhando com tipos de referência anuláveis](#)

# Tutorial: gerar e consumir fluxos assíncronos usando o C# e .NET

Artigo • 28/03/2023

Os **fluxos assíncronos** modelam uma fonte de streaming de dados. Os fluxos de dados geralmente recuperam ou geram elementos de forma assíncrona. Eles fornecem um modelo de programação natural para fontes de dados de streaming assíncronas.

Neste tutorial, você aprenderá como:

- ✓ Criar uma fonte de dados que gera uma sequência de elementos de dados de forma assíncrona.
- ✓ Consumir essa fonte de dados de forma assíncrona.
- ✓ Suporte ao cancelamento e contextos capturados para fluxos assíncronos.
- ✓ Reconhecer quando a nova interface e a fonte de dados forem preferenciais para sequências anteriores de dados síncronos.

## Pré-requisitos

Você precisa configurar o computador para executar o .NET, incluindo o compilador C#. O compilador C# está disponível com o [Visual Studio 2022](#) ou o [SDK do .NET](#).

Você precisará criar um [token de acesso do GitHub](#) para poder acessar o ponto de extremidade GitHub GraphQL. Selecione as seguintes permissões para o Token de acesso do GitHub:

- repo:status
- public\_repo

Salve o token de acesso em um local seguro, de modo que possa usá-lo para acessar o ponto de extremidade da API do GitHub.

### Aviso

Mantenha seu token de acesso pessoal protegido. Qualquer software com seu token de acesso pessoal pode fazer chamadas da API do GitHub usando seus direitos de acesso.

Este tutorial pressupõe que você esteja familiarizado com o C# e .NET, incluindo o Visual Studio ou a CLI do .NET.

# Executar o aplicativo inicial

Você pode obter o código para o aplicativo inicial usado neste tutorial em nosso repositório [dotnet/docs](#) na pasta [asynchronous-programming/snippets](#).

O aplicativo inicial é um aplicativo de console que usa a interface [GitHub GraphQL](#) para recuperar os problemas recentes gravados no repositório [dotnet/docs](#). Comece observando o código a seguir para o método `Main` do aplicativo inicial:

C#

```
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-
    //for-the-command-line/#creating-a-token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub
    access token.

    var key = GetEnvVariable("GitHubKey",
        "You must store your GitHub key in the 'GitHubKey' environment
        variable",
        "");

    var client = new GitHubClient(new
Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new
    CancellationTokenSource();

    try
    {
        var results = await RunPagedQueryAsync(client, PagedIssueQuery,
    "docs",
        cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
```

```
    }  
}
```

Você pode definir uma variável de ambiente `GitHubKey` para o token de acesso pessoal ou pode substituir o último argumento na chamada para `GetEnvVariable` com seu token de acesso pessoal. Não coloque seu código de acesso no código-fonte se você estiver compartilhando a origem com outras pessoas. Nunca carregue códigos de acesso em um repositório de origem compartilhado.

Após criar o cliente do GitHub, o código em `Main` criará um objeto de relatório de andamento e um token de cancelamento. Depois que esses objetos forem criados, `Main` chamará `RunPagedQueryAsync` para recuperar os 250 problemas mais recente criados. Depois que a tarefa for concluída, os resultados serão exibidos.

Ao executar o aplicativo inicial, você poderá realizar algumas observações importantes sobre como esse aplicativo é executado. Você verá o progresso informado para cada página retornada do GitHub. É possível observar uma pausa perceptível antes do GitHub retornar cada nova página de problemas. Por fim, os problemas só serão exibidos depois que todas as 10 páginas forem recuperadas do GitHub.

## Examinar a implementação

A implementação revela por que você observou o comportamento discutido na seção anterior. Examine o código para `RunPagedQueryAsync`:

C#

```
private static async Task<JArray> RunPagedQueryAsync(GitHubClient client,  
string queryText, string repoName, CancellationToken cancel, IProgress<int>  
progress)  
{  
    var issueAndPRQuery = new GraphQLRequest  
    {  
        Query = queryText  
    };  
    issueAndPRQuery.Variables["repo_name"] = repoName;  
  
    JArray finalResults = new JArray();  
    bool hasMorePages = true;  
    int pagesReturned = 0;  
    int issuesReturned = 0;  
  
    // Stop with 10 pages, because these are large repos:  
    while (hasMorePages && (pagesReturned++ < 10))  
    {  
        var postBody = issueAndPRQuery.ToJsonText();
```

```

        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)[ "totalCount" ]!;
hasMorePages = (bool)pageInfo(results)[ "hasPreviousPage" ]!;
issueAndPRQuery.Variables[ "start_cursor" ] = pageInfo(results)
[ "startCursor" ]!.ToString();
issuesReturned += issues(results)[ "nodes" ]!.Count();
finalResults.Merge(issues(results)[ "nodes" ]!");
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
}
return finalResults;

JObject issues(JObject result) => (JObject)result[ "data" ]!
[ "repository" ]![ "issues" ]!;
JObject pageInfo(JObject result) => (JObject)issues(result)
[ "pageInfo" ]!;
}

```

Vamos nos concentrar no algoritmo de paginação e na estrutura assíncrona do código anterior. (Você pode consultar a [documentação do GraphQL do GitHub](#) para obter detalhes sobre a API do GraphQL do GitHub). O método `RunPagedQueryAsync` enumera os problemas dos mais recentes aos mais antigos. Ele solicita 25 problemas por página e examina a estrutura `pageInfo` da resposta para continuar com a página anterior. Isso segue o suporte de paginação padrão do GraphQL para respostas com várias páginas. A resposta inclui um objeto `pageInfo` que inclui um valor `hasPreviousPages` e um valor `startCursor` usados para solicitar a página anterior. Os problemas estão na matriz `nodes`. O método `RunPagedQueryAsync` anexa esses nós em uma matriz que contém todos os resultados de todas as páginas.

Após a recuperação e a restauração de uma página de resultados, `RunPagedQueryAsync` informa o andamento e verifica o cancelamento. Se o cancelamento tiver sido solicitado, `RunPagedQueryAsync` gerará um [OperationCanceledException](#).

Há vários elementos nesse código que podem ser melhorados. Acima de tudo, `RunPagedQueryAsync` deve alojar armazenamento para todos os problemas retornados. Este exemplo é interrompido em 250 problemas porque recuperar todos os problemas exigiria muito mais memória para armazenar todos os problemas recuperados. Os protocolos para dar suporte a relatórios de progresso e cancelamento dificultam o entendimento do algoritmo em sua primeira leitura. Mais tipos e APIs estão envolvidos. Você também tem que rastrear as comunicações por meio de `CancellationTokenSource`

e seu [Cancellation Token](#) associado para entender onde o cancelamento foi solicitado e onde ele foi concedido.

## Os fluxos assíncronos fornecem uma melhor maneira

Os fluxos assíncronos e o suporte de linguagem associado lidam com todas essas preocupações. O código que gera a sequência agora pode usar `yield return` para retornar os elementos em um método que foi declarado com o modificador `async`. É possível consumir um fluxo assíncrono usando um loop `await foreach` da mesma forma que é possível consumir qualquer sequência usando um loop `foreach`.

Esses novos recursos de linguagem dependem das três novas interfaces adicionadas ao .NET Standard 2.1 e implementadas no .NET Core 3.0:

- [System.Collections.Generic.IAsyncEnumerable<T>](#)
- [System.Collections.Generic.IAsyncEnumerator<T>](#)
- [System.IAsyncDisposable](#)

Essas três interfaces devem ser familiares à maioria dos desenvolvedores C#. Elas se comportam de maneira semelhante às suas contrapartes síncronas:

- [System.Collections.Generic.IEnumerable<T>](#)
- [System.Collections.Generic.IEnumerator<T>](#)
- [System.IDisposable](#)

Um tipo que pode não ser familiar é [System.Threading.Tasks.ValueTask](#). A estrutura `ValueTask` fornece uma API semelhante para a classe [System.Threading.Tasks.Task](#). `ValueTask` é usado nas interfaces por motivos de desempenho.

## Converter para fluxos assíncronos

Em seguida, converta o método `RunPagedQueryAsync` para gerar um fluxo assíncrono. Primeiro, altere a assinatura de `RunPagedQueryAsync` para retornar um `IAsyncEnumerable<JToken>` e remova os objetos de progresso e o token de cancelamento da lista de parâmetros, conforme mostrado no código a seguir:

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
```

```
    string queryText, string repoName)
```

O código inicial processa cada página à medida que a página é recuperada, como mostrado no código a seguir:

C#

```
finalResults.Merge(issues(results)[ "nodes" ]!);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Substitua essas três linhas pelo seguinte código:

C#

```
foreach (JObject issue in issues(results)[ "nodes" ]!)
    yield return issue;
```

Você também pode remover a declaração de `finalResults` anteriormente nesse método e a instrução `return` que segue o loop que você modificou.

Você terminou as alterações para gerar um fluxo assíncrono. O método concluído deve ser semelhante ao seguinte código:

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables[ "repo_name" ] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
```

```

JObject.Parse(response.HttpResponse.Body.ToString()!);

    int totalCount = (int)issues(results)["totalCount"]!;
    hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
    issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
    ["startCursor"]!.ToString();
    issuesReturned += issues(results)["nodes"]!.Count();

    foreach (JObject issue in issues(results)["nodes"]!)
        yield return issue;
}

JObject issues(JObject result) => (JObject)result["data"]!
["repository"]![["issues"]];
JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}

```

Em seguida, você pode alterar o código que consome a coleção para consumir o fluxo assíncrono. Localize o seguinte código em `Main` que processa a coleção de problemas:

C#

```

var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}

```

Substitua o código pelo seguinte loop `await foreach`:

C#

```

int num = 0;
await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
    "docs"))
{
    Console.WriteLine(issue);
}

```

```
        Console.WriteLine($"Received {++num} issues in total");
    }
}
```

A nova interface `IAsyncEnumerator<T>` deriva de `IAsyncDisposable`. Isso significa que o loop anterior descartará o fluxo de forma assíncrona quando o loop terminar. Você pode imaginar que o loop se parece com o seguinte código:

C#

```
int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery,
"docs").GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

Por padrão, os elementos de fluxo são processados no contexto capturado. Se você quiser desabilitar a captura do contexto, use o método de extensão `TaskAsyncEnumerableExtensions.ConfigureAwait`. Para obter mais informações sobre contextos de sincronização e captura do contexto atual, consulte o artigo [Como consumir o padrão assíncrono baseado em tarefa](#).

Os fluxos assíncronos dão suporte ao cancelamento usando o mesmo protocolo que outros métodos `async`. Você modificaria a assinatura do método iterador assíncrono da seguinte forma para dar suporte ao cancelamento:

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, [EnumeratorCancellation]
CancellationToken cancellationToken = default)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;
```

```

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]![["issues"]];
    JObject pageInfo(JObject result) => (JObject)issues(result)
[["pageInfo"]!];
}

```

O atributo `System.Runtime.CompilerServices.EnumeratorCancellationAttribute` faz com que o compilador gere código para o `IAsyncEnumerable<T>` que torna o token passado para visível para `GetAsyncEnumerator` o corpo do iterador assíncrono como esse argumento. Em `runQueryAsync`, você pode examinar o estado do token e cancelar mais trabalhos, se solicitado.

Você usa outro método de extensão, `WithCancellation`, para passar o token de cancelamento para o fluxo assíncrono. Você modificaria o loop ao enumerar os problemas da seguinte forma:

C#

```

private static async Task EnumerateWithCancellation(GitHubClient client)
{
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs")
        .WithCancellation(cancellation.Token))
    {

```

```
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}
```

Você pode obter o código para o tutorial concluído do repositório [dotnet/docs](#) na pasta [asynchronous-programming/snippets](#).

## Executar o aplicativo finalizado

Execute o aplicativo novamente. Compare esse comportamento com o comportamento do aplicativo inicial. A primeira página de resultados é enumerada assim que fica disponível. Há uma pausa observável à medida que cada nova página é solicitada e recuperada, e os resultados da próxima página são rapidamente enumerados. O bloco `try / catch` não é necessário para lidar com o cancelamento: o chamador pode interromper a enumeração da coleção. O progresso é claramente informado, pois o fluxo assíncrono gera resultados à medida que cada página é baixada. O status de cada problema retornado está incluído diretamente no loop `await foreach`. Você não precisa de um objeto de retorno de chamada para acompanhar o progresso.

Você pode ver as melhorias no uso da memória examinando o código. Não é mais necessário alocar uma coleção para armazenar todos os resultados antes de serem enumerados. O chamador pode determinar como consumir os resultados e se uma coleção de armazenamento é necessária.

Execute o aplicativos inicial e o acabado, e observe você mesmo as diferenças entre as implementações. Depois de terminar, você pode excluir o token de acesso de GitHub criado ao iniciar este tutorial. Se um invasor obtiver acesso a esse token, ele poderá acessar as APIs do GitHub usando suas credenciais.

Neste tutorial, você usou fluxos assíncronos para ler itens individuais de uma API de rede que retorna páginas de dados. Fluxos assíncronos também podem ler de "fluxos sem fim", como um ticker de ações ou dispositivo sensor. A chamada para `MoveNextAsync` retorna o próximo item assim que ele estiver disponível.

# Tutorial: Escrever um manipulador de interpolação de cadeia de caracteres personalizado

Artigo • 06/04/2023

Neste tutorial, você aprenderá como:

- ✓ Implementar o padrão de manipulador de interpolação de cadeia de caracteres
- ✓ Interaja com o receptor em uma operação de interpolação de cadeia de caracteres.
- ✓ Adicionar argumentos ao manipulador de interpolação de cadeia de caracteres
- ✓ Entender os novos recursos de biblioteca para interpolação de cadeia de caracteres

## Pré-requisitos

Você precisará configurar seu computador para executar o .NET 6, incluindo o compilador C# 10. O compilador C# 10 está disponível a partir do [Visual Studio 2022](#) ↗ ou do [.NET 6 SDK](#) ↗.

Este tutorial pressupõe que você esteja familiarizado com o C# e .NET, incluindo o Visual Studio ou a CLI do .NET.

## Nova estrutura de tópicos

O C# 10 adiciona suporte para um [\*manipulador de cadeia de caracteres interpolado\*](#) personalizado. Um manipulador de cadeia de caracteres interpolado é um tipo que processa a expressão de espaço reservado em uma cadeia de caracteres interpolada. Sem um manipulador personalizado, os espaços reservados são processados de forma semelhante a [String.Format](#). Cada espaço reservado é formatado como texto e, em seguida, os componentes são concatenados para formar a cadeia de caracteres resultante.

Você pode escrever um manipulador para qualquer cenário em que use informações sobre a cadeia de caracteres resultante. Será usado? Quais restrições estão no formato? Alguns exemplos incluem:

- Você pode exigir que nenhuma das cadeias de caracteres resultantes seja maior que algum limite, como 80 caracteres. Você pode processar as cadeias de caracteres interpoladas para preencher um buffer de comprimento fixo e interromper o processamento depois que o comprimento do buffer for atingido.

- Você pode ter um formato tabular e cada espaço reservado deve ter um comprimento fixo. Um manipulador personalizado pode impor isso, em vez de forçar que todo o código do cliente esteja em conformidade.

Neste tutorial, você criará um manipulador de interpolação de cadeia de caracteres para um dos principais cenários de desempenho: registrar bibliotecas em log. Dependendo do nível de log configurado, o trabalho para construir uma mensagem de log não é necessário. Se o registro em log estiver desativado, o trabalho para construir uma cadeia de caracteres a partir de uma expressão de cadeia de caracteres interpolada não será necessário. A mensagem nunca é impressa, portanto, qualquer concatenação de cadeia de caracteres pode ser ignorada. Além disso, todas as expressões usadas nos espaços reservados, incluindo a geração de rastreamentos de pilha, não precisam ser feitas.

Um manipulador de cadeia de caracteres interpolado pode determinar se a cadeia de caracteres formatada será usada e só executará o trabalho necessário, se necessário.

## Implementação inicial

Vamos começar de uma classe básica `Logger` que dá suporte a diferentes níveis:

C#

```
public enum LogLevel
{
    Off,
    Critical,
    Error,
    Warning,
    Information,
    Trace
}

public class Logger
{
    public LogLevel EnabledLevel { get; init; } = LogLevel.Error;

    public void LogMessage(LogLevel level, string msg)
    {
        if (EnabledLevel < level) return;
        Console.WriteLine(msg);
    }
}
```

Esse `Logger` dá suporte a seis níveis diferentes. Quando uma mensagem não passa no filtro de nível de log, não há saída. A API pública do agente aceita uma cadeia de

caracteres (totalmente formatada) como a mensagem. Todo o trabalho para criar a cadeia de caracteres já foi feito.

## Implementar o padrão de manipulador

Essa etapa é para criar um *manipulador de cadeia de caracteres interpolado* que recrie o comportamento atual. Um manipulador de cadeia de caracteres interpolado é um tipo que deve ter as seguintes características:

- O `System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute` aplicado ao tipo.
- Um construtor que tem dois parâmetros `int`, `literalLength` e `formatCount`. (Mais parâmetros são permitidos).
- Um método público `AppendLiteral` com a assinatura: `public void AppendLiteral(string s)`.
- Um método público `AppendFormatted` genérico com a assinatura: `public void AppendFormatted<T>(T t)`.

Internamente, o construtor cria a cadeia de caracteres formatada e fornece um membro para um cliente recuperar essa cadeia de caracteres. O código a seguir mostra um tipo `LogInterpolatedStringHandler` que atende a esses requisitos:

C#

```
[InterpolatedStringHandler]
public ref struct LogInterpolatedStringHandler
{
    // Storage for the built-up string
    StringBuilder builder;

    public LogInterpolatedStringHandler(int literalLength, int
formattedCount)
    {
        builder = new StringBuilder(literalLength);
        Console.WriteLine($"\\tliteral length: {literalLength},
formattedCount: {formattedCount}");
    }

    public void AppendLiteral(string s)
    {
        Console.WriteLine($"\\tAppendLiteral called: {{s}}");

        builder.Append(s);
        Console.WriteLine($"\\tAppended the literal string");
    }

    public void AppendFormatted<T>(T t)
```

```
{  
    Console.WriteLine($"\\tAppendFormatted called: {{t}} is of type  
{typeof(T)}");  
  
    builder.Append(t?.ToString());  
    Console.WriteLine($"\\tAppended the formatted object");  
}  
  
internal string GetFormattedText() => builder.ToString();  
}
```

Agora você pode adicionar uma sobrecarga a `LogMessage` na classe `Logger` para experimentar o novo manipulador de cadeia de caracteres interpolado:

C#

```
public void LogMessage(LogLevel level, LogInterpolatedStringHandler builder)  
{  
    if (EnabledLevel < level) return;  
    Console.WriteLine(builder.GetFormattedText());  
}
```

Você não precisa remover o método original `LogMessage`, o compilador preferirá um método com um parâmetro de manipulador interpolado em vez de um método com um parâmetro `string` quando o argumento for uma expressão de cadeia de caracteres interpolada.

Você pode verificar se o novo manipulador é invocado usando o seguinte código como o programa principal:

C#

```
var logger = new Logger() { EnabledLevel = LogLevel.Warning };  
var time = DateTime.Now;  
  
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. This  
is an error. It will be printed.");  
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time}. This  
won't be printed.");  
logger.LogMessage(LogLevel.Warning, "Warning Level. This warning is a  
string, not an interpolated string expression.");
```

A execução do aplicativo produz uma saída semelhante ao seguinte texto:

PowerShell

```
literal length: 65, formattedCount: 1  
AppendLiteral called: {Error Level. CurrentTime: }
```

```
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This won't be printed.}
        Appended the literal string
Warning Level. This warning is a string, not an interpolated string
expression.
```

Rastreando a saída, você pode ver como o compilador adiciona código para chamar o manipulador e criar a cadeia de caracteres:

- O compilador adiciona uma chamada para construir o manipulador, passando o comprimento total do texto literal na cadeia de caracteres de formato e o número de espaços reservados.
- O compilador adiciona chamadas a `AppendLiteral` e `AppendFormatted` para cada seção da cadeia de caracteres literal e para cada espaço reservado.
- O compilador invoca o método `LogMessage` usando o `CoreInterpolatedStringHandler` como argumento.

Por fim, observe que o último aviso não invoca o manipulador de cadeia de caracteres interpolado. O argumento é um `string`, de modo que a chamada invoca a outra sobrecarga com um parâmetro de cadeia de caracteres.

## Adicionar mais recursos ao manipulador

A versão anterior do manipulador de cadeia de caracteres interpolada implementa o padrão. Para evitar o processamento de cada expressão de espaço reservado, você precisará de mais informações no manipulador. Nesta seção, você melhorará seu manipulador para que ele funcione menos quando a cadeia de caracteres construída não for gravada no log. Você usa

`System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute` para especificar um mapeamento entre parâmetros para uma API pública e parâmetros para o construtor de um manipulador. Isso fornece ao manipulador as informações necessárias para determinar se a cadeia de caracteres interpolada deve ser avaliada.

Vamos começar com alterações no Manipulador. Primeiro, adicione um campo para acompanhar se o manipulador está habilitado. Adicione dois parâmetros ao construtor: um para especificar o nível de log dessa mensagem e o outro como referência ao objeto de log:

```
C#  
  
private readonly bool enabled;  
  
public LogInterpolatedStringHandler(int literalLength, int formattedCount,  
Logger logger, LogLevel logLevel)  
{  
    enabled = logger.EnabledLevel >= logLevel;  
    builder = new StringBuilder(literalLength);  
    Console.WriteLine($"\\tliteral length: {literalLength}, formattedCount:  
{formattedCount}");  
}
```

Em seguida, use o campo para que o manipulador acrescente somente literais ou objetos formatados quando a cadeia de caracteres final for usada:

```
C#  
  
public void AppendLiteral(string s)  
{  
    Console.WriteLine($"\\tAppendLiteral called: {{s}}");  
    if (!enabled) return;  
  
    builder.Append(s);  
    Console.WriteLine($"\\tAppended the literal string");  
}  
  
public void AppendFormatted<T>(T t)  
{  
    Console.WriteLine($"\\tAppendFormatted called: {{t}} is of type  
{typeof(T)}");  
    if (!enabled) return;  
  
    builder.Append(t?.ToString());  
    Console.WriteLine($"\\tAppended the formatted object");  
}
```

Em seguida, você precisará atualizar a declaração `LogMessage` para que o compilador passe os parâmetros adicionais para o construtor do manipulador. Isso é tratado usando `System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute` no argumento do manipulador:

```
C#
```

```
public void LogMessage(LogLevel level,
    [InterpolatedStringHandlerArgument("", "level")]
    LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

Esse atributo especifica a lista de argumentos para `LogMessage` esse mapa para os parâmetros que seguem os parâmetros `literalLength` e `formattedCount` necessários. A cadeia de caracteres vazia ("") especifica o receptor. O compilador substitui o valor do objeto `Logger` representado pelo `this` pelo argumento seguinte para o construtor do manipulador. O compilador substitui o valor de `level` pelo argumento a seguir. Você pode fornecer qualquer número de argumentos para qualquer manipulador que escrever. Os argumentos que você adiciona são argumentos de cadeia de caracteres.

Você pode executar essa versão usando o mesmo código de teste. Desta vez, você verá os seguintes resultados:

PowerShell

```
literal length: 65, formattedCount: 1
AppendLiteral called: {Error Level. CurrentTime: }
Appended the literal string
AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
    Appended the formatted object
    AppendLiteral called: {. This is an error. It will be printed.}
    Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
be printed.
    literal length: 50, formattedCount: 1
    AppendLiteral called: {Trace Level. CurrentTime: }
    AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
    AppendLiteral called: {. This won't be printed.}
Warning Level. This warning is a string, not an interpolated string
expression.
```

Você pode ver que os métodos `AppendLiteral` e `AppendFormat` estão sendo chamados, mas eles não estão fazendo nenhum trabalho. O manipulador determinou que a cadeia de caracteres final não será necessária, portanto, o manipulador não a compila. Ainda há alguns aprimoramentos a serem feitos.

Primeiro, você pode adicionar uma sobrecarga de `AppendFormatted` que restringe o argumento a um tipo que implementa `System.IFormattable`. Essa sobrecarga permite que os chamadores adicionem cadeias de caracteres de formato nos espaços

reservados. Ao fazer essa alteração, também vamos alterar o tipo de retorno dos outros métodos `AppendFormatted` e `AppendLiteral`, de `void` para `bool` (se algum desses métodos tiver tipos de retorno diferentes, você receberá um erro de compilação). Essa alteração permite *curto-circuito*. Os métodos retornam `false` para indicar que o processamento da expressão de cadeia de caracteres interpolada deve ser interrompido. Retornar `true` indica que ele deve continuar. Neste exemplo, você está usando-o para interromper o processamento quando a cadeia de caracteres resultante não for necessária. O curto-circuito dá suporte a ações mais refinadas. Você pode parar de processar a expressão quando ela atingir um determinado comprimento, para dar suporte a buffers de comprimento fixo. Ou alguma condição pode indicar que elementos restantes não são necessários.

C#

```
public void AppendFormatted<T>(T t, string format) where T : IFormattable
{
    Console.WriteLine($"\\tAppendFormatted (IFormattable version) called: {t}
with format {{format}} is of type {typeof(T)},");

    builder.Append(t?.ToString(format, null));
    Console.WriteLine($"\\tAppended the formatted object");
}
```

Com essa adição, você pode especificar cadeias de caracteres de formato em sua expressão de cadeia de caracteres interpolada:

C#

```
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. The
time doesn't use formatting.");
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time:t}. This
is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time:t}. This
won't be printed.");
```

O `:t` na primeira mensagem especifica o "formato de hora curto" para a hora atual. O exemplo anterior mostrou uma das sobrecargas para o método `AppendFormatted` que você pode criar para o manipulador. Você não precisa especificar um argumento genérico para o objeto que está sendo formatado. Você pode ter maneiras mais eficientes de converter tipos criados em cadeia de caracteres. Você pode escrever sobrecargas de `AppendFormatted` que adota esses tipos em vez de um argumento genérico. O compilador escolherá a melhor sobrecarga. O runtime usa essa técnica para

converter `System.Span<T>` em saída de cadeia de caracteres. Você pode adicionar um parâmetro inteiro para especificar o *alinhamento* da saída, com ou sem um `IFormattable`. O `System.Runtime.CompilerServices.DefaultInterpolatedStringHandler` que é fornecido com o .NET 6 contém nove sobrecargas de `AppendFormatted` para usos diferentes. Você pode usá-lo como referência ao criar um manipulador para suas finalidades.

Execute o exemplo agora e você verá que, para a mensagem `Trace`, apenas o primeiro `AppendLiteral` é chamado:

PowerShell

```
literal length: 60, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted called: 10/20/2021 12:18:29 PM is of type
System.DateTime
Appended the formatted object
AppendLiteral called: . The time doesn't use formatting.
Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:18:29 PM. The time doesn't use
formatting.
literal length: 65, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted (IFormattable version) called: 10/20/2021 12:18:29
PM with format {t} is of type System.DateTime,
Appended the formatted object
AppendLiteral called: . This is an error. It will be printed.
Appended the literal string
Error Level. CurrentTime: 12:18 PM. This is an error. It will be printed.
literal length: 50, formattedCount: 1
AppendLiteral called: Trace Level. CurrentTime:
Warning Level. This warning is a string, not an interpolated string
expression.
```

Você pode fazer uma atualização final para o construtor do manipulador que melhora a eficiência. O manipulador pode adicionar um parâmetro final `out bool`. Definir esse parâmetro para `false` indica que o manipulador não deve ser chamado para processar a expressão de cadeia de caracteres interpolada:

C#

```
public LogInterpolatedStringHandler(int literalLength, int formattedCount,
Logger logger, LogLevel level, out bool isEnabled)
{
    isEnabled = logger.EnabledLevel >= level;
    Console.WriteLine($"\\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
```

```
    builder = isEnabled ? new StringBuilder(literalLength) : default!;
}
```

Essa alteração significa que você pode remover o campo `enabled`. Em seguida, você pode alterar o tipo de retorno de `AppendLiteral` e `AppendFormatted` para `void`. Agora, ao executar o exemplo, você verá a seguinte saída:

PowerShell

```
literal length: 60, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted called: 10/20/2021 12:19:10 PM is of type
System.DateTime
Appended the formatted object
AppendLiteral called: . The time doesn't use formatting.
Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. The time doesn't use
formatting.
literal length: 65, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted (IFormattable version) called: 10/20/2021 12:19:10
PM with format {t} is of type System.DateTime,
Appended the formatted object
AppendLiteral called: . This is an error. It will be printed.
Appended the literal string
Error Level. CurrentTime: 12:19 PM. This is an error. It will be printed.
literal length: 50, formattedCount: 1
Warning Level. This warning is a string, not an interpolated string
expression.
```

A única saída quando `LogLevel.Trace` foi especificado é a saída do construtor. O manipulador indicou que não está habilitado, portanto, nenhum dos métodos `Append` foi invocado.

Esse exemplo ilustra um ponto importante para manipuladores de cadeias de caracteres interpoladas, especialmente quando bibliotecas de log são usadas. Quaisquer efeitos colaterais nos espaços reservados podem não ocorrer. Adicione o seguinte código ao programa principal e veja esse comportamento em ação:

C#

```
int index = 0;
int numberOfIncrements = 0;
for (var level = LogLevel.Critical; level <= LogLevel.Trace; level++)
{
    Console.WriteLine(level);
    logger.LogMessage(level, $"{level}: Increment index a few times
```

```
{index++}, {index++}, {index++}, {index++}, {index++});  
    numberOfIncrements += 5;  
}  
Console.WriteLine($"Value of index {index}, value of numberOfIncrements:  
{numberOfIncrements}");
```

Você pode ver que a variável `index` é incrementada cinco vezes a cada iteração do loop. Como os espaços reservados são avaliados apenas para `Critical`, `Error` e `Warning` níveis, não para `Information` e `Trace`, o valor final de `index` não corresponde à expectativa:

PowerShell

```
Critical  
Critical: Increment index a few times 0, 1, 2, 3, 4  
Error  
Error: Increment index a few times 5, 6, 7, 8, 9  
Warning  
Warning: Increment index a few times 10, 11, 12, 13, 14  
Information  
Trace  
Value of index 15, value of numberOfIncrements: 25
```

Manipuladores de cadeia de caracteres interpolados fornecem maior controle sobre como uma expressão de cadeia de caracteres interpolada é convertida em uma cadeia de caracteres. A equipe de runtime do .NET já usou esse recurso para melhorar o desempenho em várias áreas. Você pode usar a mesma funcionalidade em suas próprias bibliotecas. Para explorar mais, confira

[System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#). Ele fornece uma implementação mais completa do que a que você criou aqui. Você verá muitas outras sobrecargas possíveis para os métodos `Append`.

# Usar interpolação de cadeia de caracteres para construir cadeia de caracteres formatadas

Artigo • 10/05/2023

Este tutorial ensina como usar a [interpolação de cadeias de caracteres](#) em C# para inserir valores em uma única cadeia de caracteres de resultado. Escreva o código em C# e veja os resultados da compilação e da execução. O tutorial contém uma série de lições que mostram como inserir valores em uma cadeia de caracteres e formatar esses valores de diferentes maneiras.

Este tutorial espera que você tenha um computador que possa usar para desenvolvimento. O tutorial [Olá, Mundo em 10 minutos](#) tem instruções para configurar o ambiente de desenvolvimento local no Windows, no Linux ou no macOS. Também é possível concluir a [versão interativa](#) deste tutorial em seu navegador.

## Criar uma cadeia de caracteres interpolada

Crie um diretório chamado *interpolated*. Faça com que esse seja o diretório atual e execute o seguinte comando em uma janela do console:

CLI do .NET

```
dotnet new console
```

Esse comando cria um novo aplicativo de console .NET Core no diretório atual.

Abra *Program.cs* em seu editor favorito e substitua a linha `Console.WriteLine("Hello World!");` pelo seguinte código, em que você substitui `<name>` pelo seu nome:

C#

```
var name = "<name>";  
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

Experimente este código digitando `dotnet run` na sua janela do console. Ao executar o programa, ele exibe uma única cadeia de caracteres que inclui seu nome na saudação. A cadeia de caracteres incluída na chamada de método `WriteLine` é uma *expressão de cadeia de caracteres interpolada*. Ela é um tipo de modelo que permite que você

construa uma única cadeia de caracteres (chamado de *cadeia de caracteres de resultado*) com base em uma cadeia de caracteres que inclui o código inserido. As cadeias de caracteres interpoladas são particularmente úteis para inserir valores em uma cadeia de caracteres ou para concatenar (unir) cadeias de caracteres.

Esse exemplo simples contém os dois elementos que toda cadeia de caracteres interpolada deve ter:

- Um literal de cadeia de caracteres que começa com o caractere `$` antes do caractere de aspas de abertura. Não pode haver nenhum espaço entre o símbolo `$` e o caractere de aspas. (Se você quiser ver o que acontece ao incluir um espaço, insira um após o caractere `$`, salve o arquivo e execute novamente o programa, digitando `dotnet run` na janela do console. O compilador C# exibe uma mensagem de erro: "erro CS1056: caractere inesperado '\$'".)
- Uma ou mais *expressões de interpolação*. Uma expressão de interpolação é indicada por chaves de abertura e fechamento (`{` e `}`). Você pode colocar qualquer expressão de C# que retorne um valor (incluindo `null`) dentro das chaves.

Vamos testar mais alguns exemplos de interpolação de cadeias de caracteres com outros tipos de dados.

## Incluir diferentes tipos de dados

Na seção anterior, você usou a interpolação de cadeias de caracteres para inserir uma cadeia de caracteres dentro de outra. Entretanto, o resultado de uma expressão de interpolação pode ser de qualquer tipo de dados. Vamos incluir valores de vários tipos de dados em uma cadeia de caracteres interpolada.

No exemplo a seguir, primeiramente definimos um tipo de dados de `classe Vegetable` que tem uma `propriedade Name` e um `método ToString`, que `substitui` o comportamento do método `Object.ToString()`. O `modificador de acesso public` disponibiliza esse método para qualquer código de cliente para obter a representação de cadeia de caracteres de uma instância de `Vegetable`. No exemplo, o método `Vegetable.ToString` retorna o valor da propriedade `Name` que é inicializada no `construtor Vegetable`:

C#

```
public Vegetable(string name) => Name = name;
```

Em seguida, criamos uma instância da classe `Vegetable` chamada `item` usando o `new operador` e fornecendo um nome para o construtor `Vegetable`:

C#

```
var item = new Vegetable("eggplant");
```

Por fim, incluímos a variável `item` em uma cadeia de caracteres interpolada que também contém um valor `DateTime`, um valor `Decimal` e um valor de enumeração `Unit` valor. Substitua todo o código C# em seu editor pelo seguinte código e, depois, use o comando `dotnet run` para executá-lo:

C#

```
using System;

public class Vegetable
{
    public Vegetable(string name) => Name = name;

    public string Name { get; }

    public override string ToString() => Name;
}

public class Program
{
    public enum Unit { item, kilogram, gram, dozen };

    public static void Main()
    {
        var item = new Vegetable("eggplant");
        var date = DateTime.Now;
        var price = 1.99m;
        var unit = Unit.item;
        Console.WriteLine($"On {date}, the price of {item} was {price} per
{unit}.");
    }
}
```

Observe que a expressão de interpolação `item` na cadeia de caracteres interpolada é resolvida como o texto "eggplant" na cadeia de caracteres de resultado. Isso ocorre porque, quando o tipo do resultado da expressão não é uma cadeia de caracteres, o resultado é resolvido como uma cadeia de caracteres da seguinte maneira:

- Se a expressão de interpolação for avaliada como `null`, uma cadeia de caracteres vazia ("") ou `String.Empty`) será usada.

- Se a expressão de interpolação não foi avaliada como `null`, normalmente o método `ToString` do tipo de resultado será chamado. Você pode testar isso atualizando a implementação do método `Vegetable.ToString`. Talvez nem seja necessário implementar o método `ToString`, pois cada tipo tem algum modo de implementação desse método. Para testar isso, comente a definição do método `Vegetable.ToString` no exemplo (para isso, coloque o símbolo de comentário `//` na frente dele). Na saída, a cadeia de caracteres "eggplant" é substituída pelo nome do tipo totalmente qualificado, ("Vegetable" neste exemplo), que é o comportamento padrão do método `Object.ToString()`. O comportamento padrão do método `ToString` para um valor de enumeração é retornar a representação de cadeia de caracteres do valor.

Na saída deste exemplo, a data é muito precisa (o preço de "eggplant" não muda a cada segundo) e o valor do preço não indica uma unidade monetária. Na próxima seção, você aprenderá como corrigir esses problemas controlando o formato das representações das cadeias de caracteres dos resultados de expressão.

## Controlar a formatação de expressões de interpolação

Na seção anterior, duas cadeias de caracteres formatadas de maneira inadequada foram inseridas na cadeia de caracteres de resultado. Uma era um valor de data e hora para a qual apenas a data era adequada. A segunda era um preço que não indicava a unidade monetária. Os dois problemas são fáceis de se resolver. A interpolação de cadeias de caracteres permite especificar *cadeias de caracteres de formato* que controlam a formatação de tipos específicos. Modifique a chamada a `Console.WriteLine` no exemplo anterior para incluir as cadeias de caracteres de formato para as expressões de data e de preço, conforme mostrado na linha a seguir:

C#

```
Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per {unit}.");
```

Você especifica uma cadeia de caracteres de formato colocando dois-pontos (":") e a cadeia de caracteres de formato após a expressão de interpolação. "d" é uma [cadeia de caracteres de formato de data e hora padrão](#) que representa o formato de data abreviada. "C2" é um [cadeia de caracteres de formato numérico padrão](#) que representa um número como um valor de moeda com dois dígitos após o ponto decimal.

Diversos tipos nas bibliotecas do .NET são compatíveis com um conjunto predefinido de cadeias de caracteres de formato. Isso inclui todos os tipos numéricos e os tipos de data e hora. Para obter uma lista completa dos tipos que são compatíveis com as cadeias de caracteres de formato, consulte [Cadeias de caracteres de formato e tipos da biblioteca de classes do .NET](#) no artigo [Tipos de formatação no .NET](#).

Tente modificar as cadeias de caracteres de formato em seu editor de texto e, sempre que fizer uma alteração, execute novamente o programa para ver como as alterações afetam a formatação da data e hora e do valor numérico. Altere o "d" em `{date:d}` para "t" (para exibir o formato de hora abreviada), para "y" (para exibir o ano e o mês) e para "yyyy" (para exibir o ano como um número de quatro dígitos). Altere o "C2" em `{price:C2}` para "e" (para obter notação exponencial) e para "F3" (para um valor numérico com três dígitos após o ponto decimal).

Além de controlar a formatação, você também pode controlar a largura do campo e o alinhamento das cadeias de caracteres formatadas incluídas na cadeia de caracteres de resultado. Na próxima seção, você aprenderá como fazer isso.

## Controlar a largura do campo e o alinhamento de expressões de interpolação

Normalmente, quando o resultado de uma expressão de interpolação é formatado em uma cadeia de caracteres, essa cadeia de caracteres é incluída em uma cadeia de caracteres sem espaços à esquerda nem à direita. Especialmente quando você trabalha com um conjunto de dados, poder controlar a largura do campo e o alinhamento do texto ajuda a produzir uma saída mais legível. Para ver isso, substitua todo o código em seu editor de texto pelo código a seguir e, em seguida, digite `dotnet run` para executar o programa:

C#

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        var titles = new Dictionary<string, string>()
        {
            ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
            ["London, Jack"] = "Call of the Wild, The",
            ["Shakespeare, William"] = "Tempest, The"
        };
    }
}
```

```
Console.WriteLine("Author and Title List");
Console.WriteLine();
Console.WriteLine($"|{ "Author", -25}|{ "Title", 30} |");
foreach (var title in titles)
    Console.WriteLine($"|{title.Key, -25}|{title.Value, 30} |");
}
```

Os nomes de autores são alinhados à esquerda e os títulos que eles escreveram são alinhados à direita. Você especifica o alinhamento adicionando uma vírgula (",") após a expressão de interpolação e designando a largura *mínima* do campo. Se o valor especificado for um número positivo, o campo será alinhado à direita. Se for um número negativo, o campo será alinhado à esquerda.

Tente remover os sinais negativos do código `{"Author", -25}` e `{title.Key, -25}` e execute o exemplo novamente, como feito no código a seguir:

C#

```
Console.WriteLine($"|{ "Author", 25}|{ "Title", 30} |");
foreach (var title in titles)
    Console.WriteLine($"|{title.Key, 25}|{title.Value, 30} |");
```

Desta vez, as informações sobre o autor são alinhadas à direita.

Você pode combinar um especificador de alinhamento e uma cadeia de caracteres de formato em uma única expressão de interpolação. Para fazer isso, especifique o alinhamento primeiro, seguido por dois-pontos e pela cadeia de caracteres de formato. Substitua todo o código dentro do método `Main` pelo código a seguir, que exibe três cadeias de caracteres formatadas com larguras de campo definidas. Em seguida, execute o programa inserindo o comando `dotnet run`.

C#

```
Console.WriteLine($"{ {DateTime.Now, -20:d} } Hour [{ {DateTime.Now, -10:HH} }]
[{ 1063.342, 15:N2 } ] feet");
```

A saída é semelhante ao seguinte:

Console

```
[04/14/2018] Hour [16] [1,063.34] feet
```

Você concluiu o tutorial de interpolação de cadeias de caracteres.

Para obter mais informações, confira o tópico [Interpolação de cadeia de caracteres](#) e o tutorial [Interpolação de cadeia de caracteres no C#](#).

# Interpolação de cadeias de caracteres em C#

Artigo • 10/05/2023

Este tutorial mostra como usar a [interpolação de cadeia de caracteres](#) para formatar e incluir resultados de expressão em uma cadeia de caracteres de resultado. Os exemplos pressupõem que você esteja familiarizado com os conceitos básicos do C# e a formatação de tipos do .NET. Se você não estiver familiarizado com a interpolação de cadeia de caracteres ou com a formatação de tipos do .NET, confira primeiro o [tutorial interativo sobre a interpolação de cadeia de caracteres](#). Para obter mais informações sobre como formatar tipos no .NET, confira o tópico [Formatando tipos no .NET](#).

## ⓘ Observação

Os exemplos de C# neste artigo são executados no executador de código embutido Try.NET ↗ e no playground. Clique no botão **Executar** para executar um exemplo em uma janela interativa. Ao executar o código, é possível modificá-lo e executar o código modificado clicando em **Executar** novamente. O código modificado será executado na janela interativa ou, se a compilação falhar, a janela interativa exibirá todos as mensagens de erro do compilador C#.

## Introdução

O recurso [interpolação de cadeia de caracteres](#) baseia-se no recurso [formatação composta](#) e fornece uma sintaxe mais legível e conveniente para incluir resultados de expressão formatada em uma cadeia de caracteres de resultado.

Para identificar uma literal de cadeia de caracteres como uma cadeia de caracteres interpolada, preceda-o com o símbolo `$`. Você pode inserir qualquer expressão C# válida que retorna um valor em uma cadeia de caracteres interpolada. No seguinte exemplo, assim que uma expressão é avaliada, o resultado é convertido em uma cadeia de caracteres e incluído em uma cadeia de caracteres de resultado:

C#

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is
{0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs
```

```
of {a} and {b} is {CalculateHypotenuse(a, b)}");  
  
double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 *  
leg1 + leg2 * leg2);  
  
// Expected output:  
// Area of the right triangle with legs of 3 and 4 is 6  
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

Como mostra o exemplo, você inclui uma expressão em uma cadeia de caracteres interpolada colocando-a com chaves:

C#

```
{<interpolationExpression>}
```

Cadeia de caracteres interpoladas são compatíveis com todos os recursos do recurso [formatação composta de cadeia de caracteres](#). Isso as torna uma alternativa mais legível ao uso do método [String.Format](#).

## Como especificar uma cadeia de caracteres de formato para uma expressão de interpolação

Especifique uma cadeia de caracteres de formato compatível com o tipo do resultado de expressão seguindo a expressão de interpolação com dois-pontos (":") e a cadeia de caracteres de formato:

C#

```
{<interpolationExpression>:<formatString>}
```

O seguinte exemplo mostra como especificar cadeias de caracteres padrão e personalizadas para expressões que produzem resultados numéricos ou de data e hora:

C#

```
var date = new DateTime(1731, 11, 25);  
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} Leonhard Euler introduced  
the letter e to denote {Math.E:F5} in a letter to Christian Goldbach.");  
  
// Expected output:  
// On Sunday, November 25, 1731 Leonhard Euler introduced the letter e to  
denote 2.71828 in a letter to Christian Goldbach.
```

Para obter mais informações, consulte a seção [Componente de cadeia de caracteres de formato](#) do tópico [Formatação composta](#). Esta seção fornece links para tópicos que descrevem cadeias de caracteres de formatos padrão e personalizado compatíveis com os tipos base do .NET.

## Como controlar a largura do campo e o alinhamento da expressão de interpolação formatada

Você especifica a largura mínima do campo e o alinhamento do resultado de expressão formatada seguindo a expressão de interpolação com uma vírgula (",") e a expressão de constante:

C#

```
{<interpolationExpression>,<alignment>}
```

Se o valor *alignment* for positivo, o resultado da expressão formatada será alinhado à direita; se for negativo, ele será alinhado à esquerda.

Caso precise especificar o alinhamento e uma cadeia de caracteres de formato, comece com o componente de alinhamento:

C#

```
{<interpolationExpression>,<alignment>:<formatString>}
```

O seguinte exemplo mostra como especificar o alinhamento e usa caracteres de barra vertical ("|") para delimitar campos de texto:

C#

```
const int NameAlignment = -9;
const int ValueAlignment = 7;

double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a +
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a *
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 /
b),ValueAlignment:F3}|");
```

```
// Expected output:  
// Three classical Pythagorean means of 3 and 4:  
// |Arithmetic| 3.500|  
// |Geometric| 3.464|  
// |Harmonic | 3.429|
```

Como mostra a saída de exemplo, se o tamanho do resultado da expressão formatada exceder a largura de campo especificada, o valor *alignment* será ignorado.

Para obter mais informações, consulte a seção [Componente de alinhamento](#) do tópico [Formatação composta](#).

## Como usar sequências de escape em uma cadeia de caracteres interpolada

Cadeias de caracteres interpoladas dão suporte a todas as sequências de escape que podem ser usadas em literais de cadeia de caracteres comuns. Para obter mais informações, consulte [Sequências de escape de cadeia de caracteres](#).

Para interpretar sequências de escape literalmente, use um literal de cadeia de caracteres [textual](#). Uma cadeia de caracteres verbatim interpolada começa com o caractere \$, seguido pelo caractere @. Você pode usar os tokens \$ e @ em qualquer ordem: \${@"..."} e {@"\${...}"} são cadeias de caracteres verbatim interpoladas válidas.

Para incluir uma chave, {" ou "}, em uma cadeia de caracteres de resultado, use duas chaves, {"{" ou "}}}. Para obter mais informações, consulte a seção [Chaves de escape](#) do tópico [Formatação composta](#).

O seguinte exemplo mostra como incluir chaves em uma cadeia de caracteres de resultado e construir uma cadeia de caracteres interpolada textual:

C#

```
var xs = new int[] { 1, 2, 7, 9 };  
var ys = new int[] { 7, 9, 12 };  
Console.WriteLine($"Find the intersection of the {{{{string.Join(", ",xs)}}}  
and {{{string.Join(", ",ys)}}}} sets.");  
  
var userName = "Jane";  
var stringWithEscapes = $"C:\\\\Users\\\\{userName}\\\\Documents";  
var verbatimInterpolated = ${"C:\\Users\\${userName}\\Documents"};  
Console.WriteLine(stringWithEscapes);  
Console.WriteLine(verbatimInterpolated);  
  
// Expected output:
```

```
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.  
// C:\Users\Jane\Documents  
// C:\Users\Jane\Documents
```

## Como usar um operador condicional ternário ?: em uma expressão de interpolação

Como os dois-pontos (:) têm um significado especial em um item com uma expressão de interpolação, para usar um [operador condicional](#) em uma expressão, coloque-a entre parênteses, como mostra o seguinte exemplo:

C#

```
var rand = new Random();  
for (int i = 0; i < 7; i++)  
{  
    Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" :  
    "tails")});  
}
```

## Como criar uma cadeia de caracteres de resultado específica a uma cultura com a interpolação de cadeia de caracteres

Por padrão, uma cadeia de caracteres interpolada usa a cultura atual definida pela propriedade [CultureInfo.CurrentCulture](#) para todas as operações de formatação. Use uma conversão implícita de uma cadeia de caracteres interpolada em uma instância [System.FormattableString](#) e chame seu método [ToString\(IFormatProvider\)](#) para criar uma cadeia de caracteres de resultado específica a uma cultura. O seguinte exemplo mostra como fazer isso:

C#

```
var cultures = new System.Globalization.CultureInfo[]  
{  
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),  
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),  
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),  
    System.Globalization.CultureInfo.InvariantCulture  
};  
  
var date = DateTime.Now;  
var number = 31_415_926.536;
```

```
FormattableString message = $"{date,20}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}

// Expected output is like:
// en-US      5/17/18 3:44:55 PM      31,415,926.536
// en-GB      17/05/2018 15:44:55      31,415,926.536
// nl-NL      17-05-18 15:44:55      31.415.926,536
//          05/17/2018 15:44:55      31,415,926.536
```

Como mostra o exemplo, você pode usar uma instância [FormattableString](#) para gerar várias cadeias de caracteres de resultado para várias culturas.

## Como criar uma cadeia de caracteres de resultado usando a cultura invariável

Juntamente com o método [FormattableString.ToString\(IFormatProvider\)](#), você pode usar o método [FormattableString.Invariant](#) estático para resolver uma cadeia de caracteres interpolada em uma cadeia de caracteres de resultado para a [InvariantCulture](#). O seguinte exemplo mostra como fazer isso:

C#

```
string messageInInvariantCulture = FormattableString.Invariant($"Date and
time in invariant culture: {DateTime.Now}");
Console.WriteLine(messageInInvariantCulture);

// Expected output is like:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

## Conclusão

Este tutorial descreve cenários comuns de uso da interpolação de cadeia de caracteres. Para obter mais informações sobre a interpolação de cadeia de caracteres, consulte o tópico [Interpolação de cadeia de caracteres](#). Para obter mais informações sobre como formatar tipos no .NET, confira os tópicos [Formatando tipos no .NET](#) e [Formatação composta](#).

## Confira também

- `String.Format`
- `System.FormattableString`
- `System.IFormattable`
- Cadeias de caracteres

# Aplicativo de console

Artigo • 14/03/2023

Este tutorial ensina vários recursos em .NET e na linguagem C#. O que você aprenderá:

- Noções básicas da CLI do .NET
- A estrutura de um aplicativo de console C#
- E/S do Console
- Fundamentos das APIs de E/S de arquivo no .NET
- Os fundamentos da programação assíncrona controlada por tarefas no .NET Core

Você criará um aplicativo que lê um arquivo de texto e exibe o conteúdo desse arquivo de texto no console. A saída para o console é conduzida a fim de corresponder à leitura em voz alta. É possível acelerar ou diminuir o ritmo pressionando as teclas "<" (menor que) ou ">" (maior que). Execute esse aplicativo no Windows, no Linux, no macOS ou em um contêiner do Docker.

Há vários recursos neste tutorial. Vamos criá-los individualmente.

## Pré-requisitos

- [SDK do .NET 6](#).
- Um editor de código.

## Criar o aplicativo

A primeira etapa é criar um novo aplicativo. Abra um prompt de comando e crie um novo diretório para seu aplicativo. Torne ele o diretório atual. Digite o comando `dotnet new console` no prompt de comando. Isso cria os arquivos iniciais de um aplicativo "Olá, Mundo" básico.

Antes de começar as modificações, vamos executar um simples aplicativo Olá, Mundo. Depois de criar o aplicativo, digite `dotnet run` no prompt de comando. Esse comando executa o processo de restauração do pacote NuGet, cria o executável do aplicativo e o executa.

O código do aplicativo simples Olá, Mundo está inteiro em *Program.cs*. Abra esse arquivo com o seu editor de texto favorito. Substitua o código em *Program.cs* pelo código a seguir:

```
namespace TeleprompterConsole;

internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Na parte superior do arquivo, observe uma instrução `namespace`. Assim como em outras linguagens orientadas a objeto que você pode ter usado, o C# usa namespaces para organizar tipos. Este programa Olá, Mundo não é diferente. Você pode ver que o programa está no namespace com o nome `TeleprompterConsole`.

## Como ler e exibir o arquivo

O primeiro recurso a ser adicionado é a capacidade de ler um arquivo de texto e a exibição de todo esse texto para um console. Primeiro, vamos adicionar um arquivo de texto. Copie o arquivo [sampleQuotes.txt](#) do repositório do GitHub para este exemplo no diretório de seu projeto. Isso servirá como o script de seu aplicativo. Para obter informações sobre como baixar o aplicativo de exemplo para este tutorial, consulte as instruções em [Exemplos e Tutoriais](#).

Em seguida, adicione o seguinte método em sua classe `Program` (logo abaixo do método `Main`):

```
C#

static IEnumerable<string> ReadFrom(string file)
{
    string? line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

Esse método é um tipo especial de método C# chamado de *Método iterador*. Os métodos de iterador retornam sequências que são avaliadas lentamente. Isso significa que cada item na sequência é gerado conforme a solicitação do código que está consumindo a sequência. Os métodos de iterador contêm uma ou mais instruções `yield`

`return`. O objeto retornado pelo método `ReadFrom` contém o código para gerar cada item na sequência. Neste exemplo, isso envolve a leitura da próxima linha de texto do arquivo de origem e o retorno dessa cadeia de caracteres. Toda vez que o código de chamada solicita o próximo item da sequência, o código lê a próxima linha de texto do arquivo e a retorna. Após a leitura completa do arquivo, a sequência indicará que não há mais itens.

Há dois elementos da sintaxe em C# que podem ser novidade para você. A instrução `using` nesse método gerencia a limpeza de recursos. A variável inicializada na instrução `using` (`reader`, neste exemplo) deve implementar a interface `IDisposable`. Essa interface define um único método, `Dispose`, que deve ser chamado quando o recurso for liberado. O compilador gera essa chamada quando a execução atingir a chave de fechamento da instrução `using`. O código gerado pelo compilador garante que o recurso seja liberado, mesmo se uma exceção for lançada do código no bloco definido pela instrução `using`.

A variável `reader` é definida usando a palavra-chave `var`. `var` define uma *variável local de tipo implícito*. Isso significa que o tipo da variável é determinado pelo tipo de tempo de compilação do objeto atribuído à variável. Aqui, esse é o valor retornado do método `OpenText(String)`, que é um objeto `StreamReader`.

Agora, vamos preencher o código para ler o arquivo no método `Main`:

```
C#  
  
var lines = ReadFrom("sampleQuotes.txt");  
foreach (var line in lines)  
{  
    Console.WriteLine(line);  
}
```

Execute o programa (usando `dotnet run`, e você poderá ver todas as linhas impressas no console).

## Adicionar atrasos e formatar a saída

O que você possui está sendo exibido muito rápido para permitir a leitura em voz alta. Agora você precisa adicionar os atrasos na saída. Ao começar, você criará parte do código principal que permite o processamento assíncrono. No entanto, essas primeiras etapas seguirão alguns antipadrões. Os antipadrões são indicados nos comentários durante a adição do código, e o código será atualizado em etapas posteriores.

Há duas etapas nesta seção. Primeiro, você atualizará o método iterador a fim de retornar palavras individuais em vez de linhas inteiras. Isso é feito com estas modificações. Substitua a instrução `yield return line;` pelo seguinte código:

C#

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

Em seguida, será necessário modificar a forma como você consome as linhas do arquivo, e adicionar um atraso depois de escrever cada palavra. Substitua a instrução `Console.WriteLine(line)` no método `Main` pelo seguinte bloco:

C#

```
Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

Execute o exemplo e verifique a saída. Agora, cada palavra única é impressa, seguida por um atraso de 200 ms. No entanto, a saída exibida mostra alguns problemas, pois o arquivo de texto de origem contém várias linhas com mais de 80 caracteres sem uma quebra de linha. Isso pode ser difícil de ler durante a rolagem da tela. Mas também é fácil de corrigir. Apenas mantenha o controle do comprimento de cada linha e gere uma nova linha sempre que o comprimento atingir um certo limite. Declare uma variável local após a declaração de `words` no método `ReadFrom` que contém o comprimento da linha:

C#

```
var lineLength = 0;
```

Em seguida, adicione o seguinte código após a instrução `yield return word + " "`; (antes da chave de fechamento):

C#

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Execute o exemplo e você poderá ler em voz alta de acordo com o ritmo pré-configurado.

## Tarefas assíncronas

Nesta etapa final, você adicionará o código para gravar a saída de forma assíncrona em uma tarefa, enquanto executa também outra tarefa para ler a entrada do usuário, caso ele queira acelerar ou diminuir o ritmo da exibição do texto ou interromper a exibição do texto por completo. Essa etapa tem alguns passos e, no final, você terá todas as atualizações necessárias. A primeira etapa é criar um método de retorno [Task](#) assíncrono que representa o código que você criou até agora para ler e exibir o arquivo.

Adicione este método à sua classe [Program](#) (ele é obtido do corpo de seu método [Main](#)):

C#

```
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(200);
        }
    }
}
```

Você observará duas alterações. Primeiro, no corpo do método, em vez de chamar [Wait\(\)](#) para aguardar de forma síncrona a conclusão de uma tarefa, essa versão usa a palavra-chave [await](#). Para fazer isso, você precisa adicionar o modificador [async](#) à assinatura do método. Esse método retorna [Task](#). Observe que não há instruções [return](#) que retornam um objeto [Task](#). Em vez disso, esse objeto [Task](#) é criado pelo código gerado pelo compilador quando você usa o operador [await](#). Você pode imaginar que

esse método retorna quando atinge um `await`. A `Task` retornada indica que o trabalho não foi concluído. O método será retomado quando a tarefa em espera for concluída. Após a execução completa, a `Task` retornada indicará a conclusão. O código de chamada pode monitorar essa `Task` retornada para determinar quando ela foi concluída.

Adicione uma palavra-chave `await` antes da chamada para `ShowTeleprompter`:

C#

```
await ShowTeleprompter();
```

Isso exige que você altere a assinatura do método `Main` para:

C#

```
static async Task Main(string[] args)
```

Saiba mais sobre o [método `async Main`](#) em nossa seção de conceitos básicos.

Em seguida, é necessário escrever o segundo método assíncrono a ser lido no Console e ficar atento às teclas "<" (menor que), ">" (maior que) e "X" ou "x". Este é o método que você adiciona à tarefa:

C#

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
}
```

```
    await Task.Run(work);
}
```

Isso cria uma expressão lambda para representar um delegado [Action](#) que lê uma chave no Console e modifica uma variável local que representa o atraso quando o usuário pressiona as teclas "<" (menor que) ou ">" (maior que). O método delegado termina quando o usuário pressiona a tecla "X" ou "x", que permitem ao usuário interromper a exibição de texto a qualquer momento. Esse método usa [ReadKey\(\)](#) para bloquear e aguardar até que o usuário pressione uma tecla.

Para concluir esse recurso, você precisa criar um novo método de retorno `async Task` que inicia essas duas tarefas (`GetInput` e `ShowTeleprompter`) e também gerencia os dados compartilhados entre essas tarefas.

É hora de criar uma classe que pode manipular os dados compartilhados entre essas duas tarefas. Essa classe contém duas propriedades públicas: o atraso e um sinalizador `Done` para indicar que o arquivo foi lido completamente:

C#

```
namespace TeleprompterConsole;

internal class TelePrompterConfig
{
    public int DelayInMilliseconds { get; private set; } = 200;
    public void UpdateDelay(int increment) // negative to speed up
    {
        var newDelay = Min(DelayInMilliseconds + increment, 1000);
        newDelay = Max(newDelay, 20);
        DelayInMilliseconds = newDelay;
    }
    public bool Done { get; private set; }
    public void SetDone()
    {
        Done = true;
    }
}
```

Coloque essa classe em um novo arquivo e inclua-a no namespace `TeleprompterConsole`, conforme mostrado anteriormente. Também é necessário adicionar uma instrução `using static` na parte superior do arquivo para que você possa fazer referência aos métodos `Min` e `Max` sem os nomes de classe ou namespace delimitadores. Uma instrução `using static` importa os métodos de uma classe. Isso contrasta com a instrução `using sem static`, que importa todas as classes de um namespace.

```
C#
```

```
using static System.Math;
```

Em seguida, atualize os métodos `ShowTeleprompter` e `GetInput` para usar o novo objeto `config`. Escreva um método final `async` de retorno de `Task` para iniciar as duas tarefas e sair quando a primeira tarefa for concluída:

```
C#
```

```
private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

O novo método aqui é a chamada `WhenAny(Task[])`. Isso cria uma `Task` que termina assim que qualquer uma das tarefas na lista de argumentos for concluída.

Depois, atualize os métodos `ShowTeleprompter` e `GetInput` para usar o objeto `config` para o atraso:

```
C#
```

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}

private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
        }
    };
}
```

```
        else if (key.KeyChar == '<')
            config.UpdateDelay(10);
        else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            config.SetDone();
    } while (!config.Done);
};

await Task.Run(work);
}
```

Essa nova versão de `ShowTeleprompter` chama um novo método na classe `TeleprompterConfig`. Agora, você precisa atualizar `Main` para chamar `RunTeleprompter` em vez de `ShowTeleprompter`:

C#

```
await RunTeleprompter();
```

## Conclusão

Este tutorial mostrou a você alguns recursos da linguagem C# e as bibliotecas .NET Core relacionadas ao trabalho em aplicativos de Console. Use esse conhecimento como base para explorar mais sobre a linguagem e sobre as classes apresentadas aqui. Você já viu os fundamentos de E/S do Arquivo e do Console, uso com bloqueio e sem bloqueio da programação assíncrona controlada por tarefa, um tour pela linguagem C# e como os programas em C# são organizados, além da CLI do .NET.

Para obter mais informações sobre E/S de arquivo, consulte [E/S de arquivo e de fluxo](#). Para obter mais informações sobre o modelo de programação assíncrona usado neste tutorial, consulte [Programação assíncrona controlada por tarefas](#) e [Programação assíncrona](#).

# Tutorial: Fazer solicitações HTTP em um aplicativo de console .NET usando C #

Artigo • 10/05/2023

Esse tutorial cria um aplicativo que emite solicitações HTTP para um serviço REST no GitHub. O aplicativo lê informações no formato JSON e converte o JSON em objetos C#. A conversão de objetos JSON em C# é conhecida como *deserialização*.

Este tutorial mostra como:

- ✓ Enviar solicitações HTTP.
- ✓ Dessorializar respostas JSON.
- ✓ Configurar a desserialização com atributos.

Se preferir seguir com o [exemplo final](#) desse tutorial, você pode baixá-lo. Para obter instruções de download, consulte [Exemplos e tutoriais](#).

## Pré-requisitos

- [SDK do .NET 6.0 ou posterior](#)
- Um editor de código como [Visual Studio [Code](#)] (um editor de software de código aberto e multiplataforma). Você pode executar o aplicativo de exemplo no Windows, Linux ou macOS ou em um contêiner do Docker.

## Criar o aplicativo cliente

1. Abra um prompt de comando e crie um novo diretório para seu aplicativo. Torne ele o diretório atual.
2. Insira o seguinte comando em uma janela do console:

```
CLI do .NET
dotnet new console --name WebAPIClient
```

Esse comando cria os arquivos iniciais de um aplicativo "Hello World" básico. O nome do projeto é "WebAPIClient".

3. Navegue até o diretório "WebAPIClient" e execute o aplicativo.

```
CLI do .NET
```

```
cd WebAPIClient
```

CLI do .NET

```
dotnet run
```

`dotnet run` executa `dotnet restore` automaticamente para restaurar as dependências de que o aplicativo precisa. Ele também executa `dotnet build` se necessário. Você deve ver a saída do aplicativo "Hello, World!". No terminal, pressione `Ctrl+C` para interromper o aplicativo.

## Fazer solicitações HTTP

Esse aplicativo chama a [API do GitHub](#) para obter informações sobre os projetos no escopo do [.NET Foundation](#). O ponto de extremidade é <https://api.github.com/orgs/dotnet/repos>. Para recuperar informações, ele faz uma solicitação HTTP GET. O navegador também faz solicitações HTTP GET, para que você possa colar essa URL na barra de endereços do seu navegador e ver as informações recebidas e processadas.

Use a classe `HttpClient` para fazer solicitações HTTP. `HttpClient` dá suporte apenas a métodos assíncronos para suas APIs de execução longa. Portanto, as etapas a seguir criam um método assíncrono e o chamam do método Main.

1. Abra o arquivo `Program.cs` no diretório do projeto e substitua seu conteúdo pelo seguinte:

C#

```
await ProcessRepositoriesAsync();

static async Task ProcessRepositoriesAsync(HttpClient client)
{}
```

Esse código:

- Substitui a instrução `Console.WriteLine` por uma chamada para `ProcessRepositoriesAsync`, a qual usa a palavra-chave `await`.
- Define um método `ProcessRepositoriesAsync` vazio.

2. Na classe `Program`, use um `HttpClient` para manipular solicitações e respostas ao substituir o conteúdo pelo C# a seguir.

```
C#  
  
using System.Net.Http.Headers;  
  
using HttpClient client = new();  
client.DefaultRequestHeaders.Accept.Clear();  
client.DefaultRequestHeaders.Accept.Add(  
    new  
    MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));  
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation  
Repository Reporter");  
  
await ProcessRepositoriesAsync(client);  
  
static async Task ProcessRepositoriesAsync(HttpClient client)  
{  
}
```

Esse código:

- Configura cabeçalhos HTTP de todas as solicitações:
  - Um cabeçalho `Accept` para aceitar respostas JSON
  - Um cabeçalho `User-Agent`. Esses cabeçalhos são verificados pelo código do servidor GitHub e são necessários para recuperar informações do GitHub.

3. No método `ProcessRepositoriesAsync`, chame o ponto de extremidade do GitHub que retorna uma lista de todos os repositórios na organização do .NET Foundation:

```
C#  
  
static async Task ProcessRepositoriesAsync(HttpClient client)  
{  
    var json = await client.GetStringAsync(  
        "https://api.github.com/orgs/dotnet/repos");  
  
    Console.WriteLine(json);  
}
```

Esse código:

- Aguarda a tarefa retornada do método de chamada `HttpClient.GetStringAsync(String)`. Esse método envia uma solicitação HTTP

GET para o URI especificado. O corpo da resposta é retornado como um [String](#), que está disponível quando a tarefa é concluída.

- A cadeia de caracteres de resposta `json` é impressa no console.

#### 4. Compile o aplicativo e execute-o.

```
CLI do .NET
```

```
dotnet run
```

Não há nenhum aviso de build porque o `ProcessRepositoriesAsync` agora contém um operador `await`. A saída é uma longa exibição de texto JSON.

## Desserializar o resultado JSON

As etapas a seguir convertem a resposta JSON em objetos C#. Use a classe [System.Text.Json.JsonSerializer](#) para desserializar o JSON em objetos.

1. Crie um arquivo chamado `Repository.cs` e adicione o código a seguir:

```
C#
```

```
public record class Repository(string name);
```

O código anterior define uma classe para representar o objeto JSON retornado da API do GitHub. Você usará essa classe para exibir uma lista de nomes de repositório.

O JSON de um objeto de repositório contém dezenas de propriedades, mas apenas a propriedade `name` será desserializada. O serializador ignora automaticamente as propriedades JSON para as quais não há correspondência na classe de destino. Esse recurso facilita a criação de tipos que funcionam apenas com um subconjunto de campos em um pacote JSON grande.

A convenção C# é [capitalizar a primeira letra de nomes de propriedade](#), mas a propriedade `name` aqui começa com uma letra minúscula porque corresponde exatamente ao que está no JSON. Posteriormente, você verá como usar nomes de propriedade C# que não correspondem aos nomes de propriedade JSON.

2. Use o serializador para converter JSON em objetos C#. Substitua a chamada para `GetStringAsync(String)` no método `ProcessRepositoriesAsync` pelas duas linhas a seguir:

C#

```
await using Stream stream =
    await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories =
    await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
```

O código atualizado substitui `GetStringAsync(String)` por `GetStreamAsync(String)`. Esse método de serializador usa um fluxo, em vez de uma cadeia de caracteres, como sua fonte.

O primeiro argumento `JsonSerializer.DeserializeAsync< TValue >(Stream, JsonSerializerOptions, CancellationToken)` é uma expressão `await`. Expressões `await` podem aparecer em quase todo lugar em seu código, apesar de que até o momento, você apenas viu como parte de uma instrução de atribuição. Os outros dois parâmetros `JsonSerializerOptions` e `CancellationToken`, são opcionais e são omitidos no snippet de código.

O método `DeserializeAsync` é *genérico*, o que significa que você fornece argumentos de tipo para que tipo de objetos devem ser criados a partir do texto JSON. Neste exemplo, você está desserializando para um `List<Repository>`, que é outro objeto genérico, um `System.Collections.Generic.List<T>`. A classe `List<T>` armazena uma coleção de objetos. O argumento `type` declara o tipo de objetos armazenados no `List<T>`. O argumento de tipo é seu registro `Repository`, pois o texto JSON representa uma coleção de objetos de repositório.

3. Adicione código para exibir o nome de cada repositório. Substitua as linhas que mostram:

C#

```
Console.WriteLine(json);
```

pelo código a seguir:

C#

```
foreach (var repo in repositories ?? Enumerable.Empty<Repository>())
    Console.WriteLine(repo.name);
```

4. As seguintes diretivas `using` devem estar presentes na parte superior do arquivo:

C#

```
using System.Net.Http.Headers;
using System.Text.Json;
```

5. Execute o aplicativo.

CLI do .NET

```
dotnet run
```

A saída é uma lista dos nomes dos repositórios que fazem parte do .NET Foundation.

## Configurar a desserialização

1. Em *Repository.cs*, substitua o conteúdo do arquivo pelo C# a seguir.

C#

```
using System.Text.Json.Serialization;

public record class Repository(
    [property: JsonPropertyName("name")] string Name);
```

Esse código:

- Altera o nome da propriedade `name` para `Name`.
- Adiciona o `JsonPropertyNameAttribute` para especificar como essa propriedade aparece no JSON.

2. Em *Program.cs*, atualize o código para usar a nova capitalização da propriedade

`Name`:

C#

```
foreach (var repo in repositories)
    Console.WriteLine(repo.Name);
```

3. Execute o aplicativo.

A saída é a mesma.

# Refatorar o código

O método `ProcessRepositoriesAsync` pode fazer o trabalho assíncrono e retornar uma coleção de repositórios. Altere esse método para retornar `Task<List<Repository>>` e move o código que grava no console perto do chamador.

1. Altere a assinatura de `ProcessRepositoriesAsync` para retornar uma tarefa cujo resultado é uma lista de objetos `Repository`:

```
C#
```

```
static async Task<List<Repository>> ProcessRepositoriesAsync()
```

2. Retorne os repositórios depois de processar a resposta JSON:

```
C#
```

```
await using Stream stream =
    await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories =
    await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
return repositories ?? new();
```

O compilador gera o objeto `Task<T>` para o calor de retorno porque você marcou esse método como `async`.

3. Modifique o arquivo `Program.cs`, substituindo a chamada para `ProcessRepositoriesAsync` pelo seguinte para capturar os resultados e gravar cada nome do repositório no console.

```
C#
```

```
var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
    Console.WriteLine(repo.Name);
```

4. Execute o aplicativo.

A saída é a mesma.

## Desserializar mais propriedades

As etapas a seguir adicionam código para processar mais das propriedades no pacote JSON recebido. Você provavelmente não vai querer processar todas as propriedades, mas adicionar mais algumas demonstra outros recursos de C#.

1. Substitua o conteúdo da classe `Repository` pela seguinte definição `record`:

```
C#  
  
using System.Text.Json.Serialization;  
  
public record class Repository(  
    [property: JsonPropertyName("name")] string Name,  
    [property: JsonPropertyName("description")] string Description,  
    [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,  
    [property: JsonPropertyName("homepage")] Uri Homepage,  
    [property: JsonPropertyName("watchers")] int Watchers);
```

Os tipos `Uri` e `int` têm funcionalidade interna para converter de e para a representação de cadeia de caracteres. Nenhum código extra é necessário para desserializar do formato de cadeia de caracteres JSON para esses tipos de destino. Se o pacote JSON contiver dados que não são convertidos em um tipo de destino, a ação de serialização gerará uma exceção.

2. Atualize o loop `foreach` no arquivo `Program.cs` para exibir os valores da propriedade:

```
C#  
  
foreach (var repo in repositories)  
{  
    Console.WriteLine($"Name: {repo.Name}");  
    Console.WriteLine($"Homepage: {repo.Homepage}");  
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");  
    Console.WriteLine($"Description: {repo.Description}");  
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");  
    Console.WriteLine();  
}
```

3. Execute o aplicativo.

A lista agora inclui as propriedades adicionais.

## Adicionar uma propriedade date

A data da última operação de push é formatada dessa forma na resposta JSON:

JSON

```
2016-02-08T21:27:00Z
```

Esse formato é para UTC (Tempo Universal Coordenado), portanto, o resultado da desserialização é um valor `DateTime` cuja propriedade `Kind` é `Utc`.

Para obter uma data e hora representadas em seu fuso horário, você precisa escrever um método de conversão personalizado.

1. Em `Repository.cs`, adicione uma propriedade para a representação UTC da data e hora e uma propriedade `LastPush` somente leitura que retorna a data convertida em hora local. O arquivo deve se parecer com o seguinte:

C#

```
using System.Text.Json.Serialization;

public record class Repository(
    [property: JsonPropertyName("name")] string Name,
    [property: JsonPropertyName("description")] string Description,
    [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
    [property: JsonPropertyName("homepage")] Uri Homepage,
    [property: JsonPropertyName("watchers")] int Watchers,
    [property: JsonPropertyName("pushed_at")] DateTime LastPushUtc)
{
    public DateTime LastPush => LastPushUtc.ToLocalTime();
}
```

A propriedade `LastPush` é definida usando um *membro com corpo de expressão* para o acessador `get`. Não há acessador `set`. Omitir o acessador `set` é uma maneira de definir uma propriedade *somente leitura* em C#. (Sim, você pode criar propriedades *somente gravação* em C#, mas o valor delas é limitado.)

2. Adicione outra instrução de saída em `Program.cs` novamente:

C#

```
Console.WriteLine($"Last push: {repo.LastPush}");
```

3. O aplicativo completo deve ser semelhante ao seguinte arquivo `Program.cs`:

C#

```
using System.Net.Http.Headers;
using System.Text.Json;
```

```

using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new
    MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation
Repository Reporter");

var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine($"{repo.LastPush}");
    Console.WriteLine();
}

static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient
client)
{
    await using Stream stream =
        await
    client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
    var repositories =
        await JsonSerializer.DeserializeAsync<List<Repository>>
(stream);
    return repositories ?? new();
}

```

#### 4. Execute o aplicativo.

A saída inclui a data e a hora do último push de cada repositório.

## Próximas etapas

Neste tutorial, você criou um aplicativo que faz solicitações da Web e analisa os resultados. Agora, sua versão do aplicativo deve corresponder ao [exemplo finalizado](#).

Saiba mais sobre como configurar a serialização JSON em [Como serializar e desserializar \(marshal and unmarshal\) JSON no .NET](#).

# Trabalhar com a LINQ (Consulta Integrada à Linguagem)

Artigo • 10/05/2023

## Introdução

Este tutorial ensina os recursos no .NET Core e da linguagem C#. Você aprenderá a:

- Gerar sequências com a LINQ.
- Escrever métodos que podem ser usados com facilidade em consultas LINQ.
- Distinguir entre avaliação lenta e detalhada.

Você aprenderá essas técnicas ao compilar um aplicativo que demonstra uma das habilidades básicas de qualquer mágico: o [embaralhamento faro](#). Em resumo, um embaralhamento faro é uma técnica em que você divide um baralho de cartas exatamente na metade, então as cartas de cada metade são colocadas em ordem aleatória até recriar o conjunto original.

Os mágicos usam essa técnica porque cada carta é fica em um local conhecido após o embaralhamento e a ordem é um padrão de repetição.

Para os seus propósitos, vamos examinar rapidamente as sequências de manipulação de dados. O aplicativo que você cria constrói um baralho de cartas e executa uma sequência de embaralhamento, sempre gravando a sequência de saída. Você também comparará a ordem atualizada com a ordem original.

Este tutorial tem várias etapas. Após cada etapa, você poderá executar o aplicativo e ver o progresso. Você também poderá ver o [exemplo concluído](#) no repositório `dotnet/samples` do GitHub. Para obter instruções de download, consulte [Exemplos e tutoriais](#).

## Pré-requisitos

Você precisará configurar seu computador para executar o .NET Core. Você vai encontrar as instruções de instalação na página de [download do .NET Core](#). Você pode executar esse aplicativo no Windows, no Ubuntu Linux, no OS X ou em um contêiner do Docker. Será necessário instalar o editor de código de sua preferência. As descrições a seguir usam o [Visual Studio Code](#), que é um Editor de código aberto multiplataforma. No entanto, você pode usar quaisquer ferramentas que esteja familiarizado.

# Criar o aplicativo

A primeira etapa é criar um novo aplicativo. Abra um prompt de comando e crie um novo diretório para seu aplicativo. Torne ele o diretório atual. Digite o comando `dotnet new console` no prompt de comando. Isso cria os arquivos iniciais de um aplicativo "Olá, Mundo" básico.

Se você nunca usou C# antes, [este tutorial](#) explicará a estrutura de um programa C#. Você pode ler e, em seguida, voltar aqui para saber mais sobre o LINQ.

## Criar o conjunto de dados

Antes de começar, verifique se as linhas a seguir estão na parte superior do arquivo `Program.cs` gerado pelo `dotnet new console`:

```
C#  
  
// Program.cs  
using System;  
using System.Collections.Generic;  
using System.Linq;
```

Se essas três linhas (instruções `using`) não estiverem na parte superior do arquivo, nosso programa não será compilado.

Agora que você tem todas as referências necessárias, considere o que forma um baralho de cartas. Um baralho de cartas costuma ter quatro naipes, e cada naipe tem treze valores. Normalmente, talvez você pense em criar uma classe `Card` logo de cara e preencher uma coleção de objetos `Card` manualmente. Com o LINQ, dá para ser mais conciso do que a forma comum de criação de um baralho de cartas. Em vez de criar uma classe `Card`, você pode criar duas sequências para representar naipes e valores, respectivamente. Você vai criar um par muito simples de [métodos iteradores](#) que gerará os valores e naipes como `IEnumerable<T>`s de cadeias de caracteres:

```
C#  
  
// Program.cs  
// The Main() method  
  
static IEnumerable<string> Suits()  
{  
    yield return "clubs";  
    yield return "diamonds";  
    yield return "hearts";
```

```

        yield return "spades";
    }

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
    yield return "jack";
    yield return "queen";
    yield return "king";
    yield return "ace";
}

```

Coloque-as sob o método `Main` em seu arquivo `Program.cs`. Esses dois métodos utilizam a sintaxe `yield return` para produzir uma sequência à medida que eles são executados. O compilador compila um objeto que implementa `IEnumerable<T>` e gera a sequência de cadeias de caracteres conforme solicitado.

Agora, use esses métodos iteradores para criar o baralho de cartas. Você colocará a consulta do LINQ em nosso método `Main`. Dê uma olhada:

C#

```

// Program.cs
static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                        from r in Ranks()
                        select new { Suit = s, Rank = r };

    // Display each card that we've generated and placed in startingDeck in
    // the console
    foreach (var card in startingDeck)
    {
        Console.WriteLine(card);
    }
}

```

As várias cláusulas `from` produzem um `SelectMany`, que cria uma única sequência da combinação entre cada elemento na primeira sequência com cada elemento na segunda sequência. A ordem é importante para nossos objetivos. O primeiro elemento na primeira sequência de fonte (Naipes) é combinado com cada elemento na segunda

sequência (Valores). Isso produz todas as treze cartas do primeiro naipe. Esse processo é repetido com cada elemento na primeira sequência (naipes). O resultado final é um baralho ordenado por naipes, seguido pelos valores.

É importante lembrar que se você optar por escrever seu LINQ na sintaxe de consulta usada acima, ou se decidir usar a sintaxe de método, sempre será possível alternar entre as formas de sintaxe. A consulta acima escrita em sintaxe de consulta pode ser escrita na sintaxe de método como:

C#

```
var startingDeck = Suits().SelectMany(suit => Ranks()).Select(rank => new {  
    Suit = suit, Rank = rank });
```

O compilador traduz instruções LINQ escritas com a sintaxe de consulta na sintaxe de chamada do método equivalente. Portanto, independentemente de sua escolha de sintaxe, as duas versões da consulta produzem o mesmo resultado. Escolha qual sintaxe funciona melhor para a sua situação: por exemplo, se você estiver trabalhando em uma equipe em que alguns dos membros têm dificuldade com a sintaxe de método, prefira usar a sintaxe de consulta.

Vá em frente e execute o exemplo que você criou neste momento. Ele exibirá todas as 52 cartas do baralho. Talvez seja muito útil executar esse exemplo em um depurador para observar como os métodos `Suits()` e `Ranks()` são executados. Você pode ver claramente que cada cadeia de caracteres em cada sequência é gerada apenas conforme o necessário.

```
C:\console-1inq>dotnet run  
{ Suit = clubs, Rank = two }  
{ Suit = clubs, Rank = three }  
{ Suit = clubs, Rank = four }  
{ Suit = clubs, Rank = five }  
{ Suit = clubs, Rank = six }  
{ Suit = clubs, Rank = seven }  
{ Suit = clubs, Rank = eight }  
{ Suit = clubs, Rank = nine }  
{ Suit = clubs, Rank = ten }  
{ Suit = clubs, Rank = jack }  
{ Suit = clubs, Rank = queen }  
{ Suit = clubs, Rank = king }  
{ Suit = clubs, Rank = ace }  
{ Suit = diamonds, Rank = two }  
{ Suit = diamonds, Rank = three }  
{ Suit = diamonds, Rank = four }  
{ Suit = diamonds, Rank = five }  
{ Suit = diamonds, Rank = six }  
{ Suit = diamonds, Rank = seven }  
{ Suit = diamonds, Rank = eight }  
{ Suit = diamonds, Rank = nine }  
{ Suit = diamonds, Rank = ten }
```

# Manipular a ordem

Em seguida, concentre-se em como você vai embaralhar as cartas no baralho. A primeira etapa de qualquer embaralhada é dividir o baralho em dois. Os métodos `Take` e `Skip` que fazem parte das APIs do LINQ fornecem esse recurso para você. Coloque-os sob o loop `foreach`:

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                        from r in Ranks()
                        select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    // 52 cards in a deck, so 52 / 2 = 26
    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
}
```

No entanto, não há método de embaralhamento na biblioteca padrão, portanto, você precisará escrever o seu. O método de embaralhamento que você criará ilustra várias técnicas que você usará com programas baseados em LINQ. Portanto, cada parte desse processo será explicado nas etapas.

Para adicionar funcionalidade ao seu modo de interação com o `IEnumerable<T>` recebido de volta das consultas do LINQ, precisará escrever alguns tipos especiais de métodos chamados **métodos de extensão**. Em resumo, um método de extensão é um *método estático* de objetivo especial que adiciona novas funcionalidades a um tipo já existentes, sem ter que modificar o tipo original ao qual você deseja adicionar funcionalidade.

Dê aos seus métodos de extensão uma nova casa adicionando um novo arquivo de classe *estático* ao seu programa chamado `Extensions.cs`, depois, comece a criar o primeiro método de extensão:

C#

```
// Extensions.cs
using System;
```

```
using System.Collections.Generic;
using System.Linq;

namespace LinqFaroShuffle
{
    public static class Extensions
    {
        public static IEnumerable<T> InterleaveSequenceWith<T>(this
        IEnumerable<T> first, IEnumerable<T> second)
        {
            // Your implementation will go here soon enough
        }
    }
}
```

Examine a assinatura do método por um momento, principalmente os parâmetros:

C#

```
public static IEnumerable<T> InterleaveSequenceWith<T> (this IEnumerable<T>
first, IEnumerable<T> second)
```

Você pode ver a adição do modificador `this` no primeiro argumento para o método. Isso significa que você chama o método como se fosse um método de membro do tipo do primeiro argumento. Esta declaração de método também segue um idioma padrão no qual os tipos de entrada e saídas são `IEnumerable<T>`. Essa prática permite que os métodos LINQ sejam encadeados para executar consultas mais complexas.

Naturalmente, como você dividiu o baralho em metades, precisará unir essas metades. No código, isso significa que você vai enumerar as duas sequências adquiridas por meio de `Take` e `Skip` ao mesmo tempo, *interleaving* os elementos e criará uma sequência: seu baralho não embaralhado. Escrever um método LINQ que funciona com duas sequências exige que você compreenda como `IEnumerable<T>` funciona.

A interface `IEnumerable<T>` tem um método: `GetEnumerator`. O objeto retornado por `GetEnumerator` tem um método para mover para o próximo elemento e uma propriedade que recupera o elemento atual na sequência. Você usará esses dois membros para enumerar a coleção e retornar os elementos. Esse método de Intercalação será um método iterador, portanto, em vez de criar uma coleção e retornar a coleção, você usará a sintaxe `yield return` mostrada acima.

Aqui está a implementação desse método:

C#

```
public static IEnumerable<T> InterleaveSequenceWith<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}
```

Agora que você escreveu esse método, vá até o método `Main` e embaralhe uma vez:

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                        from r in Ranks()
                        select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
    var shuffle = top.InterleaveSequenceWith(bottom);

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }
}
```

## Comparações

Quantos embaralhamentos são necessários para colocar o baralho em sua ordem original? Para descobrir, você precisará escrever um método que determina se duas sequências são iguais. Depois de ter esse método, você precisará colocar o código de embaralhamento em um loop e verificar quando a apresentação estiver na ordem.

Escrever um método para determinar se as duas sequências são iguais deve ser simples. É uma estrutura semelhante para o método que você escreveu para embaralhar as

cartas. Somente desta vez, em vez de o `yield return` rendimento retornar cada elemento, você comparará os elementos correspondentes de cada sequência. Quando toda a sequência tiver sido enumerada, se os elementos corresponderem, as sequências serão as mesmas:

C#

```
public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while ((firstIter?.MoveNext() == true) && secondIter.MoveNext())
    {
        if ((firstIter.Current is not null) &&
!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}
```

Isso mostra uma segunda linguagem LINQ: métodos de terminal. Eles consideram uma sequência como entrada (ou, neste caso, duas sequências) e retornam um único valor escalar. Ao usar métodos de terminal, eles são sempre o método final em uma cadeia de métodos para uma consulta LINQ, por isso, o nome "terminal".

Você pode ver isso em ação ao usá-lo para determinar quando o baralho está em sua ordem original. Coloque o código de embaralhamento dentro de um loop e pare quando a sequência estiver em sua ordem original, aplicando o método `SequenceEquals()`. Você pode ver que esse sempre será o método final em qualquer consulta, porque ele retorna um valor único em vez de uma sequência:

C#

```
// Program.cs
static void Main(string[] args)
{
    // Query for building the deck

    // Shuffling using InterleaveSequenceWith<T>();

    var times = 0;
    // We can re-use the shuffle variable from earlier, or you can make a
    new one
    shuffle = startingDeck;
```

```

do
{
    shuffle = shuffle.Take(26).InterleaveSequenceWith(shuffle.Skip(26));

    foreach (var card in shuffle)
    {
        Console.WriteLine(card);
    }
    Console.WriteLine();
    times++;

} while (!startingDeck.SequenceEquals(shuffle));

Console.WriteLine(times);
}

```

Execute o código que obtivemos até agora e observe como o baralho é reorganizado em cada embaralhamento. Após 8 embaralhamentos (iterações do loop do-while), o baralho retorna à configuração original que estava quando você o criou pela primeira vez a partir da consulta LINQ inicial.

## Otimizações

O exemplo que você compilou até agora executa um *embaralhamento externo*, no qual as cartas superiores e inferiores permanecem as mesmas em cada execução. Vamos fazer uma alteração: em vez disso, usaremos um *embaralhamento interno*, em que todas as 52 cartas trocam de posição. Para um embaralhamento interno, intercale o baralho para que a primeira carta da metade inferior torne-se a primeira carta do baralho. Isso significa que a última carta na metade superior torna-se a carta inferior. Essa é uma alteração simples em uma única linha de código. Atualize a consulta atual de embaralhamento, alternando as posições de `Take` e `Skip`. Isso alterará a ordem das metades superior e inferior do baralho:

C#

```
shuffle = shuffle.Skip(26).InterleaveSequenceWith(shuffle.Take(26));
```

Execute o programa novamente e você verá que leva 52 iterações para o baralho ser reordenado. Você também começará a observar algumas degradações de desempenho graves à medida que o programa continuar a ser executado.

Existem muitas razões para isso. Você pode abordar uma das principais causas dessa queda de desempenho: uso ineficiente da [avaliação lenta](#).

Em resumo, a avaliação lenta informa que a avaliação de uma instrução não será executada até que seu valor seja necessário. Consultas LINQ são instruções avaliadas lentamente. As sequências são geradas somente quando os elementos são solicitados. Geralmente, esse é o principal benefício do LINQ. No entanto, em uso como esse programa, isso causa um crescimento exponencial no tempo de execução.

Lembre-se de que geramos o baralho original usando uma consulta LINQ. Cada embaralhamento é gerado executando três consultas LINQ no baralho anterior. Todos eles são executados lentamente. Isso também significa que eles são executados novamente sempre que a sequência é solicitada. Ao obter a 52<sup>a</sup> iteração, você estará regenerando o baralho original muitas e muitas vezes. Vamos escrever um log para demonstrar esse comportamento. Em seguida, você poderá corrigir isso.

Em seu arquivo `Extensions.cs`, digite ou copie o método a seguir. Esse método de extensão cria um novo arquivo chamado `debug.log` em seu diretório do projeto, e registra qual consulta está sendo executada atualmente para o arquivo de log. Este método de extensão pode ser anexado a qualquer consulta para marcar que a consulta foi executada.

```
C#  
  
public static IEnumerable<T> LogQuery<T>  
    (this IEnumerable<T> sequence, string tag)  
{  
    // File.AppendText creates a new file if the file doesn't exist.  
    using (var writer = File.AppendText("debug.log"))  
    {  
        writer.WriteLine($"Executing Query {tag}");  
    }  
  
    return sequence;  
}
```

Você verá um rabisco vermelho sob `File`, que significa que ele não existe. Ele não será compilado, pois o compilador não sabe o que é `File`. Para resolver esse problema, é preciso que você adicione a linha de código a seguir abaixo da primeira linha em `Extensions.cs`:

```
C#  
  
using System.IO;
```

Isso deve resolver o problema e o erro vermelho desaparece.

Em seguida, instrumente a definição de cada consulta com uma mensagem de log:

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Rank Generation")
                        select new { Suit = s, Rank = r
}).LogQuery("Starting Deck");

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();
    var times = 0;
    var shuffle = startingDeck;

    do
    {
        // Out shuffle
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26)
            .LogQuery("Bottom Half"))
            .LogQuery("Shuffle");
        */

        // In shuffle
        shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top
Half"))
            .LogQuery("Shuffle");

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Observe que você não precisa fazer o registro sempre que acessar uma consulta. Você faz o registro ao criar a consulta original. O programa ainda leva muito tempo para ser executado, mas agora você pode ver o motivo. Se você não tiver paciência para executar o embaralhamento interno com o registro em log ativado, volte para o embaralhamento

externo. Você ainda verá os efeitos da avaliação lenta. Em uma execução, ele faz 2592 consultas, incluindo a geração de todos os valores e naipes.

Aqui, você pode melhorar o desempenho do código para reduzir o número de execuções feitas. Uma correção simples possível é *armazenar em cache* os resultados da consulta do LINQ original que constrói o baralho de cartas. Atualmente, você executa as consultas novamente sempre que o loop do-while passa por uma iteração, construindo novamente o baralho de cartas e o embaralhamento de novo todas as vezes. Para armazenar em cache o baralho de cartas, aproveite os métodos LINQ [ToArray](#) e [ToList](#); ao anexá-los às consultas, eles executarão as mesmas ações para quais foram instruídos, mas agora armazenarão os resultados em uma matriz ou lista, dependendo de qual método você optar por chamar. Anexe o método LINQ [ToArray](#) às duas consultas e execute o programa novamente:

C#

```
public static void Main(string[] args)
{
    IEnumerable<Suit>? suits = Suits();
    IEnumerable<Rank>? ranks = Ranks();

    if ((suits is null) || (ranks is null))
        return;

    var startingDeck = (from s in suits.LogQuery("Suit Generation")
                        from r in ranks.LogQuery("Value Generation")
                        select new { Suit = s, Rank = r })
                        .LogQuery("Starting Deck")
                        .ToArray();

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();

    var times = 0;
    var shuffle = startingDeck;

    do
    {
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom
Half"))
            .LogQuery("Shuffle")
            .ToArray();
        */
    }
```

```

        shuffle = shuffle.Skip(26)
            .LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle")
            .ToArray();

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}

```

Agora, o embaralhamento externo contém 30 consultas. Execute novamente com o embaralhamento interno e você verá melhorias semelhantes: agora, executa 162 consultas.

Observe que esse exemplo é **projetado** para realçar os casos de uso em que a avaliação lenta pode causar problemas de desempenho. Embora seja importante ver onde a avaliação lenta pode afetar o desempenho do código, é igualmente importante entender que nem todas as consultas devem ser executadas avidamente. O desempenho incorrido sem usar `ToArrayList` ocorre porque cada nova disposição do baralho de cartas é criada com base na disposição anterior. Usar a avaliação lenta significa que cada nova disposição do baralho é criada do baralho original, até mesmo a execução do código que criou o `startingDeck`. Isso causa uma grande quantidade de trabalho extra.

Na prática, alguns algoritmos funcionam bem usando a avaliação detalhada, e outros executam funcionam melhor usando a avaliação lenta. Para o uso diário, a avaliação lenta é uma opção melhor quando a fonte de dados é um processo separado, como um mecanismo de banco de dados. Para os bancos de dados, a avaliação lenta permite que as consultas mais complexas executem apenas uma viagem de ida e volta para o processo de banco de dados e de volta para o restante do seu código. O LINQ é flexível, não importa se você optar por utilizar a avaliação lenta ou detalhada, portanto, meça seus processos e escolha o tipo de avaliação que ofereça o melhor desempenho.

## Conclusão

Neste projeto, abordamos:

- o uso de consultas LINQ para agregar dados em uma sequência significativa
- a produção de métodos de Extensão para adicionar nossa própria funcionalidade personalizada a consultas LINQ
- a localização de áreas em nosso código nas quais nossas consultas LINQ podem enfrentar problemas de desempenho, como diminuição da velocidade
- avaliação lenta e detalhada com relação às consultas LINQ, e as implicações que elas podem ter no desempenho da consulta

Além do LINQ, você aprendeu um pouco sobre uma técnica usada por mágicos para truques de carta. Os mágicos usam o embaralhamento Faro porque podem controlar onde cada carta fica no baralho. Agora que você sabe, não conte para os outros!

Para saber mais sobre o LINQ, consulte:

- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Introdução ao LINQ](#)
- [Operações de consulta LINQ básica \(C#\)](#)
- [Transformações de dados com LINQ \(C#\)](#)
- [Sintaxe de consulta e sintaxe de método em LINQ \(C#\)](#)
- [funcionalidades do C# que dão suporte a LINQ](#)

# Definir e ler atributos personalizados

Artigo • 20/03/2023

Os atributos fornecem uma maneira de associar informações ao código de forma declarativa. Eles também podem fornecer um elemento reutilizável que pode ser aplicado a vários destinos. Considere o [ObsoleteAttribute](#). Ele pode ser aplicado a classes, structs, métodos, construtores e muito mais. Ele *declara* que o elemento é obsoleto. Em seguida, cabe ao compilador C# procurar esse atributo e realizar alguma ação em resposta.

Neste tutorial, você aprenderá a adicionar atributos a seu código, como criar e usar seus próprios atributos e como usar alguns atributos que são criados no .NET.

## Pré-requisitos

Você precisa configurar seu computador para executar o .NET. Você vai encontrar as instruções de instalação na página de [downloads do .NET](#). Você pode executar esse aplicativo no Windows, Ubuntu Linux, macOS ou em um contêiner do Docker. Você precisa instalar o editor de código da sua preferência. As descrições a seguir usam o [Visual Studio Code](#), que é um editor de software livre entre plataformas. No entanto, você pode usar ferramentas com que esteja familiarizado.

## Criar o aplicativo

Agora que você instalou todas as ferramentas, crie um aplicativo de console .NET. Para usar o gerador de linha de comando, execute o seguinte comando no shell de sua preferência:

CLI do .NET

```
dotnet new console
```

Esse comando cria arquivos de projeto .NET bare-bones. Execute `dotnet restore` para restaurar as dependências necessárias para compilar esse projeto.

Não é necessário executar `dotnet restore`, pois ele é executado implicitamente por todos os comandos que exigem uma restauração, como `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` e `dotnet pack`. Para desabilitar a restauração implícita, use a opção `--no-restore`.

O comando `dotnet restore` ainda é útil em determinados cenários em que realizar uma restauração explícita faz sentido, como [compilações de integração contínua no Azure DevOps Services](#) ou em sistemas de compilação que precisam controlar explicitamente quando a restauração ocorrerá.

Para obter informações sobre como gerenciar feeds do NuGet, confira a [documentação do dotnet restore](#).

Para executar o programa, use `dotnet run`. Você deve ver a saída do "Olá, Mundo" no console.

## Adicionar atributos ao código

No C#, os atributos são classes que herdam da classe base `Attribute`. Qualquer classe que herda de `Attribute` pode ser usada como uma espécie de "marcação" em outras partes do código. Por exemplo, há um atributo chamado `ObsoleteAttribute`. Esse atributo sinaliza que o código está obsoleto e não deve mais ser usado. Coloque este atributo em uma classe, por exemplo, usando colchetes.

```
C#  
  
[Obsolete]  
public class MyClass  
{  
}
```

Embora a classe seja chamada de `ObsoleteAttribute`, só é necessário usar `[Obsolete]` no código. A maioria dos códigos C# segue esta convenção. Você poderá usar o nome completo `[ObsoleteAttribute]` se escolher.

Ao marcar uma classe obsoleta, é uma boa ideia fornecer algumas informações como o motivo de estar obsoleto e/ou o que usar no lugar. Você inclui um parâmetro de cadeia de caracteres para o atributo Obsolete para fornecer essa explicação.

```
C#  
  
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]  
public class ThisClass  
{  
}
```

A cadeia de caracteres está sendo passada como um argumento para um construtor `ObsoleteAttribute`, como se você estivesse escrevendo `var attr = new`

```
ObsoleteAttribute("some string").
```

Os parâmetros para um construtor de atributo são limitados a literais/tipos simples: `bool`, `int`, `double`, `string`, `Type`, `enums`, `etc` e matrizes desses tipos. Você não pode usar uma expressão ou uma variável. Você pode usar parâmetros posicionais ou nomeados.

## Criar seu próprio atributo

Você cria um atributo definindo uma nova classe que herda da classe base `Attribute`.

C#

```
public class MySpecialAttribute : Attribute
{
}
```

Com o código anterior, agora você pode usar `[MySpecial]` (ou `[MySpecialAttribute]`) como um atributo em qualquer lugar na base do código.

C#

```
[MySpecial]
public class SomeOtherClass
{}
```

Os atributos biblioteca de classes base do .NET, como `ObsoleteAttribute`, disparam determinados comportamentos no compilador. No entanto, qualquer atributo que você cria atua como metadados e não resulta em qualquer código dentro da classe de atributo que está sendo executada. Cabe a você agir nesses metadados no seu código.

Há uma pegadinha aqui. Conforme mencionado anteriormente, somente determinados tipos podem ser passados como argumentos ao usar atributos. No entanto, ao criar um tipo de atributo, o compilador C# não impede você de criar esses parâmetros. No exemplo a seguir, você criou um atributo com um construtor que compila corretamente.

C#

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str)
    {
```

```
    }  
}
```

No entanto, não é possível usar esse construtor com a sintaxe de atributo.

C#

```
[Gotcha(new Foo(), "test")] // does not compile  
public class AttributeFail  
{  
}
```

O código anterior causa um erro do compilador como `Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type`

## Como restringir o uso do atributo

Os atributos podem ser usados nos "destinos" a seguir. Os exemplos acima mostram os atributos em classes, mas eles também podem ser usados em:

- Assembly
- Classe
- Construtor
- Delegar
- Enumeração
- Evento
- Campo
- GenericParameter
- Interface
- Método
- Módulo
- Parâmetro
- Propriedade
- ReturnValue
- Estrutura

Quando você cria uma classe de atributo, por padrão, o C# permite que você use esse atributo em qualquer um dos destinos possíveis do atributo. Se quiser restringir seu atributo a determinados destinos, você poderá fazer isso usando o `AttributeUsageAttribute` em sua classe de atributo. É isso mesmo, um atributo em um atributo!

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{
}
```

Se tentar colocar o atributo acima em algo que não é uma classe ou um struct, você obtém um erro do compilador como `Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on 'class, struct' declarations`

C#

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler
    // error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    {
    }
}
```

## Como usar atributos anexados a um elemento de código

Atributos agem como metadados. Sem nenhuma força, eles não fazem nada.

Para localizar e agir sobre os atributos, uma reflexão é necessária. A reflexão permite que você escreva código em C# que examine outro código. Por exemplo, você pode usar a Reflexão para obter informações sobre uma classe (adicione `using System.Reflection;` no início do seu código):

C#

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " +
typeInfo.AssemblyQualifiedName);
```

Isso imprime algo como: `The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`

Depois de ter um objeto `TypeInfo` (ou um `MemberInfo`, `FieldInfo` ou outro objeto), você poderá usar o método `GetCustomAttributes`. Esse método retorna uma coleção de objetos `Attribute`. Você também poderá usar `GetCustomAttribute` e especificar um tipo de atributo.

Aqui está um exemplo do uso de `GetCustomAttributes` em uma instância `MethodInfo` para `MyClass` (que vimos, anteriormente, que tem um atributo `[Obsolete]` nele).

C#

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

Isso imprime no console: `Attribute on MyClass: ObsoleteAttribute`. Tente adicionar outros atributos a `MyClass`.

É importante observar que esses objetos `Attribute` são instanciados lentamente. Ou seja, eles não serão instanciados até que você use `GetCustomAttribute` ou `GetCustomAttributes`. Eles também são instanciados a cada vez. Chamar `GetCustomAttributes` duas vezes em uma linha retornará duas instâncias diferentes do `ObsoleteAttribute`.

## Atributos comuns no runtime

Os atributos são usados por muitas ferramentas e estruturas. O NUnit usa atributos como `[Test]` e `[TestFixture]` que são usados pelo executor de teste NUnit. O ASP.NET MVC usa atributos como `[Authorize]` e fornece uma estrutura de filtro de ação para executar questões abrangentes sobre as ações do MVC. O [PostSharp](#) usa a sintaxe de atributo para permitir a programação em C# orientada ao aspecto.

Aqui estão alguns atributos importantes incorporados às bibliotecas de classes base do .NET Core:

- `[Obsolete]`. Este foi usado nos exemplos acima, e reside no namespace `System`. Isso é útil para fornecer a documentação declarativa sobre uma base de código de alteração. Uma mensagem pode ser fornecida na forma de uma cadeia de caracteres e outro parâmetro booleano pode ser usado para encaminhamento de um aviso do compilador para um erro do compilador.
- `[Conditional]`. Esse atributo está no namespace `System.Diagnostics`. Esse atributo pode ser aplicado aos métodos (ou classes de atributo). Você deve passar uma

cadeia de caracteres para o construtor. Se essa cadeia de caracteres não corresponder a uma diretiva `#define`, o compilador C# removerá todas as chamadas a esse método (mas não o próprio método). Normalmente, você usa essa técnica para fins de depuração (diagnóstico).

- `[CallerMemberName]`. Esse atributo pode ser usado em parâmetros e reside no namespace `System.Runtime.CompilerServices`. `CallerMemberName` é um atributo usado para injetar o nome do método que está chamando outro método. É uma forma de eliminar 'cadeias de caracteres mágicas' ao implementar `INotifyPropertyChanged` em diversas estruturas de interface do usuário. Por exemplo:

C#

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName =
= default!)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }

    private string? _name;
    public string? Name
    {
        get { return _name; }
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged(); // notice that "Name" is not
needed here explicitly
            }
        }
    }
}
```

No código acima, você não precisa ter uma cadeia de caracteres `"Name"` literal. Usar `CallerMemberName` evita erros relacionados à digitação e também possibilita a refatoração/renomeação mais suave. Atributos trazem poder declarativo para C#, mas eles são uma forma de metadados de código e não agem sozinhos.

# Tipos de referência anuláveis

Artigo • 09/05/2023

Em um contexto sem reconhecimento de anulável, todos os tipos de referência eram anuláveis. *Tipos de referência anuláveis* refere-se a um grupo de recursos habilitados em um contexto com reconhecimento de anulável que minimiza a probabilidade de que seu código faça com que o runtime lance [System.NullReferenceException](#). Os *tipos de referência anuláveis* incluem três recursos que ajudam você a evitar essas exceções, incluindo a capacidade de marcar explicitamente um tipo de referência como *anulável*:

- Análise de fluxo estático aprimorada que determina se uma variável pode ser `null` antes de ser desreferenciada.
- Atributos que anotam APIs para que a análise de fluxo determine o *estado nulo*.
- Anotações variáveis que os desenvolvedores usam para declarar explicitamente o *null-state* pretendido para uma variável.

A análise de estado nulo e as anotações variáveis são desabilitadas por padrão para projetos existentes, o que significa que todos os tipos de referência continuam anuláveis. A partir do .NET 6, eles são habilitados por padrão para *novos* projetos. Para obter informações sobre como habilitar esses recursos declarando um *contexto de anotação anulável*, consulte [Contextos anuláveis](#).

O restante deste artigo descreve como essas três áreas de recursos funcionam para produzir avisos quando o código pode estar **desreferenciando** um valor `null`.

Desreferenciar uma variável significa acessar um de seus membros usando o operador `.` (ponto), conforme mostrado no exemplo a seguir:

```
C#
```

```
string message = "Hello, World!";
int length = message.Length; // dereferencing "message"
```

Quando você desreferencia uma variável cujo valor é `null`, o runtime gera um [System.NullReferenceException](#).

Você também pode explorar esses conceitos em nosso módulo do Learn sobre [Segurança de anuláveis em C#](#).

## Análise de estado nulo

A **análise de estado nulo** rastreia o *estado nulo* das referências. Essa análise estática emite avisos quando o código pode desreferenciar `null`. Você pode resolver esses avisos para minimizar as incidências quando o runtime gera um [System.NullReferenceException](#). O compilador usa a análise estática para determinar o *estado nulo* de uma variável. Uma variável é *not-null* ou *maybe-null*. O compilador determina que uma variável é *not-null* de duas maneiras:

1. A variável recebeu um valor conhecido por ser *not-null*.
2. A variável foi verificada em relação a `null` e não foi modificada desde essa verificação.

Qualquer variável que o compilador não tenha determinado como *not-null* é considerada *maybe-null*. A análise fornece avisos em situações em que você pode desreferenciar acidentalmente um valor `null`. O compilador produz avisos com base no *estado nulo*.

- Quando uma variável é *not-null*, essa variável pode ser desreferenciada com segurança.
- Quando uma variável é *maybe-null*, essa variável deve ser verificada para garantir que não seja `null` antes de ser desreferenciada.

Considere o seguinte exemplo:

C#

```
string message = null;

// warning: dereference null.
Console.WriteLine($"The length of the message is {message.Length}");

var originalMessage = message;
message = "Hello, World!";

// No warning. Analysis determined "message" is not null.
Console.WriteLine($"The length of the message is {message.Length}");

// warning!
Console.WriteLine(originalMessage.Length);
```

No exemplo anterior, o compilador determina que `message` é *maybe-null* quando a primeira mensagem é impressa. Não há nenhum aviso para a segunda mensagem. A linha final de código produz um aviso, já que `originalMessage` pode ser nula. O exemplo a seguir mostra um uso mais prático para percorrer uma árvore de nós até a raiz, processando cada nó durante a passagem:

C#

```
void FindRoot(Node node, Action<Node> processNode)
{
    for (var current = node; current != null; current = current.Parent)
    {
        processNode(current);
    }
}
```

O código anterior não gera avisos para desreferenciar a variável `current`. A análise estática determina que `current` nunca é desreferenciada quando é *maybe-null*. A variável `current` é verificada em relação a `null` antes de `current.Parent` ser acessada e antes de passar `current` para a ação `ProcessNode`. Os exemplos anteriores mostram como o compilador determina o *null-state* para variáveis locais quando inicializado, atribuído ou comparado a `null`.

A análise de estado nulo não rastreia os métodos chamados. Como resultado, os campos inicializados em um método auxiliar comum chamado por construtores gerarão um aviso com o seguinte modelo:

A propriedade 'name' não anulável deve conter um valor não nulo ao sair do construtor.

Você pode resolver esses avisos de duas maneiras: *encadeamento de construtores* ou *atributos anuláveis* no método auxiliar. O código a seguir mostra um exemplo de cada um desses casos. A classe `Person` usa um construtor comum chamado por todos os outros construtores. A classe `Student` tem um método auxiliar anotado com o atributo [System.Diagnostics.CodeAnalysis.MemberNotNullAttribute](#):

C#

```
using System.Diagnostics.CodeAnalysis;

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

```
    public Person() : this("John", "Doe") { }

public class Student : Person
{
    public string Major { get; set; }

    public Student(string firstName, string lastName, string major)
        : base(firstName, lastName)
    {
        SetMajor(major);
    }

    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
        SetMajor();
    }

    public Student()
    {
        SetMajor();
    }

    [MemberNotNull(nameof(Major))]
    private void SetMajor(string? major = default)
    {
        Major = major ?? "Undeclared";
    }
}
```

### ① Observação

Foram adicionadas várias melhorias à atribuição definitiva e à análise de estado nulo no C# 10. Ao atualizar para o C# 10, você poderá encontrar menos avisos anuláveis que são falsos positivos. Você pode saber mais sobre as melhorias na [especificação de recursos para melhorias de atribuição definitivas](#).

Análise de estado anulável e os avisos gerados pelo compilador ajudam você a evitar erros de programa ao desreferenciar `null`. O artigo sobre [Como resolver avisos anuláveis](#) fornece técnicas para corrigir os avisos que você provavelmente verá em seu código.

## Atributos em assinaturas de API

A análise de estado nulo precisa de dicas dos desenvolvedores para entender a semântica das APIs. Algumas APIs fornecem verificações nulas e devem alterar o `null`-

*state* de uma variável de *maybe-null* para *not-null*. Outras APIs retornam expressões que são *not-null* ou *maybe-null*, dependendo do *null-state* dos argumentos de entrada. Por exemplo, considere o seguinte código que exibe uma mensagem:

C#

```
public void PrintMessage(string message)
{
    if (!string.IsNullOrWhiteSpace(message))
    {
        Console.WriteLine($"{DateTime.Now}: {message}");
    }
}
```

Com base na inspeção, qualquer desenvolvedor consideraria esse código seguro e não deveria gerar avisos. O compilador não sabe que `IsNullOrEmpty` fornece uma verificação nula. Quando `IsNullOrEmpty` retorna `false`, o *estado nulo* da cadeia de caracteres é *não nulo*. Quando `IsNullOrEmpty` retorna `true`, o *estado nulo* não é alterado. No exemplo anterior, a assinatura inclui o `NotNullWhen` para indicar estado nulo de `message`:

C#

```
public static bool IsNullOrEmpty([NotNullWhen(false)] string message);
```

Os atributos fornecem informações detalhadas sobre o estado nulo de argumentos, valores retornados e membros da instância de objeto usada para invocar um membro. Os detalhes sobre cada atributo podem ser encontrados no artigo de referência de idioma sobre [atributos de referência anuláveis](#). As APIs de runtime do .NET foram todas anotadas no .NET 5. Você aprimora a análise estática anotando suas APIs para fornecer informações semânticas sobre o *null-state* dos argumentos e os valores retornados.

## Anotações de variáveis anuláveis

A análise *null-state* fornece uma análise robusta para a maioria das variáveis. O compilador precisa de mais informações de você para variáveis membro. O compilador não pode fazer suposições sobre a ordem na qual os membros públicos são acessados. Qualquer membro público pode ser acessado em qualquer ordem. Qualquer um dos construtores acessíveis pode ser usado para inicializar o objeto. Se um campo membro for definido como `null`, o compilador deve assumir que seu *null-state* seja *maybe-null* no início de cada método.

Você usa anotações que podem declarar se uma variável é um **tipo de referência anulável** ou um **tipo de referência não anulável**. Essas anotações fazem instruções importantes sobre o *null-state* das variáveis:

- **Uma referência não deve ser nula.** O estado padrão de uma variável de referência não nula é *not-null*. O compilador impõe regras que garantem que seja seguro desreferenciar essas variáveis sem verificar primeiro que ela não está nula:
  - A variável deve ser inicializada para um valor não nulo.
  - A variável nunca pode receber o valor `null`. O compilador emite um aviso quando o código atribui uma expressão *maybe-null* a uma variável que não deve ser nula.
- **Uma referência pode ser nula.** O estado padrão de uma variável de referência anulável é *maybe-null*. O compilador impõe regras para garantir que você tenha verificado corretamente uma referência `null`:
  - A variável só poderá ser desreferenciada quando o compilador puder garantir que o valor não é `null`.
  - Essas variáveis podem ser inicializadas com o valor `null` padrão e receber o valor `null` em outro código.
  - O compilador não emite avisos quando o código atribui uma expressão *maybe-null* a uma variável que pode ser nula.

Qualquer variável de referência que não deveria ser `null` tem um *null-state* de *not-null*. Qualquer variável de referência que possa ser `null` inicialmente tem o *null-state* de *maybe-null*.

Um **tipo de referência que permite valor nulo** é indicado usando a mesma sintaxe que [tipos de valor que permitem valor nulo](#): um `?` é acrescentado ao tipo da variável. Por exemplo, a seguinte declaração de variável representa uma variável de cadeia de caracteres que permite valor nulo, `name`:

C#

```
string? name;
```

Qualquer variável em que o `?` não é acrescentado ao nome do tipo é um **tipo de referência que não permite valor nulo**. Isso inclui todas as variáveis de tipo de referência no código existente quando você habilitou esse recurso. No entanto, todas as variáveis locais tipadas implicitamente (declaradas usando `var`) são **tipos de referência anuláveis**. Como as seções anteriores mostraram, a análise estática determina o *null-state* das variáveis locais para determinar se elas são *maybe-null*.

Às vezes, você deve substituir um aviso quando souber que uma variável não é nula, mas o compilador determina que seu *null-state* é *maybe-null*. Você usa o [operador null-forgiving](#)! seguindo um nome de variável para forçar o *null-state* a ser *not-null*. Por exemplo, se você sabe que a variável `name` não é `null`, mas o compilador emite um aviso, é possível escrever o seguinte código para substituir a análise do compilador:

```
C#  
name!.Length;
```

Tipos de referência anuláveis e tipos de valor anulável fornecem um conceito semântico semelhante: uma variável pode representar um valor ou objeto ou essa variável pode ser `null`. No entanto, tipos de referência anuláveis e tipos de valor anulável são implementados de forma diferente: tipos de valor anuláveis são implementados usando [System.Nullable<T>](#) e tipos de referência anuláveis são implementados por atributos lidos pelo compilador. Por exemplo, `string?` e `string` são ambos representados pelo mesmo tipo: [System.String](#). No entanto, `int?` e `int` são representados por [System.Nullable<System.Int32>](#) e [System.Int32](#), respectivamente.

Tipos de referência anuláveis são um recurso de tempo de compilação. Isso significa que é possível que os chamadores ignorem avisos, usar `null` intencionalmente como um argumento para um método que espera uma referência não anulável. Os autores da biblioteca devem incluir verificações de runtime em relação a valores de argumento nulos. O [ArgumentNullException.ThrowIfNull](#) é a opção preferencial para verificar um parâmetro em relação ao nulo no tempo de execução.

### ⓘ Importante

Habilitar anotações anuláveis pode alterar a forma como o Entity Framework Core determina se um membro de dados é necessário. Você pode conferir mais detalhes no artigo sobre [Conceitos básicos do Entity Framework Core: Trabalhando com tipos de referência anuláveis](#).

## Genéricos

Os genéricos exigem regras detalhadas para lidar com `T?` para qualquer parâmetro de tipo `T`. As regras são necessariamente detalhadas devido ao histórico e à implementação diferente para um tipo de valor anulável e um tipo de referência anulável. [Tipos de valor anuláveis](#) são implementados usando o struct

`System.Nullable<T>`. Tipos de referência anuláveis são implementados como anotações de tipo que fornecem regras semânticas para o compilador.

- Se o argumento de tipo para `T` for um tipo de referência, `T?` referencia o tipo de referência anulável correspondente. Por exemplo, se `T` for um `string`, então `T?` será um `string?`.
- Se o argumento de tipo for `T` um tipo de valor, `T?` faça referência ao mesmo tipo de valor, `T`. Por exemplo, se `T` for um `int`, o `T?` também será um `int`.
- Se o argumento de tipo para `T` for um tipo de referência anulável, `T?` referencia o mesmo tipo de referência anulável correspondente. Por exemplo, se `T` for um `string?`, então `T?` será um `string?`.
- Se o argumento de tipo para `T` for um tipo de valor anulável, `T?` referencia o mesmo tipo de valor anulável correspondente. Por exemplo, se `T` for um `int?`, então `T?` será um `int?`.

Para valores retornados, `T?` é equivalente a `[MaybeNull]T`; para valores de argumento, `T?` é equivalente a `[AllowNull]T`. Para obter mais informações, consulte o artigo sobre [Atributos para análise de null-state](#) na referência da linguagem.

Você pode especificar um comportamento diferente usando [restrições](#):

- A restrição `class` significa que `T` deve ser um tipo de referência não anulável (por exemplo, `string`). O compilador produz um aviso se você usar um tipo de referência anulável, como `string?` para `T`.
- A restrição `class?` significa que `T` deve ser um tipo de referência, não anulável (`string`) ou um tipo de referência anulável (por exemplo, `string?`). Quando o parâmetro de tipo é um tipo de referência anulável, como `string?`, uma expressão de `T?` referencia esse mesmo tipo de referência anulável, como `string?`.
- A restrição `notnull` significa que `T` deve ser um tipo de referência não anulável ou um tipo de valor não anulável. Se você usar um tipo de referência anulável ou um tipo de valor anulável para o parâmetro de tipo, o compilador produzirá um aviso. Além disso, quando `T` é um tipo de valor, o valor retornado é esse tipo de valor, não o tipo de valor anulável correspondente.

Essas restrições ajudam a fornecer mais informações ao compilador sobre como `T` será usado. Isso ajuda quando os desenvolvedores escolhem o tipo para `T`, e fornece uma melhor análise de *null-state* quando uma instância do tipo genérico é usada.

## Contextos que permitem valor nulo

Os novos recursos que protegem contra o lançamento de um [System.NullReferenceException](#) podem ser disruptivos quando ativados em uma base de código existente:

- Todas as variáveis de referência tipadas explicitamente são interpretadas como tipos de referência não anuláveis.
- O significado da restrição `class` em genéricos foi alterado para significar um tipo de referência não anulável.
- Novos avisos são gerados devido a essas novas regras.

Você deve aceitar explicitamente o uso desses recursos em seus projetos existentes. Isso fornece um caminho de migração e preserva a compatibilidade com versões anteriores. Contextos que permitem valor nulo habilitam o controle refinado para a maneira como o compilador interpreta variáveis de tipo de referência. O **contexto de anotação anulável** determina o comportamento do compilador. Há quatro valores para o **contexto de notação anulável**:

- *disable*: o código *não reconhece anulável*.
  - Avisos anuláveis estão desabilitados.
  - Todas as variáveis de tipo de referência são tipos de referência anuláveis.
  - Você não pode declarar uma variável como um tipo de referência anulável usando o sufixo `?` no tipo.
  - Você pode usar o operador tolerante a nulo, `!`, mas ele não tem efeito.
- *enable*: o compilador habilita toda a análise de referência nula e todos os recursos de linguagem.
  - Todos os novos avisos anuláveis estão habilitados.
  - Você pode usar o sufixo `?` para declarar um tipo de referência anulável.
  - Todas as outras variáveis de tipo de referência são tipos de referência não anuláveis.
  - O operador de tolerância a nulo suprime avisos para uma possível atribuição a `null`.
- *warnings*: o compilador executa todas as análises nulas e emite avisos quando o código pode desreferenciar `null`.
  - Todos os novos avisos anuláveis estão habilitados.
  - O uso do sufixo `?` para declarar um tipo de referência anulável produz um aviso.
  - Todas as variáveis de tipo de referência têm permissão para serem nulas. No entanto, os membros têm o *null-state* de *not-null* na chave de abertura de todos os métodos, a menos que declarados com o sufixo `?`.
  - Você pode usar o operador de tolerância a nulo, `!`.

- *annotations*: o compilador não executa análise nula ou avisos de emissão quando o código pode desreferenciar `null`.
  - Todos os avisos anuláveis estão desabilitados.
  - Você pode usar o sufixo `?` para declarar um tipo de referência anulável.
  - Todas as outras variáveis de tipo de referência são tipos de referência não anuláveis.
  - Você pode usar o operador tolerante a nulo, `!`, mas ele não tem efeito.

O contexto de anotação que permite valor nulo e o contexto de aviso que permite valor nulo podem ser definidos para um projeto que usa o elemento `<Nullable>` em seu arquivo `.csproj`. Esse elemento configura como o compilador interpreta a nulidade de tipos e quais avisos são gerados. A tabela a seguir mostra os valores permitidos e resume os contextos que eles especificam.

Contexto	Avisos de desreferência	Avisos de atribuição	Tipos de referência	Sufixo	Operador
<code>disable</code>	Desabilitado	Desabilitado	Todos são anuláveis	<code>?</code>	<code>!</code>
<code>enable</code>	habilitado	habilitado	Não anulável, a menos que declarado com <code>?</code>	Declara tipo anulável	Suprime avisos para uma possível atribuição <code>null</code>
<code>warnings</code>	habilitado	Não aplicável	Todos são anuláveis, mas os membros são considerados <i>não nulos</i> na abertura da chave de métodos	Produz um aviso	Suprime avisos para uma possível atribuição <code>null</code>
<code>annotations</code>	Desabilitado	Desabilitado	Não anulável, a menos que declarado com <code>?</code>	Declara tipo anulável	Não tem efeito

As variáveis de tipo de referência no código compilado em um contexto *disabled* são *nullable-oblivious*. Você pode atribuir uma variável literal `null` ou *maybe-null* a uma variável que seja *nullable-oblivious*. No entanto, o estado padrão de uma variável *nullable-oblivious* é *not-null*.

Você pode escolher qual configuração é melhor para seu projeto:

- Escolha *disable* em projetos herdados que você não deseja atualizar com base no diagnóstico ou em novos recursos.
- Escolha *avisos* para determinar onde seu código pode gerar [System.NullReferenceExceptions](#). Você pode resolver esses avisos antes de modificar o código para habilitar tipos de referência não anuláveis.
- Escolha *annotations* para expressar sua intenção de design antes de habilitar avisos.
- Escolha *enable* para novos projetos e projetos ativos em que você deseja proteger contra exceções de referência nulas.

**Exemplo:**

XML

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

Também é possível usar diretivas para definir esses mesmos contextos em qualquer lugar no código-fonte: Elas são mais úteis quando você está migrando uma base de código grande.

- `#nullable enable`: define o contexto de anotação que permite valor nulo e o contexto de aviso que permite valor nulo como **enable**.
- `#nullable disable`: define o contexto de anotação que permite valor nulo e o contexto de aviso que permite valor nulo como **disable**.
- `#nullable restore`: restaura o contexto de anotação que permite valor nulo e o contexto de aviso que permite valor nulo para as configurações do projeto.
- `#nullable disable warnings`: definir o contexto de aviso anulável para **disable**.
- `#nullable enable warnings`: definir o contexto de aviso anulável para **enable**.
- `#nullable restore warnings`: restaura o contexto de aviso anulável para as configurações do projeto.
- `#nullable disable annotations`: definir o contexto de anotação anulável para **disable**.
- `#nullable enable annotations`: definir o contexto de anotação anulável para **enable**.
- `#nullable restore annotations`: restaura o contexto de aviso de anotação para as configurações do projeto.

Para qualquer linha de código, você pode definir qualquer uma das seguintes combinações:

Contexto de aviso	Contexto de anotação	Use
-------------------	----------------------	-----

Contexto de aviso	Contexto de anotação	Use
Padrão do projeto	Padrão do projeto	Padrão
enable	disable	Corrigir avisos da análise
enable	Padrão do projeto	Corrigir avisos da análise
Padrão do projeto	enable	Adicionar anotações de tipo:
enable	enable	Código já migrado
disable	enable	Anotar código antes de corrigir avisos
disable	disable	Adicionando código herdado ao projeto migrado
Padrão do projeto	disable	Raramente
disable	Padrão do projeto	Raramente

Essas nove combinações fornecem controle refinado sobre os diagnósticos que o compilador emite para seu código. Você pode habilitar mais recursos em qualquer área que estiver atualizando, sem ver avisos adicionais que ainda não está pronto para resolver.

### ⓘ Importante

O contexto global anulável não se aplica aos arquivos de código gerados. Em qualquer das estratégias, o contexto anulável é *desabilitado* para todo arquivo de origem marcado como gerado. Isso significa que todas as APIs em arquivos gerados não são anotadas. Há quatro maneiras de um arquivo ser marcado como gerado:

1. No `.editorconfig`, especifique `generated_code = true` em uma seção que se aplica a esse arquivo.
2. Coloque `<auto-generated>` ou `<auto-generated/>` em um comentário na parte superior do arquivo. Ele pode estar em qualquer linha nesse comentário, mas o bloco de comentários deve ser o primeiro elemento do arquivo.
3. Inicie o nome do arquivo com `TemporaryGeneratedFile_`
4. Termine o nome do arquivo com `.designer.cs`, `.generated.cs`, `.g.cs` ou `.g.i.cs`.

Os geradores podem aceitar usando a diretiva de pré-processador `#nullable`.

Por padrão, os contextos de aviso e de anotação de anulável são **desabilitados**. Isso significa que seu código existente compila sem alterações e sem gerar nenhum aviso

novo. A partir do .NET 6, novos projetos incluem o elemento `<Nullable>enable</Nullable>` em todos os modelos de projeto.

Essas opções fornecem duas estratégias distintas para [atualizar uma base de código existente](#) para usar tipos de referência anuláveis.

## Armadilhas conhecidas

Matrizes e structs que contêm tipos de referência são armadilhas conhecidas em referências anuláveis e na análise estática que determina a segurança nula. Em ambas as situações, uma referência não anulável pode ser inicializada para `null` sem gerar avisos.

## Estruturas

Um struct que contém tipos de referência não anuláveis permite atribuir `default` para ele sem avisos. Considere o seguinte exemplo:

C#

```
using System;

(nullable enable

public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}

public static class Program
{
    public static void PrintStudent(Student student)
    {
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");
        Console.WriteLine($"Middle name: {student.MiddleName?.ToUpper()}");
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");
    }

    public static void Main() => PrintStudent(default);
}
```

No exemplo anterior, não há nenhum aviso em `PrintStudent(default)`, enquanto os tipos de referência não anuláveis `FirstName` e `LastName` são nulos.

Outro caso mais comum é quando você lida com structs genéricos. Considere o seguinte exemplo:

```
C#  
  
#nullable enable  
  
public struct Foo<T>  
{  
    public T Bar { get; set; }  
}  
  
public static class Program  
{  
    public static void Main()  
    {  
        string s = default(Foo<string>).Bar;  
    }  
}
```

No exemplo anterior, a propriedade `Bar` será `null` no tempo de execução e será atribuída a uma cadeia de caracteres não anulável sem avisos.

## Matrizes

Matrizes também são uma armadilha conhecida em tipos de referência anuláveis. Considere o exemplo a seguir que não produz avisos:

```
C#  
  
using System;  
  
#nullable enable  
  
public static class Program  
{  
    public static void Main()  
    {  
        string[] values = new string[10];  
        string s = values[0];  
        Console.WriteLine(s.ToUpper());  
    }  
}
```

No exemplo anterior, a declaração da matriz mostra que ela contém cadeias de caracteres não anuláveis, enquanto seus elementos são inicializados para `null`. Em

seguida, a variável `s` recebe um valor (`null` o primeiro elemento da matriz). Por fim, a variável `s` é desreferenciada, causando uma exceção de runtime.

## Confira também

- [Proposta de tipos de referência anulável](#)
- [Especificação de tipos de referência anuláveis de rascunho](#)
- [Anotações de parâmetro de tipo sem restrição](#)
- [Introdução ao tutorial de referências que permitem valor nulo](#)
- [Anulável \(Opção do compilador C#\)](#)

# Atualizar uma base de código com tipos de referência anuláveis para melhorar os avisos de diagnóstico nulos

Artigo • 07/04/2023

[Tipos de referência anuláveis](#) permitem declarar se variáveis de um tipo de referência devem ou não ser atribuídas a um valor `null`. A análise estática e os avisos do compilador quando seu código pode desreferenciar `null` são o benefício mais importante desse recurso. Depois de habilitado, o compilador gera avisos que ajudam você a evitar a geração de [System.NullReferenceException](#) quando o código é executado.

Se a base de código for relativamente pequena, você poderá ativar o [recurso em seu projeto](#), endereçar avisos e aproveitar os benefícios do diagnóstico aprimorado. Bases de código maiores podem exigir uma abordagem mais estruturada para endereçar avisos ao longo do tempo, habilitando o recurso para alguns, conforme você aborda avisos em diferentes tipos ou arquivos. Este artigo descreve diferentes estratégias para atualizar uma base de código e as compensações associadas a essas estratégias. Antes de iniciar a migração, leia a visão geral conceitual dos [tipos de referência anuláveis](#). Ele aborda a análise estática do compilador, valores do *estado de nulo* iguais a *talvez nulo* e *não nulo* e anotações anuláveis. Depois de conhecer esses conceitos e termos, você estará pronto para migrar seu código.

## Planeje sua migração

Independentemente de como você atualiza sua base de código, a meta é que avisos anuláveis e anotações anuláveis sejam habilitados em seu projeto. Depois de atingir essa meta, você terá a configuração `<nullable>Enable</nullable>` em seu projeto. Você não precisará de nenhuma das diretivas de pré-processador para ajustar as configurações em outro lugar.

A primeira opção é definir o padrão do projeto. Suas opções são:

1. **Anulável desabilitado como padrão:** `Disable` será o padrão se você não adicionar um elemento `Nullable` ao arquivo de projeto. Use esse padrão quando não estiver ativamente adicionando novos arquivos à base de código. A atividade principal é atualizar a biblioteca para usar tipos de referência anuláveis. Usar esse padrão significa que você adiciona uma diretiva de pré-processador anulável a cada arquivo à medida que atualiza o código deles.

2. **Anulável habilitado como padrão:** defina esse padrão quando estiver desenvolvendo ativamente novos recursos. Você deseja que todos os novos códigos beneficiem tipos de referência anuláveis e análise estática anulável. Usar esse padrão significa que você deve adicionar `#nullable disable` à parte superior de cada arquivo. Você removerá essas diretivas de pré-processador ao resolver os avisos em cada arquivo.
3. **Avisos anuláveis como padrão:** escolha esse padrão para uma migração em duas fases. Na primeira fase, aborde os avisos. Na segunda fase, ative as anotações para declarar o *estado de nulo* esperado de uma variável. Usar esse padrão significa que você deve adicionar `#nullable disable` à parte superior de cada arquivo.
4. **Anotações anuláveis** como padrão. Faça anotações no código antes de abordar os avisos.

Habilitar anulável como padrão cria um trabalho mais inicial para adicionar as diretivas de pré-processador a cada arquivo. A vantagem é que cada novo arquivo de código adicionado ao projeto será habilitado para anulável. Qualquer novo trabalho será habilitado para anulável; somente o código existente deve ser atualizado. Desabilitar anulável como o padrão funciona melhor se a biblioteca estiver estável e o foco principal do desenvolvimento for adotar tipos de referência anuláveis. Você ativa tipos de referência anuláveis à medida que anota as APIs. Ao terminar, você habilita os tipos de referência anuláveis para o projeto inteiro. Ao criar um arquivo, você deve adicionar as diretivas de pré-processador e habilitá-lo para anulável. Se algum desenvolvedor em sua equipe esquecer, esse novo código agora estará no backlog do trabalho para tornar todo o código habilitado para anulável.

A sua escolha entre essas estratégias depende da quantidade de desenvolvimento ativo que está ocorrendo em seu projeto. Quanto mais maduro e estável o projeto, melhor será a segunda estratégia. Quanto mais recursos sendo desenvolvidos, melhor será a primeira estratégia.

### ⓘ Importante

O contexto global anulável não se aplica aos arquivos de código gerados. Em qualquer das estratégias, o contexto anulável é *desabilitado* para todo arquivo de origem marcado como gerado. Isso significa que todas as APIs em arquivos gerados não são anotadas. Há quatro maneiras de um arquivo ser marcado como gerado:

1. No `.editorconfig`, especifique `generated_code = true` em uma seção que se aplica a esse arquivo.

2. Coloque `<auto-generated>` ou `<auto-generated/>` em um comentário na parte superior do arquivo. Ele pode estar em qualquer linha nesse comentário, mas o bloco de comentários deve ser o primeiro elemento do arquivo.
3. Inicie o nome do arquivo com `TemporaryGeneratedFile_`
4. Termine o nome do arquivo com `.designer.cs`, `.generated.cs`, `.g.cs` ou `.g.i.cs`.

Os geradores podem aceitar usando a diretiva de pré-processador `#nullable`.

## Entender contextos e avisos

Habilitar avisos e anotações controla como o compilador exibe tipos de referência e nulidade. Cada tipo tem uma entre três nulidades:

- *oblivious*: todos os tipos de referência são *nullable oblivious* quando o contexto de anotação está desabilitado.
- *nonnullable*: um tipo de referência não anotado, `c` é *nonnullable* quando o contexto de anotação está habilitado.
- *nullable*: um tipo de referência anotado, `c?`, é *nullable*, mas um aviso pode ser emitido quando o contexto de anotação está desabilitado. As variáveis declaradas com `var` são *nullable* quando o contexto de anotação está habilitado.

O compilador gera avisos com base nessa nulidade:

- Tipos *nonnullable* geram avisos se um valor potencial `null` é atribuído a eles.
- Tipos *nullable* geram avisos se são desreferenciados quando *talvez nulos*.
- Tipos *oblivious* geram avisos se são desreferenciados quando *talvez nulo* e o contexto de aviso está habilitado.

Cada variável tem um estado anulável padrão que depende de sua nulidade:

- As variáveis nullable têm um *estado de nulo* padrão igual a *talvez nulo*.
- As variáveis non-nullable têm um *estado de nulo* padrão igual a *não nulo*.
- As variáveis nullable oblivious têm um *estado de nulo* padrão igual a *não nulo*.

Antes de habilitar tipos de referência anuláveis, todas as declarações da sua base de código podem ser *nullable oblivious*. Isso é importante porque significa que todos os tipos de referência têm um *estado de nulo* padrão igual a *não nulo*.

## Abordar avisos

Se o projeto usa o Entity Framework Core, você deve ler as diretrizes dele sobre [Como trabalhar com tipos de referência anuláveis](#).

Ao iniciar a migração, você deve começar habilitando apenas avisos. Todas as declarações permanecem *nullable oblivious*, mas você verá avisos quando desreferenciar um valor após o *estado de nulo* dele mudar para *talvez nulo*. Ao abordar esses avisos, você verificará nulo em mais locais e sua base de código se tornará mais resiliente. Para aprender técnicas específicas adequadas a diferentes situações, confira o artigo sobre [Técnicas para resolver avisos anuláveis](#).

Você pode abordar avisos e habilitar anotações em cada arquivo ou classe antes de continuar com outro código. No entanto, geralmente é mais eficiente resolver os avisos gerados enquanto o contexto é *avisos*, antes de habilitar as anotações de tipo. Assim, todos os tipos são *oblivious* até você abordar o primeiro conjunto de avisos.

## Habilitar anotações de tipo

Depois de abordar o primeiro conjunto de avisos, você pode habilitar o *contexto de anotação*. Isso altera os tipos de referência de *oblivious* para *nonnullable*. Todas as variáveis declaradas com `var` são *anuláveis*. Essa alteração geralmente introduz novos avisos. A primeira etapa para abordar os avisos do compilador é usar anotações `?` em tipos de retorno e parâmetro para indicar quando os argumentos ou os valores retornados podem ser `null`. Ao realizar essa tarefa, sua meta não é apenas corrigir avisos. A meta mais importante é fazer com que o compilador entenda sua intenção com os possíveis valores nulos.

## Atributos estendem anotações de tipo

Vários atributos foram adicionados para expressar informações adicionais sobre o estado de nulo das variáveis. As regras das suas APIs provavelmente são mais complicadas do que *não nulas* ou *talvez nulo* para todos os parâmetros e valores retornados. Muitas das suas APIs têm regras mais complexas para quando as variáveis podem ou não ser `null`. Nesses casos, você usará atributos para expressar essas regras. Os atributos que descrevem a semântica da sua API são encontrados no artigo sobre [Atributos que afetam a análise anulável](#).

## Próximas etapas

Depois de resolver todos os avisos após habilitar anotações, você pode definir o contexto padrão do seu projeto como *habilitado*. Se você adicionou pragmas ao seu

código para o contexto de anotações ou avisos anuláveis, você pode removê-los. Com o tempo, você pode ver novos avisos. Você pode escrever um código que introduz avisos. Uma dependência de biblioteca pode ser atualizada para tipos de referência anuláveis. Essas atualizações vão alterar os tipos dessa biblioteca de *nullable oblivious* para *nonnullable* ou *nullable*.

Você também pode explorar esses conceitos em nosso módulo Learn sobre [segurança anulável em C#](#).

# Resolver avisos anuláveis

Artigo • 14/03/2023

Este artigo aborda os seguintes avisos do compilador:

- [CS8597](#) - *Valor gerado pode ser nulo.*
- [CS8600](#) - *Convertendo literal nulo ou possível valor nulo em tipo não anulável.*
- [CS8601](#) - *Possível atribuição de referência nula.*
- [CS8602](#) - *Desreferência de uma referência possivelmente nula.*
- [CS8603](#) - *Possível retorno de referência nula.*
- [CS8604](#) - *Possível argumento de referência nula para parâmetro.*
- [CS8605](#) - *Conversão unboxing de um valor possivelmente nulo.*
- [CS8607](#) - *Um possível valor nulo pode não ser usado para um tipo marcado com `[NotNull]` ou `[DisallowNull]`.*
- [CS8608](#) - *A nulidade de tipos de referência no tipo não corresponde ao membro substituído.*
- [CS8609](#) - *A nulidade de tipos de referência no tipo de retorno não corresponde ao membro substituído.*
- [CS8610](#) - *A nulidade de tipos de referência no parâmetro de tipo não corresponde ao membro substituído.*
- [CS8611](#) - *A nulidade de tipos de referência em tipo de parâmetro não corresponde à declaração de método parcial.*
- [CS8612](#) - *A nulidade de tipos de referência no tipo não corresponde ao membro implementado implicitamente.*
- [CS8613](#) - *A anulabilidade de tipos de referência em tipo de retorno não corresponde ao membro implicitamente implementado.*
- [CS8614](#) - *A anulabilidade de tipos de referência em tipo de parâmetro não corresponde ao membro implicitamente implementado.*
- [CS8615](#) - *A anulabilidade de tipos de referência em tipo não corresponde ao membro implementado.*
- [CS8616](#) - *A anulabilidade de tipos de referência em tipo de retorno não corresponde ao membro implementado.*
- [CS8617](#) - *A anulabilidade de tipos de referência em tipo de parâmetro não corresponde ao membro implementado.*
- [CS8618](#) - *Variável não anulável deve conter um valor não nulo ao sair do construtor. Considere declará-lo como anulável.*
- [CS8619](#) - *A nulidade de tipos de referência no valor não corresponde ao tipo de destino.*
- [CS8620](#) - *O argumento não pode ser usado para o parâmetro devido a diferenças na nulidade dos tipos de referência.*

- **CS8621** - A nulidade de tipos de referência no tipo de retorno não corresponde ao delegado de destino (possivelmente devido a atributos de nulidade).
- **CS8622** - A nulidade de tipos de referência no tipo de parâmetro não corresponde ao delegado de destino (possivelmente devido a atributos de nulidade).
- **CS8624** - O argumento não pode ser usado como uma saída devido a diferenças na nulidade dos tipos de referência.
- **CS8625** - Não é possível converter literal nulo em tipo de referência não anulável.
- **CS8629** - tipo de valor anulável pode ser nulo.
- **CS8631** - O tipo não pode ser usado como parâmetro de tipo no tipo genérico ou método. A nulidade do argumento de tipo não corresponde ao tipo de restrição.
- **CS8633** - A nulidade em restrições para o parâmetro de tipo do método não corresponde às restrições para o parâmetro de tipo do método de interface. Em vez disso, considere usar uma implementação de interface explícita.
- **CS8634** - O tipo não pode ser usado como parâmetro de tipo no tipo genérico ou método. A nulidade do argumento de tipo não corresponde à restrição 'class'.
- **CS8643** - A nulidade dos tipos de referência no especificador de interface explícito não corresponde à interface implementada pelo tipo.
- **CS8644** - O tipo não implementa o membro da interface. A nulidade dos tipos de referência na interface implementados pelo tipo base não corresponde.
- **CS8645** - O membro já está listado na lista de interfaces no tipo com nulidade diferente de tipos de referência.
- **CS8655** - A expressão switch não manipula algumas entradas nulas (ela não é finita).
- **CS8667** - Declarações de método parcial têm nulidade inconsistente em restrições para o parâmetro de tipo.
- **CS8670** - Objeto ou inicializador de coleção desreferencia implicitamente possivelmente membro nulo.
- **CS8714** - O tipo não pode ser usado como parâmetro de tipo no tipo genérico ou método. A nulidade do argumento de tipo não corresponde à restrição 'notnull'.
- **CS8762** - O parâmetro deve ter um valor não nulo ao sair.
- **O método CS8763** - A marcado como `[DoesNotReturn]` não deve retornar.
- **CS8764** - A nulidade do tipo de retorno não corresponde ao membro substituído (possivelmente devido a atributos de nulidade).
- **CS8765** - A nulidade do tipo de parâmetro não corresponde ao membro substituído (possivelmente devido a atributos de nulidade).
- **CS8766** - A nulidade de tipos de referência no tipo de retorno não corresponde ao membro implementado implicitamente (possivelmente devido a atributos de nulidade).
- **CS8767** - A nulidade de tipos de referência no tipo de parâmetro não corresponde ao membro implementado implicitamente (possivelmente devido a atributos de

*nulidade).*

- **CS8768** - A nulidade de tipos de referência no tipo de retorno não corresponde ao membro implementado (possivelmente devido a atributos de nulidade).
- **CS8769** - A nulidade de tipos de referência no tipo de parâmetro não corresponde ao membro implementado (possivelmente devido a atributos de nulidade).
- **CS8770** - O método não tem a anotação `[DoesNotReturn]` para corresponder ao membro implementado ou substituído.
- **CS8774** - O membro deve ter um valor não nulo ao sair.
- **CS8776** - O membro não pode ser usado nesse atributo.
- **CS8775** - O membro deve ter um valor não nulo ao sair.
- **CS8777** - O parâmetro deve ter um valor não nulo ao sair.
- **CS8819** - A nulidade de tipos de referência no tipo de retorno não corresponde à declaração de método parcial.
- **CS8824** - O parâmetro deve ter um valor não nulo ao sair porque o parâmetro não é nulo.
- **CS8825** - O valor retornado precisa ser não nulo porque o parâmetro não é nulo.
- **CS8847** - A expressão switch não manipula algumas entradas nulas (não é exaustiva). No entanto, um padrão com uma cláusula 'when' pode corresponder com êxito a esse valor.

A finalidade dos avisos anuláveis é minimizar a chance do aplicativo gerar `System.NullReferenceException` quando executado. Para atingir essa meta, o compilador usa análise estática e emite avisos quando o código tem constructos que podem gerar exceções de referência nulas. Você fornece ao compilador informações para análise estática aplicando anotações de tipo e atributos. Essas anotações e atributos descrevem a nulidade de argumentos, parâmetros e membros de seus tipos. Neste artigo, você aprenderá diferentes técnicas para abordar os avisos anuláveis gerados pelo compilador a partir de sua análise estática. As técnicas descritas aqui são para o código C# geral. Saiba como trabalhar com tipos de referência anuláveis e o Entity Framework Core em [Trabalhar com tipos de referência anuláveis](#).

Quase todos os avisos serão abordados usando uma das quatro técnicas:

- Adicionando verificações nulas necessárias.
- Adicionando anotações anuláveis `? ou !`.
- Adicionando atributos que descrevem a semântica nula.
- Inicializando variáveis corretamente.

## Possível desreferência do nulo

Esse conjunto de avisos alerta que você está desreferenciando uma variável cujo *estado nulo* é *talvez nulo*. Estes avisos são:

- CS8602 - *Desreferência de uma referência possivelmente nula.*
- CS8670 - *Objeto ou inicializador de coleção desreferencia implicitamente possivelmente membro nulo.*

O seguinte código demonstra um exemplo de cada um dos avisos anteriores:

```
C#  
  
class Container  
{  
    public List<string>? States { get; set; }  
}  
  
internal void PossibleDereferenceNullExamples(string? message)  
{  
    Console.WriteLine(message.Length); // CS8602  
  
    var c = new Container { States = { "Red", "Yellow", "Green" } }; //  
CS8670  
}
```

No exemplo acima, o aviso ocorre porque o `Container`, `c`, pode ter um valor nulo para a propriedade `States`. Atribuir novos estados a uma coleção que pode ser nula causa o aviso.

Para remover esses avisos, é necessário adicionar código para alterar o *estado nulo* dessa variável para *não nulo* antes de desreferenciá-la. O aviso do inicializador de coleção pode ser mais difícil de detectar. O compilador detecta que a coleção *talvez seja nula* quando o inicializador adiciona elementos a ela.

Em muitas instâncias, você pode corrigir esses avisos verificando se uma variável não é nula antes de desreferenciá-la. Por exemplo, o exemplo acima pode ser gravado novamente como:

```
C#  
  
void WriteMessageLength(string? message)  
{  
    if (message is not null)  
    {  
        Console.WriteLine(message.Length);  
    }  
}
```

Outras instâncias ao receber esses avisos podem ser falsas positivas. Você pode ter um método utilitário privado que testa nulo. O compilador não sabe que o método fornece uma verificação nula. Considere o exemplo a seguir que usa um método utilitário privado `IsNotNull`:

```
C#  
  
public void WriteMessage(string? message)  
{  
    if (IsNotNull(message))  
        Console.WriteLine(message.Length);  
}
```

O compilador avisa que você pode estar desreferenciando o nulo ao gravar a propriedade `message.Length` porque sua análise estática determina que `message` pode ser `null`. Você pode saber que `IsNotNull` fornece uma verificação nula e, quando retorna `true`, o *estado nulo* de `message` deve ser *não nulo*. Você deve informar esses fatos ao compilador. Uma maneira é usar o operador tolerante a nulo `!`. Você pode alterar a instrução `WriteLine` para corresponder ao seguinte código:

```
C#  
  
Console.WriteLine(message!.Length);
```

O operador tolerante a nulo torna a expressão *não nula* ainda que ela fosse *talvez nula* sem o `!` aplicado. Neste exemplo, uma solução melhor é a inclusão de um atributo à assinatura de `IsNotNull`:

```
C#  
  
private static bool IsNotNull([NotNullWhen(true)] object? obj) => obj != null;
```

`System.Diagnostics.CodeAnalysis.NotNullWhenAttribute` informa ao compilador que o argumento usado para o parâmetro `obj` é *não nulo* quando o método retorna `true`. Quando o método retorna `false`, o argumento tem o mesmo *estado nulo* que tinha antes do método ser chamado.

### 💡 Dica

Há um conjunto avançado de atributos que podem ser usados para descrever como seus métodos e propriedades afetam o *estado nulo*. Você pode aprender

sobre eles no artigo de referência de idioma em [Análise estática que permite valor nulo](#).

A correção de um aviso para desreferenciar uma variável *talvez nula* envolve uma das três técnicas:

- Adicionar uma verificação nula ausente.
- Adicione atributos de análise nulos em APIs para afetar a análise estática do *estado nulo* do compilador. Esses atributos informam o compilador quando um valor ou argumento retornado deve ser *talvez nulo* ou *não nulo* depois de chamar o método.
- Aplique o operador tolerante a nulo `!` à expressão para forçar o estado para *não nulo*.

## Possível nulo atribuído a uma referência não anulável

Esse conjunto de avisos alerta que você está atribuindo uma variável cujo tipo não pode ser nulo a uma expressão cujo *estado nulo* é *talvez nulo*. Estes avisos são:

- CS8597 - *Valor gerado pode ser nulo*.
- CS8600 - *Convertendo literal nulo ou possível valor nulo em tipo não anulável*.
- CS8601 - *Possível atribuição de referência nula*.
- CS8603 - *Possível retorno de referência nula*.
- CS8604 - *Possível argumento de referência nula para parâmetro*.
- CS8605 - *Conversão unboxing de um valor possivelmente nulo*.
- CS8625 - *Não é possível converter literal nulo em tipo de referência não anulável*.
- CS8629 - *O tipo de valor anulável pode ser nulo*.

O compilador emite esses avisos quando você tenta atribuir uma expressão *talvez nula* a uma variável não anulável. Por exemplo:

C#

```
string? TryGetMessage(int id) => "";

string msg = TryGetMessage(42); // Possible null assignment.
```

Os avisos diferentes fornecem detalhes sobre o código, como atribuição, atribuição de unboxing, instruções de retorno, argumentos para métodos e gerar expressões.

Você pode executar uma das três ações para resolver esses avisos. Uma delas é adicionar a anotação `?` para tornar a variável um tipo de referência anulável. Essa alteração pode gerar outros avisos. Alterar uma variável de uma referência não anulável para uma referência anulável altera o *estado nulo* padrão de *não nulo* para *talvez nulo*. A análise estática do compilador pode encontrar instâncias em que você desreferencia uma variável *talvez nula*.

As outras ações informam ao compilador que o lado direito da atribuição é *não nulo*. A expressão no lado direito pode ser marcada como nula antes da atribuição, conforme mostrado no exemplo a seguir:

C#

```
string notNullMsg = TryGetMessage(42) ?? "Unknown message id: 42";
```

Os exemplos anteriores demonstram a atribuição ao valor retornado de um método. Você pode anotar o método (ou propriedade) para indicar quando um método retorna um valor não nulo. Geralmente

[System.Diagnostics.CodeAnalysis.NotNullIfNotNullAttribute](#) especifica que um valor retornado é *não nulo* quando um argumento de entrada é *não nulo*. Uma alternativa é adicionar o operador tolerante a nulo `!` ao lado direito:

C#

```
string msg = TryGetMessage(42)!;
```

A correção de um aviso para atribuir uma expressão *talvez nula* a uma variável *não nula* envolve uma das quatro técnicas:

- Alterar o lado esquerdo da atribuição para um tipo anulável. Essa ação pode introduzir novos avisos ao desreferenciar essa variável.
- Fornecer uma verificação nula antes da atribuição.
- Anotar a API que produz o lado direito da atribuição.
- Adicionar o operador tolerante a nulo ao lado direito da atribuição.

## Referência não anulável não inicializada

Esse conjunto de avisos alerta que você está atribuindo uma variável cujo tipo não pode ser nulo a uma expressão cujo *estado nulo* é *talvez nulo*. Estes avisos são:

- **CS8618 - Variável não anulável deve conter um valor não nulo ao sair do construtor.** Considere declará-lo como anulável.

- CS8762 - O parâmetro deve ter um valor não nulo ao sair.

Considere a seguinte classe a título de exemplo:

C#

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

`FirstName` ou `LastName` não são garantidos inicializados. Se o código for novo, considere alterar a interface pública. O exemplo acima pode ser atualizado da seguinte maneira:

C#

```
public class Person
{
    public Person(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Caso seja necessário criar um objeto `Person` antes de definir o nome, será possível inicializar as propriedades usando um valor não nulo padrão:

C#

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;
    public string LastName { get; set; } = string.Empty;
}
```

Uma alternativa pode ser alterar esses membros para tipos de referência anuláveis. A classe `Person` poderá ser definida da seguinte maneira se `null` for permitido para o nome:

C#

```
public class Person
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
}
```

O código existente pode exigir outras alterações para informar o compilador sobre a semântica nula para esses membros. Você pode ter criado vários construtores e sua classe pode ter um método auxiliar privado que inicializa um ou mais membros. Você pode mover o código de inicialização para um único construtor e garantir que todos os construtores chamem aquele com o código de inicialização comum. Ou você pode usar os atributos [System.Diagnostics.CodeAnalysis.MemberNotNullAttribute](#) e [System.Diagnostics.CodeAnalysis.MemberNotNullWhenAttribute](#). Esses atributos informam ao compilador que um membro é *não nulo* depois que o método é chamado. O código a seguir mostra um exemplo de cada um desses casos. A classe `Person` usa um construtor comum chamado por todos os outros construtores. A classe `Student` tem um método auxiliar anotado com o atributo [System.Diagnostics.CodeAnalysis.MemberNotNullAttribute](#):

C#

```
using System.Diagnostics.CodeAnalysis;

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public Person() : this("John", "Doe") { }
}

public class Student : Person
{
    public string Major { get; set; }

    public Student(string firstName, string lastName, string major)
        : base(firstName, lastName)
    {
        SetMajor(major);
    }
}
```

```

public Student(string firstName, string lastName) :
    base(firstName, lastName)
{
    SetMajor();
}

public Student()
{
    SetMajor();
}

[MemberNotNull(nameof(Major))]
private void SetMajor(string? major = default)
{
    Major = major ?? "Undeclared";
}
}

```

Por fim, você pode usar o operador tolerante a nulo para indicar que um membro é inicializado em outro código. Para obter outro exemplo, considere as seguintes classes que representam um modelo Entity Framework Core:

C#

```

public class TodoItem
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}

public class TodoContext : DbContext
{
    public TodoContext(DbContextOptions<TodoContext> options)
        : base(options)
    {
    }

    public DbSet<TodoItem> TodoItems { get; set; } = null!;
}

```

A propriedade `DbSet` é inicializada com `null!`. Isso informa ao compilador que a propriedade foi definida como um valor *não nulo*. Na verdade, a base `DbContext` executa a inicialização do conjunto. A análise estática do compilador não capta isso. Para obter mais informações sobre como trabalhar com tipos de referência anuláveis e o Entity Framework Core, consulte o artigo sobre [Como trabalhar com Tipos de Referência Anuláveis no EF Core](#).

A correção de um aviso para não inicializar um membro não anulável envolve uma das quatro técnicas:

- Alterar os construtores ou inicializadores de campo para garantir que todos os membros não anuláveis sejam inicializados.
- Alterar um ou mais membros para serem tipos anuláveis.
- Anotar quaisquer métodos auxiliares para indicar quais membros são atribuídos.
- Adicione um inicializador para `null!` indicar que o membro é inicializado em outro código.

## Incompatibilidade na declaração de nulidade

Muitos avisos indicam incompatibilidades de nulidade entre assinaturas de métodos, delegados ou parâmetros de tipo.

- **CS8608** - *A nulidade de tipos de referência no tipo não corresponde ao membro substituído.*
- **CS8609** - *A nulidade de tipos de referência no tipo de retorno não corresponde ao membro substituído.*
- **CS8610** - *A nulidade de tipos de referência no parâmetro de tipo não corresponde ao membro substituído.*
- **CS8611** - *A nulidade de tipos de referência em tipo de parâmetro não corresponde à declaração de método parcial.*
- **CS8612** - *A nulidade de tipos de referência no tipo não corresponde ao membro implementado implicitamente.*
- **CS8613** - *A nulidade de tipos de referência no tipo de retorno não corresponde ao membro implementado implicitamente.*
- **CS8614** - *A nulidade de tipos de referência no tipo de parâmetro não corresponde ao membro implementado implicitamente.*
- **CS8615** - *A nulidade de tipos de referência no tipo não corresponde ao membro implementado.*
- **CS8616** - *A nulidade de tipos de referência no tipo de retorno não corresponde ao membro implementado.*
- **CS8617** - *A nulidade de tipos de referência no tipo de parâmetro não corresponde ao membro implementado.*
- **CS8619** - *A nulidade de tipos de referência no valor não corresponde ao tipo de destino.*
- **CS8620** - *O argumento não pode ser usado para o parâmetro devido a diferenças na nulidade dos tipos de referência.*
- **CS8621** - *A nulidade de tipos de referência no tipo de retorno não corresponde ao delegado de destino (possivelmente devido a atributos de nulidade).*

- CS8622 - A nulidade de tipos de referência no tipo de parâmetro não corresponde ao delegado de destino (possivelmente devido a atributos de nulidade).
- CS8624 - O argumento não pode ser usado como uma saída devido a diferenças na nulidade dos tipos de referência.
- CS8631 - O tipo não pode ser usado como parâmetro de tipo no tipo genérico ou método. A nulidade do argumento de tipo não corresponde ao tipo de restrição.
- CS8633 - A nulidade em restrições para o parâmetro de tipo do método não corresponde às restrições para o parâmetro de tipo do método de interface. Em vez disso, considere usar uma implementação de interface explícita.
- CS8634 - O tipo não pode ser usado como parâmetro de tipo no tipo genérico ou método. A nulidade do argumento de tipo não corresponde à restrição 'class'.
- CS8643 - A nulidade dos tipos de referência no especificador de interface explícito não corresponde à interface implementada pelo tipo.
- CS8644 - O tipo não implementa o membro da interface. A nulidade de tipos de referência na interface implementada pelo tipo base não corresponde.
- CS8645 - O membro já está listado na lista de interfaces no tipo com nulidade diferente de tipos de referência.
- CS8667 - Declarações de método parcial têm nulidade inconsistente em restrições para o parâmetro de tipo.
- CS8714 - O tipo não pode ser usado como parâmetro de tipo no tipo genérico ou método. A nulidade do argumento de tipo não corresponde à restrição 'notnull'.
- CS8764 - A nulidade do tipo de retorno não corresponde ao membro substituído (possivelmente devido a atributos de nulidade).
- CS8765 - A nulidade do tipo de parâmetro não corresponde ao membro substituído (possivelmente devido a atributos de nulidade).
- CS8766 - A nulidade de tipos de referência no tipo de retorno não corresponde ao membro implementado implicitamente (possivelmente devido a atributos de nulidade).
- CS8767 - A nulidade de tipos de referência no tipo de parâmetro não corresponde ao membro implementado implicitamente (possivelmente devido a atributos de nulidade).
- CS8768 - A nulidade de tipos de referência no tipo de retorno não corresponde ao membro implementado (possivelmente devido a atributos de nulidade).
- CS8769 - A nulidade de tipos de referência no tipo de parâmetro não corresponde ao membro implementado (possivelmente devido a atributos de nulidade).
- CS8819 - A nulidade de tipos de referência no tipo de retorno não corresponde à declaração de método parcial.

O seguinte código demonstra o erro CS8764:

```
public class B
{
    public virtual string GetMessage(string id) => string.Empty;
}
public class D : B
{
    public override string? GetMessage(string? id) => default;
}
```

O exemplo anterior mostra um método `virtual` em uma classe base e um `override` com nulidade diferente. A classe base retorna uma cadeia de caracteres não anulável, mas a classe derivada retorna uma cadeia de caracteres anulável. Se `string` e `string?` forem invertidos, ele será permitido porque a classe derivada é mais restritiva. Da mesma forma, as declarações de parâmetro devem corresponder. Os parâmetros no método de substituição podem permitir o nulo mesmo quando a classe base não permite.

Outras situações podem gerar esses avisos. Você pode ter uma incompatibilidade em uma declaração de método de interface e na implementação desse método. Ou um tipo de delegado e a expressão desse delegado podem ser diferentes. Um parâmetro de tipo e o argumento de tipo podem diferir na nulidade.

Para corrigir esses avisos, atualize a declaração apropriada.

## O código não corresponde à declaração de atributo

As seções anteriores discutiram como você pode usar os [Atributos para análise estática anulável](#) para informar o compilador sobre a semântica nula do código. O compilador avisará se o código não cumprir as promessas desse atributo:

- CS8607 - *Um possível valor nulo pode não ser usado para um tipo marcado com `[NotNull]` ou `[DisallowNull]`*
- O método CS8763 - *A marcado como `[DoesNotReturn]` não deve retornar.*
- CS8770 - *O método não tem a anotação `[DoesNotReturn]` para corresponder ao membro implementado ou substituído.*
- CS8774 - *O membro deve ter um valor não nulo ao sair.*
- CS8775 - *O membro deve ter um valor não nulo ao sair.*
- CS8776 - *O membro não pode ser usado nesse atributo.*
- CS8777 - *O parâmetro deve ter um valor não nulo ao sair.*

- CS8824 - O parâmetro deve ter um valor não nulo ao sair porque o parâmetro não é nulo.
- CS8825 - O valor retornado precisa ser não nulo porque o parâmetro não é nulo.

Considere o método a seguir:

C#

```
public bool TryGetMessage(int id, [NotNullWhen(true)] out string? message)
{
    message = null;
    return true;

}
```

O compilador produz um aviso porque o parâmetro `message` é atribuído `null` e o método retorna `true`. O atributo `NotNullWhen` indica que isso não deve acontecer.

Para resolver esses avisos, atualize o código para que ele corresponda às expectativas dos atributos aplicados. É possível alterar os atributos ou o algoritmo.

## Expressões de switch exaustivas

As expressões de comutador devem ser *exaustivas*, o que significa que todos os valores de entrada devem ser tratados. Mesmo para tipos de referência não anuláveis, o valor `null` deve ser contabilizado. O compilador emite avisos quando o valor nulo não é tratado:

- CS8655 - A expressão `switch` não manipula algumas entradas nulas (ela não é finita).
- CS8847 - A expressão `switch` não manipula algumas entradas nulas (não é exaustiva). No entanto, um padrão com uma cláusula 'when' pode corresponder com êxito a esse valor.

O código de exemplo a seguir demonstra essa condição:

C#

```
int AsScale(string status) =>
    status switch
    {
        "Red" => 0,
        "Yellow" => 5,
        "Green" => 10,
```

```
{ } => -1  
};
```

A expressão de entrada é um `string`, não um `string?`. O compilador ainda gera esse aviso. O padrão `{ }` manipula todos os valores não nulos, mas não corresponde a `null`. Para resolver esses erros, você pode adicionar um caso de `null` explícito ou substituir o `{ }` pelo padrão `_` (descarte). O padrão de descarte corresponde a nulo, bem como a qualquer outro valor.

# Métodos em C#

Artigo • 07/04/2023

Um método é um bloco de código que contém uma série de instruções. Um programa faz com que as instruções sejam executadas chamando o método e especificando os argumentos de método necessários. No C#, todas as instruções executadas são realizadas no contexto de um método. O método `Main` é o ponto de entrada para todos os aplicativos C# e é chamado pelo CLR (Common Language Runtime) quando o programa é iniciado.

## ⓘ Observação

Este tópico aborda os métodos nomeados. Para obter mais informações sobre funções anônimas, consulte [Expressões lambda](#).

## Assinaturas de método

Os métodos são declarados em `class`, `record` ou `struct` especificando:

- Um nível de acesso opcional, como `public` ou `private`. O padrão é `private`.
- Modificadores opcionais como `abstract` ou `sealed`.
- O valor retornado ou `void` se o método não tiver nenhum.
- O nome do método.
- Quaisquer parâmetros de método. Os parâmetros de método estão entre parênteses e separados por vírgulas. Parênteses vazios indicam que o método não requer parâmetros.

Essas partes juntas formam a assinatura do método.

## ⓘ Importante

Um tipo de retorno de um método não faz parte da assinatura do método para fins de sobrecarga de método. No entanto, ele faz parte da assinatura do método ao determinar a compatibilidade entre um delegado e o método para o qual ele aponta.

O exemplo a seguir define uma classe chamada `Motorcycle` que contém cinco métodos:

C#

```

using System;

abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() /* Method statements here */

    // Only derived classes can call this.
    protected void AddGas(int gallons) /* Method statements here */

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) /* Method statements
here */ return 1;

    // Derived classes can override the base class implementation.
    public virtual int Drive(TimeSpan time, int speed) /* Method
statements here */ return 0;

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}

```

A classe `Motorcycle` inclui um método sobrecarregado, `Drive`. Dois métodos têm o mesmo nome, mas devem ser diferenciados por seus tipos de parâmetro.

## Invocação de método

Os métodos podem ser de *instância* ou *estáticos*. Invocar um método de instância requer que você crie uma instância de um objeto e chame o método nesse objeto. Um método de instância opera nessa instância e seus dados. Você invoca um método estático referenciando o nome do tipo ao qual o método pertence; os métodos estáticos não operam nos dados da instância. Tentar chamar um método estático por meio de uma instância do objeto gera um erro do compilador.

Chamar um método é como acessar um campo. Após o nome do objeto (se você estiver chamando um método de instância) ou o nome do tipo (se você estiver chamando um método `static`), adicione um ponto, o nome do método e parênteses. Os argumentos são listados dentro dos parênteses e são separados por vírgulas.

A definição do método especifica os nomes e tipos de quaisquer parâmetros obrigatórios. Quando um chamador invoca o método, ele fornece valores concretos, chamados argumentos, para cada parâmetro. Os argumentos devem ser compatíveis com o tipo de parâmetro, mas o nome do argumento, se for usado no código de chamada, não precisa ser o mesmo do parâmetro nomeado definido no método. No exemplo a seguir, o método `Square` inclui um único parâmetro do tipo `int` chamado *i*.

A primeira chamada do método passa para o método `Square` uma variável do tipo `int` chamada `num`, a segunda, uma constante numérica e a terceira, uma expressão.

```
C#  
  
public class SquareExample  
{  
    public static void Main()  
    {  
        // Call with an int variable.  
        int num = 4;  
        int productA = Square(num);  
  
        // Call with an integer literal.  
        int productB = Square(12);  
  
        // Call with an expression that evaluates to int.  
        int productC = Square(productA * 3);  
    }  
  
    static int Square(int i)  
    {  
        // Store input argument in a local variable.  
        int input = i;  
        return input * input;  
    }  
}
```

A forma mais comum de invocação de método usa argumentos posicionais, ela fornece os argumentos na mesma ordem que os parâmetros de método. Os métodos da classe `Motorcycle`, podem, portanto, ser chamados como no exemplo a seguir. A chamada para o método `Drive`, por exemplo, inclui dois argumentos que correspondem aos dois parâmetros na sintaxe do método. O primeiro se torna o valor do parâmetro `miles`, o segundo o valor do parâmetro `speed`.

```
C#  
  
class TestMotorcycle : Motorcycle  
{  
    public override double GetTopSpeed()  
    {  
        return 108.4;  
    }  
  
    static void Main()  
    {  
        TestMotorcycle moto = new TestMotorcycle();  
  
        moto.StartEngine();  
    }  
}
```

```

        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}

```

Você também pode usar *argumentos nomeados* em vez de argumentos posicionais ao invocar um método. Ao usar argumentos nomeados, você especifica o nome do parâmetro seguido por dois pontos (":") e o argumento. Os argumentos do método podem aparecer em qualquer ordem, desde que todos os argumentos necessários estejam presentes. O exemplo a seguir usa argumentos nomeados para invocar o método `TestMotorcycle.Drive`. Neste exemplo, os argumentos nomeados são passados na ordem oposta da lista de parâmetros do método.

C#

```

using System;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed)
    {
        return (int)Math.Round(((double)miles) / speed, 0);
    }

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        var travelTime = moto.Drive(speed: 60, miles: 170);
        Console.WriteLine("Travel time: approx. {0} hours", travelTime);
    }
}

// The example displays the following output:
//      Travel time: approx. 3 hours

```

Você pode invocar um método usando argumentos posicionais e argumentos nomeados. No entanto, os argumentos posicionais só podem seguir argumentos nomeados quando os argumentos nomeados estiverem nas posições corretas. O

exemplo a seguir invoca o método `TestMotorcycle.Drive` do exemplo anterior usando um argumento posicional e um argumento nomeado.

```
C#
```

```
var travelTime = moto.Drive(170, speed: 55);
```

## Métodos herdados e substituídos

Além dos membros que são definidos explicitamente em um tipo, um tipo herda membros definidos em suas classes base. Como todos os tipos no sistema de tipos gerenciado são herdados direta ou indiretamente da classe `Object`, todos os tipos herdam seus membros, como `Equals(Object)`, `GetType()` e `ToString()`. O exemplo a seguir define uma classe `Person`, instancia dois objetos `Person` e chama o método `Person.Equals` para determinar se os dois objetos são iguais. O método `Equals`, no entanto, não é definido na classe `Person`; ele é herdado do `Object`.

```
C#
```

```
using System;

public class Person
{
    public String FirstName;
}

public class ClassTypeExample
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
        var p2 = new Person();
        p2.FirstName = "John";
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}
// The example displays the following output:
//      p1 = p2: False
```

Tipos podem substituir membros herdados usando a palavra-chave `override` e fornecendo uma implementação para o método substituído. A assinatura do método precisa ser igual à do método substituído. O exemplo a seguir é semelhante ao anterior, exceto que ele substitui o método `Equals(Object)`. (Ele também substitui o método

`GetHashCode()`, uma vez que os dois métodos destinam-se a fornecer resultados consistentes.)

```
C#  
  
using System;  
  
public class Person  
{  
    public String FirstName;  
  
    public override bool Equals(object obj)  
    {  
        var p2 = obj as Person;  
        if (p2 == null)  
            return false;  
        else  
            return FirstName.Equals(p2.FirstName);  
    }  
  
    public override int GetHashCode()  
    {  
        return FirstName.GetHashCode();  
    }  
}  
  
public class Example  
{  
    public static void Main()  
    {  
        var p1 = new Person();  
        p1.FirstName = "John";  
        var p2 = new Person();  
        p2.FirstName = "John";  
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));  
    }  
}  
// The example displays the following output:  
//      p1 = p2: True
```

## Passando parâmetros

Os tipos no C# são *tipos de valor* ou *tipos de referência*. Para obter uma lista de tipos de valor internos, consulte [Tipos](#). Por padrão, os tipos de referência e tipos de valor são passados para um método por valor.

### Passando parâmetros por valor

Quando um tipo de valor é passado para um método por valor, uma cópia do objeto, em vez do próprio objeto, é passada para o método. Portanto, as alterações no objeto do método chamado não têm efeito no objeto original quando o controle retorna ao chamador.

O exemplo a seguir passa um tipo de valor para um método por valor e o método chamado tenta alterar o valor do tipo de valor. Ele define uma variável do tipo `int`, que é um tipo de valor, inicializa o valor para 20 e o passa para um método chamado `ModifyValue` que altera o valor da variável para 30. No entanto, quando o método retorna, o valor da variável permanece inalterado.

C#

```
using System;

public class ByValueExample
{
    public static void Main()
    {
        int value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    static void ModifyValue(int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}

// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

Quando um objeto do tipo de referência é passado para um método por valor, uma referência ao objeto é passada por valor. Ou seja, o método recebe não o objeto em si, mas um argumento que indica o local do objeto. Se você alterar um membro do objeto usando essa referência, a alteração será refletida no objeto quando o controle retornar para o método de chamada. No entanto, substituir o objeto passado para o método não tem efeito no objeto original quando o controle retorna para o chamador.

O exemplo a seguir define uma classe (que é um tipo de referência) chamada `SampleRefType`. Ele cria uma instância de um objeto `SampleRefType`, atribui 44 ao seu campo `value` e passa o objeto para o método `ModifyObject`. Este exemplo faz

essencialmente a mesma coisa que o exemplo anterior: ele passa um argumento por valor para um método. Mas como um tipo de referência é usado, o resultado é diferente. A modificação feita em `ModifyObject` para o campo `obj.value` também muda o campo `value` do argumento, `rt`, no método `Main` para 33, como a saída do exemplo mostra.

```
C#  
  
using System;  
  
public class SampleRefType  
{  
    public int value;  
}  
  
public class ByRefTypeExample  
{  
    public static void Main()  
    {  
        var rt = new SampleRefType();  
        rt.value = 44;  
        ModifyObject(rt);  
        Console.WriteLine(rt.value);  
    }  
  
    static void ModifyObject(SampleRefType obj)  
    {  
        obj.value = 33;  
    }  
}
```

## Passando parâmetros por referência

Você passa um parâmetro por referência quando deseja alterar o valor de um argumento em um método e deseja refletir essa alteração quando o controle retorna para o método de chamada. Para passar um parâmetro por referência, use a palavra-chave `ref` ou `out`. Você também pode passar um valor por referência para evitar a cópia e ainda evitar modificações usando a palavra-chave `in`.

O exemplo a seguir é idêntico ao anterior, exceto que o valor é passado por referência para o método `ModifyValue`. Quando o valor do parâmetro é modificado no método `ModifyValue`, a alteração no valor é refletida quando o controle retorna ao chamador.

```
C#  
  
using System;
```

```

public class ByRefExample
{
    public static void Main()
    {
        int value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(ref value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    static void ModifyValue(ref int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}

// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30

```

Um padrão comum que usa parâmetros pela referência envolve a troca os valores das variáveis. Você passa duas variáveis para um método por referência e o método troca seus conteúdos. O exemplo a seguir troca valores inteiros.

C#

```

using System;

public class RefSwapExample
{
    static void Main()
    {
        int i = 2, j = 3;
        System.Console.WriteLine("i = {0}  j = {1}" , i, j);

        Swap(ref i, ref j);

        System.Console.WriteLine("i = {0}  j = {1}" , i, j);
    }

    static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
}

// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2

```

Passar um parâmetro de tipo de referência permite que você altere o valor da própria referência, em vez de o valor de seus campos ou elementos individuais.

## Matrizes de parâmetros

Às vezes, o requisito de que você especifique o número exato de argumentos para o método é restritivo. Usando a palavra-chave `params` para indicar que um parâmetro é uma matriz de parâmetros, você permite que o método seja chamado com um número variável de argumentos. O parâmetro marcado com a palavra-chave `params` deve ser um tipo de matriz e ele deve ser o último parâmetro na lista de parâmetros do método.

Um chamador pode, então, invocar o método de uma das quatro maneiras:

- Passando uma matriz do tipo apropriado que contém o número de elementos desejado.
- Passando uma lista separada por vírgulas de argumentos individuais do tipo apropriado para o método.
- Passando `null`.
- Não fornecendo um argumento para a matriz de parâmetros.

O exemplo a seguir define um método chamado `GetVowels` que retorna todas as vogais de uma matriz de parâmetros. O método `Main` ilustra todas as quatro maneiras de invocar o método. Os chamadores não precisam fornecer argumentos para parâmetros que incluem o modificador `params`. Nesse caso, o parâmetro é uma matriz vazia.

C#

```
using System;
using System.Linq;

class ParamsExample
{
    static void Main()
    {
        string fromArray = GetVowels(new[] { "apple", "banana", "pear" });
        Console.WriteLine($"Vowels from array: '{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments:
'{fromMultipleArguments}'");

        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }
}
```

```

static string GetVowels(params string[] input)
{
    if (input == null || input.Length == 0)
    {
        return string.Empty;
    }

    var vowels = new char[] { 'A', 'E', 'I', 'O', 'U' };
    return string.Concat(
        input.SelectMany(
            word => word.Where(letter =>
vowels.Contains(char.ToUpper(letter)))));
}
}

// The example displays the following output:
//      Vowels from array: 'aeaaaaea'
//      Vowels from multiple arguments: 'aeaaaaea'
//      Vowels from null: ''
//      Vowels from no value: ''

```

## Parâmetros e argumentos opcionais

Uma definição de método pode especificar que os parâmetros são obrigatórios ou que são opcionais. Por padrão, os parâmetros são obrigatórios. Os parâmetros opcionais são especificados incluindo o valor padrão do parâmetro na definição do método. Quando o método for chamado, se nenhum argumento for fornecido para um parâmetro opcional, o valor padrão será usado em vez disso.

O valor padrão do parâmetro deve ser atribuído por um dos tipos de expressões a seguir:

- Uma constante, como um número ou uma cadeia de caracteres literal.
- Uma expressão do formulário `default(SomeType)`, em que `SomeType` pode ser um tipo de valor ou um tipo de referência. Se for um tipo de referência, ele será efetivamente o mesmo que especificar `null`. Você pode usar o literal `default`, pois o compilador pode inferir o tipo a partir da declaração do parâmetro.
- Uma expressão da forma `new ValType()`, em que `ValType` é um tipo de valor. Ela invoca o construtor sem parâmetros implícito do tipo de valor, que não é de fato um membro do tipo.

 Observação

No C# 10 e posterior, quando uma expressão do formulário `new ValueType()` invoca o construtor sem parâmetros definido explicitamente de um tipo de valor, o compilador gera um erro, pois o valor do parâmetro padrão deve ser uma constante de tempo de compilação. Use a expressão `default(ValueType)` ou o literal `default` para fornecer o valor padrão do parâmetro. Para obter mais informações sobre construtores sem parâmetros, consulte a seção [Inicialização de struct e valores padrão](#) do artigo [Tipos de estrutura](#).

Se um método inclui parâmetros obrigatórios e opcionais, os parâmetros opcionais são definidos no final da lista de parâmetros, após todos os parâmetros obrigatórios.

O exemplo a seguir define um método, `ExampleMethod`, que tem um parâmetro obrigatório e dois opcionais.

C#

```
using System;

public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                             string? description = default)
    {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} =
{required + optionalInt}";
        Console.WriteLine(msg);
    }
}
```

Se um método com vários argumentos opcionais for invocado usando argumentos posicionais, o chamador deverá fornecer um argumento para todos os parâmetros opcionais do primeiro ao último para o qual um argumento é fornecido. No caso do método `ExampleMethod`, por exemplo, se o chamador fornecer um argumento para o parâmetro `description`, ele deverá fornecer também um para o parâmetro `optionalInt`. `opt.ExampleMethod(2, 2, "Addition of 2 and 2");` é uma chamada de método válida, `opt.ExampleMethod(2, , "Addition of 2 and 0");` gera um erro do compilador de “Argumento ausente”.

Se um método for chamado usando argumentos nomeados ou uma combinação de argumentos posicionais e nomeados, o chamador poderá omitir todos os argumentos após o último argumento posicional na chamada do método.

A exemplo a seguir chama o método `ExampleMethod` três vezes. As duas primeiras chamadas de método usam argumentos posicionais. O primeiro omite ambos os

argumentos opcionais, enquanto o segundo omite o último argumento. A terceira chamada de método fornece um argumento posicional para o parâmetro obrigatório, mas usa um argumento nomeado para fornecer um valor para o parâmetro `description` enquanto omite o argumento `optionalInt`.

C#

```
public class OptionsExample
{
    public static void Main()
    {
        var opt = new Options();
        opt.ExampleMethod(10);
        opt.ExampleMethod(10, 2);
        opt.ExampleMethod(12, description: "Addition with zero:");
    }
}
// The example displays the following output:
//      N/A: 10 + 0 = 10
//      N/A: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

O uso de parâmetros opcionais afeta a *resolução de sobrecarga* ou a maneira em que o compilador C# determina qual sobrecarga específica deve ser invocada pela invocada de método, da seguinte maneira:

- Um método, indexador ou construtor é um candidato para a execução se cada um dos parâmetros é opcional ou corresponde, por nome ou posição, a um único argumento na instrução de chamada e esse argumento pode ser convertido para o tipo do parâmetro.
- Se mais de um candidato for encontrado, as regras de resolução de sobrecarga de conversões preferenciais serão aplicadas aos argumentos que são especificados explicitamente. Os argumentos omitidos para parâmetros opcionais são ignorados.
- Se dois candidatos são considerados igualmente bons, a preferência vai para um candidato que não tenha parâmetros opcionais para os quais argumentos foram omitidos na chamada. Esta é uma consequência da preferência geral na resolução de sobrecarga de candidatos que têm menos parâmetros.

## Valores retornados

Os métodos podem retornar um valor para o chamador. Se o tipo de retorno (o tipo listado antes do nome do método) não for `void`, o método poderá retornar o valor usando a palavra-chave `return`. Uma instrução com a palavra-chave `return` seguida por uma variável, constante ou expressão que corresponde ao tipo de retorno retornará

esse valor para o chamador do método. Métodos com um tipo de retorno não nulo devem usar a palavra-chave `return` para retornar um valor. A palavra-chave `return` também interrompe a execução do método.

Se o tipo de retorno for `void`, uma instrução `return` sem um valor ainda será útil para interromper a execução do método. Sem a palavra-chave `return`, a execução do método será interrompida quando chegar ao final do bloco de código.

Por exemplo, esses dois métodos usam a palavra-chave `return` para retornar inteiros:

```
C#  
  
class SimpleMath  
{  
    public int AddTwoNumbers(int number1, int number2)  
    {  
        return number1 + number2;  
    }  
  
    public int SquareANumber(int number)  
    {  
        return number * number;  
    }  
}
```

Para usar um valor retornado de um método, o método de chamada pode usar a chamada de método em si em qualquer lugar que um valor do mesmo tipo seria suficiente. Você também pode atribuir o valor retornado a uma variável. Por exemplo, os dois exemplos de código a seguir obtêm a mesma meta:

```
C#  
  
int result = obj.AddTwoNumbers(1, 2);  
result = obj.SquareANumber(result);  
// The result is 9.  
Console.WriteLine(result);
```

```
C#  
  
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));  
// The result is 9.  
Console.WriteLine(result);
```

Usar uma variável local, nesse caso, `result`, para armazenar um valor é opcional. Isso pode ajudar a legibilidade do código ou pode ser necessário se você precisar armazenar o valor original do argumento para todo o escopo do método.

Às vezes, você deseja que seu método retorne mais de um único valor. Você pode fazer isso facilmente usando *tipos de tupla* e *literais de tupla*. O tipo de tupla define os tipos de dados dos elementos da tupla. Os literais de tupla fornecem os valores reais da tupla retornada. No exemplo a seguir, `(string, string, string, int)` define o tipo de tupla que é retornado pelo método `GetPersonalInfo`. A expressão `(per.FirstName, per.MiddleName, per.LastName, per.Age)` é a tupla literal, o método retorna o nome, o nome do meio e o sobrenome, juntamente com a idade, de um objeto `PersonInfo`.

C#

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

O chamador pode então consumir a tupla retornada com o código semelhante ao seguinte:

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

Os nomes também podem ser atribuídos aos elementos da tupla na definição de tipo de tupla. O exemplo a seguir mostra uma versão alternativa do método `GetPersonalInfo` que usa elementos nomeados:

C#

```
public (string FName, string MName, string LName, int Age)
GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

A chamada anterior para o método `GetPersonalInfo` pode ser modificada da seguinte maneira:

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

Se um método passa uma matriz como um argumento e modifica o valor de elementos individuais, o método não precisa retornar a matriz, embora você possa optar por fazer isso para obter um bom estilo ou um fluxo de valores funcional. Isso ocorre porque o C# passa todos os tipos de referência por valor e o valor de uma referência de matriz é o ponteiro para a matriz. No exemplo a seguir, as alterações no conteúdo da matriz `values` realizados pelo método `DoubleValues` são observáveis por qualquer código que faz referência à matriz.

C#

```
using System;

public class ArrayValueExample
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8 };
        DoubleValues(values);
        foreach (var value in values)
            Console.Write("{0} ", value);
    }

    public static void DoubleValues(int[] arr)
    {
        for (int ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
            arr[ctr] = arr[ctr] * 2;
    }
}

// The example displays the following output:
//      4 8 12 16
```

## Métodos de extensão

Normalmente, há duas maneiras de adicionar um método a um tipo existente:

- Modificar o código-fonte para esse tipo. Você não pode fazer isso, é claro, se não tiver o código-fonte do tipo. E isso se torna uma alteração significativa se você também adicionar campos de dados privados para dar suporte ao método.
- Definir o novo método em uma classe derivada. Não é possível adicionar um método dessa forma usando a herança para outros tipos, como estruturas e enumerações. Isso também não pode ser usado para “adicionar” um método a uma classe selada.

Os métodos de extensão permitem que você “adicione” um método a um tipo existente sem modificar o tipo em si ou implementar o novo método em um tipo herdado. O

método de extensão também não precisa residir no mesmo assembly do tipo que ele estende. Você chama um método de extensão como se fosse um membro definido de um tipo.

Para obter mais informações, consulte [Métodos de extensão](#).

## Métodos assíncronos

Usando o recurso `async`, você pode invocar métodos assíncronos sem usar retornos de chamada explícitos ou dividir manualmente seu código entre vários métodos ou expressões lambda.

Se marcar um método com o modificador `async`, você poderá usar o operador `await` no método. Quando o controle atingir uma expressão `await` no método assíncrono, o controle retornará para o chamador se a tarefa aguardada não estiver concluída, e o progresso no método com a palavra-chave `await` será suspenso até a tarefa aguardada ser concluída. Quando a tarefa for concluída, a execução poderá ser retomada no método.

### ⓘ Observação

Um método assíncrono retorna para o chamador quando encontra o primeiro objeto esperado que ainda não está completo ou chega ao final do método assíncrono, o que ocorrer primeiro.

Um método assíncrono normalmente tem um tipo de retorno `Task<TResult>`, `Task`, `IAsyncEnumerable<T>` ou `void`. O tipo de retorno `void` é usado principalmente para definir manipuladores de eventos, nos quais o tipo de retorno `void` é necessário. Um método assíncrono que retorna `void` não pode ser aguardado e o chamador de um método de retorno nulo não pode capturar as exceções que esse método gera. Um método assíncrono pode ter [qualquer tipo de retorno como os de tarefa](#).

No exemplo a seguir, `DelayAsync` é um método assíncrono que contém uma instrução `return` que retorna um inteiro. Como é um método assíncrono, sua declaração de método deve ter um tipo de retorno `Task<int>`. Como o tipo de retorno é `Task<int>`, a avaliação da expressão `await` em `DoSomethingAsync` produz um inteiro, como a instrução `int result = await delayTask` a seguir demonstra.

C#

```

class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}

// Example output:
//   Result: 5

```

Um método assíncrono não pode declarar os parâmetros `in`, `ref` nem `out`, mas pode chamar métodos que tenham esses parâmetros.

Para obter mais informações sobre os métodos assíncronos, consulte [Programação assíncrona com `async` e `await`](#) e [Tipos de retorno Async](#).

## Membros aptos para expressão

É comum ter definições de método que simplesmente retornam imediatamente com o resultado de uma expressão ou que têm uma única instrução como o corpo do método. Há um atalho de sintaxe para definir esses métodos usando `=>`:

C#

```

public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);

```

Se o método retornar `void` ou for um método assíncrono, o corpo do método deverá ser uma expressão de instrução (igual aos lambdas). Para propriedades e indexadores, eles devem ser somente leitura e você não usa a palavra-chave do acessador `get`.

## Iterators

Um iterador realiza uma iteração personalizada em uma coleção, como uma lista ou uma matriz. Um iterador usa a instrução `yield return` para retornar um elemento de cada vez. Quando uma instrução `yield return` for atingida, o local atual será lembrado para que o chamador possa solicitar o próximo elemento na sequência.

O tipo de retorno de um iterador pode ser `IEnumerable`, `IEnumerable<T>`, `IAsyncEnumerable<T>`, `IEnumerator` ou `IEnumerator<T>`.

Para obter mais informações, consulte [Iteradores](#).

## Confira também

- [Modificadores de acesso](#)
- [Classes static e membros de classes static](#)
- [Herança](#)
- [Classes e membros de classes abstract e sealed](#)
- [params](#)
- [out](#)
- [ref](#)
- [Em](#)
- [Passando parâmetros](#)

# Propriedades

Artigo • 07/04/2023

As propriedades são cidadãos de primeira classe no C#. A linguagem define uma sintaxe que permite aos desenvolvedores escrever código que expresse sua intenção de design com precisão.

As propriedades se comportam como campos quando são acessadas. No entanto, diferentemente dos campos, as propriedades são implementadas com acessadores, que definem as instruções que são executadas quando uma propriedade é acessada ou atribuída.

## Sintaxe de propriedade

A sintaxe para propriedades é uma extensão natural para os campos. Um campo define um local de armazenamento:

```
C#  
  
public class Person  
{  
    public string? FirstName;  
  
    // Omitted for brevity.  
}
```

Uma definição de propriedade contém declarações para um acessador `get` e `set` que recupera e atribui o valor dessa propriedade:

```
C#  
  
public class Person  
{  
    public string? FirstName { get; set; }  
  
    // Omitted for brevity.  
}
```

A sintaxe mostrada acima é a sintaxe da *propriedade automática*. O compilador gera o local de armazenamento para o campo que dá suporte à propriedade. O compilador também implementa o corpo dos acessadores `get` e `set`.

Às vezes, você precisa inicializar uma propriedade para um valor diferente do padrão para seu tipo. O C# permite isso definindo um valor após a chave de fechamento da propriedade. Você pode preferir que o valor inicial para a propriedade `FirstName` seja a cadeia de caracteres vazia em vez de `null`. Você deve especificar isso conforme mostrado abaixo:

```
C#  
  
public class Person  
{  
    public string FirstName { get; set; } = string.Empty;  
  
    // Omitted for brevity.  
}
```

A inicialização específica é mais útil para propriedades somente leitura, como você verá adiante neste artigo.

Você mesmo também pode definir o armazenamento, conforme mostrado abaixo:

```
C#  
  
public class Person  
{  
    public string? FirstName  
    {  
        get { return _firstName; }  
        set { _firstName = value; }  
    }  
    private string? _firstName;  
  
    // Omitted for brevity.  
}
```

Quando uma implementação de propriedade é uma única expressão, você pode usar *membros aptos para expressão* para o getter ou setter:

```
C#  
  
public class Person  
{  
    public string? FirstName  
    {  
        get => _firstName;  
        set => _firstName = value;  
    }  
    private string? _firstName;
```

```
// Omitted for brevity.  
}
```

Essa sintaxe simplificada será usada quando aplicável ao longo deste artigo.

A definição da propriedade mostrada acima é uma propriedade de leitura/gravação. Observe a palavra-chave `value` no acessador `set`. O acessador `set` sempre tem um parâmetro único chamado `value`. O acessador `get` deve retornar um valor que seja conversível para o tipo da propriedade (`string`, neste exemplo).

Essas são as noções básicas sobre a sintaxe. Há muitas variações diferentes que oferecem suporte a uma variedade de linguagens de design diferentes. Vamos explorá-las e conhecer as opções de sintaxe para cada uma.

## Validação

Os exemplos acima mostraram um dos casos mais simples de definição de propriedade: uma propriedade de leitura/gravação sem validação. Ao escrever o código que você deseja nos acessadores `get` e `set`, você pode criar vários cenários diferentes.

Você pode escrever código no acessador `set` para garantir que os valores representados por uma propriedade sejam sempre válidos. Por exemplo, suponha que uma regra para a classe `Person` é que o nome não pode ser um espaço em branco. Você escreveria isso da seguinte maneira:

```
C#  
  
public class Person  
{  
    public string? FirstName  
    {  
        get => _firstName;  
        set  
        {  
            if (string.IsNullOrWhiteSpace(value))  
                throw new ArgumentException("First name must not be blank");  
            _firstName = value;  
        }  
    }  
    private string? _firstName;  
  
    // Omitted for brevity.  
}
```

O exemplo anterior pode ser simplificado usando uma expressão `throw` como parte da validação do setter da propriedade:

```
C#  
  
public class Person  
{  
    public string? FirstName  
    {  
        get => _firstName;  
        set => _firstName = (!string.IsNullOrWhiteSpace(value)) ? value :  
throw new ArgumentException("First name must not be blank");  
    }  
    private string? _firstName;  
  
    // Omitted for brevity.  
}
```

O exemplo acima aplica a regra de que o nome não pode ser em branco ou espaço em branco. Se um desenvolvedor escreve

```
C#  
  
hero.FirstName = "";
```

Essa atribuição lança uma `ArgumentException`. Como um acessador set de propriedade deve ter um tipo de retorno `void`, você relata erros no acessador set lançando uma exceção.

Você pode estender essa mesma sintaxe para qualquer coisa necessária em seu cenário. Você pode verificar as relações entre diferentes propriedades ou validar em relação a qualquer condição externa. Todas as instruções de C# válidas são válidas em um acessador de propriedade.

## Controle de acesso

Até aqui, todas as definições de propriedade que você viu são de propriedades de leitura/gravação com acessadores públicos. Essa não é a única acessibilidade válida para as propriedades. Você pode criar propriedades somente leitura ou dar acessibilidade diferente aos acessadores `get` e `set`. Suponha que sua classe `Person` só deva habilitar a alteração do valor da propriedade `FirstName` em outros métodos naquela classe. Você pode dar acessibilidade `private` ao acessador `set`, em vez de `public`:

```
C#
```

```
public class Person
{
    public string? FirstName { get; private set; }

    // Omitted for brevity.
}
```

Agora, a propriedade `FirstName` pode ser acessada de qualquer código, mas só pode ser atribuída de outro código na classe `Person`.

Você pode adicionar qualquer modificador de acesso restritivo aos acessadores `get` ou `set`. Nenhum modificador de acesso que você colocar no acessador individual deve ser mais limitado que o modificador de acesso da definição de propriedade. O que está acima é válido porque a propriedade `FirstName` é `public`, mas o acessador `set` é `private`. Você não poderia declarar uma propriedade `private` com um acessador `public`. As declarações de propriedade também podem ser declaradas `protected`, `internal`, `protected internal` ou até mesmo `private`.

Também é válido colocar o modificador mais restritivo no acessador `get`. Por exemplo, você poderia ter uma propriedade `public`, mas restringir o acessador `get` como `private`. Esse cenário raramente acontece na prática.

## Somente leitura

Você também pode restringir modificações a uma propriedade para que ela possa ser definida somente em um construtor. Você pode modificar a classe `Person` da seguinte maneira:

C#

```
public class Person
{
    public Person(string firstName) => FirstName = firstName;

    public string FirstName { get; }

    // Omitted for brevity.
}
```

## Somente init

O exemplo anterior requer que os chamadores usem o construtor que inclui o parâmetro `FirstName`. Os chamadores não podem usar [inicializadores de objeto](#) para atribuir um valor à propriedade. Para dar suporte a inicializadores, você pode transformar o `set` em um `init`, conforme mostrado no seguinte código:

C#

```
public class Person
{
    public Person() { }
    public Person(string firstName) => FirstName = firstName;

    public string? FirstName { get; init; }

    // Omitted for brevity.
}
```

O exemplo anterior permite que um chamador crie um `Person` usando o construtor padrão, mesmo que esse código não defina a propriedade `FirstName`. Começando com o C# 11, você pode *exigir* que os chamadores definam essa propriedade:

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName) => FirstName = firstName;

    public required string FirstName { get; init; }

    // Omitted for brevity.
}
```

O código anterior faz duas adições à classe `Person`. Primeiro, a declaração de propriedade `FirstName` inclui o modificador `required`. Isso significa que qualquer código que cria um novo `Person` deve definir essa propriedade. Em segundo lugar, o construtor que usa um parâmetro `firstName` tem o atributo [System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute](#). Esse atributo informa ao compilador que esse construtor define *todos* `required` os membros.

 **Importante**

Não confunda `required` com *não anulável*. É válido definir uma propriedade `required` como `null` ou `default`. Se o tipo for não anulável, como `string` nesses exemplos, o compilador emitirá um aviso.

Os chamadores devem usar o construtor com `SetsRequiredMembers` ou definir a propriedade `FirstName` usando um inicializador de objeto, conforme mostrado no seguinte código:

C#

```
var person = new VersionNinePoint2.Person("John");
person = new VersionNinePoint2.Person{ FirstName = "John"};
// Error CS9035: Required member `Person.FirstName` must be set:
//person = new VersionNinePoint2.Person();
```

## Propriedades computadas

Uma propriedade não precisa simplesmente retornar o valor de um campo de membro. Você pode criar propriedades que retornam um valor computado. Vamos expandir o objeto `Person` para retornar o nome completo, computado pela concatenação dos nomes e sobrenomes:

C#

```
public class Person
{
    public string? FirstName { get; set; }

    public string? LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```

O exemplo acima usa o recurso de [interpolação de cadeia de caracteres](#) para criar a cadeia de caracteres formatada do nome completo.

Use também um *membro com corpo da expressão*, que fornece uma maneira mais sucinta de criar a propriedade `FullName` computada:

C#

```
public class Person
{
    public string? FirstName { get; set; }
```

```
    public string? LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Os *membros com corpo da expressão* usam a sintaxe *expressão lambda* para definir métodos que contêm uma única expressão. Aqui, essa expressão retorna o nome completo do objeto person.

## Propriedades avaliadas armazenadas em cache

Combine o conceito de uma propriedade computada com o armazenamento e crie uma *propriedade avaliada armazenada em cache*. Por exemplo, você poderia atualizar a propriedade `FullName` para que a formatação da cadeia de caracteres só acontecesse na primeira vez que ela foi acessada:

```
C#

public class Person
{
    public string? FirstName { get; set; }

    public string? LastName { get; set; }

    private string? _fullName;
    public string FullName
    {
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

No entanto, o código acima contém um bug. Se o código atualizar o valor das propriedades `FirstName` OU `LastName`, o campo `fullName`, anteriormente avaliado, será inválido. Modifique os acessadores `set` das propriedades `FirstName` e `LastName` para que o campo `fullName` seja calculado novamente:

```
C#

public class Person
{
    private string? _firstName;
```

```

public string? FirstName
{
    get => _firstName;
    set
    {
        _firstName = value;
        _fullName = null;
    }
}

private string? _lastName;
public string? LastName
{
    get => _lastName;
    set
    {
        _lastName = value;
        _fullName = null;
    }
}

private string? _fullName;
public string FullName
{
    get
    {
        if (_fullName is null)
            _fullName = $"{FirstName} {LastName}";
        return _fullName;
    }
}
}

```

Esta versão final avalia a propriedade `FullName` apenas quando necessário. Se a versão calculada anteriormente for válida, ela será usada. Se outra alteração de estado invalidar a versão calculada anteriormente, ela será recalculada. Os desenvolvedores que usam essa classe não precisam saber dos detalhes da implementação. Nenhuma dessas alterações internas afetam o uso do objeto `Person`. Esse é o motivo principal para o uso de propriedades para expor os membros de dados de um objeto.

## Anexando atributos a propriedades autoimplementadas

Os atributos de campo podem ser anexados ao campo de backup gerado pelo compilador em propriedades implementadas automaticamente. Por exemplo, considere uma revisão da classe `Person` que adiciona uma propriedade `Id` de inteiro exclusivo. A propriedade `Id` é escrita usando uma propriedade autoimplementada, mas o design

não exige a persistência da propriedade `Id`. O `NonSerializedAttribute` pode ser anexado apenas a campos, não a propriedades. Anexe o `NonSerializedAttribute` ao campo de suporte da propriedade `Id` usando o especificador `field:` no atributo, conforme mostrado no seguinte exemplo:

C#

```
public class Person
{
    public string? FirstName { get; set; }

    public string? LastName { get; set; }

    [field:NonSerialized]
    public int Id { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Essa técnica funciona para qualquer atributo anexado ao campo de suporte na propriedade autoimplementada.

## Implementando `INotifyPropertyChanged`

A última situação em que você precisa escrever código em um acessador de propriedade é para oferecer suporte à interface `INotifyPropertyChanged`, usada para notificar os clientes de vinculação de dados que um valor foi alterado. Quando o valor de uma propriedade for alterado, o objeto aciona o evento `INotifyPropertyChanged.PropertyChanged` para indicar a alteração. As bibliotecas de vinculação de dados, por sua vez, atualizam os elementos de exibição com base nessa alteração. O código a seguir mostra como você implementaria `INotifyPropertyChanged` para a propriedade `FirstName` dessa classe person.

C#

```
public class Person : INotifyPropertyChanged
{
    public string? FirstName
    {
        get => _firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            if (value != _firstName)
            {

```

```
        _firstName = value;
        PropertyChanged?.Invoke(this,
            new PropertyChangedEventArgs(nameof(FirstName)));
    }
}
private string? _firstName;

public event PropertyChangedEventHandler? PropertyChanged;
}
```

O operador `?.` é chamado de *operador condicional nulo*. Ele verifica uma referência nula antes de avaliar o lado direito do operador. O resultado final é que, se não houver nenhum assinante para o evento `PropertyChanged`, o código para acionar o evento não é executado. Ela lançaria uma `NullReferenceException` sem essa verificação, nesse caso. Para obter mais informações, consulte [events](#). Este exemplo também usa o novo operador `nameof` para converter o símbolo de nome da propriedade em sua representação de texto. O uso de `nameof` pode reduzir erros no local em que o nome da propriedade foi digitado errado.

Novamente, a implementação de [INotifyPropertyChanged](#) é um exemplo de um caso em que você pode escrever o código nos acessadores para dar suporte aos cenários necessários.

## Resumindo

As propriedades são uma forma de campos inteligentes em uma classe ou objeto. De fora do objeto, elas parecem como campos no objeto. No entanto, as propriedades podem ser implementadas usando a paleta completa de funcionalidades do C#. Você pode fornecer validação, acessibilidade diferente, avaliação lenta ou quaisquer requisitos necessários aos seus cenários.

# Indexadores

Artigo • 10/05/2023

Os *indexadores* são semelhantes às propriedades. De muitas maneiras, os indexadores baseiam-se nos mesmos recursos de linguagem que as [propriedades](#). Os indexadores habilitam as propriedades *indexadas*: propriedades referenciadas com o uso de um ou mais argumentos. Esses argumentos fornecem um índice em um conjunto de valores.

## Sintaxe do indexador

Você pode acessar um indexador por meio de um nome de variável e colchetes. Coloque os argumentos do indexador dentro de colchetes:

C#

```
var item = someObject["key"];
someObject["AnotherKey"] = item;
```

Você declara os indexadores usando a palavra-chave `this` como o nome da propriedade e declarando os argumentos entre colchetes. Essa declaração corresponderia à utilização mostrada no parágrafo anterior:

C#

```
public int this[string key]
{
    get { return storage.Find(key); }
    set { storage.SetAt(key, value); }
}
```

Neste exemplo inicial, você pode ver a relação entre a sintaxe das propriedades e dos indexadores. Essa analogia impulsiona a maioria das regras de sintaxe para indexadores. Os indexadores podem ter qualquer modificador de acesso válido (público, protegido interno, protegido, interno, particular ou protegido de forma privada). Eles podem ser sealed, virtual ou abstract. Assim como acontece com as propriedades, você pode especificar modificadores de acesso diferentes para os acessadores get e set em um indexador. Você também pode especificar indexadores somente leitura (omitindo o acessador set) ou indexadores somente gravação (omitindo o acessador get).

Você pode aplicar aos indexadores quase tudo o que aprendeu ao trabalhar com propriedades. A única exceção a essa regra são as *propriedades autoimplementadas*. O compilador não pode gerar sempre o armazenamento correto para um indexador.

A presença dos argumentos para referenciar um item em um conjunto de itens distingue os indexadores das propriedades. Você pode definir vários indexadores em um tipo, contanto que as listas de argumentos para cada indexador seja exclusiva. Vamos explorar diferentes cenários em que você pode usar um ou mais indexadores em uma definição de classe.

## Cenários

Você deve definir *indexadores* em seu tipo quando a API do tipo modela alguma coleção na qual você define os argumentos para essa coleção. Seu indexadores podem ou não mapear diretamente para os tipos de coleção que fazem parte da estrutura principal do .NET. O tipo pode ter outras responsabilidades, além da modelagem de uma coleção. Os indexadores permitem que você forneça a API que corresponda à abstração do tipo, sem expor os detalhes internos de como os valores dessa abstração são armazenados ou computados.

Vamos examinar alguns dos cenários comuns de uso de *indexadores*. Você pode acessar a [pasta de exemplo para indexadores](#). Para obter instruções de download, consulte [Exemplos e tutoriais](#).

## Matrizes e vetores

Um dos cenários mais comuns para a criação de indexadores é quando seu tipo modela uma matriz ou um vetor. Você pode criar um indexador para modelar uma lista ordenada de dados.

A vantagem de criar seu próprio indexador é que você pode definir o armazenamento dessa coleção para atender às suas necessidades. Imagine um cenário em que seu tipo modela dados históricos que são muito grandes para serem carregados na memória ao mesmo tempo. Você precisa carregar e descarregar seções da coleção com base na utilização. O exemplo a seguir modela esse comportamento. Ele relata quantos pontos de dados existem. Ele cria páginas para manter as seções de dados sob demanda. Ele remove páginas da memória a fim de liberar espaço para as páginas necessárias para as solicitações mais recentes.

C#

```
public class DataSamples
{
    private class Page
    {
        private readonly List<Measurements> pageData = new
List<Measurements>();
```

```

private readonly int startingIndex;
private readonly int length;
private bool dirty;
private DateTime lastAccess;

public Page(int startingIndex, int length)
{
    this.startingIndex = startingIndex;
    this.length = length;
    lastAccess = DateTime.Now;

    // This stays as random stuff:
    var generator = new Random();
    for(int i=0; i < length; i++)
    {
        var m = new Measurements
        {
            HiTemp = generator.Next(50, 95),
            LoTemp = generator.Next(12, 49),
            AirPressure = 28.0 + generator.NextDouble() * 4
        };
        pageData.Add(m);
    }
}

public bool HasItem(int index) =>
((index >= startingIndex) &&
(index < startingIndex + length));

public Measurements this[int index]
{
    get
    {
        lastAccess = DateTime.Now;
        return pageData[index - startingIndex];
    }
    set
    {
        pageData[index - startingIndex] = value;
        dirty = true;
        lastAccess = DateTime.Now;
    }
}

public bool Dirty => dirty;
public DateTime LastAccess => lastAccess;
}

private readonly int totalSize;
private readonly List<Page> pagesInMemory = new List<Page>();

public DataSamples(int totalSize)
{
    this.totalSize = totalSize;
}

```

```

public Measurements this[int index]
{
    get
    {
        if (index < 0)
            throw new IndexOutOfRangeException("Cannot index less than
0");
        if (index >= totalSize)
            throw new IndexOutOfRangeException("Cannot index past the
end of storage");

        var page = updateCachedPagesForAccess(index);
        return page[index];
    }
    set
    {
        if (index < 0)
            throw new IndexOutOfRangeException("Cannot index less than
0");
        if (index >= totalSize)
            throw new IndexOutOfRangeException("Cannot index past the
end of storage");
        var page = updateCachedPagesForAccess(index);

        page[index] = value;
    }
}

private Page updateCachedPagesForAccess(int index)
{
    foreach (var p in pagesInMemory)
    {
        if (p.HasItem(index))
        {
            return p;
        }
    }
    var startingIndex = (index / 1000) * 1000;
    var newPassword = new Page(startingIndex, 1000);
    addPageToCache(newPassword);
    return newPassword;
}

private void addPageToCache(Page p)
{
    if (pagesInMemory.Count > 4)
    {
        // remove oldest non-dirty page:
        var oldest = pagesInMemory
            .Where(page => !page.Dirty)
            .OrderBy(page => page.LastAccess)
            .FirstOrDefault();
        // Note that this may keep more than 5 pages in memory
        // if too much is dirty
        if (oldest != null)

```

```
        pagesInMemory.Remove(oldest);
    }
    pagesInMemory.Add(p);
}
}
```

Você pode seguir esta linguagem de design para modelar qualquer tipo de coleção na qual há bons motivos para não carregar todo o conjunto de dados em uma coleção na memória. Observe que a classe `Page` é uma classe particular aninhada que não faz parte da interface pública. Esses detalhes estão ocultos de qualquer usuário dessa classe.

## Dicionários

Outro cenário comum é quando você precisa modelar um dicionário ou um mapa. Esse cenário é quando o seu tipo armazena valores com base na chave, normalmente chaves de texto. Este exemplo cria um dicionário que mapeia os argumentos de linha de comando para [expressões lambda](#) que gerenciam essas opções. O exemplo a seguir mostra duas classes: uma classe `ArgsActions` que mapeia uma opção de linha de comando para um delegado `Action` e uma `ArgsProcessor`, que usa a `ArgsActions` para executar cada `Action`, quando encontrar essa opção.

C#

```
public class ArgsProcessor
{
    private readonly ArgsActions actions;

    public ArgsProcessor(ArgsActions actions)
    {
        this.actions = actions;
    }

    public void Process(string[] args)
    {
        foreach(var arg in args)
        {
            actions[arg]?.Invoke();
        }
    }
}

public class ArgsActions
{
    readonly private Dictionary<string, Action> argsActions = new
    Dictionary<string, Action>();

    public Action this[string s]
    {
```

```

    get
    {
        Action action;
        Action defaultAction = () => {} ;
        return argsActions.TryGetValue(s, out action) ? action :
defaultAction;
    }
}

public void SetOption(string s, Action a)
{
    argsActions[s] = a;
}
}

```

Neste exemplo, a coleção `ArgsAction` mapeia próximo à coleção subjacente. O `get` determina se uma opção específica foi configurada. Se sim, ele retorna a `Action` associada a essa opção. Se não, ele retorna uma `Action` que não faz nada. O acessador público não inclui um acessador `set`. Em vez disso, o design usa um método público para a configuração de opções.

## Mapas multidimensionais

Você pode criar indexadores que usam vários argumentos. Além disso, esses argumentos não estão restritos a serem do mesmo tipo. Vamos analisar dois exemplos.

O primeiro exemplo mostra uma classe que gera valores para um conjunto de Mandelbrot. Para obter mais informações sobre a matemática por trás desse conjunto, leia [este artigo ↗](#). O indexador usa dois duplos para definir um ponto no plano X, Y. O acessador `get` calcula o número de iterações até que um ponto seja considerado como fora do conjunto. Se o número máximo de iterações for atingido, o ponto está no conjunto e o valor da classe `maxIterations` será retornado. (As imagens geradas por computador, popularizadas para o conjunto de Mandelbrot, definem cores para o número de iterações necessárias para determinar que um ponto está fora do conjunto.)

C#

```

public class Mandelbrot
{
    readonly private int maxIterations;

    public Mandelbrot(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public int this [double x, double y]

```

```

    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x*x + y * y < 4) &&
                   (iterations < maxIterations))
            {
                var newX = x * x - y * y + x0;
                y = 2 * x * y + y0;
                x = newX;
                iterations++;
            }
            return iterations;
        }
    }
}

```

O conjunto de Mandelbrot define valores em cada coordenada (x,y) para valores de número real. Isso define um dicionário que poderia conter um número infinito de valores. Portanto, não há armazenamento por trás desse conjunto. Em vez disso, essa classe calcula o valor de cada ponto quando o código chama o acessador `get`. Não há nenhum armazenamento subjacente usado.

Vamos examinar um último uso de indexadores, em que o indexador recebe vários argumentos de tipos diferentes. Considere um programa que gerencia os dados históricos de temperatura. Esse indexador utiliza uma cidade e uma data para definir ou obter as temperaturas máximas e mínimas desse local:

C#

```

using DateMeasurements =
    System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements =
    System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;

public class HistoricalWeatherData
{
    readonly CityDataMeasurements storage = new CityDataMeasurements();

    public Measurements this[string city, DateTime date]
    {
        get
        {
            var cityData = default(DateMeasurements);

```

```

        if (!storage.TryGetValue(city, out cityData))
            throw new ArgumentOutOfRangeException(nameof(city), "City
not found");

        // strip out any time portion:
        var index = date.Date;
        var measure = default(Measurements);
        if (cityData.TryGetValue(index, out measure))
            return measure;
        throw new ArgumentOutOfRangeException(nameof(date), "Date not
found");
    }
    set
    {
        var cityData = default(DateMeasurements);

        if (!storage.TryGetValue(city, out cityData))
        {
            cityData = new DateMeasurements();
            storage.Add(city, cityData);
        }

        // Strip out any time portion:
        var index = date.Date;
        cityData[index] = value;
    }
}

```

Este exemplo cria um indexador que mapeia dados meteorológicos de dois argumentos diferentes: uma cidade (representada por uma `string`) e uma data (representada por uma `DateTime`). O armazenamento interno usa duas classes `Dictionary` para representar o dicionário bidimensional. A API pública não representa mais o armazenamento subjacente. Em vez disso, os recursos de linguagem dos indexadores permite que você crie uma interface pública que representa a sua abstração, mesmo que o armazenamento subjacente deva usar diferentes tipos principais de coleção.

Há duas partes desse código que podem não ser familiares para alguns desenvolvedores. Estas duas diretivas `using`:

C#

```

using DateMeasurements =
System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements = System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;

```

criam um *alias* para um tipo genérico construído. Essas instruções habilitam o código a usar, mais adiante, os nomes `DateMeasurements` e `CityDataMeasurements` mais descritivos, em vez da construção genérica de `Dictionary<DateTime, Measurements>` e `Dictionary<string, Dictionary<DateTime, Measurements>>`. Esse constructo exige o uso de nomes de tipo totalmente qualificados no lado direito do sinal `=`.

A segunda técnica é para remover as partes de hora de qualquer objeto `DateTime` usado para indexar na coleção. O .NET não inclui um tipo somente data. Os desenvolvedores usam o tipo `DateTime`, mas usam a propriedade `Date` para garantir que qualquer objeto `DateTime` daquele dia sejam iguais.

## Resumindo

Você deve criar indexadores sempre que tiver um elemento semelhante a uma propriedade em sua classe, em que essa propriedade representa não um único valor, mas uma coleção de valores em que cada item individual é identificado por um conjunto de argumentos. Esses argumentos podem identificar exclusivamente qual item da coleção deve ser referenciado. Os indexadores ampliam o conceito de [propriedades](#), em que um membro é tratado como um item de dados de fora da classe, mas como um método do lado de dentro. Os indexadores permitem que os argumentos localizem um único item em uma propriedade que representa um conjunto de itens.

# Iterators

Artigo • 10/05/2023

Quase todos os programas que você escrever terão alguma necessidade de iterar em uma coleção. Você escreverá um código que examina cada item em uma coleção.

Você também criará métodos de iterador, que são métodos que produzem um *iterator* para os elementos dessa classe. Um *iterator* é um objeto que percorre contêineres, particularmente listas. Os iteradores podem ser usados para:

- Executar uma ação em cada item em uma coleção.
- Enumerar uma coleção personalizada.
- Estender [LINQ](#) ou outras bibliotecas.
- Criar um pipeline de dados em que os dados fluem com eficiência pelos métodos de iterador.

A linguagem C# fornece recursos para gerar e consumir sequências. Essas sequências podem ser produzidas e consumidas de maneira síncrona ou assíncrona. Este artigo fornece uma visão geral desses recursos.

## iterando com foreach

Enumerar uma coleção é simples: a palavra-chave `foreach` enumera uma coleção, executando a instrução inserida uma vez para cada elemento na coleção:

```
C#  
  
foreach (var item in collection)  
{  
    Console.WriteLine(item?.ToString());  
}
```

Isso é tudo. Para iterar em todo o conteúdo de uma coleção, a instrução `foreach` é tudo o que você precisa. No entanto, a instrução `foreach` não é mágica. Ele se baseia em duas interfaces genéricas definidas na biblioteca do .NET Core a fim de gerar o código necessário para iterar uma coleção: `IEnumerable<T>` e `IEnumerator<T>`. Esse mecanismo é explicado mais detalhadamente abaixo.

Essas duas interfaces também têm contrapartes não genéricas: `IEnumerable` e `IEnumerator`. As versões [genéricas](#) são preferenciais para o código moderno.

Quando uma sequência é gerada de maneira assíncrona, você pode usar a instrução `await foreach` para consumir essa sequência assincronamente:

```
C#  
  
await foreach (var item in asyncSequence)  
{  
    Console.WriteLine(item?.ToString());  
}
```

Quando a sequência é uma `System.Collections.Generic.IEnumerable<T>`, você usa `foreach`. Quando a sequência é uma `System.Collections.Generic.IAsyncEnumerable<T>`, você usa `await foreach`. No último caso, a sequência é gerada de maneira assíncrona.

## Fontes de enumeração com métodos de iterador

Outro ótimo recurso da linguagem C# permite que você crie métodos que criam uma fonte para uma enumeração. Esses métodos são chamados de *métodos de iterador*. Um método de iterador define como gerar os objetos em uma sequência quando solicitado. Você usa as palavras-chave contextuais `yield return` para definir um método iterador.

Você poderia escrever esse método para produzir a sequência de inteiros de 0 a 9:

```
C#  
  
public IEnumerable<int> GetSingleDigitNumbers()  
{  
    yield return 0;  
    yield return 1;  
    yield return 2;  
    yield return 3;  
    yield return 4;  
    yield return 5;  
    yield return 6;  
    yield return 7;  
    yield return 8;  
    yield return 9;  
}
```

O código acima mostra instruções `yield return` distintas para destacar o fato de que você pode usar várias instruções `yield return` discretas em um método iterador. Você pode (e frequentemente o faz) usar outros constructos de linguagem para simplificar o

código de um método iterador. A definição do método abaixo produz a mesma sequência exata de números:

```
C#  
  
public IEnumerable<int> GetSingleDigitNumbersLoop()  
{  
    int index = 0;  
    while (index < 10)  
        yield return index++;  
}
```

Você não precisa determinar uma ou a outra. Você pode ter quantas instruções `yield` `return` forem necessárias para atender as necessidades do seu método:

```
C#  
  
public IEnumerable<int> GetSetsOfNumbers()  
{  
    int index = 0;  
    while (index < 10)  
        yield return index++;  
  
    yield return 50;  
  
    index = 100;  
    while (index < 110)  
        yield return index++;  
}
```

Todos esses exemplos anteriores teriam um equivalente assíncrono. Em cada caso, você substituiria o tipo de retorno `IEnumerable<T>` por `IAsyncEnumerable<T>`. O exemplo anterior, por exemplo, teria a seguinte versão assíncrona:

```
C#  
  
public async IAsyncEnumerable<int> GetSetsOfNumbersAsync()  
{  
    int index = 0;  
    while (index < 10)  
        yield return index++;  
  
    await Task.Delay(500);  
  
    yield return 50;  
  
    await Task.Delay(500);  
  
    index = 100;
```

```
        while (index < 110)
            yield return index++;
    }
```

Essa é a sintaxe para iteradores síncronos e assíncronos. Vamos considerar um exemplo do mundo real. Imagine que você está em um projeto de IoT e os sensores de dispositivo geram um enorme fluxo de dados. Para ter uma noção dos dados, você pode escrever um método realiza a amostragem a cada N elementos de dados. Esse pequeno método iterador resolve:

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sourceSequence,
int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Se a leitura do dispositivo IoT produzir uma sequência assíncrona, você modificará o método como mostra o seguinte exemplo:

C#

```
public static async IAsyncEnumerable<T> Sample<T>(this IAsyncEnumerable<T>
sourceSequence, int interval)
{
    int index = 0;
    await foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Há uma restrição importante em métodos de iterador: você não pode ter uma instrução `return` e uma instrução `yield return` no mesmo método. O seguinte código não será compilado:

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
```

```

        while (index < 10)
            yield return index++;

        yield return 50;

        // generates a compile time error:
        var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109
    };
        return items;
}

```

Essa restrição normalmente não é um problema. Você tem a opção de usar `yield return` em todo o método ou separar o método original em vários métodos, alguns usando `return` e alguns usando `yield return`.

Você pode modificar um pouco o último método para usar `yield return` em todos os lugares:

C#

```

public IEnumerable<int> GetFirstDecile()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109
};
    foreach (var item in items)
        yield return item;
}

```

Às vezes, a resposta certa é dividir um método iterador em dois métodos diferentes. Um que usa `return` e outro que usa `yield return`. Considere a situação em que você talvez deseja retornar uma coleção vazia ou os primeiros cinco números ímpares, com base em um argumento booleano. Você poderia escrever isso como esses dois métodos:

C#

```

public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

```

```
private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
        index++;
    }
}
```

Observe os métodos acima. O primeiro usa a instrução `return` padrão para retornar uma coleção vazia ou o iterador criado pelo segundo método. O segundo método usa a instrução `yield return` para criar a sequência solicitada.

## Aprofundamento em `foreach`

A instrução `foreach` se expande em uma expressão padrão que usa as interfaces `IEnumerable<T>` e `IEnumerator<T>` para iterar em todos os elementos de uma coleção. Ela também minimiza os erros cometidos pelos desenvolvedores por não gerenciarem os recursos adequadamente.

O compilador converte o loop `foreach` mostrado no primeiro exemplo em algo semelhante a esse constructo:

C#

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

O código exato gerado pelo compilador é mais complicado e lida com situações em que o objeto retornado por `GetEnumerator()` implementa a interface `IDisposable`. A expansão completa gera um código mais semelhante a esse:

C#

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
```

```
        var item = enumerator.Current;
        Console.WriteLine(item.ToString());
    }
}
finally
{
    // dispose of enumerator.
}
}
```

O compilador converte a primeira amostra assíncrona em algo semelhante a este constructo:

```
C#
{
    var enumerator = collection.GetAsyncEnumerator();
    try
    {
        while (await enumerator.MoveNextAsync())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of async enumerator.
    }
}
```

A maneira na qual o enumerador é descartado depende das características do tipo de `enumerator`. No caso síncrono geral, a cláusula `finally` se expande para:

```
C#
finally
{
    (enumerator as IDisposable)?.Dispose();
}
```

O caso assíncrono geral se expande para:

```
C#
finally
{
    if (enumerator is IAsyncDisposable asyncDisposable)
```

```
        await asyncDisposable.DisposeAsync();
    }
```

No entanto, se `enumerator` é de um tipo selado e não há nenhuma conversão implícita do tipo de `enumerator` para `IDisposable` ou `IAsyncDisposable`, a cláusula `finally` se expande para um bloco vazio:

C#

```
finally
{
}
```

Se houver uma conversão implícita do tipo de `enumerator` para `IDisposable` e `enumerator` for um tipo de valor não anulável, a cláusula `finally` se expandirá para:

C#

```
finally
{
    ((IDisposable)enumerator).Dispose();
}
```

Felizmente, você não precisa se lembrar de todos esses detalhes. A instrução `foreach` trata todas essas nuances para você. O compilador gerará o código correto para qualquer um desses constructos.

# Introdução a delegados e eventos em C#

Artigo • 10/05/2023

Os delegados fornecem um mecanismo de *associação tardia* no .NET. Associação tardia significa que você cria um algoritmo em que o chamador também fornece pelo menos um método que implementa a parte do algoritmo.

Por exemplo, considere classificar uma lista de estrelas em um aplicativo de astronomia. Você pode optar por classificar as estrelas por sua distância da terra ou a magnitude da estrela ou seu brilho percebido.

Em todos esses casos, o método `Sort()` faz essencialmente a mesma coisa: organiza os itens na lista com base em alguma comparação. O código que compara duas estrelas é diferente para cada uma das ordenações de classificação.

Esses tipos de soluções foram usados no software por meio século. O conceito de delegado de linguagem C# fornece suporte de linguagem de primeira classe e segurança de tipos em torno do conceito.

Como você verá mais à frente nesta série, o código C# que você escreve para algoritmos como esse é de fortemente tipado. O compilador garante que os tipos correspondam a argumentos e tipos de retorno.

[Ponteiros de função](#) foram adicionados ao C# 9 para cenários semelhantes, em que você precisa de mais controle sobre a convenção de chamada. O código associado a um delegado é invocado usando um método virtual adicionado a um tipo delegado. Usando ponteiros de função, você pode especificar convenções diferentes.

## Metas de design da linguagem para delegados

Os designers de linguagem enumeraram várias metas para o recurso que eventualmente se tornaram delegados.

A equipe queria um constructo de linguagem comum que pudesse ser usada qualquer algoritmo de associação tardia. Delegados permitem aos desenvolvedores aprender um conceito e usar esse mesmo conceito em muitos problemas de software diferentes.

Em segundo lugar, a equipe queria dar suporte a chamadas de método single ou multicast. (Representantes multicast são delegados que encadeiam várias chamadas de método. Você verá exemplos [mais adiante nesta série](#).)

A equipe queria delegados para dar suporte à mesma segurança de tipos que os desenvolvedores esperam de todos os constructos de C#.

Por fim, a equipe reconheceu que um padrão de eventos é um padrão específico em que delegados ou qualquer algoritmo de associação tardia é útil. A equipe quis garantir que o código para delegados pudesse fornecer a base para o padrão de evento .NET.

O resultado de todo esse trabalho era o suporte do delegado e do evento no C# e .NET.

Os demais artigos nessa série abordarão os recursos da linguagem, o suporte da biblioteca e as expressões comuns que são usadas ao trabalhar com delegados e eventos. Você saberá mais sobre:

- A palavra-chave `delegate` e qual código ela gera.
- Os recursos na classe `System.Delegate` e como esses recursos são usados.
- Como criar delegados fortemente tipados.
- Como criar métodos que podem ser invocados por meio de delegados.
- Como trabalhar com eventos e delegados usando expressões lambda.
- Como os delegados se tornam um dos blocos de construção para LINQ.
- Como os delegados são a base para o padrão de evento do .NET e como eles são diferentes.

Vamos começar.

[Próximo](#)

# System.Delegate e a palavra-chave delegate

Artigo • 10/05/2023

[Anterior](#)

Este artigo abordará as classes do .NET Framework que dão suporte a delegados e como eles são mapeados para a palavra-chave `delegate`.

## Definir tipos de delegado

Vamos começar com a palavra-chave ‘`delegate`’, pois ela é basicamente o que você usará ao trabalhar com delegados. O código que o compilador gera quando você usa a palavra-chave `delegate` será mapeado para chamadas de método que invocam membros das classes [Delegate](#) e [MulticastDelegate](#).

Você define um tipo de delegado usando uma sintaxe semelhante à definição de uma assinatura de método. Basta adicionar a palavra-chave `delegate` à definição.

Vamos continuar a usar o método `List.Sort()` como nosso exemplo. A primeira etapa é criar um tipo para o delegado de comparação:

C#

```
// From the .NET Core library

// Define the delegate type:
public delegate int Comparison<in T>(T left, T right);
```

O compilador gera uma classe, derivada de `System.Delegate`, que corresponde à assinatura usada (nesse caso, um método que retorna um inteiro e tem dois argumentos). O tipo do delegado é `Comparison`. O tipo delegado `comparison` é um tipo genérico. Para obter detalhes sobre os genéricos, consulte [aqui](#).

Observe que a sintaxe pode aparecer como se estivesse declarando uma variável, mas na verdade está declarando um *tipo*. Você pode definir tipos de delegado dentro de classes, diretamente dentro de namespaces ou até mesmo no namespace global.

ⓘ Observação

Declarar tipos de delegado (ou outros tipos) diretamente no namespace global não é recomendado.

O compilador também gera manipuladores de adição e remoção para esse novo tipo de forma que os clientes dessa classe podem adicionar e remover métodos de uma lista de invocação de instância. O compilador imporá que a assinatura do método que está sendo adicionado ou removido corresponda à assinatura usada ao declarar o método.

## Declarar instâncias de delegados

Depois de definir o delegado, você pode criar uma instância desse tipo. Como todas as variáveis em C#, você não pode declarar instâncias de delegado diretamente em um namespace ou no namespace global.

C#

```
// inside a class definition:  
  
// Declare an instance of that type:  
public Comparison<T> comparator;
```

O tipo da variável é `Comparison<T>`, o tipo de delegado definido anteriormente. O nome da variável é `comparator`.

Esse snippet de código acima declarou uma variável de membro dentro de uma classe. Você também pode declarar variáveis de delegado que são variáveis locais ou argumentos para métodos.

## Invocar delegados

Você invoca os métodos que estão na lista de invocação de um delegado chamando esse delegado. Dentro do método `Sort()`, o código chamará o método de comparação para determinar em qual ordem posicionar objetos:

C#

```
int result = comparator(left, right);
```

Na linha acima, o código *invoca* o método anexado ao delegado. Você trata a variável como um nome de método e a invoca usando a sintaxe de chamada de método normal.

Essa linha de código faz uma suposição não segura: não há garantia de que um destino foi adicionado ao delegado. Se nenhum destino tiver sido anexado, a linha acima fará com que um `NullReferenceException` seja lançado. As expressões usadas para resolver esse problema são mais complicadas do que uma simples verificação de null e são abordadas posteriormente nesta [série](#).

## Atribuir, adicionar e remover destinos de invocação

Essa é a forma como o tipo do delegado é definido e como as instâncias de delegado são declaradas e invocadas.

Os desenvolvedores que desejam usar o método `List.Sort()` precisa definir um método cuja assinatura corresponde à definição de tipo de delegado e atribuí-lo ao delegado usado pelo método de classificação. Esta atribuição adiciona o método à lista de invocação do objeto de delegado.

Suponha que você queira classificar uma lista de cadeias de caracteres pelo seu comprimento. A função de comparação pode ser a seguinte:

C#

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

O método é declarado como um método particular. Tudo bem. Você pode não desejar que esse método seja parte da sua interface pública. Ele ainda pode ser usado como o método de comparação ao anexar a um delegado. O código de chamada terá esse método anexado à lista de destino do objeto de delegado e pode acessá-lo por meio do delegado.

Você cria essa relação passando esse método para o método `List.Sort()`:

C#

```
phrases.Sort(CompareLength);
```

Observe que o nome do método é usado, sem parênteses. Usar o método como um argumento informa ao compilador para converter a referência de método em uma referência que pode ser usada como um destino de invocação do delegado e anexar esse método como um destino de invocação.

Você também poderia ter sido explícito declarando uma variável do tipo

`Comparison<string>` e fazendo uma atribuição:

C#

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

Em utilizações em que o método que está sendo usado como um destinos de delegado é um método pequeno, é comum usar a sintaxe da [expressão lambda](#) para executar a atribuição:

C#

```
Comparison<string> comparer = (left, right) =>
    left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

O uso de expressões lambda para destinos de delegado será abordado em uma [seção posterior](#).

O exemplo de Sort() normalmente anexa um único método de destino ao delegado. No entanto, objetos delegados dão suporte a listas de invocação que têm vários métodos de destino anexados a um objeto de delegado.

## Classes Delegate e MulticastDelegate

O suporte de linguagem descrito acima fornece os recursos e o suporte que você normalmente precisará para trabalhar com delegados. Esses recursos são criados com base em duas classes no .NET Core Framework: [Delegate](#) e [MulticastDelegate](#).

A classe `System.Delegate` e sua única subclasse direta, `System.MulticastDelegate`, fornecem o suporte de estrutura para criar delegados, registrar métodos como destinos de delegado e invocar todos os métodos que são registrados como um destino de delegado.

Curiosamente, as classes `System.Delegate` e `System.MulticastDelegate` não são em si tipos de delegado. Elas fornecem a base para todos os tipos de delegado específicos. Esse mesmo processo de design de linguagem determinou que você não pode declarar uma classe que deriva de `Delegate` ou `MulticastDelegate`. As regras da linguagem C# proíbem isso.

Em vez disso, o compilador C# cria instâncias de uma classe derivada de `MulticastDelegate` quando você usa a palavra-chave da linguagem C# para declarar os tipos de delegado.

Esse design tem suas raízes na primeira versão do C# e do .NET. Uma meta da equipe de design era garantir que a linguagem aplicava a segurança de tipos ao usar delegados. Isso significava garantir que os delegados fossem invocados com o tipo e o número de argumentos certos. E, que algum tipo de retorno fosse indicado no tempo de compilação. Os delegados faziam parte da versão 1.0 do .NET, que era anterior aos genéricos.

A melhor maneira de reforçar essa segurança de tipos foi o compilador criar as classes de delegado concretas que representavam a assinatura do método sendo usado.

Embora não seja possível criar classes derivadas diretamente, você usará os métodos definidos nessas classes. Vamos percorrer os métodos mais comuns que você usará ao trabalhar com delegados.

O primeiro e mais importante fato a se lembrar é que todos os delegados com os quais você trabalha são derivados de `MulticastDelegate`. Um delegado multicast significa que mais de um destino de método pode ser invocado durante a invocação através de um delegado. O design original considerava fazer uma distinção entre delegados em que somente um método de destino poderia ser anexado e invocado e delegados em que vários métodos de destino poderiam ser anexados e invocados. Essa distinção provou ser menos útil na prática do que pensado originalmente. As duas classes diferentes já foram criadas e estão na estrutura desde seu lançamento público inicial.

Os métodos que você usará mais com delegados são `Invoke()` e `BeginInvoke() / EndInvoke()`. `Invoke()` invocará todos os métodos que foram anexados a uma instância de delegado específica. Como você viu anteriormente, normalmente invoca delegados usando a sintaxe de chamada de método na variável de delegado. Como você verá [posteriormente nesta série](#), existem padrões que trabalham diretamente com esses métodos.

Agora que você viu a sintaxe da linguagem e as classes que dão suporte a delegados, vamos examinar como os delegados fortemente tipados são usados, criados e invocados.

[Próximo](#)

# Delegados Fortemente Tipados

Artigo • 10/05/2023

[Anterior](#)

No artigo anterior, você viu como criar tipos de delegado específicos usando a palavra-chave `delegate`.

A classe de Delegado abstrata fornece a infraestrutura para a invocação e acoplamento fraco. Os tipos de delegado concretos se tornam muito mais úteis adotando e impondo a segurança de tipos para os métodos que são adicionados à lista de invocação para um objeto delegado. Quando você usa a palavra-chave `delegate` e define um tipo de delegado concreto, o compilador gera esses métodos.

Na prática, isso poderia levar à criação de novos tipos de delegado sempre que precisar de uma assinatura de método diferente. Esse trabalho pode se tornar entediante depois de um tempo. Cada novo recurso exige novos tipos de delegado.

Felizmente, isso não é necessário. O .NET Core Framework contém vários tipos que podem ser reutilizados sempre que você precisar de tipos de delegado. Essas são definições [genéricas](#) para que você possa declarar personalizações quando precisar de novas declarações de método.

O primeiro desses tipos é o tipo [Action](#) e diversas variações:

C#

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

O modificador `in` no argumento de tipo genérico é abordado neste artigo sobre covariância.

Há variações do delegado `Action` que contêm até 16 argumentos como `Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>`. É importante que essas definições usem argumentos genéricos diferentes para cada um dos argumentos do delegado: isso proporciona a máxima flexibilidade. Os argumentos de método não precisam ser, mas podem ser, do mesmo tipo.

Use um dos tipos `Action` para qualquer tipo de delegado que tenha um tipo de retorno nulo.

A estrutura também inclui vários tipos de delegado genérico que você pode usar para tipos de delegado que retornam valores:

C#

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

O modificador `out` no argumento de tipo genérico de resultado é abordado neste artigo sobre covariância.

Há variações do delegado `Func` com até 16 argumentos de entrada como `Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>`. O tipo do resultado é sempre o último parâmetro de tipo em todas as declarações `Func`, por convenção.

Use um dos tipos `Func` para qualquer tipo de delegado que retorna um valor.

Há também um tipo `Predicate<T>` especializado para um delegado que retorna um teste em um único valor:

C#

```
public delegate bool Predicate<in T>(T obj);
```

Você pode observar que para qualquer tipo `Predicate`, há um tipo `Func` estruturalmente equivalente, por exemplo:

C#

```
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

Você pode pensar que esses dois tipos são equivalentes. Eles não são. Essas duas variáveis não podem ser usadas alternadamente. Uma variável de um tipo não pode ser atribuída o outro tipo. O sistema de tipo do C# usa os nomes dos tipos definidos, não a estrutura.

Todas essas definições de tipo de delegado na Biblioteca do .NET Core devem significar que você não precisa definir um novo tipo de delegado para qualquer novo recurso criado que exige delegados. Essas definições genéricas devem fornecer todos os tipos de delegado necessários na maioria das situações. Você pode simplesmente instanciar

um desses tipos com os parâmetros de tipo necessários. No caso de algoritmos que podem ser tornados genéricos, esses delegados podem ser usados como tipos genéricos.

Isso deve economizar tempo e minimizar o número de novos tipos de que você precisa criar a fim de trabalhar com delegados.

No próximo artigo, você verá vários padrões comuns para trabalhar com delegados na prática.

[Próximo](#)

# Padrões comuns para delegados

Artigo • 07/04/2023

[Anterior](#)

Os delegados fornecem um mecanismo que permite designs de software que envolvem acoplamento mínimo entre os componentes.

Um exemplo excelente desse tipo de design é o LINQ. O padrão de expressão de consulta LINQ se baseia em delegados para todos os seus recursos. Considere este exemplo simples:

C#

```
var smallNumbers = numbers.Where(n => n < 10);
```

Isso filtra a sequência de números para somente aqueles com valor menor que 10. O método `Where` usa um delegado que determina quais elementos de uma sequência são passados no filtro. Quando cria uma consulta LINQ, você fornece a implementação do delegado para essa finalidade específica.

O protótipo para o método `Where` é:

C#

```
public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

Este exemplo é repetido com todos os métodos que fazem parte do LINQ. Todos eles contam com delegados para o código que gerencia a consulta específica. Esse padrão de design de API é poderoso para aprender e entender.

Este exemplo simples ilustra como delegados requerem muito pouco acoplamento entre componentes. Você não precisa criar uma classe que deriva de uma classe base específica. Você não precisa implementar uma interface específica. O único requisito é fornecer a implementação de um método que é fundamental para a tarefa em questão.

## Criar componentes próprios com delegados

Vamos trabalhar naquele exemplo criando um componente usando um design que se baseia em delegados.

Vamos definir um componente que poderia ser usado para mensagens de log em um sistema grande. Os componentes da biblioteca poderiam ser usados em muitos ambientes diferentes, em várias plataformas diferentes. Há muitos recursos comuns no componente que gerencia os logs. Ele precisará aceitar mensagens de qualquer componente do sistema. Essas mensagens terão prioridades diferentes, que o componente de núcleo pode gerenciar. As mensagens devem ter carimbos de data/hora em sua forma final arquivada. Para cenários mais avançados, é possível filtrar mensagens pelo componente de origem.

Há um aspecto do recurso que é alterado com frequência: onde as mensagens são gravadas. Em alguns ambientes, elas podem ser gravadas no console de erro. Em outros, em um arquivo. Outras possibilidades incluem o armazenamento em banco de dados, logs de eventos do sistema operacional ou outro armazenamento de documentos.

Também há combinações de saídas que podem ser usadas em cenários diferentes. Talvez você queira gravar mensagens no console e em um arquivo.

Um design baseado em delegados fornece muita flexibilidade e facilitam o suporte a mecanismos de armazenamento que podem ser adicionados no futuro.

Nesse design, o componente de log primário pode ser uma classe não virtual, até mesmo lacrada. Você pode conectar qualquer conjunto de delegados para gravar as mensagens em diferentes mídias de armazenamento. O suporte interno para delegados multicast facilita o suporte a cenários em que as mensagens devem ser gravadas em vários locais (um arquivo e um console).

## Uma primeira implementação

Vamos começar pequeno: a implementação inicial aceitará novas mensagens e as gravará usando qualquer delegado anexo. Você pode começar com um delegado que grava mensagens no console.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(string msg)
    {
        if (WriteMessage is not null)
            WriteMessage(msg);
    }
}
```

A classe estática acima é a coisa mais simples que pode funcionar. Precisamos escrever a única implementação do método que grava mensagens no console:

```
C#  
  
public static class LoggingMethods  
{  
    public static void LogToConsole(string message)  
    {  
        Console.Error.WriteLine(message);  
    }  
}
```

Por fim, você precisa conectar o delegado anexando-o ao delegado WriteMessage declarado no agente:

```
C#  
  
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

## Práticas

Até agora, nossa amostra é bastante simples, mas ainda demonstra algumas das diretrizes importantes para designs que envolvem delegados.

Usar os tipos de delegados definidos no Core Framework torna mais fácil para os usuários trabalhem com os delegados. Você não precisa definir novos tipos e os desenvolvedores que usam sua biblioteca não precisam aprender novos tipos de delegados especializadas.

As interfaces usadas são tão mínimas e flexíveis quanto possível: para criar um novo agente de saída, você precisa criar um método. O método pode ser um método estático ou um método de instância. Ele pode ter qualquer acesso.

## Formatar saída

Vamos fazer esta primeira versão um pouco mais robusta e, então, começar a criar outros mecanismos de registro em log.

Em seguida, vamos adicionar alguns argumentos para o método `LogMessage()` para que sua classe de log crie mensagens mais estruturadas:

```
C#
```

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

Em seguida, vamos usar aquele argumento `Severity` para filtrar as mensagens que são enviadas para o log de saída.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static Severity LogLevel { get; set; } = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

## Práticas

Você adicionou novos recursos à infraestrutura de registro em log. Como somente o componente do agente está acoplado de forma muito flexível a qualquer mecanismo de saída, esses novos recursos podem ser adicionados sem afetar o código que implementa o delegado do agente.

Conforme prosseguir com sua criação, você verá mais exemplos de como esse acoplamento flexível permite maior versatilidade para atualizar partes do site sem alterar outros locais. De fato, em um aplicativo maior, as classes de saída do agente podem estar em um assembly diferente e nem mesmo precisar ser recriadas.

## Criar um segundo mecanismo de saída

O componente de Log estará indo bem. Vamos adicionar mais um mecanismo de saída que registra as mensagens em um arquivo. Esse será um mecanismo de saída de um pouco mais envolvido. Ele será uma classe que encapsula as operações de arquivo e garante que o arquivo sempre seja fechado após cada gravação. Isso garante que todos os dados sejam liberados no disco após cada mensagem ser gerada.

Aqui está o agente baseado em arquivo:

C#

```
public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
        }
    }
}
```

```
// So, while normally, catching an exception  
// and doing nothing isn't wise, it's really the  
// only reasonable option here.  
}  
}  
}
```

Após ter criado essa classe, você pode instanciá-la e ela anexa o método LogMessage ao componente do agente:

C#

```
var file = new FileLogger("log.txt");
```

Os dois não são mutuamente exclusivos. Você pode anexar os dois métodos de log e gerar mensagens para o console e um arquivo:

C#

```
var fileOutput = new FileLogger("log.txt");  
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the  
static class we utilized earlier
```

Posteriormente, mesmo no mesmo aplicativo, você pode remover um dos delegados sem problemas para o sistema:

C#

```
Logger.WriteMessage -= LoggingMethods.LogToConsole;
```

## Práticas

Agora, você adicionou um segundo manipulador de saída para o subsistema de registro em log. Este precisa de um pouco mais de infraestrutura para dar suporte ao sistema de arquivos corretamente. O delegado tem um método de instância. Ele também é um método particular. Não é necessário ter mais acessibilidade porque a infraestrutura de delegados pode conectar os delegados.

Em segundo lugar, o design baseado em delegados permite vários métodos de saída sem nenhum código extra. Não é necessário criar nenhuma infraestrutura adicional para dar suporte a vários métodos de saída. Eles simplesmente se tornam outro método na lista de invocação.

Dedique atenção especial ao código no método de saída do registro em log de arquivos. Ele é codificado para garantir que não gere nenhuma exceção. Embora isso nem sempre seja estritamente necessário, geralmente é uma boa prática. Se um dos métodos de delegado gerar uma exceção, os delegados restantes que fazem parte da invocação não serão invocados.

Como uma última observação, o agente de arquivos deve gerenciar seus recursos abrindo e fechando o arquivo em cada mensagem de log. Você pode optar por manter o arquivo aberto e implementar `IDisposable` para fechar o arquivo quando terminar. Cada método tem suas vantagens e desvantagens. Ambos criam um pouco mais de acoplamento entre as classes.

Nenhum código na classe `Logger` precisaria ser atualizado para dar suporte a qualquer um dos cenários.

## Manipular delegados nulos

Por fim, vamos atualizar o método `LogMessage` para que ele seja robusto para os casos em que nenhum mecanismo de saída está selecionado. A implementação atual gerará um `NullReferenceException` quando o delegado `WriteMessage` não tiver uma lista de invocação anexada. Talvez você prefira um design que continua silenciosamente quando não nenhum método tiver sido anexado. Isso é fácil usando o operador condicional nulo, combinado com o método `Delegate.Invoke()`:

```
C#  
  
public static void LogMessage(string msg)  
{  
    WriteMessage?.Invoke(msg);  
}
```

O operador condicional nulo (`?.`) entra em curto-círcuito quando o operando esquerdo (`WriteMessage` nesse caso) for nulo, o que significa que não é feita nenhuma tentativa de registrar uma mensagem.

Você não encontrará o método `Invoke()` listado na documentação de `System.Delegate` ou `System.MulticastDelegate`. O compilador gera um método `Invoke` fortemente tipado para qualquer tipo de delegado declarado. Neste exemplo, isso significa que `Invoke` usa um único argumento `string` e tem um tipo de retorno nulo.

## Resumo das práticas

Você já viu o início de um componente de log que poderia ser expandido com outros gravadores e outros recursos. Com o uso de delegados no design, esses diferentes componentes ficam acoplados de maneira flexível. Isso traz vários benefícios. É fácil criar mecanismos de saída e anexá-los ao sistema. Esses outros mecanismos precisam de apenas um método: o método que grava a mensagem de log. Trata-se de um design que é resiliente quando novos recursos são adicionados. O contrato necessário para qualquer gravador é implementar um método. Esse método pode ser estático ou de instância. Ele pode ser ter acesso público, privado ou qualquer outro acesso válido.

A classe de agente pode fazer vários aprimoramentos ou alterações sem introduzir alterações interruptivas. Assim como qualquer classe, você não pode modificar a API pública sem o risco de fazer alterações interruptivas. Mas, como o acoplamento entre o agente e qualquer mecanismo de saída ocorre somente por meio do delegado, nenhum outro tipo (como interfaces ou classes base) é envolvido. O acoplamento é o menor possível.

[Próximo](#)

# Introdução a eventos

Artigo • 10/05/2023

[Anterior](#)

Eventos são, assim como delegados, um mecanismo de *associação tardia*. De fato, os eventos são criados com base no suporte de linguagem para delegados.

Os eventos são uma forma de um objeto difundir (para todos os componentes interessados do sistema) que algo aconteceu. Qualquer outro componente pode assinar ao evento e ser notificado quando um evento for gerado.

Provavelmente, você usou eventos em alguma parte de sua programação. Muitos sistemas gráficos têm um modelo de evento para informar a interação do usuário. Esses eventos informariam movimentos do mouse, pressionamentos de botão e interações semelhantes. Esse é um dos cenários mais comuns, mas certamente não o único cenário em que eventos são usados.

Você pode definir os eventos que devem ser gerados para suas classes. Uma consideração importante ao trabalhar com eventos é que pode não haver nenhum objeto registrado para um determinado evento. Você deve escrever seu código de modo que ele não gere eventos quando nenhum ouvinte estiver configurado.

Assinar um evento também cria um acoplamento entre dois objetos (a origem do evento e o coletor do evento). Você precisa garantir que o coletor do evento cancele a assinatura da origem do evento quando não houver mais interesse nos eventos.

## Metas de design para o suporte a eventos

O design de linguagem para eventos tem como alvo as seguintes metas:

- Habilitar um acoplamento muito mínimo entre uma origem do evento e um coletor de eventos. Esses dois componentes não podem ter sido escritos pela mesma organização e podem até mesmo ser atualizados segundo cronogramas totalmente diferentes.
- Deve ser muito simples assinar um evento e cancelar a assinatura desse mesmo evento.
- As origens do evento devem dar suporte a vários assinantes de eventos. Elas também devem dar suporte a não ter assinantes de evento anexados.

Você pode ver que as metas para os eventos são muito semelhantes às metas para delegados. É por isso que o suporte à linguagem do evento é baseado no suporte à linguagem do delegado.

## Suporte de linguagem para eventos

A sintaxe para definir eventos e se inscrever ou cancelar a inscrição em eventos é uma extensão da sintaxe de delegados.

Para definir um evento, você use a palavra-chave `event`:

C#

```
public event EventHandler<FileListArgs> Progress;
```

O tipo de evento (`EventHandler<FileListArgs>` neste exemplo) deve ser um tipo delegado. Há uma série de convenções que você deve seguir ao declarar um evento. Normalmente, o tipo de delegado do evento tem um retorno nulo. Declarações de evento devem ser um verbo ou uma frase verbal. Use o tempo passado quando o evento relatar algo que aconteceu. Use um tempo verbal presente (por exemplo, `Closing`) para informar algo que está prestes a ocorrer. Frequentemente, usar o tempo presente indica que sua classe dá suporte a algum tipo de comportamento de personalização. Um dos cenários mais comuns é dar suporte ao cancelamento. Por exemplo, um evento `Closing` pode incluir um argumento que indicaria se a operação de encerramento deve continuar ou não. Outros cenários podem permitir que os chamadores modifiquem o comportamento atualizando propriedades dos argumentos do evento. Você pode acionar um evento para indicar uma próxima ação proposta que um algoritmo usará. O manipulador de eventos pode forçar uma ação diferente modificando as propriedades do argumento do evento.

Quando quiser acionar o evento, você pode chamar os manipuladores de eventos usando a sintaxe de invocação de delegado:

C#

```
Progress?.Invoke(this, new FileListArgs(file));
```

Conforme discutido na seção sobre [delegados](#), o operador `?` torna fácil garantir que você não tente acionar o evento quando não houver nenhum assinante do evento.

Assine um evento usando o operador `+=`:

C#

```
EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>
    Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;
```

O método de manipulador normalmente tem o prefixo "On" seguido pelo nome do evento, conforme mostrado acima.

Cancele a assinatura usando o operador `-=`:

C#

```
fileLister.Progress -= onProgress;
```

É importante declarar uma variável local para a expressão que representa o manipulador de eventos. Isso garante que o cancelamento da assinatura remova o manipulador. Se, em vez disso, você tiver usado o corpo da expressão lambda, você estará tentando remover um manipulador que nunca foi anexado, o que não faz nada.

No próximo artigo, você aprenderá mais sobre padrões de evento típicos e diferentes variações deste exemplo.

[Próximo](#)

# Padrões de evento .NET padrão

Artigo • 10/05/2023

[Anterior](#)

Os eventos do .NET geralmente seguem alguns padrões conhecidos. Adotar esses padrões significa que os desenvolvedores podem aproveitar o conhecimento desses padrões, que podem ser aplicados a qualquer programa de evento do .NET.

Vamos analisar esses padrões para que você obtenha todo o conhecimento que precisa a fim de criar origens do evento padrão e também assinar e processar eventos padrão em seu código.

## Assinaturas de delegado de evento

A assinatura padrão de um delegado de evento do .NET é:

C#

```
void EventRaised(object sender, EventArgs args);
```

O tipo de retorno é nulo. Os eventos são baseados em delegados e são delegados multicast. Isso dá suporte a vários assinantes de qualquer origem do evento. O único valor retornado de um método não ajusta a escala para vários assinantes do evento. Qual valor retornado a origem do evento vê depois de gerar um evento? Neste artigo, você verá como criar protocolos de evento que oferecem suporte a assinantes de evento que relatam informações para a origem do evento.

A lista de argumentos contém dois argumentos: o remetente e os argumentos do evento. O tipo de tempo de compilação de `sender` é `System.Object`, mas é provável que você conheça um tipo mais derivado que sempre estaria correto. Por convenção, use `object`.

O segundo argumento normalmente tem sido um tipo derivado de `System.EventArgs`. (Você verá na [próxima seção](#) que essa convenção não é mais imposta.) Se o tipo de evento não precisar de argumentos adicionais, você ainda fornecerá ambos os argumentos. Há um valor especial, o `EventArgs.Empty`, que você deve usar para indicar que o evento não contém nenhuma informação adicional.

Vamos criar uma classe que lista os arquivos em um diretório ou em qualquer um de seus subdiretórios, que seguem um padrão. Esse componente aciona um evento para

cada arquivo encontrado que corresponde ao padrão.

O uso de um modelo de evento fornece algumas vantagens de design. Você pode criar vários ouvintes de eventos que realizam ações diferentes quando um arquivo procurado é encontrado. A combinação de diferentes ouvintes pode criar algoritmos mais robustos.

Aqui está a declaração de argumento de evento inicial para localizar um arquivo pesquisado:

```
C#
```

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

Embora esse tipo se pareça com um tipo pequeno de somente dados, você deve seguir a convenção e torná-lo um tipo de referência (`class`). Isso significa que o objeto de argumento será passado por referência e todas as atualizações nos dados serão visualizadas por todos os assinantes. A primeira versão é um objeto imutável. É preferível tornar as propriedades em seu tipo de argumento de evento imutáveis. Dessa forma, um assinante não poderá alterar os valores antes que outro assinante os veja. (Há exceções, como você verá abaixo).

Em seguida, precisamos criar a declaração de evento na classe `FileSearcher`. O aproveitamento do tipo `EventHandler<T>` significa que não é necessário criar outra definição de tipo. Você simplesmente usa uma especialização genérica.

Vamos preencher a classe `FileSearcher` para pesquisar arquivos que correspondam a um padrão e acionar o evento correto quando uma correspondência for descoberta.

```
C#
```

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs>? FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory,
searchPattern))
        {
            RaiseFileFound(file);
        }
    }
}
```

```
}

    private void RaiseFileFound(string file) =>
        FileFound?.Invoke(this, new FileFoundArgs(file));
}
```

## Definir e gerar eventos semelhantes a campos

A maneira mais simples de adicionar um evento à sua classe é declarar esse evento como um campo público, como no exemplo anterior:

C#

```
public event EventHandler<FileFoundArgs>? FileFound;
```

Isso é semelhante à declaração de um campo público, o que parece ser uma prática ruim orientada a objetos. Você deseja proteger o acesso a dados por meio de propriedades ou métodos. Embora isso possa parecer uma prática ruim, o código gerado pelo compilador cria wrappers para que os objetos de evento só possam ser acessados de maneiras seguras. As únicas operações disponíveis em um evento semelhante a campo são adicionar manipulador:

C#

```
var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

e remover manipulador:

C#

```
fileLister.FileFound -= onFileFound;
```

Observe que há uma variável local para o manipulador. Se você usou o corpo de lambda, a operação remover não funcionará corretamente. Ela seria uma instância diferente do delegado e silenciosamente não faria nada.

O código fora da classe não pode acionar o evento nem executar outras operações.

## Valor retornados de assinantes de evento

Sua versão simples está funcionando bem. Vamos adicionar outro recurso: cancelamento.

Quando você acionar o evento encontrado, os ouvintes devem conseguir parar o processamento, se esse arquivo for o procurado.

Os manipuladores de eventos não retornam um valor, por isso você precisa comunicar isso de outra forma. O padrão de evento usa o objeto `EventArgs` para incluir campos que os assinantes de evento podem usar para comunicar o cancelamento.

Há dois padrões diferentes que podem ser usados, com base na semântica do contrato de cancelamento. Em ambos os casos, você adicionará um campo booleano no `EventArgs` para o evento de arquivo encontrado.

Um padrão permitiria a qualquer assinante cancelar a operação. Para esse padrão, o novo campo é inicializado para `false`. Qualquer assinante pode alterá-lo para `true`. Depois que todos os assinantes viram o evento acionado, o componente `FileSearcher` examina o valor booleano e toma uma ação.

O segundo padrão apenas cancelaria a operação se todos os assinantes quisessem que a operação fosse cancelada. Nesse padrão, o novo campo é inicializado para indicar que a operação deve ser cancelada e qualquer assinante poderia alterá-lo para indicar que a operação deve continuar. Depois que todos os assinantes viram o evento acionado, o componente `FileSearcher` examina o booleano e toma uma ação. Há uma etapa adicional nesse padrão: o componente precisa saber se algum assinante viu o evento. Se não houver nenhum assinante, o campo indicaria incorretamente um cancelamento.

Vamos implementar a primeira versão deste exemplo. Você precisa adicionar um campo booleano chamado `CancelRequested` ao tipo `FileFoundArgs`:

C#

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

Este novo campo é inicializado automaticamente para `false`, o valor padrão para um campo `Boolean`, para que você não cancele acidentalmente. A única alteração adicional no componente é verificar o sinalizador depois de acionar o evento, para ver se qualquer um dos assinantes solicitou um cancelamento:

```
C#  
  
private void SearchDirectory(string directory, string searchPattern)  
{  
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))  
    {  
        FileFoundArgs args = RaiseFileFound(file);  
        if (args.CancelRequested)  
        {  
            break;  
        }  
    }  
}  
  
private FileFoundArgs RaiseFileFound(string file)  
{  
    var args = new FileFoundArgs(file);  
    FileFound?.Invoke(this, args);  
    return args;  
}
```

Uma vantagem desse padrão é que ele não é uma alteração significativa. Nenhum dos assinantes solicitou um cancelamento antes e ainda não fizeram. Nenhuma parte do código de assinante precisa ser atualizado, a menos que eles queiram dar suporte ao novo protocolo de cancelamento. Ele é acoplado de forma bem livre.

Vamos atualizar o assinante para que ele solicite um cancelamento, depois de encontrar o primeiro executável:

```
C#  
  
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>  
{  
    Console.WriteLine(eventArgs.FoundFile);  
    eventArgs.CancelRequested = true;  
};
```

## Adicionar outra declaração de evento

Vamos adicionar mais um recurso e demonstrar outras expressões de linguagem para eventos. Vamos adicionar uma sobrecarga do método `Search` que percorre todas os

subdiretórios pesquisando arquivos.

Isso poderia se tornar uma operação demorada em um diretório com muitos subdiretórios. Vamos adicionar um evento que é acionado no início de cada nova pesquisa de diretório. Isso permite que os assinantes acompanhem o progresso e atualizem o usuário sobre o progresso. Todos os exemplos que você criou até agora são públicos. Vamos fazer com que esse seja um evento interno. Isso significa que você também pode fazer com que os tipos usados para os argumentos sejam internos.

Você começará criando a nova classe derivada EventArgs para relatar o novo diretório e o andamento.

C#

```
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int
completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Novamente, você pode seguir as recomendações para criar um tipo de referência imutável para os argumentos do evento.

Em seguida, defina o evento. Desta vez, você usará uma sintaxe diferente. Além de usar a sintaxe de campo, você pode explicitamente criar a propriedade com os manipuladores adicionar e remover. Nesta amostra, você não precisará de código extra nos manipuladores, mas será mostrado como criá-los.

C#

```
internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { _directoryChanged += value; }
    remove { _directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs>? _directoryChanged;
```

De muitas formas, o código que você escreverá aqui é bem parecido com o código que o compilador gera para as definições de evento de campo vistas anteriormente. Você cria o evento usando uma sintaxe muito parecida àquela utilizada para [propriedades](#). Observe que os manipuladores têm nomes diferentes: `add` e `remove`. Eles são chamados para assinar o evento ou cancelar a inscrição do evento. Observe que você também deve declarar um campo de suporte particular para armazenar a variável de evento. Ele é inicializado como null.

Em seguida, vamos adicionar a sobrecarga do método `Search` que percorre os subdiretórios e aciona os dois eventos. A maneira mais fácil de fazer isso é usar um argumento padrão para especificar que você deseja pesquisar todas as pastas:

```
C#  
  
public void Search(string directory, string searchPattern, bool  
searchSubDirs = false)  
{  
    if (searchSubDirs)  
    {  
        var allDirectories = Directory.GetDirectories(directory, "*.*",  
SearchOption.AllDirectories);  
        var completedDirs = 0;  
        var totalDirs = allDirectories.Length + 1;  
        foreach (var dir in allDirectories)  
        {  
            RaiseSearchDirectoryChanged(dir, totalDirs, completedDirs++);  
            // Search 'dir' and its subdirectories for files that match the  
search pattern:  
            SearchDirectory(dir, searchPattern);  
        }  
        // Include the Current Directory:  
        RaiseSearchDirectoryChanged(directory, totalDirs, completedDirs++);  
  
        SearchDirectory(directory, searchPattern);  
    }  
    else  
    {  
        SearchDirectory(directory, searchPattern);  
    }  
}  
  
private void SearchDirectory(string directory, string searchPattern)  
{  
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))  
    {  
        FileFoundArgs args = RaiseFileFound(file);  
        if (args.CancelRequested)  
        {  
            break;  
        }  
    }  
}
```

```
}

private void RaiseSearchDirectoryChanged(
    string directory, int totalDirs, int completedDirs) =>
    _directoryChanged?.Invoke(
        this,
        new SearchDirectoryArgs(directory, totalDirs, completedDirs));

private FileFoundArgs RaiseFileFound(string file)
{
    var args = new FileFoundArgs(file);
    FileFound?.Invoke(this, args);
    return args;
}
```

Neste momento, você pode executar o aplicativo, chamando a sobrecarga para pesquisar todos os subdiretórios. Não há nenhum assinante no novo evento `DirectoryChanged`, mas o uso da expressão `??.Invoke()` garante que isso funcione corretamente.

Vamos adicionar um manipulador para escrever uma linha que mostra o andamento na janela do console.

C#

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.WriteLine($"Entering '{eventArgs.CurrentSearchDirectory}' .");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs}
completed...");
```

Você viu os padrões que são seguidos em todo o ecossistema do .NET. Ao aprender esses padrões e convenções, você escreverá expressões idiomáticas de C# e .NET rapidamente.

## Confira também

- [Introdução a eventos](#)
- [Design de evento](#)
- [Manipular e gerar eventos](#)

Em seguida, você verá algumas alterações nesses padrões na versão mais recente do .NET.

[Próximo](#)

# O padrão de eventos atualizado do .NET Core Event

Artigo • 10/05/2023

[Anterior](#)

O artigo anterior abordou os padrões de eventos mais comuns. O .NET Core tem um padrão mais flexível. Nesta versão, a definição `EventHandler<TEventArgs>` não tem a restrição de que `TEventArgs` deve ser uma classe derivada de `System.EventArgs`.

Isso aumenta a flexibilidade para você e é compatível com versões anteriores. Vamos começar com a flexibilidade. A classe `System.EventArgs` introduz um método:

`MemberwiseClone()`, que cria uma cópia superficial do objeto. Esse método deve usar a reflexão para implementar sua funcionalidade para qualquer classe derivada de `EventArgs`. Essa funcionalidade é mais fácil de criar em uma classe derivada específica. Na prática, isso significa que a derivação de `System.EventArgs` é uma restrição que limita seus designs, mas não oferece nenhum benefício adicional. Na verdade, você pode alterar as definições de `FileEventArgs` e `SearchDirectoryEventArgs` para que eles não derivem de `EventArgs`. O programa funcionará exatamente da mesma forma.

Você também pode alterar o `SearchDirectoryEventArgs` para um struct, se você fizer mais uma alteração:

C#

```
internal struct SearchDirectoryEventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryEventArgs(string dir, int totalDirs, int
completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

A alteração adicional é chamar o construtor sem parâmetro antes de inserir o construtor que inicializa todos os campos. Sem esse acréscimo, as regras de C# informariam que as propriedades estão sendo acessadas antes de terem sido atribuídas.

Você não deve alterar o `FileEventArgs` de uma classe (tipo de referência) para um struct (tipo de valor). Isso ocorre porque o protocolo para manipular cancelamentos exige que os argumentos do evento sejam passados por referência. Se você fizesse a mesma alteração, a classe de pesquisa de arquivo nunca observaria as alterações feitas por qualquer um dos assinantes do evento. Uma nova cópia da estrutura seria usada para cada assinante e essa cópia seria uma cópia diferente daquela vista pelo objeto de pesquisa de arquivo.

Em seguida, vamos considerar como essa alteração pode ser compatível com versões anteriores. A remoção da restrição não afeta nenhum código existente. Qualquer tipo de argumento de evento existente ainda deriva de `System.EventArgs`. A compatibilidade com versões anteriores é um dos principais motivos pelos quais eles vão continuar derivando de `System.EventArgs`. Assinantes de eventos existentes serão assinantes de um evento que seguiu o padrão clássico.

Seguindo uma lógica semelhante, qualquer tipo de argumento de evento criado agora não teria assinantes nas bases de código existentes. Novos tipos de evento que não derivam de `System.EventArgs` não quebram essas bases de código.

## Eventos com assinantes assíncronos

Você tem um último padrão para aprender: como escrever corretamente os assinantes do evento que chamam o código assíncrono. O desafio é descrito no artigo em [async e await](#). Métodos assíncronos podem ter um tipo de retorno nulo, mas isso não é recomendável. Quando seu código de assinante de evento chama um método assíncrono, você não terá outra escolha além de criar um método `async void`. A assinatura do manipulador de eventos o exige.

Você precisa conciliar essas diretrizes opostas. De alguma forma, você precisa criar um método `async void` seguro. As noções básicas do padrão que você precisa implementar estão abaixo:

C#

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
    }
}
```

```
// Consider gracefully, and quickly exiting.  
}  
};
```

Primeiro, observe que o manipulador está marcado como um manipulador assíncrono. Como está sendo atribuído a um tipo de delegado de manipulador de eventos, ele terá um tipo de retorno nulo. Isso significa que você deve seguir o padrão mostrado no manipulador e não permitir que qualquer exceção seja gerada fora do contexto do manipulador assíncrono. Como ele não retorna uma tarefa, não há nenhuma tarefa que pode relatar o erro entrando no estado de falha. Como o método é assíncrono, ele não pode simplesmente gerar a exceção. (O método de chamada continuou a execução porque é `async`.) O comportamento de runtime real será definido de forma diferente para ambientes diferentes. Ele pode encerrar o thread ou o processo que possui o thread ou deixar o processo em um estado indeterminado. Todos esses resultados potenciais são altamente indesejáveis.

É por isso que você deve encapsular a instrução `await` para a tarefa assíncrona em seu próprio bloco de teste. Se isso causar uma tarefa com falha, você pode registrar o erro em log. Se for um erro do qual não é possível recuperar o aplicativo, você pode sair do programa rápida e normalmente

Essas são as principais atualizações do padrão de eventos do .NET. Você verá muitos exemplos das versões anteriores nas bibliotecas com que trabalhar. No entanto, você também precisa compreender quais são os padrões mais recentes.

O próximo artigo desta série ajuda a distinguir entre o uso de `delegates` e de `events` em seus designs. Eles são conceitos similares e artigo o ajudará a tomar a melhor decisão para seus programas.

[Próximo](#)

# Distinção entre Delegados e Eventos

Artigo • 09/05/2023

[Anterior](#)

Desenvolvedores que são novos na plataforma .NET Core geralmente têm dificuldades para decidir entre um design baseado em `delegates` e um design baseado em `events`. A escolha entre delegados ou eventos geralmente é difícil, pois os dois recursos de idioma são semelhantes. De fato, os eventos são criados usando o suporte de linguagem para delegados.

Ambos oferecem um cenário de associação tardia: eles permitem cenários em que um componente se comunica chamando um método conhecido somente em tempo de execução. Ambas dão suporte a métodos de assinante único e vários assinantes. Você pode ver esse suporte ser chamado de singlecast e multicast. Ambas dão suporte a uma sintaxe semelhante para adicionar e remover manipuladores. Por fim, acionar um evento e chamar um delegado usam exatamente a mesma sintaxe de chamada de método. As duas até mesmo dão suporte à mesma sintaxe de método `Invoke()` para uso com o operador `?`.

Com todas essas semelhanças, é fácil de ter problemas para determinar quando usar qual.

## Ouvir eventos é opcional

O aspecto mais importante para determinar qual recurso da linguagem usar é se é necessário ou não que haja um assinante anexado. Se o seu código precisar chamar o código fornecido pelo assinante, você deverá usar um design baseado em delegados quando precisar implementar o retorno de chamada. Se seu código puder concluir todo o seu trabalho sem chamar nenhum assinante, você deverá usar um design baseado em eventos.

Considere os exemplos criados durante esta seção. O código que você criou usando `List.Sort()` deve receber uma função de comparador para classificar corretamente os elementos. Consultas de LINQ devem receber delegados para determinar quais elementos retornar. Ambos usaram um design criado com delegados.

Considere o evento `Progress`. Ele relata o progresso de uma tarefa. A tarefa continua quer haja ouvintes ou não. O `FileSearcher` é outro exemplo. Ele ainda pesquisaria e localizaria todos os arquivos que foram buscados, mesmo que não houvesse assinantes do evento anexados. Controles de UX ainda funcionam corretamente, mesmo quando

não houver nenhum assinante ouvindo os eventos. Ambos usam os designs baseados em eventos.

## Valores retornados exigem delegados

Outra consideração é o protótipo do método que você gostaria de ter para seu método de delegado. Como você viu, todos os delegados usados para os eventos têm um tipo retornado nulo. Você também viu que há expressões para criar manipuladores de eventos que passam informações para as origens dos eventos modificando propriedades do objeto de argumento de evento. Embora essas expressões funcionem, elas não são tão naturais quanto retornar um valor de um método.

Observe que essas duas heurísticas geralmente podem estar presentes: se o método de delegado retornar um valor, provavelmente ele terá impacto sobre o algoritmo de alguma forma.

## Eventos têm invocação privada

Classes diferentes daquela em que um evento está contido só podem adicionar e remover ouvintes de eventos; só a classe que contém um evento pode invocá-lo. Eventos normalmente são membros de classe pública. Em comparação, os delegados geralmente são passados como parâmetros e armazenados como membros de classe privada, se forem armazenados.

## Ouvintes de evento frequentemente têm vida útil mais longa

O fato de que os ouvintes de evento têm tempos de vida mais longos é uma justificativa um pouco mais fraca. No entanto, você pode descobrir que designs baseados em eventos são mais naturais quando a origem do evento for gerar eventos durante um longo período de tempo. Você pode ver exemplos de design baseado em evento para controles de UX em muitos sistemas. Quando você assina um evento, a origem do evento pode gerar eventos durante o tempo de vida do programa. (Você pode cancelar a assinatura de eventos quando não precisar mais deles.)

Compare isso com vários designs baseados em delegados, em que um delegado é usado como um argumento para um método e o delegado não é usado depois que o método é retornado.

## Avalie cuidadosamente

As considerações acima não são regras rígidas e óbvias. Em vez disso, elas são diretrizes que podem ajudá-lo a decidir qual opção é melhor para seu uso específico. Como elas são semelhantes, você pode até mesmo fazer protótipos das suas e considerar com qual seria mais natural trabalhar. Ambas lidam bem com cenários de associação tardia. Use a que comunica melhor o seu design.

# LINQ (Consulta Integrada à Linguagem)

Artigo • 15/02/2023

O LINQ (consulta integrada à linguagem) é o nome de um conjunto de tecnologias com base na integração de recursos de consulta diretamente na linguagem C#.

Tradicionalmente, consultas feitas em dados são expressas como cadeias de caracteres simples sem verificação de tipo no tempo de compilação ou suporte a IntelliSense. Além disso, você precisará aprender uma linguagem de consulta diferente para cada tipo de fonte de dados: bancos de dados SQL, documentos XML, vários serviços Web etc. Com o LINQ, uma consulta é um constructo de linguagem de primeira classe, como classes, métodos, eventos.

Para um desenvolvedor que escreve consultas, a parte mais visível "integrada à linguagem" do LINQ é a expressão de consulta. As expressões de consulta são uma *sintaxe declarativa de consulta*. Usando a sintaxe de consulta, você pode executar operações de filtragem, ordenação e agrupamento em fontes de dados com o mínimo de código. Você pode usar os mesmos padrões de expressão de consulta básica para consultar e transformar dados em bancos de dados SQL, conjuntos de dados do ADO.NET, documentos XML e fluxos e coleções .NET.

O exemplo a seguir mostra a operação de consulta completa. A operação completa inclui a criação de uma fonte de dados, definição da expressão de consulta e execução da consulta em uma instrução `foreach`.

```
C#  
  
// Specify the data source.  
int[] scores = { 97, 92, 81, 60 };  
  
// Define the query expression.  
IQueryable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;  
  
// Execute the query.  
foreach (int i in scoreQuery)  
{  
    Console.Write(i + " ");  
}  
  
// Output: 97 92 81
```

# Visão geral da expressão de consulta

- Expressões de consulta podem ser usadas para consultar e transformar dados de qualquer fonte de dados habilitada para LINQ. Por exemplo, uma única consulta pode recuperar dados de um Banco de Dados SQL e produzir um fluxo XML como saída.
- As expressões de consulta são fáceis de entender porque elas usam muitos constructos de linguagem C# familiares.
- As variáveis em uma expressão de consulta são fortemente tipadas, embora em muitos casos você não precise fornecer o tipo explicitamente, pois o compilador pode inferir nele. Para obter mais informações, consulte [Relacionamentos de tipo em operações de consulta LINQ](#).
- Uma consulta não é executada até que você itere sobre a variável de consulta, por exemplo, em uma instrução `foreach`. Para obter mais informações, consulte [Introdução a consultas LINQ](#).
- No tempo de compilação, as expressões de consulta são convertidas em chamadas de método do operador de consulta padrão de acordo com as regras definidas na especificação do C#. Qualquer consulta que pode ser expressa usando sintaxe de consulta também pode ser expressa usando sintaxe de método. No entanto, na maioria dos casos, a sintaxe de consulta é mais legível e concisa. Para obter mais informações, consulte [Especificação da linguagem C#](#) e [Visão geral de operadores de consulta padrão](#).
- Como uma regra ao escrever consultas LINQ, recomendamos que você use a sintaxe de consulta sempre que possível e a sintaxe de método sempre que necessário. Não há semântica ou diferença de desempenho entre as duas formas. As expressões de consulta são geralmente mais legíveis do que as expressões equivalentes escritas na sintaxe de método.
- Algumas operações de consulta, como [Count](#) ou [Max](#), não apresentam cláusulas de expressão de consulta equivalentes e, portanto, devem ser expressas como chamadas de método. A sintaxe de método pode ser combinada com a sintaxe de consulta de várias maneiras. Para obter mais informações, consulte [Sintaxe de consulta e sintaxe de método em LINQ](#).
- As expressões de consulta podem ser compiladas para árvores de expressão ou delegados, dependendo do tipo ao qual a consulta é aplicada. As consultas `IEnumerable<T>` são compiladas para representantes. As consultas `IQueryable` e

`IQueryable<T>` são compiladas para árvores de expressão. Para obter mais informações, consulte [Árvores de expressão](#).

## Próximas etapas

Para obter mais detalhes sobre o LINQ, comece se familiarizando com alguns conceitos básicos em [Noções básicas sobre expressões de consulta](#), e, em seguida, leia a documentação para a tecnologia LINQ na qual você está interessado:

- Documentos XML: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to Entities](#)
- Coleções do .NET, arquivos, cadeias de caracteres, etc.: [LINQ to Objects](#)

Para saber mais sobre o LINQ, consulte [LINQ em C#](#).

Para começar a trabalhar com o LINQ em C#, consulte o tutorial [Trabalhando com LINQ](#).

# Noções básicas sobre expressões de consulta

Artigo • 07/04/2023

Este artigo apresenta os conceitos básicos relacionados a expressões de consulta em C#.

## O que é uma consulta e o que ela faz?

Uma *consulta* é um conjunto de instruções que descreve quais dados recuperar de uma determinada fonte de dados (ou fontes) e que forma e organização os dados retornados devem ter. Uma consulta é diferente dos resultados que ela produz.

Em geral, os dados de origem são organizados de forma lógica como uma sequência de elementos do mesmo tipo. Por exemplo, uma tabela de banco de dados SQL contém uma sequência de linhas. Em um arquivo XML, há uma "sequência" de elementos XML (embora eles estejam organizados hierarquicamente em uma estrutura de árvore). Uma coleção na memória contém uma sequência de objetos.

Do ponto de vista do aplicativo, o tipo específico e a estrutura dos dados de origem original não são importantes. O aplicativo sempre vê os dados de origem como uma coleção de `IEnumerable<T>` ou de `IQueryable<T>`. Por exemplo, no LINQ to XML, os dados de origem ficam visíveis como um `IEnumerable< XElement >`.

Dada essa sequência de origem, uma consulta pode executar uma das três ações:

- Recuperar um subconjunto dos elementos para produzir uma nova sequência sem modificar os elementos individuais. A consulta pode, em seguida, classificar ou agrupar a sequência retornada de várias maneiras, conforme mostrado no exemplo a seguir (suponha que `scores` é um `int[]`):

C#

```
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
```

- Recuperar uma sequência de elementos, como no exemplo anterior, mas transformá-los em um novo tipo de objeto. Por exemplo, uma consulta pode recuperar apenas os sobrenomes de determinados registros de cliente em uma

fonte de dados. Ou pode recuperar o registro completo e, em seguida, usá-lo para construir outro tipo de objeto na memória ou até mesmo dados XML antes de gerar a sequência de resultados final. O exemplo a seguir mostra uma projeção de um `int` para um `string`. Observe o novo tipo de `highScoresQuery`.

C#

```
IEnumerable<string> highScoresQuery2 =  
    from score in scores  
    where score > 80  
    orderby score descending  
    select $"The score is {score}";
```

- Recuperar um valor singleton sobre os dados de origem, como:
  - O número de elementos que correspondem a uma determinada condição.
  - O elemento que tem o maior ou menor valor.
  - O primeiro elemento que corresponde a uma condição ou a soma dos valores específicos em um conjunto de elementos especificado. Por exemplo, a consulta a seguir retorna o número pontuações maiores que 80 na matriz de inteiros `scores`:

C#

```
int highScoreCount = (  
    from score in scores  
    where score > 80  
    select score  
) .Count();
```

No exemplo anterior, observe o uso de parênteses ao redor da expressão de consulta antes da chamada para o método `Count`. Também é possível expressar isso usando uma nova variável para armazenar o resultado concreto. Essa técnica é mais legível porque mantém a variável que armazena a consulta separada da consulta que armazena um resultado.

C#

```
IEnumerable<int> highScoresQuery3 =  
    from score in scores  
    where score > 80  
    select score;  
  
int scoreCount = highScoresQuery3.Count();
```

No exemplo anterior, a consulta é executada na chamada para `Count`, pois `Count` deve iterar os resultados para determinar o número de elementos retornados por `highScoresQuery`.

## O que é uma expressão de consulta?

Uma *expressão de consulta* é uma consulta expressada na sintaxe da consulta. Uma expressão de consulta é um constructo de linguagem de primeira classe. É exatamente como qualquer outra expressão e pode ser usada em qualquer contexto em que uma expressão C# é válida. Uma expressão de consulta consiste em um conjunto de cláusulas escritas em uma sintaxe declarativa semelhante ao SQL ou XQuery. Cada cláusula, por sua vez, contém uma ou mais expressões C# e essas expressões podem ser uma expressão de consulta ou conter uma expressão de consulta.

Uma expressão de consulta deve começar com uma cláusula `from` e deve terminar com uma cláusula `select` ou `group`. Entre a primeira cláusula `from` e a última cláusula `select` ou `group`, ela pode conter uma ou mais dessas cláusulas opcionais: `where`, `orderby`, `join`, `let` e até mesmo cláusulas `from` adicionais. Você também pode usar a palavra-chave `into` para permitir que o resultado de uma cláusula `join` ou `group` sirva como a fonte para cláusulas de consulta adicionais na mesma expressão de consulta.

## Variável da consulta

Em LINQ, uma variável de consulta é qualquer variável que armazena uma *consulta* em vez dos *resultados* de uma consulta. Mais especificamente, uma variável de consulta é sempre um tipo enumerável que produzirá uma sequência de elementos quando for iterada em uma instrução `foreach` ou uma chamada direta para seu método `IEnumerator.MoveNext`.

O exemplo de código a seguir mostra uma expressão de consulta simples com uma fonte de dados, uma cláusula de filtragem, uma cláusula de ordenação e nenhuma transformação dos elementos de origem. A cláusula `select` termina a consulta.

```
C#  
  
// Data source.  
int[] scores = { 90, 71, 82, 93, 75, 82 };  
  
// Query Expression.  
IEnumerable<int> scoreQuery = //query variable  
    from score in scores //required  
    where score > 80 // optional  
    orderby score descending // optional
```

```

    select score; //must end with select or group

// Execute the query to produce the results
foreach (int testScore in scoreQuery)
{
    Console.WriteLine(testScore);
}

// Output: 93 90 82 82

```

No exemplo anterior, `scoreQuery` é uma *variável de consulta*, o que às vezes é chamado apenas de uma *consulta*. A variável de consulta não armazena nenhum dado de resultado real, que é produzido no loop `foreach`. E quando instrução `foreach` é executada, os resultados da consulta não são retornados pela variável de consulta `scoreQuery`. Em vez disso, eles são retornados pela variável de iteração `testScore`. A variável `scoreQuery` pode ser iterada em um segundo loop `foreach`. Ele produzirá os mesmos resultados contanto que nem ele nem a fonte de dados tenham sido modificados.

Uma variável de consulta pode armazenar uma consulta que é expressada na sintaxe de consulta ou na sintaxe de método ou uma combinação das duas. Nos exemplos a seguir, `queryMajorCities` e `queryMajorCities2` são variáveis de consulta:

C#

```

//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population >
100000);

```

Por outro lado, os dois exemplos a seguir mostram variáveis que não são variáveis de consulta, embora sejam inicializadas com uma consulta. Elas não são variáveis de consulta porque armazenam resultados:

C#

```

int highestScore = (
    from score in scores
    select score
).Max();

// or split the expression

```

```
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
highScore = scores.Max();
```

C#

```
List<City> largeCitiesList = (
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city
).ToList();

// or split the expression
IQueryable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();
```

Para obter mais informações sobre as diferentes maneiras de expressar consultas, consulte [Sintaxe de consulta e sintaxe de método em LINQ](#).

## Tipagem explícita e implícita de variáveis de consulta

Esta documentação normalmente fornece o tipo explícito da variável de consulta para mostrar a relação de tipo entre a variável de consulta e a [cláusula select](#). No entanto, você também pode usar a palavra-chave `var` para instruir o compilador a inferir o tipo de uma variável de consulta (ou qualquer outra variável local) em tempo de compilação. Por exemplo, o exemplo de consulta que foi mostrado anteriormente neste tópico também pode ser expressado usando a tipagem implícita:

C#

```
// Use of var is optional here and in all queries.
// queryCities is an IEnumerable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

Para obter mais informações, consulte [Variáveis locais de tipo implícito](#) e [Relacionamentos de tipo em operações de consulta LINQ](#).

## Iniciando uma expressão de consulta

Uma expressão de consulta deve começar com uma cláusula `from`. Especifica uma fonte de dados junto com uma variável de intervalo. A variável de intervalo representa cada elemento sucessivo na sequência de origem como a sequência de origem que está sendo percorrida. A variável de intervalo é fortemente tipada com base no tipo dos elementos na fonte de dados. No exemplo a seguir, como `countries` é uma matriz de objetos `Country`, a variável de intervalo também é tipada como `Country`. Como a variável de intervalo é fortemente tipada, você pode usar o operador ponto para acessar todos os membros disponíveis do tipo.

C#

```
IEnumerable<Country> countryAreaQuery =  
    from country in countries  
    where country.Area > 500000 //sq km  
    select country;
```

A variável de intervalo está no escopo até a consulta ser encerrada com ponto e vírgula ou com uma cláusula [continuation](#).

Uma expressão de consulta pode conter várias cláusulas `from`. Use cláusulas `from` adicionais quando cada elemento na sequência de origem for uma coleção em si ou contiver uma coleção. Por exemplo, suponha que você tem uma coleção de objetos `Country` e cada um dos quais contém uma coleção de objetos `City` chamada `Cities`. Para consultar os objetos `City` em cada `Country`, use duas cláusulas `from` como mostrado aqui:

C#

```
IEnumerable<City> cityQuery =  
    from country in countries  
    from city in country.Cities  
    where city.Population > 10000  
    select city;
```

Para obter mais informações, consulte [Cláusula from](#).

## Encerrando uma expressão de consulta

Uma expressão de consulta deve ser encerrada com uma cláusula `group` ou uma cláusula `select`.

## Cláusula group

Use a cláusula `group` para produzir uma sequência de grupos organizada por uma chave que você especificar. A chave pode ter qualquer tipo de dados. Por exemplo, a consulta a seguir cria uma sequência de grupos que contém um ou mais objetos `Country` e cuja chave é um tipo `char` com o valor sendo a primeira letra dos nomes dos países.

C#

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

Para obter mais informações sobre o agrupamento, consulte [Cláusula group](#).

## Cláusula select

Use a cláusula `select` para produzir todos os outros tipos de sequências. Uma cláusula `select` simples produz apenas uma sequência do mesmo tipo dos objetos contidos na fonte de dados. Neste exemplo, a fonte de dados contém objetos `Country`. A cláusula `orderby` simplesmente classifica os elementos em uma nova ordem e a cláusula `select` produz uma sequência dos objetos `Country` reordenados.

C#

```
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

A cláusula `select` pode ser usada para transformar dados de origem em sequências de novos tipos. Essa transformação também é chamada de *projeção*. No exemplo a seguir, a cláusula `select` projeta uma sequência de tipos anônimos que contém apenas um subconjunto dos campos no elemento original. Observe que os novos objetos são inicializados usando um inicializador de objeto.

C#

```
// Here var is required because the query
// produces an anonymous type.
var queryNameAndPop =
    from country in countries
    select new
    {
        Name = country.Name,
        Pop = country.Population
    };

```

Para obter mais informações sobre todas as maneiras que uma cláusula `select` pode ser usada para transformar os dados de origem, consulte [Cláusula select](#).

## Continuações com *into*

Você pode usar a palavra-chave `into` em uma cláusula `select` ou `group` para criar um identificador temporário que armazena uma consulta. Faça isso quando precisar executar operações de consulta adicionais em uma consulta após a operação de agrupamento ou seleção. No exemplo a seguir `countries` são agrupados de acordo com a população em intervalos de 10 milhões. Depois que esses grupos são criados, cláusulas adicionais filtram alguns grupos e, em seguida, classificam os grupos em ordem crescente. Para executar essas operações adicionais, a continuação representada por `countryGroup` é necessária.

C#

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int)country.Population / 10_000_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
    {
        Console.WriteLine(country.Name + ":" + country.Population);
    }
}
```

Para obter mais informações, consulte [into](#).

# Filtragem, ordenação e junção

Entre a cláusula `from` inicial e a cláusula `select` ou `group` final, todas as outras cláusulas (`where`, `join`, `orderby`, `from`, `let`) são opcionais. Todas as cláusulas opcionais podem ser usadas várias vezes ou nenhuma vez no corpo de uma consulta.

## Cláusula where

Use a cláusula `where` para filtrar os elementos dos dados de origem com base em uma ou mais expressões de predicado. A cláusula `where` no exemplo a seguir tem um predicado com duas condições.

C#

```
IEnumerable<City> queryCityPop =  
    from city in cities  
    where city.Population < 200000 && city.Population > 100000  
    select city;
```

Para obter mais informações, consulte [Cláusula where](#).

## Cláusula orderby

Use a cláusula `orderby` para classificar os resultados em ordem crescente ou decrescente. Você também pode especificar as ordens de classificação secundárias. O exemplo a seguir executa uma classificação primária nos objetos `country` usando a propriedade `Area`. Em seguida, ele executa a classificação secundária usando a propriedade `Population`.

C#

```
IEnumerable<Country> querySortedCountries =  
    from country in countries  
    orderby country.Area, country.Population descending  
    select country;
```

A palavra-chave `ascending` é opcional. Será a ordem de classificação padrão se nenhuma ordem for especificada. Para obter mais informações, consulte [Cláusula orderby](#).

## Cláusula join

Use a cláusula `join` para associar e/ou combinar elementos de uma fonte de dados com elementos de outra fonte de dados com base em uma comparação de igualdade entre as chaves especificadas em cada elemento. Na LINQ, as operações `join` são executadas em sequências de objetos cujos elementos são de tipos diferentes. Após ter unido duas sequências, você deve usar uma instrução `select` ou `group` para especificar qual elemento armazenar na sequência de saída. Você também pode usar um tipo anônimo para combinar propriedades de cada conjunto de elementos associados em um novo tipo para a sequência de saída. O exemplo a seguir associa objetos `prod` cuja propriedade `Category` corresponde a uma das categorias na matriz de cadeias de caracteres `categories`. Os produtos cujos `category` não correspondem a nenhuma cadeia de caracteres `categories` são filtrados. A instrução `select` projeta um novo tipo cujas propriedades são obtidas de ambos `cat` e `prod`.

C#

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new
    {
        Category = cat,
        Name = prod.Name
    };
```

Você também pode executar uma junção de grupo armazenando os resultados da operação `join` em uma variável temporária usando a palavra-chave `into`. Para obter mais informações, consulte [Cláusula join](#).

## Cláusula let

Use a cláusula `let` para armazenar o resultado de uma expressão, como uma chamada de método, em uma nova variável de intervalo. No exemplo a seguir, a variável de intervalo `firstName` armazena o primeiro elemento da matriz de cadeias de caracteres que é retornado pelo `Split`.

C#

```
string[] names = { "Svetlana Omelchenko", "Claire O'Donnell", "Sven
Mortensen", "Cesar Garcia" };
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;
```

```
foreach (string s in queryFirstNames)
{
    Console.WriteLine(s + " ");
}

//Output: Svetlana Claire Sven Cesar
```

Para obter mais informações, consulte [Cláusula let](#).

## Subconsultas em uma expressão de consulta

Uma cláusula de consulta pode conter uma expressão de consulta, que às vezes é chamada de *subconsulta*. Cada subconsulta começa com sua própria cláusula `from` que não necessariamente aponta para a mesma fonte de dados na primeira cláusula `from`. Por exemplo, a consulta a seguir mostra uma expressão de consulta que é usada na instrução `select` para recuperar os resultados de uma operação de agrupamento.

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.ExamScores.Average()
        ).Max()
    };
}
```

Para obter mais informações, confira [Executar uma subconsulta em uma operação de agrupamento](#).

## Confira também

- [Guia de programação em C#](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Palavras-chave de consulta \(LINQ\)](#)
- [Visão geral de operadores de consulta padrão](#)

# LINQ (Consulta Integrada à Linguagem)

Artigo • 15/02/2023

O LINQ (consulta integrada à linguagem) é o nome de um conjunto de tecnologias com base na integração de recursos de consulta diretamente na linguagem C#.

Tradicionalmente, consultas feitas em dados são expressas como cadeias de caracteres simples sem verificação de tipo no tempo de compilação ou suporte a IntelliSense. Além disso, você precisará aprender uma linguagem de consulta diferente para cada tipo de fonte de dados: bancos de dados SQL, documentos XML, vários serviços Web etc. Com o LINQ, uma consulta é um constructo de linguagem de primeira classe, como classes, métodos, eventos.

Para um desenvolvedor que escreve consultas, a parte mais visível "integrada à linguagem" do LINQ é a expressão de consulta. As expressões de consulta são uma *sintaxe declarativa de consulta*. Usando a sintaxe de consulta, você pode executar operações de filtragem, ordenação e agrupamento em fontes de dados com o mínimo de código. Você pode usar os mesmos padrões de expressão de consulta básica para consultar e transformar dados em bancos de dados SQL, conjuntos de dados do ADO.NET, documentos XML e fluxos e coleções .NET.

O exemplo a seguir mostra a operação de consulta completa. A operação completa inclui a criação de uma fonte de dados, definição da expressão de consulta e execução da consulta em uma instrução `foreach`.

```
C#  
  
// Specify the data source.  
int[] scores = { 97, 92, 81, 60 };  
  
// Define the query expression.  
IQueryable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;  
  
// Execute the query.  
foreach (int i in scoreQuery)  
{  
    Console.Write(i + " ");  
}  
  
// Output: 97 92 81
```

# Visão geral da expressão de consulta

- Expressões de consulta podem ser usadas para consultar e transformar dados de qualquer fonte de dados habilitada para LINQ. Por exemplo, uma única consulta pode recuperar dados de um Banco de Dados SQL e produzir um fluxo XML como saída.
- As expressões de consulta são fáceis de entender porque elas usam muitos constructos de linguagem C# familiares.
- As variáveis em uma expressão de consulta são fortemente tipadas, embora em muitos casos você não precise fornecer o tipo explicitamente, pois o compilador pode inferir nele. Para obter mais informações, consulte [Relacionamentos de tipo em operações de consulta LINQ](#).
- Uma consulta não é executada até que você itere sobre a variável de consulta, por exemplo, em uma instrução `foreach`. Para obter mais informações, consulte [Introdução a consultas LINQ](#).
- No tempo de compilação, as expressões de consulta são convertidas em chamadas de método do operador de consulta padrão de acordo com as regras definidas na especificação do C#. Qualquer consulta que pode ser expressa usando sintaxe de consulta também pode ser expressa usando sintaxe de método. No entanto, na maioria dos casos, a sintaxe de consulta é mais legível e concisa. Para obter mais informações, consulte [Especificação da linguagem C#](#) e [Visão geral de operadores de consulta padrão](#).
- Como uma regra ao escrever consultas LINQ, recomendamos que você use a sintaxe de consulta sempre que possível e a sintaxe de método sempre que necessário. Não há semântica ou diferença de desempenho entre as duas formas. As expressões de consulta são geralmente mais legíveis do que as expressões equivalentes escritas na sintaxe de método.
- Algumas operações de consulta, como [Count](#) ou [Max](#), não apresentam cláusulas de expressão de consulta equivalentes e, portanto, devem ser expressas como chamadas de método. A sintaxe de método pode ser combinada com a sintaxe de consulta de várias maneiras. Para obter mais informações, consulte [Sintaxe de consulta e sintaxe de método em LINQ](#).
- As expressões de consulta podem ser compiladas para árvores de expressão ou delegados, dependendo do tipo ao qual a consulta é aplicada. As consultas `IEnumerable<T>` são compiladas para representantes. As consultas `IQueryable` e

`IQueryable<T>` são compiladas para árvores de expressão. Para obter mais informações, consulte [Árvores de expressão](#).

## Próximas etapas

Para obter mais detalhes sobre o LINQ, comece se familiarizando com alguns conceitos básicos em [Noções básicas sobre expressões de consulta](#), e, em seguida, leia a documentação para a tecnologia LINQ na qual você está interessado:

- Documentos XML: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to Entities](#)
- Coleções do .NET, arquivos, cadeias de caracteres, etc.: [LINQ to Objects](#)

Para saber mais sobre o LINQ, consulte [LINQ em C#](#).

Para começar a trabalhar com o LINQ em C#, consulte o tutorial [Trabalhando com LINQ](#).

# Escrever consultas LINQ em C#

Artigo • 20/07/2023

A maioria das consultas na documentação introdutória do LINQ (Consulta Integrada à Linguagem) é escrita com o uso da sintaxe de consulta declarativa do LINQ. No entanto, a sintaxe de consulta deve ser convertida em chamadas de método para o CLR (Common Language Runtime) do .NET quando o código for compilado. Essas chamadas de método invocam os operadores de consulta padrão, que têm nomes como `Where`, `Select`, `GroupBy`, `Join`, `Max` e `Average`. Você pode chamá-los diretamente usando a sintaxe de método em vez da sintaxe de consulta.

A sintaxe de consulta e a sintaxe de método são semanticamente idênticas, mas muitas pessoas acham a sintaxe de consulta mais simples e fácil de ler. Algumas consultas devem ser expressadas como chamadas de método. Por exemplo, você deve usar uma chamada de método para expressar uma consulta que recupera o número de elementos que correspondem a uma condição especificada. Você também deve usar uma chamada de método para uma consulta que recupera o elemento que tem o valor máximo em uma sequência de origem. A documentação de referência para os operadores de consulta padrão no namespace `System.Linq` geralmente usa a sintaxe de método. Portanto, mesmo ao começar a escrever consultas LINQ, é útil que você esteja familiarizado com a forma de uso da sintaxe de método em consultas e nas próprias expressões de consulta.

## Métodos de extensão do operador de consulta padrão

O exemplo a seguir mostra uma *expressão de consulta* simples e a consulta semanticamente equivalente escrita como uma *consulta baseada em método*.

C#

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
```

```

    select num;

    //Method syntax:
    IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 ==
0).OrderBy(n => n);

        foreach (int i in numQuery1)
    {
        Console.Write(i + " ");
    }
    Console.WriteLine(System.Environment.NewLine);
    foreach (int i in numQuery2)
    {
        Console.Write(i + " ");
    }

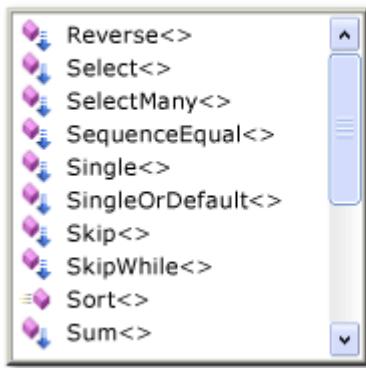
    // Keep the console open in debug mode.
    Console.WriteLine(System.Environment.NewLine);
    Console.WriteLine("Press any key to exit");
    Console.ReadKey();
}
/*
Output:
6 8 10 12
6 8 10 12
*/

```

A saída dos dois exemplos é idêntica. Você pode ver que o tipo da variável de consulta é o mesmo em ambas as formas: `IEnumerable<T>`.

Para entender a consulta baseada em método, vamos examiná-la melhor. No lado direito da expressão, observe que a cláusula `where` agora é expressa como um método de instância no objeto `numbers`, que, como você deve se lembrar, tem um tipo de `IEnumerable<int>`. Se você estiver familiarizado com a interface `IEnumerable<T>` genérica, você saberá que ela não tem um método `Where`. No entanto, se você invocar a lista de conclusão do IntelliSense no IDE do Visual Studio, verá não apenas um método `Where`, mas muitos outros métodos como `Select`, `SelectMany`, `Join` e `Orderby`. Esses são todos os operadores de consulta padrão.

```
List<string> list = new List<string>();  
list.|
```



Embora pareça como se `IEnumerable<T>` tivesse sido redefinido para incluir esses métodos adicionais, na verdade esse não é o caso. Os operadores de consulta padrão são implementados como um novo tipo de método chamado *métodos de extensão*. Métodos de extensão "estendem" um tipo existente, eles podem ser chamados como se fossem métodos de instância no tipo. Os operadores de consulta padrão estendem `IEnumerable<T>` e que é por esse motivo que você pode escrever `numbers.Where(...)`.

Para começar a usar LINQ, tudo o que você realmente precisa saber sobre os métodos de extensão é como colocá-los no escopo no seu aplicativo usando as diretivas `using` corretas. Do ponto de vista do aplicativo, um método de extensão e um método de instância normal são iguais.

Para obter mais informações sobre os métodos de extensão, consulte [Métodos de extensão](#). Para obter mais informações sobre os operadores de consulta padrão, consulte [Visão geral de operadores de consulta padrão \(C#\)](#). Alguns provedores LINQ, como LINQ to SQL e LINQ to XML, implementam seus próprios operadores de consulta padrão e métodos de extensão adicionais para outros tipos além de `IEnumerable<T>`.

## Expressões lambda

No exemplo anterior, observe que a expressão condicional (`num % 2 == 0`) é passada como um argumento embutido para o método `Where`: `Where(num => num % 2 == 0)`. Essa expressão embutida é chamada de uma expressão lambda. É uma maneira conveniente de escrever um código que de outra forma precisaria ser escrito de forma mais complicada como um método anônimo, um delegado genérico ou uma árvore de expressão. No C# `=>` é o operador lambda, que é lido como "vai para". O `num` à esquerda do operador é a variável de entrada que corresponde ao `num` na expressão de consulta. O compilador pode inferir o tipo de `num` porque ele sabe que `numbers` é um tipo `IEnumerable<T>` genérico. O corpo do lambda é exatamente igual à expressão na sintaxe de consulta ou em qualquer outra expressão ou instrução C#, ele pode incluir

chamadas de método e outra lógica complexa. O "valor retornado" é apenas o resultado da expressão.

Para começar a usar o LINQ, você não precisa usar lambdas extensivamente. No entanto, determinadas consultas só podem ser expressadas em sintaxe de método e algumas delas requerem expressões lambda. Após se familiarizar com lambdas, você verá que eles são uma ferramenta avançada e flexível na sua caixa de ferramentas do LINQ. Para obter mais informações, consulte [Expressões Lambda](#).

## Possibilidade de composição das consultas

No exemplo de código anterior, observe que o método `OrderBy` é invocado usando o operador ponto na chamada para `Where`. `Where` produz uma sequência filtrada e, em seguida, `Orderby` opera nessa sequência classificando-a. Como as consultas retornam uma `IEnumerable`, você pode escrevê-las na sintaxe de método encadeando as chamadas de método. Isso é o que o compilador faz nos bastidores quando você escreve consultas usando a sintaxe de consulta. E como uma variável de consulta não armazena os resultados da consulta, você pode modificá-la ou usá-la como base para uma nova consulta a qualquer momento, mesmo depois que ela foi executada.

Os exemplos a seguir demonstram algumas consultas LINQ simples usando cada abordagem listada anteriormente. Em geral, a regra é usar (1) sempre que possível e usar (2) e (3) sempre que necessário.

### ① Observação

Essas consultas funcionam em coleções na memória simples, no entanto, a sintaxe básica é idêntica àquela usada no LINQ to Entities e no LINQ to XML.

## Exemplo – sintaxe de consulta

A maneira recomendada de escrever a maioria das consultas é usar a *sintaxe de consulta* para criar *expressões de consulta*. O exemplo a seguir mostra três expressões de consulta. A primeira expressão de consulta demonstra como filtrar ou restringir os resultados aplicando condições com uma cláusula `where`. Ela retorna todos os elementos na sequência de origem cujos valores são maiores que 7 ou menores que 3. A segunda expressão demonstra como ordenar os resultados retornados. A terceira expressão demonstra como agrupar resultados de acordo com uma chave. Esta consulta retorna dois grupos com base na primeira letra da palavra.

C#

```
List<int> numbers = new() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

// The query variables can also be implicitly typed by using var

// Query #1.
IEnumerable<int> filteringQuery =
    from num in numbers
    where num < 3 || num > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num < 3 || num > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans",
    "barley" };
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

Observe que o tipo das consultas é `IEnumerable<T>`. Todas essas consultas poderiam ser escritas usando `var` conforme mostrado no exemplo a seguir:

```
var query = from num in numbers...
```

Em cada exemplo anterior, as consultas não são de fato executadas até você iterar na variável de consulta em uma instrução `foreach` ou outra instrução. Para obter mais informações, consulte [Introdução a Consultas LINQ](#).

## Exemplo – sintaxe de método

Algumas operações de consulta devem ser expressas como uma chamada de método. Os mais comuns desses métodos são aqueles que retornam valores numéricos singleton como `Sum`, `Max`, `Min`, `Average` e assim por diante. Esses métodos devem sempre ser chamados por último em qualquer consulta porque representam apenas um único valor e não podem atuar como a fonte para uma operação de consulta adicional. O exemplo a seguir mostra uma chamada de método em uma expressão de consulta:

C#

```
List<int> numbers1 = new() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };

// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

Se o método tiver os parâmetros Action ou Func, eles serão fornecidos na forma de uma expressão [lambda](#), como mostra o exemplo a seguir:

C#

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

Nas consultas anteriores, apenas a Query #4 é executada imediatamente. Isso ocorre porque ele retorna um único valor e não uma coleção [IEnumerable<T>](#) genérica. O próprio método tem que usar [foreach](#) para calcular seu valor.

Cada uma das consultas anteriores pode ser escrita usando a tipagem implícita com [var](#), como mostrado no exemplo a seguir:

C#

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

## Exemplo – sintaxe mista de consulta e do método

Este exemplo mostra como usar a sintaxe do método nos resultados de uma cláusula de consulta. Simplesmente coloque a expressão de consulta entre parênteses e, em seguida, aplique o operador de ponto e chame o método. No exemplo a seguir, a Query #7 retorna uma contagem dos números cujo valor está entre 3 e 7. Em geral, no entanto, é melhor usar uma segunda variável para armazenar o resultado da chamada do método. Dessa forma, é menos provável que a consulta seja confundida com os resultados da consulta.

C#

```
// Query #7.

// Using a query expression with method syntax
int numCount1 = (
    from num in numbers1
    where num < 3 || num > 7
    select num
).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

Como a Query #7 retorna um único valor e não uma coleção, a consulta é executada imediatamente.

A consulta anterior pode ser escrita usando a tipagem implícita com `var`, da seguinte maneira:

C#

```
var numCount = (from num in numbers...
```

Ela pode ser escrita na sintaxe de método da seguinte maneira:

C#

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

Ela pode ser escrita usando a tipagem explícita da seguinte maneira:

C#

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

## Confira também

- [Passo a passo: escrevendo consultas em C#](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Cláusula where](#)

# Consultar uma coleção de objetos

Artigo • 20/07/2023

O termo "LINQ to Objects" refere-se ao uso de consultas LINQ com qualquer coleção `IEnumerable` ou `IEnumerable<T>` diretamente, sem o uso de uma API ou provedor LINQ intermediário como o [LINQ to SQL](#) ou [LINQ to XML](#). Você pode usar o LINQ para consultar qualquer coleção enumerável como `List<T>`, `Array` ou `Dictionary< TKey, TValue >`. A coleção pode ser definida pelo usuário ou pode ser devolvida por uma API do .NET. Na abordagem da LINQ, você escreve o código declarativo que descreve o que você deseja recuperar.

Além disso, as consultas LINQ oferecem três principais vantagens sobre os loops `foreach` tradicionais:

- Elas são mais concisas e legíveis, especialmente quando você filtra várias condições.
- Elas fornecem poderosos recursos de filtragem, ordenação e agrupamento com um mínimo de código do aplicativo.
- Elas podem ser movidas para outras fontes de dados com pouca ou nenhuma modificação.

Em geral, quanto mais complexa a operação que você deseja executar sobre os dados, maior benefício você perceberá usando consultas LINQs em vez de técnicas tradicionais de iteração.

Este exemplo mostra como executar uma consulta simples em uma lista de objetos `Student`. Cada objeto `Student` contém algumas informações básicas sobre o aluno e uma lista que representa as pontuações do aluno em quatro provas.

## ⓘ Observação

Muitos outros exemplos nesta seção usam a mesma `Student` classe e coleção `students`.

C#

```
class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
```

```
public GradeLevel? Year { get; set; }
public List<int> ExamScores { get; set; }

public Student(string FirstName, string LastName, int ID, GradeLevel
Year, List<int> ExamScores)
{
    this.FirstName = FirstName;
    this.LastName = LastName;
    this.ID = ID;
    this.Year = Year;
    this.ExamScores = ExamScores;
}

public Student(string FirstName, string LastName, int StudentID,
List<int>? ExamScores = null)
{
    this.FirstName = FirstName;
    this.LastName = LastName;
    ID = StudentID;
    this.ExamScores = ExamScores ?? Enumerable.Empty<int>().ToList();
}

public static List<Student> students = new()
{
    new(
        FirstName: "Terry", LastName: "Adams", ID: 120,
        Year: GradeLevel.SecondYear,
        ExamScores: new() { 99, 82, 81, 79 }
    ),
    new(
        "Fadi", "Fakhouri", 116,
        GradeLevel.ThirdYear,
        new() { 99, 86, 90, 94 }
    ),
    new(
        "Hanying", "Feng", 117,
        GradeLevel.FirstYear,
        new() { 93, 92, 80, 87 }
    ),
    new(
        "Cesar", "Garcia", 114,
        GradeLevel.FourthYear,
        new() { 97, 89, 85, 82 }
    ),
    new(
        "Debra", "Garcia", 115,
        GradeLevel.ThirdYear,
        new() { 35, 72, 91, 70 }
    ),
    new(
        "Hugo", "Garcia", 118,
        GradeLevel.SecondYear,
        new() { 92, 90, 83, 78 }
    ),
    new(

```

```

        "Sven", "Mortensen", 113,
        GradeLevel.FirstYear,
        new() { 88, 94, 65, 91 }
    ),
    new(
        "Claire", "O'Donnell", 112,
        GradeLevel.FourthYear,
        new() { 75, 84, 91, 39 }
    ),
    new(
        "Svetlana", "Omelchenko", 111,
        GradeLevel.SecondYear,
        new() { 97, 92, 81, 60 }
    ),
    new(
        "Lance", "Tucker", 119,
        GradeLevel.ThirdYear,
        new() { 68, 79, 88, 92 }
    ),
    new(
        "Michael", "Tucker", 122,
        GradeLevel.FirstYear,
        new() { 94, 92, 91, 91 }
    ),
    new(
        "Eugene", "Zabokritski", 121,
        GradeLevel.FourthYear,
        new() { 96, 85, 91, 60 }
    )
);
}

enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

```

## Exemplo

A consulta a seguir retorna os alunos que receberam uma pontuação de 90 ou mais em sua primeira prova.

C#

```

void QueryHighScores(int exam, int score)
{
    var highScores =
        from student in students

```

```
    where student.ExamScores[exam] > score
    select new
    {
        Name = student.FirstName,
        Score = student.ExamScores[exam]
    };

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name},-{15}{item.Score}");
    }
}

QueryHighScores(1, 90);
```

Essa consulta é intencionalmente simples para que você possa testar. Por exemplo, você pode testar mais condições na cláusula `where` ou usar uma cláusula `orderby` para classificar os resultados.

## Classificação de operadores de consulta padrão por maneira de execução

As implementações de LINQ to Objects dos métodos de operador de consulta padrão em uma das duas maneiras principais: imediata ou adiada. Os operadores de consulta que usam a execução adiada podem ser divididos em mais duas categorias: streaming e não streaming. Se você souber como os operadores de consulta diferentes são executados, isso poderá ajudá-lo a entender os resultados que serão obtidos de uma determinada consulta. Isso é especialmente verdadeiro se a fonte de dados está sendo alterada ou se você estiver criando uma consulta sobre outra consulta. Este tópico classifica os operadores de consulta padrão de acordo com o modo de execução.

### Imediata

A execução imediata significa que a fonte de dados é lida e a operação é executada uma vez. Todos os operadores de consulta padrão que retornam um resultado escalar são executados imediatamente. Você pode forçar uma consulta a ser executada imediatamente usando o método [Enumerable.ToList](#) ou [Enumerable.ToArray](#). A execução imediata fornece reutilização dos resultados da consulta, não uma declaração de consulta. Os resultados são recuperados uma vez e armazenados para uso futuro.

### Adiado

A execução adiada significa que a operação não será realizada no ponto do código em que a consulta estiver declarada. A operação será realizada somente quando a variável de consulta for enumerada, por exemplo, usando uma instrução `foreach`. Isso significa que os resultados da execução da consulta dependerão do conteúdo da fonte de dados quando a consulta for executada em vez de quando a consulta for definida. Se a variável de consulta for enumerada várias vezes, os resultados poderão ser diferentes a cada vez. Quase todos os operadores de consulta padrão cujo tipo de retorno é `IEnumerable<T>` ou `IOrderedEnumerable<TElement>` executam de maneira adiada. A execução adiada oferece a facilidade da reutilização da consulta, uma vez que ela busca os dados atualizados da fonte de dados sempre que os seus resultados são iterados.

Os operadores de consulta que usam a execução adiada podem ser, adicionalmente, classificados como streaming ou não streaming.

## Streaming

Operadores streaming não precisam ler todos os dados de origem antes de gerar elementos. No momento da execução, um operador streaming realiza sua operação em cada elemento de origem enquanto eles são lidos, gerando o elemento, se apropriado. Um operador streaming continua a ler os elementos de origem até que um elemento de resultado possa ser produzido. Isso significa que mais de um elemento de origem poderá ser lido para produzir um elemento de resultado.

## Não streaming

Os operadores não streaming devem ler todos os dados de origem antes de produzirem um elemento de resultado. Operações como classificação ou agrupamento se enquadram nesta categoria. No momento da execução, os operadores de consulta não streaming leem todos os dados de origem, colocam-nos em uma estrutura de dados, realizam a operação e geram os elementos de resultado.

## Tabela de classificação

A tabela a seguir classifica cada método de operador de consulta padrão de acordo com o respectivo método de execução.

### ⓘ Observação

Se um operador estiver marcado em duas colunas, duas sequências de entrada estarão envolvidas na operação e cada sequência será avaliada de forma diferente.

Nesses casos, a primeira sequência na lista de parâmetros é a que sempre será avaliada de maneira adiada e em modo streaming.

<b>Operador de consulta padrão</b>	<b>Tipo de retorno</b>	<b>Execução imediata</b>	<b>Execução adiada de streaming</b>	<b>Execução adiada de não streaming</b>
Aggregate	TSource	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	Valor numérico único	X		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	TSource	X		
ElementAtOrDefault	TSource?		X	
Empty	IEnumerable<T>		X	
Except	IEnumerable<T>		X	X
First	TSource	X		
FirstOrDefault	TSource?		X	
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X
Join	IEnumerable<T>		X	X
Last	TSource	X		

<b>Operador de consulta padrão</b>	<b>Tipo de retorno</b>	<b>Execução imediata</b>	<b>Execução adiada de streaming</b>	<b>Execução adiada de não streaming</b>
LastOrDefault	<code>TSource?</code>	X		
LongCount	<code>Int64</code>	X		
Max	Valor numérico único, <code>TSource</code> OU <code>TResult?</code>	X		
Min	Valor numérico único, <code>TSource</code> OU <code>TResult?</code>	X		
OfType	<code>IEnumerable&lt;T&gt;</code>	X		
OrderBy	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
OrderByDescending	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
Range	<code>IEnumerable&lt;T&gt;</code>	X		
Repeat	<code>IEnumerable&lt;T&gt;</code>	X		
Reverse	<code>IEnumerable&lt;T&gt;</code>		X	
Select	<code>IEnumerable&lt;T&gt;</code>	X		
SelectMany	<code>IEnumerable&lt;T&gt;</code>	X		
SequenceEqual	<code>Boolean</code>	X		
Single	<code>TSource</code>	X		
SingleOrDefault	<code>TSource?</code>	X		
Skip	<code>IEnumerable&lt;T&gt;</code>	X		
SkipWhile	<code>IEnumerable&lt;T&gt;</code>	X		
Sum	Valor numérico único	X		
Take	<code>IEnumerable&lt;T&gt;</code>	X		
TakeWhile	<code>IEnumerable&lt;T&gt;</code>	X		
ThenBy	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
ThenByDescending	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
ToArray	Matriz <code>TSource[]</code>	X		

<b>Operador de consulta padrão</b>	<b>Tipo de retorno</b>	<b>Execução imediata</b>	<b>Execução adiada de streaming</b>	<b>Execução adiada de não streaming</b>
ToDictionary	Dictionary<TKey,TValue>	X		
ToList	IList<T>	X		
ToLookup	ILookup<TKey,TElement>	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

## Confira também

- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Interpolação de cadeia de caracteres](#)

# Como retornar uma consulta de um método

Artigo • 07/04/2023

Este exemplo mostra como retornar uma consulta de um método como o valor retornado e como um parâmetro `out`.

Os objetos de consulta são combináveis, o que significa que você pode retornar uma consulta de um método. Os objetos que representam consultas não armazenam a coleção resultante, mas as etapas para gerar os resultados quando necessário. A vantagem de retornar objetos de consulta de métodos é que eles podem ser ainda mais modificados e combinados. Portanto, qualquer valor retornado ou parâmetro `out` de um método que retorna uma consulta também deve ter o tipo. Se um método materializa uma consulta em um tipo `List<T>` ou `Array` concreto, considera-se que ele está retornando os resultados da consulta em vez da consulta em si. Uma variável de consulta retornada de um método ainda pode ser combinada ou modificada.

## Exemplo

No exemplo a seguir, o primeiro método retorna uma consulta como um valor retornado e o segundo método retorna uma consulta como um parâmetro `out`. Observe que em ambos os casos é uma consulta que é retornada, não os resultados da consulta.

C#

```
// QueryMethod1 returns a query as its value.
IEnumerable<string> QueryMethod1(int[] ints) =>
    from i in ints
    where i > 4
    select i.ToString();

// QueryMethod2 returns a query as the value of the out parameter returnQ
void QueryMethod2(int[] ints, out IEnumerable<string> returnQ) =>
    returnQ =
        from i in ints
        where i < 4
        select i.ToString();

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

// QueryMethod1 returns a query as the value of the method.
var myQuery1 = QueryMethod1(nums);
```

```

// Query myQuery1 is executed in the following foreach loop.
Console.WriteLine("Results of executing myQuery1:");
// Rest the mouse pointer over myQuery1 to see its type.
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}

// You also can execute the query returned from QueryMethod1
// directly, without using myQuery1.
Console.WriteLine("\nResults of executing myQuery1 directly:");
// Rest the mouse pointer over the call to QueryMethod1 to see its
// return type.
foreach (var s in QueryMethod1(nums))
{
    Console.WriteLine(s);
}

// QueryMethod2 returns a query as the value of its out parameter.
QueryMethod2(nums, out IEnumerable<string> myQuery2);

// Execute the returned query.
Console.WriteLine("\nResults of executing myQuery2:");
foreach (var s in myQuery2)
{
    Console.WriteLine(s);
}

// You can modify a query by using query composition. In this case, the
// previous query object is used to create a new query object; this new
// object
// will return different results than the original query object.
myQuery1 =
    from item in myQuery1
    orderby item descending
    select item;

// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}

```

## Confira também

- LINQ (Consulta Integrada à Linguagem)

# Armazenar os resultados de uma consulta na memória

Artigo • 07/04/2023

Uma consulta é basicamente um conjunto de instruções sobre como recuperar e organizar os dados. As consultas são executadas lentamente, conforme cada item subsequente no resultado é solicitado. Quando você usa `foreach` para iterar os resultados, os itens são retornados conforme acessado. Para avaliar uma consulta e armazenar os resultados sem executar um loop `foreach`, basta chamar um dos métodos a seguir na variável de consulta:

- [ToListAsync](#)
- [ToArray](#)
- [ToDictionary](#)
- [ToLookup](#)

Recomendamos que ao armazenar os resultados da consulta, você atribua o objeto da coleção retornado a uma nova variável conforme mostrado no exemplo a seguir:

## Exemplo

C#

```
List<int> numbers = new() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

IEnumerable<int> queryFactorsOfFour =
    from num in numbers
    where num % 4 == 0
    select num;

// Store the results in a new variable
// without executing a foreach loop.
List<int> factorsofFourList = queryFactorsOfFour.ToList();

// Read and write from the newly created list to demonstrate that it holds
// data.
Console.WriteLine(factorsofFourList[2]);
factorsofFourList[2] = 0;
Console.WriteLine(factorsofFourList[2]);
```

## Confira também

- LINQ (Consulta Integrada à Linguagem)

# Agrupar resultados de consultas

Artigo • 07/04/2023

O agrupamento é um dos recursos mais poderosos do LINQ. Os exemplos a seguir mostram como agrupar dados de várias maneiras:

- Por uma única propriedade.
- Pela primeira letra de uma propriedade de cadeia de caracteres.
- Por um intervalo numérico calculado.
- Por predicado booleano ou outra expressão.
- Por uma chave composta.

Além disso, as duas últimas consultas projetam seus resultados em um novo tipo anônimo que contém somente o primeiro nome e sobrenome do aluno. Para obter mais informações, consulte a [cláusula group](#).

## ⓘ Observação

O exemplo neste tópico usa a classe `Student` e a lista `students` do código de exemplo em [Consulta de uma coleção de objetos](#).

## Exemplo de agrupamento por propriedade única

O exemplo a seguir mostra como agrupar elementos de origem usando uma única propriedade do elemento como a chave de grupo. Nesse caso, a chave é uma `string`, o sobrenome do aluno. Também é possível usar uma substring para a chave; confira [o próximo exemplo](#). A operação de agrupamento usa o comparador de igualdade padrão para o tipo.

C#

```
// Variable groupByLastNamesQuery is an IEnumerable<IGrouping<string,
// DataClass.Student>>.
var groupByLastNamesQuery =
    from student in students
    group student by student.LastName into newGroup
    orderby newGroup.Key
```

```

    select newGroup;

    foreach (var nameGroup in groupByLastNamesQuery)
    {
        Console.WriteLine($"Key: {nameGroup.Key}");
        foreach (var student in nameGroup)
        {
            Console.WriteLine($"{student.LastName}, {student.FirstName}");
        }
    }

/* Output:
   Key: Adams
      Adams, Terry
   Key: Fakhouri
      Fakhouri, Fadi
   Key: Feng
      Feng, Hanying
   Key: Garcia
      Garcia, Cesar
      Garcia, Debra
      Garcia, Hugo
   Key: Mortensen
      Mortensen, Sven
   Key: O'Donnell
      O'Donnell, Claire
   Key: Omelchenko
      Omelchenko, Svetlana
   Key: Tucker
      Tucker, Lance
      Tucker, Michael
   Key: Zabokritski
      Zabokritski, Eugene
*/

```

## Exemplo de agrupamento por valor

O exemplo a seguir mostra como agrupar elementos de origem usando algo diferente de uma propriedade do objeto para a chave de grupo. Neste exemplo, a chave é a primeira letra do sobrenome do aluno.

C#

```

var groupByFirstLetterQuery =
    from student in students
    group student by student.LastName[0];

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    // Nested foreach is required to access group items.

```

```

        foreach (var student in studentGroup)
    {
        Console.WriteLine($"\\t{student.LastName}, {student.FirstName}");
    }
}

/* Output:
Key: A
    Adams, Terry
Key: F
    Fakhouri, Fadi
    Feng, Hanying
Key: G
    Garcia, Cesar
    Garcia, Debra
    Garcia, Hugo
Key: M
    Mortensen, Sven
Key: O
    O'Donnell, Claire
    Omelchenko, Svetlana
Key: T
    Tucker, Lance
    Tucker, Michael
Key: Z
    Zabokritski, Eugene
*/

```

## Exemplo de agrupamento por intervalo

O exemplo a seguir mostra como agrupar elementos de origem usando um intervalo numérico como a chave de grupo. Em seguida, a consulta, projeta os resultados em um tipo anônimo que contém apenas o nome e o sobrenome e o intervalo de percentil ao qual o aluno pertence. Um tipo anônimo é usado porque não é necessário usar o objeto `Student` completo para exibir os resultados. `GetPercentile` é uma função auxiliar que calcula um percentil com base na pontuação média do aluno. O método retorna um número inteiro entre 0 e 10.

C#

```

int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

var groupByPercentileQuery =
    from student in students
    let percentile = GetPercentile(student)

```

```

group new
{
    student.FirstName,
    student.LastName
} by percentile into percentGroup
orderby percentGroup.Key
select percentGroup;

// Nested foreach required to iterate over groups and group items.
foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup)
    {
        Console.WriteLine($"{item.LastName}, {item.FirstName}");
    }
}

/* Output:
Key: 60
    Garcia, Debra
Key: 70
    O'Donnell, Claire
Key: 80
    Adams, Terry
    Feng, Hanying
    Garcia, Cesar
    Garcia, Hugo
    Mortensen, Sven
    Omelchenko, Svetlana
    Tucker, Lance
    Zabokritski, Eugene
Key: 90
    Fakhouri, Fadi
    Tucker, Michael
*/

```

## Exemplo de agrupamento por comparação

O exemplo a seguir mostra como agrupar elementos de origem usando uma expressão de comparação booleana. Neste exemplo, a expressão booleana testa se a pontuação média de provas do aluno é maior que 75. Como nos exemplos anteriores, os resultados são projetados em um tipo anônimo porque o elemento de origem completo não é necessário. Observe que as propriedades no tipo anônimo se tornam propriedades no membro `Key` e podem ser acessadas pelo nome quando a consulta é executada.

C#

```

var groupByHighAverageQuery =
    from student in students

```

```

group new
{
    student.FirstName,
    student.LastName
} by student.ExamScores.Average() > 75 into studentGroup
select studentGroup;

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"{student.FirstName} {student.LastName}");
    }
}

/* Output:
Key: True
    Terry Adams
    Fadi Fakhouri
    Hanying Feng
    Cesar Garcia
    Hugo Garcia
    Sven Mortensen
    Svetlana Omelchenko
    Lance Tucker
    Michael Tucker
    Eugene Zabokritski
Key: False
    Debra Garcia
    Claire O'Donnell
*/

```

## Exemplo de agrupamento por tipo anônimo

O exemplo a seguir mostra como usar um tipo anônimo para encapsular uma chave que contém vários valores. Neste exemplo, o primeiro valor da chave é a primeira letra do sobrenome do aluno. O segundo valor da chave é um booleano que especifica se o aluno tirou mais que 85 na primeira prova. Você pode ordenar os grupos por qualquer propriedade na chave.

C#

```

var groupByCompoundKey =
    from student in students
    group student by new
    {
        FirstLetter = student.LastName[0],
        IsScoreOver85 = student.ExamScores[0] > 85
    } into studentGroup

```

```

    orderby studentGroup.Key.FirstLetter
    select studentGroup;

foreach (var scoreGroup in groupByCompoundKey)
{
    string s = scoreGroup.Key.IsScoreOver85 == true ? "more than 85" : "less
than 85";
    Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetter} who
scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"{item.FirstName} {item.LastName}");
    }
}

/* Output:
   Name starts with A who scored more than 85
      Terry Adams
   Name starts with F who scored more than 85
      Fadi Fakhouri
      Hanying Feng
   Name starts with G who scored more than 85
      Cesar Garcia
      Hugo Garcia
   Name starts with G who scored less than 85
      Debra Garcia
   Name starts with M who scored more than 85
      Sven Mortensen
   Name starts with O who scored less than 85
      Claire O'Donnell
   Name starts with O who scored more than 85
      Svetlana Omelchenko
   Name starts with T who scored less than 85
      Lance Tucker
   Name starts with T who scored more than 85
      Michael Tucker
   Name starts with Z who scored more than 85
      Eugene Zabokritski
*/

```

## Confira também

- GroupBy
- IGrouping< TKey, TElement >
- LINQ (Consulta Integrada à Linguagem)
- Cláusula group
- Tipos anônimos
- Executar uma subconsulta em uma operação de agrupamento
- Criar um grupo aninhado
- Agrupar dados



# Criar um grupo aninhado

Artigo • 07/04/2023

O exemplo a seguir mostra como criar grupos aninhados em uma expressão de consulta LINQ. Cada grupo que é criado de acordo com o ano do aluno ou nível de ensino, é subdividido em grupos com base nos nomes das pessoas.

## Exemplo

### ① Observação

O exemplo neste tópico usa a classe `students` e a lista `Student` do código de exemplo em [Consulta de uma coleção de objetos](#).

C#

```
var nestedGroupsQuery =
    from student in students
    group student by student.Year into newGroup1
    from newGroup2 in (
        from student in newGroup1
        group student by student.LastName
    )
    group newGroup2 by newGroup1.Key;

// Three nested foreach loops are required to iterate
// over all elements of a grouped group. Hover the mouse
// cursor over the iteration variables to see their actual type.
foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup)
    {
        Console.WriteLine($"    Names that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {
            Console.WriteLine($"        \t{innerGroupElement.LastName}
{innerGroupElement.FirstName}");
        }
    }
}

/* Output:
   DataClass.Student Level = SecondYear
       Names that begin with: Adams
           Adams Terry
       Names that begin with: Garcia
```

```
        Garcia Hugo
    Names that begin with: Omelchenko
        Omelchenko Svetlana
DataClass.Student Level = ThirdYear
    Names that begin with: Fakhouri
        Fakhouri Fadi
    Names that begin with: Garcia
        Garcia Debra
    Names that begin with: Tucker
        Tucker Lance
DataClass.Student Level = FirstYear
    Names that begin with: Feng
        Feng Hanying
    Names that begin with: Mortensen
        Mortensen Sven
    Names that begin with: Tucker
        Tucker Michael
DataClass.Student Level = FourthYear
    Names that begin with: Garcia
        Garcia Cesar
    Names that begin with: O'Donnell
        O'Donnell Claire
    Names that begin with: Zabokritski
        Zabokritski Eugene
*/
```

Observe que três loops `foreach` aninhados são necessários para iterar sobre os elementos internos de um grupo aninhado.

## Confira também

- LINQ (Consulta Integrada à Linguagem)

# Executar uma subconsulta em uma operação de agrupamento

Artigo • 07/04/2023

Este artigo mostra duas maneiras diferentes de criar uma consulta que ordena os dados de origem em grupos e, em seguida, realiza uma subconsulta em cada grupo individualmente. A técnica básica em cada exemplo é agrupar os elementos de origem usando uma *continuação* chamada `newGroup` e, em seguida, gerar uma nova subconsulta de `newGroup`. Essa subconsulta é executada em cada novo grupo criado pela consulta externa. Observe que, nesse exemplo específico, a saída final não é um grupo, mas uma sequência simples de tipos anônimos.

Para obter mais informações sobre como agrupar, consulte [Cláusula group](#).

Para obter mais informações sobre continuações, consulte [into](#). O exemplo a seguir usa uma estrutura de dados na memória como a fonte de dados, mas os mesmos princípios se aplicam a qualquer tipo de fonte de dados do LINQ.

## Exemplo

### ① Observação

O exemplo neste tópico usa a classe `Student` e a lista `students` do código de exemplo em [Consulta de uma coleção de objetos](#).

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.ExamScores.Average()
        ).Max()
    };
int count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
```

```
{  
    Console.WriteLine($" {item.Level} Highest Score={item.HighestScore}");  
}
```

A consulta no snippet acima também pode ser escrita usando a sintaxe de método. O snippet de código a seguir tem uma consulta semanticamente equivalente escrita usando a sintaxe de método.

C#

```
var queryGroupMax =  
    students  
        .GroupBy(student => student.Year)  
        .Select(studentGroup => new  
        {  
            Level = studentGroup.Key,  
            HighestScore = studentGroup.Select(student2 =>  
                student2.ExamScores.Average()).Max()  
        });  
  
int count = queryGroupMax.Count();  
Console.WriteLine($"Number of groups = {count}");  
  
foreach (var item in queryGroupMax)  
{  
    Console.WriteLine($" {item.Level} Highest Score={item.HighestScore}");  
}
```

## Confira também

- [LINQ \(Consulta Integrada à Linguagem\)](#)

# Agrupar resultados por chaves contíguas

Artigo • 07/04/2023

O exemplo a seguir mostra como agrupar elementos em partes que representam subsequências de chaves contíguas. Por exemplo, suponha que você receba a seguinte sequência de pares chave-valor:

Chave	Valor
Um	We
Um	think
Um	that
B	Linq
C	is
Um	really
B	cool
B	!

Os seguintes grupos serão criados nesta ordem:

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

A solução é implementada como um método de extensão que é thread-safe e que retorna os resultados de uma maneira de streaming. Em outras palavras, ela produz seus grupos à medida que percorre a sequência de origem. Diferentemente dos operadores `group` ou `orderby`, ela pode começar a retornar grupos para o chamador antes que todas as sequências sejam lidas.

O acesso thread-safe é alcançado ao fazer uma cópia de cada grupo ou parte enquanto a sequência de origem é iterada, conforme explicado nos comentários do código-fonte. Se a sequência de origem tiver uma sequência grande de itens contíguos, o Common Language Runtime poderá lançar uma [OutOfMemoryException](#).

# Exemplo

O exemplo a seguir mostra o método de extensão e o código do cliente que o usa:

C#

```
public static class ChunkExtensions
{
    public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource,
    TKey>(
        this IEnumerable<TSource> source,
        Func<TSource, TKey> keySelector
    ) =>
        source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);

    public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource,
    TKey>(
        this IEnumerable<TSource> source,
        Func<TSource, TKey> keySelector,
        IEqualityComparer<TKey> comparer
    )
    {
        // Flag to signal end of source sequence.
        const bool noMoreSourceElements = true;

        // Auto-generated iterator for the source array.
        var enumerator = source.GetEnumerator();

        // Move to the first element in the source sequence.
        if (!enumerator.MoveNext())
        {
            yield break;
        }

        // Iterate through source sequence and create a copy of each Chunk.
        // On each pass, the iterator advances to the first element of the
        next "Chunk"
        // in the source sequence. This loop corresponds to the outer
        foreach loop that
        // executes the query.
        Chunk<TKey, TSource> current = null;
        while (true)
        {
            // Get the key for the current Chunk. The source iterator will
            churn through
            // the source sequence until it finds an element with a key that
            doesn't match.
            var key = keySelector(enumerator.Current);

            // Make a new Chunk (group) object that initially has one
            GroupItem, which is a copy of the current source element.
            current = new Chunk<TKey, TSource>(key, enumerator, value =>
            comparer.Equals(key, keySelector(value)));
        }
    }
}
```

```

        // Return the Chunk. A Chunk is an IGrouping< TKey, TSource >,
        which is the return value of the ChunkBy method.
        // At this point the Chunk only has the first element in its
        source sequence. The remaining elements will be
        // returned only when the client code foreach's over this chunk.
        See Chunk.GetEnumerator for more info.
        yield return current;

        // Check to see whether (a) the chunk has made a copy of all its
        source elements or
        // (b) the iterator has reached the end of the source sequence.
        If the caller uses an inner
            // foreach loop to iterate the chunk items, and that loop ran to
            completion,
            // then the Chunk.GetEnumerator method will already have made
            // copies of all chunk items before we get here. If the
            Chunk.GetEnumerator loop did not
            // enumerate all elements in the chunk, we need to do it here to
            avoid corrupting the iterator
            // for clients that may be calling us on a separate thread.
            if (current.CopyAllChunkElements() == noMoreSourceElements)
            {
                yield break;
            }
        }

    }

// A Chunk is a contiguous group of one or more source elements that have
// the same key. A Chunk
// has a key and a list of ChunkItem objects, which are copies of the
elements in the source sequence.
class Chunk< TKey, TSource > : IGrouping< TKey, TSource >
{
    // INVARIANT: DoneCopyingChunk == true ||
    // (predicate != null && predicate(enumerator.Current) &&
current.Value == enumerator.Current)

    // A Chunk has a linked list of ChunkItems, which represent the elements
in the current chunk. Each ChunkItem
    // has a reference to the next ChunkItem in the list.
    class ChunkItem
    {
        public ChunkItem(TSource value)
        {
            Value = value;
        }
        public readonly TSource Value;
        public ChunkItem? Next = null;
    }

    // Stores a reference to the enumerator for the source sequence
    private IEnumerator< TSource > enumerator;
}

```

```

// A reference to the predicate that is used to compare keys.
private Func<TSource, bool> predicate;

// Stores the contents of the first source element that
// belongs with this chunk.
private readonly ChunkItem head;

// End of the list. It is repositioned each time a new
// ChunkItem is added.
private ChunkItem tail;

// Flag to indicate the source iterator has reached the end of the
source sequence.
internal bool isLastSourceElement = false;

// Private object for thread synchronization
private readonly object m_Lock;

// REQUIRES: enumerator != null && predicate != null
public Chunk(TKey key, IEnumerator<TSource> enumerator, Func<TSource,
bool> predicate)
{
    Key = key;
    this.enumerator = enumerator;
    this.predicate = predicate;

    // A Chunk always contains at least one element.
    head = new ChunkItem(enumerator.Current);

    // The end and beginning are the same until the list contains > 1
elements.
    tail = head;

    m_Lock = new object();
}

// Indicates that all chunk elements have been copied to the list of
ChunkItems,
// and the source enumerator is either at the end, or else on an element
with a new key.
// the tail of the linked list is set to null in the
CopyNextChunkElement method if the
// key of the next element does not match the current chunk's key, or
there are no more elements in the source.
private bool DoneCopyingChunk => tail == null;

// Adds one ChunkItem to the current group
// REQUIRES: !DoneCopyingChunk && lock(this)
private void CopyNextChunkElement()
{
    // Try to advance the iterator on the source sequence.
    // If MoveNext returns false we are at the end, and
isLastSourceElement is set to true
    isLastSourceElement = !enumerator.MoveNext();
}

```

```

        // If we are (a) at the end of the source, or (b) at the end of the
        current chunk
        // then null out the enumerator and predicate for reuse with the
        next chunk.
        if (isLastSourceElement || !predicate(enumerator.Current))
        {
            enumerator = null;
            predicate = null;
        }
        else
        {
            tail.Next = new ChunkItem(enumerator.Current);
        }

        // tail will be null if we are at the end of the chunk elements
        // This check is made in DoneCopyingChunk.
        tail = tail.Next!;
    }

    // Called after the end of the last chunk was reached. It first checks
    whether
    // there are more elements in the source sequence. If there are, it
    // Returns true if enumerator for this chunk was exhausted.
    internal bool CopyAllChunkElements()
    {
        while (true)
        {
            lock (m_Lock)
            {
                if (DoneCopyingChunk)
                {
                    // If isLastSourceElement is false,
                    // it signals to the outer iterator
                    // to continue iterating.
                    return isLastSourceElement;
                }
                else
                {
                    CopyNextChunkElement();
                }
            }
        }
    }

    public TKey Key { get; }

    // Invoked by the inner foreach loop. This method stays just one step
    ahead
    // of the client requests. It adds the next element of the chunk only
    after
    // the clients requests the last element in the list so far.
    public IEnumarator<TSource> GetEnumerator()
    {
        //Specify the initial element to enumerate.

```

```

        ChunkItem current = head;

        // There should always be at least one ChunkItem in a Chunk.
        while (current != null)
        {
            // Yield the current item in the list.
            yield return current.Value;

            // Copy the next item from the source sequence,
            // if we are at the end of our local list.
            lock (m_Lock)
            {
                if (current == tail)
                {
                    CopyNextChunkElement();
                }
            }

            // Move to the next ChunkItem in the list.
            current = current.Next;
        }
    }

    System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() => GetEnumerator();
}

public static class GroupByContiguousKeys
{
    // The source sequence.
    static readonly KeyValuePair<string, string>[] list = {
        new("A", "We"),
        new("A", "think"),
        new("A", "that"),
        new("B", "LINQ"),
        new("C", "is"),
        new("A", "really"),
        new("B", "cool"),
        new("B", "!")
    };

    // Query variable declared as class member to be available
    // on different threads.
    static readonly IEnumerable<IGrouping<string, KeyValuePair<string,
    string>>> query =
        list.ChunkBy(p => p.Key);

    public static void GroupByContiguousKeys1()
    {
        // ChunkBy returns IGrouping objects, therefore a nested
        // foreach loop is required to access the elements in each "chunk".
        foreach (var item in query)
        {
            Console.WriteLine($"Group key = {item.Key}");
            foreach (var inner in item)

```

```
        {
            Console.WriteLine($"\\t{inner.Value}");
        }
    }
}
```

## Confira também

- [LINQ \(Consulta Integrada à Linguagem\)](#)

# Especificar filtros de predicado dinamicamente em tempo de execução

Artigo • 09/03/2023

Em alguns casos, você não sabe até o tempo de execução quantos predicados precisa aplicar aos elementos de origem na cláusula `where`. Uma maneira de especificar dinamicamente vários filtros de predicados é usar o método `Contains`, conforme mostrado no exemplo a seguir. A consulta retornará resultados diferentes com base no valor de `id` quando a consulta for executada.

C#

```
int[] ids = { 111, 114, 112 };

var queryNames =
    from student in students
    where ids.Contains(student.ID)
    select new
    {
        student.LastName,
        student.ID
    };

foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
   Garcia: 114
   O'Donnell: 112
   Omelchenko: 111
 */

// Change the ids.
ids = new[] { 122, 117, 120, 115 };

// The query will now return different results
foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
   Adams: 120
   Feng: 117
   Garcia: 115
   Tucker: 122
 */
```

# Usando consultas diferentes no runtime

Você pode usar instruções de fluxo de controle, como `if... else` ou `switch`, para selecionar entre consultas alternativas predeterminadas. No exemplo a seguir, `studentQuery` usará outra cláusula `where` se o valor do runtime de `oddYear` é `true` ou `false`.

C#

```
void FilterByYearType(bool oddYear)
{
    IEnumerable<Student> studentQuery;
    if (oddYear)
    {
        studentQuery =
            from student in students
            where student.Year == GradeLevel.FirstYear || student.Year ==
GradeLevel.ThirdYear
            select student;
    }
    else
    {
        studentQuery =
            from student in students
            where student.Year == GradeLevel.SecondYear || student.Year ==
GradeLevel.FourthYear
            select student;
    }

    string descr = oddYear ? "odd" : "even";
    Console.WriteLine($"The following students are at an {descr} year
level:");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

FilterByYearType(true);

/* Output:
   The following students are at an odd year level:
   Fakhouri: 116
   Feng: 117
   Garcia: 115
   Mortensen: 113
   Tucker: 119
   Tucker: 122
*/
```

```
FilterByYearType(false);

/* Output:
The following students are at an even year level:
Adams: 120
Garcia: 114
Garcia: 118
O'Donnell: 112
Omelchenko: 111
Zabokritski: 121
*/
```

## Confira também

- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Cláusula where](#)
- [Como realizar consultas com base no estado do runtime](#)

# Executar junções internas

Artigo • 31/07/2023

Em termos de banco de dados relacionais, uma *junção interna* produz um conjunto de resultados no qual cada elemento da primeira coleção aparece uma vez para todo elemento correspondente na segunda coleção. Se um elemento na primeira coleção não tiver nenhum elemento correspondente, ele não aparece no conjunto de resultados. O método `Join`, que é chamado pela cláusula `join` no C#, implementa uma junção interna.

Este artigo mostra como executar quatro variações de uma junção interna:

- Uma junção interna simples que correlaciona os elementos de duas fontes de dados com base em uma chave simples.
- Uma junção interna simples que correlaciona os elementos de duas fontes de dados com base em uma chave *composta*. Uma chave composta, que é uma chave que consiste em mais de um valor, permite que você correlacione os elementos com base em mais de uma propriedade.
- Uma *junção múltipla* na qual operações `join` sucessivas são acrescentadas umas às outras.
- Uma junção interna que é implementada por meio de uma junção de grupo.

## ① Observação

Os exemplos deste tópico usam as seguintes classes de dados:

C#

```
record Person(string FirstName, string LastName);
record Pet(string Name, Person Owner);
record Employee(string FirstName, string LastName, int EmployeeID);
record Cat(string Name, Person Owner) : Pet(Name, Owner);
record Dog(string Name, Person Owner) : Pet(Name, Owner);
```

bem como a classe `Student` de Consulta de uma coleção de objetos.

## Exemplo – junção de chave simples

O exemplo a seguir cria duas coleções que contêm objetos de dois tipos definidos pelo usuário, `Person` e `Pet`. A consulta usa a cláusula `join` em C# para corresponder objetos

`Person` com objetos `Pet` cujo `Owner` é `Person`. A cláusula `select` em C# define a aparência dos objetos resultantes. Neste exemplo, os objetos resultantes são tipos anônimos que consistem no nome do proprietário e no nome do animal de estimação.

C#

```
Person magnus = new(FirstName: "Magnus", LastName: "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");
Person rui = new("Rui", "Raposo");

List<Person> people = new() { magnus, terry, charlotte, arlene, rui };

List<Pet> pets = new()
{
    new(Name: "Barley", Owner: terry),
    new("Boots", terry),
    new("Whiskers", charlotte),
    new("Blue Moon", rui),
    new("Daisy", magnus),
};

// Create a collection of person-pet pairs. Each element in the collection
// is an anonymous type containing both the person's name and their pet's
// name.
var query =
    from person in people
    join pet in pets on person equals pet.Owner
    select new
    {
        OwnerName = person.FirstName,
        PetName = pet.Name
    };

string result = "";
foreach (var ownerAndPet in query)
{
    result += $"{ownerAndPet.PetName}\\" is owned by
{ownerAndPet.OwnerName}\r\n";
}
Console.WriteLine(result);
return result;

/* Output:
   "Daisy" is owned by Magnus
   "Barley" is owned by Terry
   "Boots" is owned by Terry
   "Whiskers" is owned by Charlotte
   "Blue Moon" is owned by Rui
*/
```

Você obtém os mesmos resultados usando a sintaxe do método [Join](#):

C#

```
var query =
    people.Join(pets,
        person => person,
        pet => pet.Owner,
        (person, pet) =>
            new { OwnerName = person.FirstName, PetName = pet.Name
});
```

Observe que o objeto `Person` cujo `LastName` é "Huff" não aparecerá no conjunto de resultados porque não há nenhum objeto `Pet` que tenha `Pet.Owner` igual a esse `Person`.

## Exemplo – junção de chave composta

Em vez de correlacionar os elementos com base em apenas uma propriedade, você pode usar uma chave composta para comparar elementos com base em várias propriedades. Para fazer isso, especifique a função de seletor de chave para cada coleção retornar um tipo anônimo que consiste nas propriedades que você deseja comparar. Se você rotular as propriedades, elas devem ter o mesmo rótulo no tipo anônimo cada chave. As propriedades também devem aparecer na mesma ordem.

O exemplo a seguir usa uma lista de objetos `Employee` e uma lista de objetos `Student` para determinar quais funcionários também são alunos. Ambos os tipos têm uma propriedade `FirstName` e uma `LastName` do tipo `String`. As funções que criam as chaves de junção dos elementos de cada lista retornam um tipo anônimo que consiste nas propriedades `FirstName` e `LastName` de cada elemento. A operação `join` compara essas chaves compostas quanto à igualdade e retorna pares de objetos de cada lista em que o nome e o sobrenome correspondem.

C#

```
List<Employee> employees = new()
{
    new(FirstName: "Terry", LastName: "Adams", EmployeeID: 522459),
    new("Charlotte", "Weiss", 204467),
    new("Magnus", "Hedland", 866200),
    new("Vernette", "Price", 437139)
};

List<Student> students = new()
{
    new(FirstName: "Vernette", LastName: "Price", StudentID: 9562),
    new("Terry", "Earls", 9870),
```

```

        new("Terry", "Adams", 9913)
    };

    // Join the two data sources based on a composite key consisting of first
    // and last name,
    // to determine which employees are also students.
    var query =
        from employee in employees
        join student in students on new
        {
            employee.FirstName,
            employee.LastName
        } equals new
        {
            student.FirstName,
            student.LastName
        }
        select employee.FirstName + " " + employee.LastName;

    string result = "";
    result += "The following people are both employees and students:\r\n";
    foreach (string name in query)
    {
        result += $"{name}\r\n";
    }
    Console.WriteLine(result);
    return result;

    /* Output:
       The following people are both employees and students:
       Terry Adams
       Vernette Price
    */
}

```

Você pode usar o método [Join](#), conforme mostrado no exemplo a seguir:

C#

```

var query = employees.Join(
    students,
    employee => new { FirstName = employee.FirstName, LastName =
employee.LastName },
    student => new { FirstName = student.FirstName, student.LastName },
    (employee, student) => $"{employee.FirstName} {employee.LastName}"
);

```

## Exemplo – junção múltipla

Qualquer número de operações join pode ser acrescentado entre si para realizar uma junção múltipla. Cada cláusula `join` em C# correlaciona uma fonte de dados

especificada com os resultados da junção anterior.

O exemplo a seguir cria três coleções: uma lista de objetos `Person`, uma lista de objetos `Cat` e uma lista de objetos `Dog`.

A primeira cláusula `join` em C# corresponde a pessoas e gatos com base em um objeto `Person` correspondendo a `Cat.Owner`. Ela retorna uma sequência de tipos anônimos que contêm o objeto `Person` e `Cat.Name`.

A segunda cláusula `join` em C# correlaciona os tipos anônimos retornados pela primeira junção com objetos `Dog` na lista de cães fornecida, com base em uma chave composta que consiste na propriedade `Owner` do tipo `Person` e na primeira letra do nome do animal. Ela retorna uma sequência de tipos anônimos que contêm as propriedades `Cat.Name` e `Dog.Name` de cada par correspondente. Como esta é uma junção interna, apenas os objetos da primeira fonte de dados que têm uma correspondência na segunda fonte de dados são retornados.

C#

```
Person magnus = new(FirstName: "Magnus", LastName: "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");
Person rui = new("Rui", "Raposo");
Person phyllis = new("Phyllis", "Harris");

List<Person> people = new() { magnus, terry, charlotte, arlene, rui, phyllis };

List<Cat> cats = new()
{
    new(Name: "Barley", Owner: terry),
    new("Boots", terry),
    new("Whiskers", charlotte),
    new("Blue Moon", rui),
    new("Daisy", magnus),
};

List<Dog> dogs = new()
{
    new(Name: "Four Wheel Drive", Owner: phyllis),
    new("Duke", magnus),
    new("Denim", terry),
    new("Wiley", charlotte),
    new("Snoopy", rui),
    new("Snickers", arlene),
};

// The first join matches Person and Cat.Owner from the list of people and
```

```

// cats, based on a common Person. The second join matches dogs whose names
start
// with the same letter as the cats that have the same owner.
var query =
    from person in people
    join cat in cats on person equals cat.Owner
    join dog in dogs on new
    {
        Owner = person,
        Letter = cat.Name.Substring(0, 1)
    } equals new
    {
        dog.Owner,
        Letter = dog.Name.Substring(0, 1)
    }
    select new
    {
        CatName = cat.Name,
        DogName = dog.Name
    };

string result = "";
foreach (var obj in query)
{
    result += $"The cat \"{obj.CatName}\" shares a house, and the first
letter of their name, with \"{obj.DogName}\"\r\n";
}
Console.WriteLine(result);
return result;

/* Output:
   The cat "Daisy" shares a house, and the first letter of their name,
with "Duke".
   The cat "Whiskers" shares a house, and the first letter of their name,
with "Wiley".
 */

```

O equivalente que usa vários métodos `Join` usa a mesma abordagem com o tipo anônimo (no exemplo abaixo é nomeado `commonOwner`):

C#

```

var query = people.Join(cats,
    person => person,
    cat => cat.Owner,
    (person, cat) => new { person, cat })
    .Join(dogs,
        commonOwner => new { Owner = commonOwner.person, Letter =
commonOwner.cat.Name.Substring(0, 1) },
        dog => new { dog.Owner, Letter = dog.Name.Substring(0, 1) },
        (commonOwner, dog) => new { CatName = commonOwner.cat.Name, DogName
= dog.Name });

```

# Exemplo – junção interna usando junção agrupada

O exemplo a seguir mostra como implementar uma junção interna usando uma junção de grupo.

Em `query1`, a lista de objetos `Person` é unida por grupo à lista de objetos `Pet` com base no `Person` correspondente à propriedade `Pet.Owner`. A junção de grupo cria uma coleção de grupos intermediários, em que cada grupo é composto por um objeto `Person` e uma sequência de objetos `Pet` correspondentes.

Ao adicionar uma segunda cláusula `from` à consulta, essa sequência de sequências é combinada (ou mesclada) na sequência mais longa. O tipo dos elementos da sequência de final é especificado pela cláusula `select`. Neste exemplo, o tipo é um tipo anônimo que consiste nas propriedades `Person.FirstName` e `Pet.Name` para cada par correspondente.

O resultado de `query1` é equivalente ao conjunto de resultados que seria obtido usando a cláusula `join` sem a cláusula `into` para realizar uma junção interna. A variável `query2` demonstra essa consulta equivalente.

C#

```
Person magnus = new(FirstName: "Magnus", LastName: "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");

List<Person> people = new() { magnus, terry, charlotte, arlene };

List<Pet> pets = new()
{
    new(Name: "Barley", Owner: terry),
    new("Boots", terry),
    new("Whiskers", charlotte),
    new("Blue Moon", terry),
    new("Daisy", magnus),
};

var query1 =
    from person in people
    join pet in pets on person equals pet.Owner into gj
    from subpet in gj
    select new
    {
        OwnerName = person.FirstName,
        PetName = subpet.Name
```

```

};

string result = "";
result += "Inner join using GroupJoin():\r\n";
foreach (var v in query1)
{
    result += $"{v.OwnerName} - {v.PetName}\r\n";
}

var query2 =
    from person in people
    join pet in pets on person equals pet.Owner
    select new
    {
        OwnerName = person.FirstName,
        PetName = pet.Name
    };

result += "\r\nThe equivalent operation using Join():\r\n";
foreach (var v in query2)
{
    result += $"{v.OwnerName} - {v.PetName}\r\n";
}
Console.WriteLine(result);
return result;

/* Output:
   Inner join using GroupJoin():
   Magnus - Daisy
   Terry - Barley
   Terry - Boots
   Terry - Blue Moon
   Charlotte - Whiskers

   The equivalent operation using Join():
   Magnus - Daisy
   Terry - Barley
   Terry - Boots
   Terry - Blue Moon
   Charlotte - Whiskers
*/

```

Os mesmos resultados podem ser obtidos usando o método [GroupJoin](#) da seguinte forma:

C#

```

var query1 = people.GroupJoin(pets,
    person => person,
    pet => pet.Owner,
    (person, gj) => new { person, gj })
.SelectMany(pet => pet.gj,

```

```
(groupJoinPet, subpet) => new { OwnerName =  
groupJoinPet.person.FirstName, PetName = subpet.Name } );
```

Observe que essa abordagem requer o encadeamento dos resultados da consulta com [SelectMany](#) para criar a lista final da relação Proprietário - Animal de estimação com base nos resultados da junção do grupo. Para evitar o encadeamento, o método único [Join](#) pode ser usado conforme apresentado abaixo:

C#

```
var query2 = people.Join(pets,  
    person => person,  
    pet => pet.Owner,  
    (person, pet) => new { OwnerName = person.FirstName, PetName = pet.Name  
});
```

## Confira também

- [Join](#)
- [GroupJoin](#)
- [Executar junções agrupadas](#)
- [Executar junções externas esquerdas](#)
- [Tipos anônimos](#)

# Executar junções agrupadas

Artigo • 07/04/2023

A junção de grupo é útil para a produção de estruturas de dados hierárquicos. Ela combina cada elemento da primeira coleção com um conjunto de elementos correlacionados da segunda coleção.

Por exemplo, uma classe ou uma tabela de banco de dados relacional chamada `Student` pode conter dois campos: `Id` e `Name`. Uma segunda classe ou tabela de banco de dados relacional chamada `Course` pode conter dois campos: `StudentId` e `CourseTitle`. Uma junção de grupo dessas duas fontes de dados, com base na correspondência de `Student.Id` e `Course.StudentId`, agruparia cada `Student` com uma coleção de objetos `Course` (que pode estar vazia).

## ⓘ Observação

Cada elemento da primeira coleção aparece no conjunto de resultados de uma junção de grupo, independentemente de se os elementos correlacionados encontram-se na segunda coleção. Caso nenhum elemento correlacionado seja encontrado, a sequência de elementos correlacionados desse elemento ficará vazia. O seletor de resultado, portanto, tem acesso a todos os elementos da primeira coleção. Isso difere do seletor de resultado de uma junção que não é de grupo, que não pode acessar os elementos da primeira coleção que não têm correspondência na segunda coleção.

## ⚠ Aviso

`Enumerable.GroupJoin` não tem equivalente direto em termos de banco de dados relacionais tradicionais. No entanto, esse método implementa um superconjunto de junções internas e junções externas à esquerda. Ambas as operações podem ser gravadas em termos de uma junção agrupada. Para obter mais informações, consulte [Operações de Junção e Entity Framework Core, GroupJoin](#).

O primeiro exemplo neste artigo mostra como executar uma junção de grupo. O segundo exemplo mostra como usar uma junção de grupo para criar elementos XML.

## ⓘ Observação

Os exemplos neste tópico usam as classes de dados Person e Pet de Executar junções internas.

## Exemplo – junção de grupo

O exemplo a seguir realiza uma junção de grupo de objetos do tipo Person e Pet com base em Person correspondente à propriedade Pet.Owner. Ao contrário de uma junção que não é de grupo, que produziria um par de elementos para cada correspondência, a junção de grupo produz apenas um objeto resultante para cada elemento da primeira coleção, que neste exemplo é um objeto Person. Os elementos correspondentes da segunda coleção, que neste exemplo são objetos Pet, são agrupados em uma coleção. Por fim, a função de seletor de resultado cria um tipo anônimo para cada correspondência que consiste em Person.FirstName e em uma coleção de objetos Pet.

C#

```
Person magnus = new(FirstName: "Magnus", LastName: "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");

List<Person> people = new() { magnus, terry, charlotte, arlene };

List<Pet> pets = new()
{
    new(Name: "Barley", Owner: terry),
    new("Boots", terry),
    new("Whiskers", charlotte),
    new("Blue Moon", terry),
    new("Daisy", magnus),
};

// Create a list where each element is an anonymous type
// that contains the person's first name and a collection of
// pets that are owned by them.
var query =
    from person in people
    join pet in pets on person equals pet.Owner into gj
    select new
    {
        OwnerName = person.FirstName,
        Pets = gj
    };

foreach (var v in query)
{
    // Output the owner's name.
    Console.WriteLine($"{v.OwnerName}:");
```

```

// Output each of the owner's pet's names.
foreach (var pet in v.Pets)
{
    Console.WriteLine($" {pet.Name}");
}
}

/* Output:
Magnus:
Daisy
Terry:
Barley
Boots
Blue Moon
Charlotte:
Whiskers
Arlene:
*/

```

## Exemplo – junção de grupo para criar XML

As junções de grupo são ideais para a criação de XML usando o LINQ to XML. O exemplo a seguir é semelhante ao exemplo anterior, exceto que em vez de criar tipos anônimos, a função de seletor de resultado cria elementos XML que representam os objetos associados.

C#

```

// using System.Xml.Linq;

Person magnus = new(FirstName: "Magnus", LastName: "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");

List<Person> people = new() { magnus, terry, charlotte, arlene };

List<Pet> pets = new()
{
    new(Name: "Barley", Owner: terry),
    new("Boots", terry),
    new("Whiskers", charlotte),
    new("Blue Moon", terry),
    new("Daisy", magnus),
};

 XElement ownersAndPets = new("PetOwners",
    from person in people
    join pet in pets on person equals pet.Owner into gj

```

```

        select new XElement("Person",
            new XAttribute("FirstName", person.FirstName),
            new XAttribute("LastName", person.LastName),
            from subpet in gj
            select new XElement("Pet", subpet.Name)
    )
);

Console.WriteLine(ownersAndPets);

/* Output:
<PetOwners>
  <Person FirstName="Magnus" LastName="Hedlund">
    <Pet>Daisy</Pet>
  </Person>
  <Person FirstName="Terry" LastName="Adams">
    <Pet>Barley</Pet>
    <Pet>Boots</Pet>
    <Pet>Blue Moon</Pet>
  </Person>
  <Person FirstName="Charlotte" LastName="Weiss">
    <Pet>Whiskers</Pet>
  </Person>
  <Person FirstName="Arlene" LastName="Huff" />
</PetOwners>
*/

```

## Confira também

- Join
- GroupJoin
- Executar junções internas
- Executar junções externas esquerdas
- Tipos anônimos

# Executar junções externas esquerdas

Artigo • 07/04/2023

Uma junção externa esquerda é uma junção em que cada elemento da primeira coleção é retornado, mesmo que ele tenha elementos correlacionados na segunda coleção. É possível usar o LINQ para executar uma junção externa esquerda chamando o método [DefaultIfEmpty](#) nos resultados de uma junção de grupo.

## ⓘ Observação

O exemplo neste tópico usa as classes de dados `Pet` e `Person` de [Executar junções internas](#).

## Exemplo

O exemplo a seguir demonstra como usar o método [DefaultIfEmpty](#) nos resultados de uma junção de grupo para executar uma junção externa esquerda.

A primeira etapa da produção de uma junção externa esquerda de duas coleções é executar uma junção interna usando uma junção de grupo. (Confira [Executar junções internas](#) para obter uma explicação desse processo.) Neste exemplo, a lista de objetos `Person` é combinada por junção interna com a lista de objetos `Pet` com base em um objeto `Person` que corresponde a `Pet.Owner`.

A segunda etapa é incluir cada elemento da primeira coleção (esquerda) no conjunto de resultados, mesmo que esse elemento não tenha nenhuma correspondência na coleção direita. Isso é feito chamando [DefaultIfEmpty](#) em cada sequência de elementos correspondentes da junção de grupo. Neste exemplo, [DefaultIfEmpty](#) é chamado em cada sequência de objetos `Pet` correspondentes. O método retorna uma coleção que contém um valor padrão único se a sequência de objetos `Pet` correspondentes estiver vazia para qualquer objeto `Person`, garantindo assim que cada objeto `Person` seja representado no conjunto de resultados.

## ⓘ Observação

O valor padrão para um tipo de referência é `null`; portanto, o exemplo procura uma referência nula antes de acessar cada elemento de cada coleção `Pet`.

C#

```
Person magnus = new("Magnus", "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");

Pet barley = new("Barley", terry);
Pet boots = new("Boots", terry);
Pet whiskers = new("Whiskers", charlotte);
Pet bluemoon = new("Blue Moon", terry);
Pet daisy = new("Daisy", magnus);

// Create two lists.
List<Person> people = new() { magnus, terry, charlotte, arlene };
List<Pet> pets = new() { barley, boots, whiskers, bluemoon, daisy };

var query =
    from person in people
    join pet in pets on person equals pet.Owner into gj
    from subpet in gj.DefaultIfEmpty()
    select new
    {
        person.FirstName,
        PetName = subpet?.Name ?? string.Empty
    };

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName + ":",-15}{v.PetName}");
}

record class Person(string FirstName, string LastName);
record class Pet(string Name, Person Owner);

// This code produces the following output:
// 
// Magnus:      Daisy
// Terry:       Barley
// Terry:       Boots
// Terry:       Blue Moon
// Charlotte:   Whiskers
// Arlene:
```

## Confira também

- Join
- GroupJoin
- Executar junções internas
- Executar junções agrupadas

- Tipos anônimos

# Ordenar os resultados de uma cláusula join

Artigo • 07/04/2023

Este exemplo mostra como ordenar os resultados de uma operação de junção. Observe que a ordenação é executada após a junção. Embora você possa usar uma cláusula `orderby` com uma ou mais sequências de origem antes da junção, normalmente não é recomendável. Alguns provedores LINQ não podem preservar essa ordem após a junção.

## ① Observação

O exemplo deste tópico usa as seguintes classes de dados:

C#

```
record Product(string Name, int CategoryID);
record Category(string Name, int ID);
```

## Exemplo

Esta consulta cria uma junção de grupos e classifica os grupos com base no elemento de categoria, que ainda está no escopo. Dentro do inicializador de tipo anônimo, uma subconsulta ordena todos os elementos de correspondência da sequência de produtos.

C#

```
List<Category> categories = new()
{
    new(Name: "Beverages", ID: 001),
    new("Condiments", 002),
    new("Vegetables", 003),
    new("Grains", 004),
    new("Fruit", 005)
};

List<Product> products = new()
{
    new(Name: "Cola", CategoryID: 001),
    new("Tea", 001),
    new("Mustard", 002),
    new("Pickles", 002),
    new("Carrots", 003),
```

```

        new("Bok Choy", 003),
        new("Peaches", 005),
        new("Melons", 005),
    };

var groupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    orderby category.Name
    select new
    {
        Category = category.Name,
        Products =
            from prod2 in prodGroup
            orderby prod2.Name
            select prod2
    };

foreach (var productGroup in groupJoinQuery2)
{
    Console.WriteLine(productGroup.Category);
    foreach (var prodItem in productGroup.Products)
    {
        Console.WriteLine($" {prodItem.Name,-10} {prodItem.CategoryID}");
    }
}

/* Output:
   Beverages
      Cola      1
      Tea       1
   Condiments
      Mustard   2
      Pickles   2
   Fruit
      Melons    5
      Peaches   5
   Grains
   Vegetables
      Bok Choy  3
      Carrots   3
*/

```

## Confira também

- LINQ (Consulta Integrada à Linguagem)
- Cláusula orderby
- Cláusula join

# Unir usando chaves compostas

Artigo • 07/04/2023

Este exemplo mostra como realizar operações de junção nas quais você deseja usar mais de uma chave para definir uma correspondência. Isso é realizado por meio de uma chave composta. Uma chave composta é criada como um tipo anônimo ou como um tipo nomeado com os valores que você deseja comparar. Se a variável de consulta será passada entre limites de método, use um tipo nomeado que substitui [Equals](#) e [GetHashCode](#) para a chave. Os nomes das propriedades e a ordem em que elas ocorrem, devem ser idênticas em cada chave.

## Exemplo

O exemplo a seguir demonstra como usar uma chave composta para unir dados de três tabelas:

C#

```
var query = from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new {o.OrderID, p.ProductID} equals new {d.OrderID, d.ProductID}
    into details
        from d in details
    select new {o.OrderID, p.ProductID, d.UnitPrice};
```

A inferência de tipos em chaves compostas depende dos nomes das propriedades nas chaves e da ordem em que elas ocorrem. Quando as propriedades nas sequências de origem não têm os mesmos nomes, você precisa atribuir novos nomes nas chaves. Por exemplo, se a tabela `Orders` e a tabela `OrderDetails` usaram nomes diferentes para suas colunas, você poderia criar chaves compostas ao atribuir nomes idênticos nos tipos anônimos:

C#

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
    new {Name = d.CustName, ID = d.CustID }
```

As chaves compostas também podem ser usadas em uma cláusula `group`.

## Confira também

- LINQ (Consulta Integrada à Linguagem)
- Cláusula join
- Cláusula group

# Executar operações de junção personalizadas

Artigo • 07/04/2023

Este exemplo mostra como executar operações de junção que não são possíveis com a cláusula `join`. Em uma expressão de consulta, a cláusula `join` é limitada e otimizada para junções por igualdade, que são, de longe, o tipo de operação de junção mais comum. Ao realizar uma junção por igualdade, provavelmente você terá o melhor desempenho usando a cláusula `join`.

No entanto, a cláusula `join` não pode ser usada nos seguintes casos:

- Quando a junção se baseia em uma expressão de desigualdade (uma junção que não é por igualdade).
- Quando a junção se baseia em mais de uma expressão de igualdade ou desigualdade.
- Quando for necessário introduzir uma variável de intervalo temporária para a sequência do lado direito (interna) antes da operação de junção.

Para executar junções que não são junções por igualdade, você pode usar várias cláusulas `from` para introduzir cada fonte de dados de forma independente. Em seguida, você aplica uma expressão de predicado em uma cláusula `where` à variável de intervalo para cada fonte. A expressão também pode assumir a forma de uma chamada de método.

## ⓘ Observação

Não confunda esse tipo de operação de junção personalizada com o uso de várias cláusulas `from` para acessar coleções internas. Para obter mais informações, consulte [Cláusula join](#).

## Junção cruzada

## ⓘ Observação

Este exemplo é aquele após o uso das definições `Product` e `Category` em [Ordenar os resultados de uma cláusula join](#).

Essa consulta mostra uma união cruzada simples. Uniões cruzadas devem ser usadas com cuidado porque podem produzir conjuntos de resultados muito grandes. No entanto, elas podem ser úteis em alguns cenários para criar sequências de origem em que são executadas consultas adicionais.

C#

```
List<Category> categories = new()
{
    new(Name: "Beverages", ID: 001),
    new("Condiments", 002),
    new("Vegetables", 003)
};

List<Product> products = new()
{
    new(Name: "Tea", CategoryID: 001),
    new("Mustard", 002),
    new("Pickles", 002),
    new("Carrots", 003),
    new("Bok Choy", 003),
    new("Peaches", 005),
    new("Melons", 005),
    new("Ice Cream", 007),
    new("Mackerel", 012)
};

var crossJoinQuery =
    from c in categories
    from p in products
    select new
    {
        c.ID,
        p.Name
    };

Console.WriteLine("Cross Join Query:");
foreach (var v in crossJoinQuery)
{
    Console.WriteLine($"{v.ID,-5}{v.Name}");
}

/* Output:
   Cross Join Query:
   1    Tea
   1    Mustard
   1    Pickles
   1    Carrots
   1    Bok Choy
   1    Peaches
   1    Melons
   1    Ice Cream
   1    Mackerel
```

```
2   Tea
2   Mustard
2   Pickles
2   Carrots
2   Bok Choy
2   Peaches
2   Melons
2   Ice Cream
2   Mackerel
3   Tea
3   Mustard
3   Pickles
3   Carrots
3   Bok Choy
3   Peaches
3   Melons
3   Ice Cream
3   Mackerel
*/

```

## Sem junção por igualdade

Essa consulta produz uma sequência de todos os produtos cuja ID da categoria está na lista de categorias no lado esquerdo. Observe o uso da cláusula `let` e do método `Contains` para criar uma matriz temporária. Também é possível criar a matriz antes da consulta e eliminar a primeira cláusula `from`.

C#

```
var nonEquijoinQuery =
    from p in products
    let catIds =
        from c in categories
        select c.ID
    where catIds.Contains(p.CategoryID) == true
    select new
    {
        Product = p.Name,
        p.CategoryID
    };

Console.WriteLine("Non-equijoin query:");
foreach (var v in nonEquijoinQuery)
{
    Console.WriteLine($"{v.CategoryID,-5}{v.Product}");
}

/* Output:
   Non-equijoin query:
   1      Tea

```

```
2   Mustard
2   Pickles
3   Carrots
3   Bok Choy
*/
```

## Mesclar arquivos CSV

No exemplo a seguir, a consulta deve unir duas sequências com base nas chaves correspondentes que, no caso da sequência interna (lado direito), não podem ser obtidas antes da cláusula `join`. Se essa junção tiver sido executada com uma cláusula `join`, o método `Split` precisará ser chamado para cada elemento. O uso de várias cláusulas `from` permite que a consulta evite a sobrecarga da chamada de método repetida. No entanto, como `join` é otimizado, neste caso em particular ainda pode ser mais rápido do que usar várias cláusulas `from`. Os resultados variam dependendo principalmente do quanto a chamada de método é cara.

C#

```
string[] names = File.ReadAllLines(@"csv/names.csv");
string[] scores = File.ReadAllLines(@"csv/scores.csv");

// Merge the data sources using a named type.
// You could use var instead of an explicit type for the query.
IEnumerable<Student> queryNamesScores =
    // Split each line in the data files into an array of strings.
    from name in names
    let x = name.Split(',')
    from score in scores
    let s = score.Split(',')
    // Look for matching IDs from the two data files.
    where x[2] == s[0]
    // If the IDs match, build a Student object.
    select new Student(
        FirstName: x[0],
        LastName: x[1],
        StudentID: int.Parse(x[2]),
        ExamScores: (
            from scoreAsText in s.Skip(1)
            select int.Parse(scoreAsText)
        ).ToList()
    );
    // Optional. Store the newly created student objects in memory
    // for faster access in future queries
    List<Student> students = queryNamesScores.ToList();

foreach (var student in students)
{
```

```
        Console.WriteLine($"The average score of {student.FirstName}  
{student.LastName} is {student.ExamScores.Average()}."));  
    }  
  
/* Output:  
   The average score of Omelchenko Svetlana is 82.5.  
   The average score of O'Donnell Claire is 72.25.  
   The average score of Mortensen Sven is 84.5.  
   The average score of Garcia Cesar is 88.25.  
   The average score of Garcia Debra is 67.  
   The average score of Fakhouri Fadi is 92.25.  
   The average score of Feng Hanying is 88.  
   The average score of Garcia Hugo is 85.75.  
   The average score of Tucker Lance is 81.75.  
   The average score of Adams Terry is 85.25.  
   The average score of Zabokritski Eugene is 83.  
   The average score of Tucker Michael is 92.  
*/
```

## Confira também

- LINQ (Consulta Integrada à Linguagem)
- Cláusula join
- Ordenar os resultados de uma cláusula join

# Manipular valores nulos em expressões de consulta

Artigo • 07/04/2023

Este exemplo mostra como tratar os possíveis valores nulos em coleções de origem. Uma coleção de objetos, tal como uma `IEnumerable<T>`, pode conter elementos cujo valor é `null`. Se uma coleção de origem for `null` ou contiver um elemento cujo valor seja `null` e a consulta não lidar com valores `null`, uma `NullReferenceException` será gerada ao executar a consulta.

Você pode escrever o código defensivamente para evitar uma exceção de referência nula conforme mostrado no exemplo a seguir:

C#

```
var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals p?.CategoryID
    select new
    {
        Category = c.Name,
        Name = p.Name
    };
}
```

No exemplo anterior, a cláusula `where` filtra todos os elementos nulos na sequência de categorias. Essa técnica é independente da verificação de nulos na cláusula `join`. A expressão condicional com `null` nesse exemplo funciona porque `Products.CategoryID` é do tipo `int?`, que é uma abreviação para `Nullable<int>`.

Em uma cláusula `join`, se apenas uma das chaves de comparação for um tipo de valor anulável que, você pode converter a outra para um tipo de valor anulável na expressão de consulta. No exemplo a seguir, suponha que `EmployeeID` é uma coluna que contém os valores do tipo `int?`:

C#

```
void TestMethod(Northwind db)
{
    var query =
        from o in db.Orders
        join e in db.Employees
        on o.EmployeeID equals (int?)e.EmployeeID
```

```
        select new { o.OrderID, e.FirstName };  
    }
```

Em cada um dos exemplos, a palavra-chave de consulta `equals` é usada. O C# 9 adiciona [padrões correspondentes](#), o que inclui padrões para `is null` e `is not null`. Esses padrões não são recomendados em consultas LINQ porque os provedores de consulta podem não interpretar a nova sintaxe C# corretamente. Um provedor de consultas é uma biblioteca que converte expressões de consulta C# em um formato de dados nativo, como o Entity Framework Core. Os provedores de consulta implementam a interface [System.Linq.IQueryProvider](#) para criar fontes de dados que implementam a interface [System.Linq.IQueryable<T>](#).

## Confira também

- [Nullable<T>](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Tipos de valor anuláveis](#)

# Tratar exceções em expressões de consulta

Artigo • 10/05/2023

É possível chamar qualquer método no contexto de uma expressão de consulta. No entanto, é recomendável que você evite chamar qualquer método em uma expressão de consulta que possa criar um efeito colateral, como modificar o conteúdo da fonte de dados ou gerar uma exceção. Este exemplo mostra como evitar exceções ao chamar métodos em uma expressão de consulta, sem violar as diretrizes gerais sobre tratamento de exceção do .NET. Essas diretrizes declaram que é aceitável capturar uma exceção específica quando você entende por que ela é gerada em um determinado contexto. Para obter mais informações, consulte [Melhores práticas para exceções](#).

O último exemplo mostra como tratar os casos em que é necessário lançar uma exceção durante a execução de uma consulta.

## Exemplo 1

O exemplo a seguir mostra como mover o código de tratamento de exceção para fora de uma expressão de consulta. Isso só é possível quando o método não depende de nenhuma variável que seja local para a consulta.

C#

```
// A data source that is very likely to throw an exception!
IEnumerable<int> GetData() => throw new InvalidOperationException();

// DO THIS with a datasource that might
// throw an exception. It is easier to deal with
// outside of the query expression.
IEnumerable<int>? dataSource = null;
try
{
    dataSource = GetData();
}
catch (InvalidOperationException)
{
    // Handle (or don't handle) the exception
    // in the way that is appropriate for your application.
    Console.WriteLine("Invalid operation");
}

if (dataSource is not null)
{
    // If we get here, it is safe to proceed.
```

```

var query =
    from i in dataSource
    select i * i;

foreach (var i in query)
{
    Console.WriteLine(i.ToString());
}
}

```

## Exemplo 2

Em alguns casos, a melhor resposta para uma exceção que é lançada de dentro de uma consulta poderá ser a interrupção imediata da execução da consulta. O exemplo a seguir mostra como tratar exceções que podem ser geradas de dentro de um corpo de consulta. Suponha que `SomeMethodThatMightThrow` possa causar uma exceção que exija que a execução da consulta seja interrompida.

Observe que o bloco `try` inclui o loop `foreach` e não a própria consulta. Isso ocorre porque o loop `foreach` é o ponto em que a consulta é realmente executada. Para obter mais informações, consulte [Introdução a consultas LINQ](#).

C#

```

// Not very useful as a general purpose method.
string SomeMethodThatMightThrow(string s) =>
    s[4] == 'C' ?
        throw new InvalidOperationException() :
        @"C:\newFolder\" + s;

// Data source.
string[] files = { "fileA.txt", "fileB.txt", "fileC.txt" };

// Demonstration query that throws.
var exceptionDemoQuery =
    from file in files
    let n = SomeMethodThatMightThrow(file)
    select n;

// The runtime exception will only be thrown when the query is executed.
// Therefore they must be handled in the foreach loop.
try
{
    foreach (var item in exceptionDemoQuery)
    {
        Console.WriteLine($"Processing {item}");
    }
}

```

```
// Catch whatever exception you expect to raise
// and/or do any necessary cleanup in a finally block
catch (InvalidOperationException e)
{
    Console.WriteLine(e.Message);
}

/* Output:
   Processing C:\newFolder\fileA.txt
   Processing C:\newFolder\fileB.txt
   Operation is not valid due to the current state of the object.
*/
```

## Confira também

- [LINQ \(Consulta Integrada à Linguagem\)](#)

# Reduce memory allocations using new C# features

Article • 04/06/2023

## ⓘ Important

The techniques described in this section improve performance when applied to *hot paths* in your code. *Hot paths* are those sections of your codebase that are executed often and repeatedly in normal operations. Applying these techniques to code that isn't often executed will have minimal impact. Before making any changes to improve performance, it's critical to measure a baseline. Then, analyze that baseline to determine where memory bottlenecks occur. You can learn about many cross platform tools to measure your application's performance in the section on [Diagnostics and instrumentation](#). You can practice a profiling session in the tutorial to [Measure memory usage](#) in the Visual Studio documentation.

Once you've measured memory usage and have determined that you can reduce allocations, use the techniques in this section to reduce allocations. After each successive change, measure memory usage again. Make sure each change has a positive impact on the memory usage in your application.

Performance work in .NET often means removing allocations from your code. Every block of memory you allocate must eventually be freed. Fewer allocations reduce time spent in garbage collection. It allows for more predictable execution time by removing garbage collections from specific code paths.

A common tactic to reduce allocations is to change critical data structures from `class` types to `struct` types. This change impacts the semantics of using those types. Parameters and returns are now passed by value instead of by reference. The cost of copying a value is negligible if the types are small, three words or less (considering one word being of natural size of one integer). It's measurable and can have real performance impact for larger types. To combat the effect of copying, developers can pass these types by `ref` to get back the intended semantics.

The C# `ref` features give you the ability to express the desired semantics for `struct` types without negatively impacting their overall usability. Prior to these enhancements, developers needed to resort to `unsafe` constructs with pointers and raw memory to achieve the same performance impact. The compiler generates *verifiably safe code* for the new `ref` related features. *Verifiably safe code* means the compiler detects possible

buffer overruns or accessing unallocated or freed memory. The compiler detects and prevents some errors.

## Pass and return by reference

Variables in C# store *values*. In `struct` types, the value is the contents of an instance of the type. In `class` types, the value is a reference to a block of memory that stores an instance of the type. Adding the `ref` modifier means that the variable stores the *reference* to the value. In `struct` types, the reference points to the storage containing the value. In `class` types, the reference points to the storage containing the reference to the block of memory.

In C#, parameters to methods are *passed by value*, and return values are *return by value*. The *value* of the argument is passed to the method. The *value* of the return argument is the return value.

The `ref`, `in`, or `out` modifier indicates that parameter is *passed by reference*. The *reference* to the storage location is passed to the method. Adding `ref` to the method signature means the return value is *returned by reference*. The *reference* to the storage location is the return value.

You can also use *ref assignment* to have a variable refer to another variable. A typical assignment copies the *value* of the right hand side to the variable on the left hand side of the assignment. A *ref assignment* copies the memory location of the variable on the right hand side to the variable on the left hand side. The `ref` now refers to the original variable:

```
C#
```

```
int anInteger = 42; // assignment.  
ref int location = ref anInteger; // ref assignment.  
ref int sameLocation = ref location; // ref assignment  
  
Console.WriteLine(location); // output: 42  
  
sameLocation = 19; // assignment  
  
Console.WriteLine(anInteger); // output: 19
```

When you *assign* a variable, you change its *value*. When you *ref assign* a variable, you change what it refers to.

You can work directly with the storage for values using `ref` variables, pass by reference, and `ref` assignment. Scope rules enforced by the compiler ensure safety when working directly with storage.

## Ref safe to escape scope

C# includes rules for `ref` expressions to ensure that a `ref` expression can't be accessed where the storage it refers to is no longer valid. Consider the following example:

C#

```
public ref int CantEscape()
{
    int index = 42;
    return ref index; // Error: index's ref safe to escape scope is the body
                      of CantEscape
}
```

The compiler reports an error because you can't return a reference to a local variable from a method. The caller can't access the storage being referred to. The *ref safe to escape scope* defines the scope in which a `ref` expression is safe to access or modify. The following table lists the *ref safe to escape scopes* for variable types. `ref` fields can't be declared in a `class` or a non-ref `struct`, so those rows aren't in the table:

Declaration	<b>ref safe to escape scope</b>
non-ref local	block where local is declared
non-ref parameter	current method
<code>ref</code> , <code>in</code> parameter	calling method
<code>out</code> parameter	current method
<code>class</code> field	calling method
non-ref <code>struct</code> field	current method
<code>ref</code> field of <code>ref struct</code>	calling method

A variable can be `ref` returned if its *ref safe to escape scope* is the calling method. If its *ref safe to escape scope* is the current method or a block, `ref` return is disallowed. The following snippet shows two examples. A member field can be accessed from the scope calling a method, so a class or struct field's *ref safe to escape scope* is the calling method.

The *ref safe to escape scope* for a parameter with the `ref`, or `in` modifiers is the entire method. Both can be `ref` returned from a member method:

```
C#  
  
private int anIndex;  
  
public ref int RetrieveIndexRef()  
{  
    return ref anIndex;  
}  
  
public ref int RefMin(ref int left, ref int right)  
{  
    if (left < right)  
        return ref left;  
    else  
        return ref right;  
}
```

### ⓘ Note

When the `in` modifier is applied to a parameter, that parameter can be returned by `ref readonly`, not `ref`.

The compiler ensures that a reference can't escape its *ref safe to escape scope*. You can use `ref` parameters, `ref return` and `ref` local variables safely because the compiler detects if you've accidentally written code where a `ref` expression could be accessed when its storage isn't valid.

## Safe to escape scope and ref structs

`ref struct` types require more rules to ensure they can be used safely. A `ref struct` type may include `ref` fields. That requires the introduction of a *safe to escape scope*. For most types, the *safe to escape scope* is the calling method. In other words, a value that's not a `ref struct` can always be returned from a method.

Informally, the *safe to escape scope* for a `ref struct` is the scope where all of its `ref` fields can be accessed. In other words, it's the intersection of the *ref safe to escape scopes* of all its `ref` fields. The following method returns a `ReadOnlySpan<char>` to a member field, so its *safe to escape scope* is the method:

```
C#
```

```

private string longMessage = "This is a long message";

public ReadOnlySpan<char> Safe()
{
    var span = longMessage.AsSpan();
    return span;
}

```

In contrast, the following code emits an error because the `ref field` member of the `Span<int>` refers to the stack allocated array of integers. It can't escape the method:

C#

```

public Span<int> M()
{
    int length = 3;
    Span<int> numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
    return numbers; // Error! numbers can't escape this method.
}

```

## Unify memory types

The introduction of `System.Span<T>` and `System.Memory<T>` provide a unified model for working with memory. `System.ReadOnlySpan<T>` and `System.ReadOnlyMemory<T>` provide readonly versions for accessing memory. They all provide an abstraction over a block of memory storing an array of similar elements. The difference is that `Span<T>` and `ReadOnlySpan<T>` are `ref struct` types whereas `Memory<T>` and `ReadOnlyMemory<T>` are `struct` types. Spans contain a `ref field`. Therefore instances of a span can't leave its *safe to escape scope*. The *safe to escape scope* of a `ref struct` is the *ref safe to escape scope* of its `ref field`. The implementation of `Memory<T>` and `ReadOnlyMemory<T>` remove this restriction. You use these types to directly access memory buffers.

## Improve performance with ref safety

Using these features to improve performance involves these tasks:

- *Avoid allocations*: When you change a type from a `class` to a `struct`, you change how it's stored. Local variables are stored on the stack. Members are stored inline when the container object is allocated. This change means fewer allocations and

that decreases the work the garbage collector does. It may also decrease memory pressure so the garbage collector runs less often.

- *Preserve reference semantics*: Changing a type from a `class` to a `struct` changes the semantics of passing a variable to a method. Code that modified the state of its parameters needs modification. Now that the parameter is a `struct`, the method is modifying a copy of the original object. You can restore the original semantics by passing that parameter as a `ref` parameter. After that change, the method modifies the original `struct` again.
- *Avoid copying data*: Copying larger `struct` types can impact performance in some code paths. You can also add the `ref` modifier to pass larger data structures to methods by reference instead of by value.
- *Restrict modifications*: When a `struct` type is passed by reference, the called method could modify the state of the struct. You can replace the `ref` modifier with the `in` modifier to indicate that the argument can't be modified. You can also create `readonly struct` types or `struct` types with `readonly` members to provide more control over what members of a `struct` can be modified.
- *Directly manipulate memory*: Some algorithms are most efficient when treating data structures as a block of memory containing a sequence of elements. The `Span` and `Memory` types provide safe access to blocks of memory.

None of these techniques require `unsafe` code. Used wisely, you can get performance characteristics from safe code that was previously only possible by using unsafe techniques. You can try the techniques yourself in the tutorial on [reducing memory allocations](#).

# Expression Trees

Article • 03/09/2023

*Expression trees* represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

If you have used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you aren't familiar with LINQ, you probably want to read [the LINQ tutorial](#) and the article about [lambda expressions](#) before this one.) Expression Trees provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

You've likely already written code that uses Expression trees. Entity Framework's LINQ APIs accept Expression trees as the arguments for the LINQ Query Expression Pattern. That enables [Entity Framework](#) to translate the query you wrote in C# into SQL that executes in the database engine. Another example is [Moq](#) ↗, which is a popular mocking framework for .NET.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you examine, modify, or execute. These tools give you the power to manipulate code during run time. You write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you modify running algorithms and even translate C# expressions into another form for execution in another environment.

You compile and run code represented by expression trees. Building and running expression trees enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see [How to use expression trees to build dynamic queries](#).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see [Dynamic Language Runtime Overview](#).

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the `System.Linq.Expressions` namespace.

When a lambda expression is assigned to a variable of type `Expression<TDelegate>`, the compiler emits code to build an expression tree that represents the lambda expression.

The C# compiler generates expression trees only from expression lambdas (or single-line lambdas). It can't parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see [Lambda Expressions](#).

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

C#

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

You create expression trees in your code. You build the tree by creating each node and attaching the nodes into a tree structure. You learn how to create expressions in the article on [building expression trees](#).

Expression trees are immutable. If you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You use an expression tree visitor to traverse the existing expression tree. For more information, see the article on [translating expression trees](#).

Once you build an expression tree, you [execute the code represented by the expression tree](#).

## Limitations

There are some newer C# language elements that don't translate well into expression trees. Expression trees can't contain `await` expressions, or `async` lambda expressions. Many of the features added in C# 6 and later don't appear exactly as written in expression trees. Instead, newer features are exposed in expression trees in the equivalent, earlier syntax, where possible. Other constructs aren't available. It means that code that interprets expression trees works the same when new language features are introduced. However, even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It enables rich libraries such as Entity Framework to accomplish what they do.

Expression trees won't support new expression node types. It would be a breaking change for all libraries interpreting expression trees to introduce new node types. The following list includes most C# language elements that can't be used:

- Conditional methods that have been removed
- base access
- Method group expressions, including *address-of* (&) a method group, and anonymous method expressions
- References to local functions
- Statements, including assignment (=) and statement bodied expressions
- Partial methods with only a defining declaration
- Unsafe pointer operations
- dynamic operations
- Coalescing operators with null or default literal left side, null coalescing assignment, and the null propagating operator (?)
- Multi-dimensional array initializers, indexed properties, and dictionary initializers
- throw expressions
- Accessing static virtual or abstract interface members
- Lambda expressions that have attributes
- Interpolated strings
- UTF-8 string conversions or UTF-8 string literals
- Method invocations using variable arguments, named arguments or optional arguments
- Expressions using System.Index or System.Range, index "from end" (^) operator or range expressions (..)
- async lambda expressions or await expressions, including await foreach and await using
- Tuple literals, tuple conversions, tuple == or !=, or with expressions
- Discards (\_), deconstructing assignment, pattern matching is operator or the pattern matching switch expression
- COM call with ref omitted on the arguments
- ref, in or out parameters, ref return values, out arguments, or any values of ref struct type

# Árvores de expressão – Dados que definem o código

Artigo • 16/03/2023

Uma Árvore de expressão é uma estrutura de dados que define o código. As árvores de expressão se baseiam nas mesmas estruturas que um compilador usa para analisar o código e gerar a saída compilada. Ao ler este artigo, você notará certa semelhança entre as árvores de expressão e os tipos usados nas APIs Roslyn para criar [Analyzers](#) e [CodeFixes](#). (Os Analyzers e os CodeFixes são pacotes NuGet que executam análise estática no código e sugerem possíveis correções para um desenvolvedor.) Os conceitos são semelhantes e o resultado final é uma estrutura de dados que permite o exame do código-fonte de maneira significativa. No entanto, as árvores de expressão são baseadas em um conjunto de classes e APIs totalmente diferentes das APIs Roslyn. Aqui está uma linha de código:

C#

```
var sum = 1 + 2;
```

Ao analisar o código anterior como uma árvore de expressão, é possível perceber que a árvore contém vários nós. O nó mais externo é uma instrução de declaração de variável com atribuição (`var sum = 1 + 2;`). Esse nó mais externo contém vários nós filho: uma declaração de variável, um operador de atribuição e uma expressão que representa o lado direito do sinal de igual. Essa expressão é ainda subdividida em expressões que representam a operação de adição e os operandos esquerdo e direito da adição.

Vamos detalhar um pouco mais as expressões que compõem o lado direito do sinal de igual. A expressão é `1 + 2`, uma expressão binária. Mais especificamente, ela é uma expressão de adição binária. Uma expressão de adição binária tem dois filhos, que representam os nós esquerdo e direito da expressão de adição. Aqui, ambos os nós são expressões constantes: o operando esquerdo é o valor `1` e o operando direito é o valor `2`.

Visualmente, a declaração inteira é uma árvore: você pode começar no nó raiz e viajar até cada nó da árvore para ver o código que constitui a instrução:

- Instrução de declaração de variável com atribuição (`var sum = 1 + 2;`)
  - Declaração de tipo de variável implícita (`var sum`)
    - Palavra-chave var implícita (`var`)
    - Declaração de nome de variável (`sum`)

- Operador de atribuição (=)
- Expressão de adição binária (1 + 2)
  - Operando esquerdo (1)
  - Operador de adição (+)
  - Operando direito (2)

A árvore anterior pode parecer complicada, mas é muito eficiente. Seguindo o mesmo processo, é possível decompor expressões muito mais complicadas. Considere esta expressão:

C#

```
var finalAnswer = this.SecretSauceFunction(
    currentState.createInterimResult(), currentState.createSecondValue(1,
2),
    decisionServer.considerFinalOptions("hello")) +
MoreSecretSauce('A', DateTime.Now, true);
```

A expressão anterior também é uma declaração de variável com uma atribuição. Neste exemplo, o lado direito da atribuição é uma árvore muito mais complicada. Você não vai decompor essa expressão, mas considere quais seriam os diferentes nós. Há chamadas de método usando o objeto atual como receptor, uma com um receptor `this` explícito e outra não. Há chamadas de método usando outros objetos receptores, há argumentos constantes de tipos diferentes. E, por fim, há um operador de adição binário.

Dependendo do tipo de retorno de `SecretSauceFunction()` ou `MoreSecretSauce()`, esse operador de adição binária pode ser uma chamada de método para um operador de adição substituído, resolvendo em uma chamada de método estático ao operador de adição binária definido para uma classe.

Apesar da complexidade, a expressão anterior cria uma estrutura de árvore tão fácil de se navegar quanto a do primeiro exemplo. Você continua percorrendo os nós filho para encontrar os nós folha na expressão. Os nós pai terão referências aos filhos, sendo que cada nó tem uma propriedade que descreve o tipo de nó.

A estrutura de uma árvore de expressão é muito consistente. Depois de aprender os conceitos básicos, você entenderá até mesmo o código mais complexo, quando ele for representado como uma árvore de expressão. A elegância na estrutura de dados explica como o compilador C# analisa os programas em C# mais complexos e cria a saída apropriada desse código-fonte complicado.

Depois de se familiarizar com a estrutura das árvores de expressão, descobrirá que o conhecimento adquirido permitirá que você trabalhe com muitos outros cenários ainda mais avançados. O potencial das árvores de expressão é incrível.

Além de converter algoritmos para serem executados em outros ambientes, com as árvores de expressão, você pode escrever facilmente algoritmos que inspecionam o código antes de executá-lo. Você escreverá um método cujos argumentos são expressões e, depois, examinará essas expressões antes de executar o código. A árvore de expressão é uma representação completa do código: é possível ver os valores de qualquer subexpressão. Você vê os nomes dos métodos e das propriedades. Você vê o valor de qualquer expressão de constante. Você converte uma árvore de expressão em um delegado executável e executa o código.

As APIs para árvores de expressão permitem criar árvores que representam quase todos os constructos de código válidos. No entanto, para manter as coisas o mais simples possível, não é possível criar algumas expressões em C# em uma árvore de expressão. Um exemplo são as expressões assíncronas (usando as palavras-chave `async` e `await`). Se suas necessidades requerem algoritmos assíncronos, você precisa manipular diretamente os objetos `Task`, em vez de contar com o suporte do compilador. Outro exemplo é na criação de loops. Normalmente, você cria esses loops usando `for`, `foreach`, `while` ou `do`. Como você verá [mais adiante nesta série](#), as APIs para árvores de expressão dão suporte a uma expressão de loop individual, com expressões `break` e `continue` controlando a repetição do loop.

A única coisa que você não pode fazer é modificar uma árvore de expressão. As árvores de expressão são estruturas de dados imutáveis. Se quiser modificar (alterar) uma árvore de expressão, você deverá criar uma nova árvore, que seja uma cópia da original, com as alterações desejadas.

# Suporte ao runtime do .NET para árvores de expressão

Artigo • 16/03/2023

Há uma grande lista de classes no runtime do .NET que funcionam com árvores de expressão. Veja a lista completa em [System.Linq.Expressions](#). Em vez de enumerar a lista completa, vamos entender como as classes de runtime foram projetadas.

No design de linguagem, uma expressão é um corpo de código que calcula e retorna um valor. As expressões podem ser muito simples: a expressão constante `1` retorna o valor constante de 1. Elas podem ser mais complicados: a expressão `( -B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` retorna uma raiz de uma equação quadrática (no caso em que a equação tem uma solução).

## System.Linq.Expression e tipos derivados

Uma das complexidades de se trabalhar com árvores de expressão é que muitos tipos diferentes de expressões são válidos em muitos locais nos programas. Considere uma expressão de atribuição. O lado direito de uma atribuição pode ser um valor constante, uma variável, uma expressão de chamada de método ou outros. Essa flexibilidade de linguagem significa que você poderá encontrar muitos tipos diferentes de expressão em qualquer lugar nos nós de uma árvore ao percorrer uma árvore de expressão. Portanto, a maneira mais simples de se trabalhar é quando você pode trabalhar com o tipo de expressão de base. No entanto, às vezes você precisa saber mais. A classe Expressão de base contém uma propriedade `NodeType` para essa finalidade. Ela retorna um `ExpressionType` que é uma enumeração dos tipos de expressão possíveis. Quando você sabe qual é o tipo do nó, pode convertê-la para esse tipo e executar ações específicas, conhecendo o tipo do nó de expressão. Você pode pesquisar determinados tipos de nó e trabalhar com as propriedades específicas desse tipo de expressão.

Por exemplo, esse código imprimirá o nome de uma variável para uma expressão de acesso variável. O código abaixo mostra a prática de verificar o tipo de nó para convertê-lo em uma expressão de acesso variável e verificar as propriedades do tipo de expressão específico:

C#

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive is LambdaExpression lambdaExp)
```

```
{  
    var parameter = lambdaExp.Parameters[0]; -- first  
  
    Console.WriteLine(parameter.Name);  
    Console.WriteLine(parameter.Type);  
}
```

## Criar árvores de expressão

A classe `System.Linq.Expression` também contém vários métodos estáticos para criar expressões. Esses métodos criam um nó de expressão usando os argumentos fornecidos para seus filhos. Dessa forma, você cria uma expressão com os nós folha. Por exemplo, esse código cria uma expressão de Adicionar:

C#

```
// Addition is an add expression for "1 + 2"  
var one = Expression.Constant(1, typeof(int));  
var two = Expression.Constant(2, typeof(int));  
var addition = Expression.Add(one, two);
```

Você pode ver neste exemplo simples que há muitos tipos envolvidos na criação e no trabalho com árvores de expressão. Essa complexidade é necessária para fornecer os recursos do vocabulário avançado fornecido pela linguagem C#.

## Navegar pelas APIs

Há tipos de Nó de expressão que mapeiam para quase todos os elementos de sintaxe da linguagem C#. Cada tipo tem métodos específicos para esse tipo de elemento de linguagem. É muita coisa para guardar na memória ao mesmo tempo. Em vez de tentar memorizar tudo, confira técnicas úteis para trabalhar com árvores de expressão:

1. Observar os membros da enumeração `ExpressionType` para determinar possíveis nós que devem ser examinados. Essa lista é útil se você deseja percorrer e entender uma árvore de expressão.
2. Observar os membros estáticos da classe `Expression` para compilar uma expressão. Esses métodos podem compilar qualquer tipo de expressão em um conjunto de seu nós filho.
3. Examinar a classe `ExpressionVisitor` para compilar uma árvore de expressão modificada.

Você encontrará mais informações ao examinar cada uma dessas três áreas. Você sempre encontrará o que precisa ao começar com uma dessas três etapas.

# Executar árvores de expressão

Artigo • 16/03/2023

Uma *árvore de expressão* é uma estrutura de dados que representa algum código. Não se trata de código compilado nem executável. Se você quiser executar o código do .NET representado por uma árvore de expressão, precisará convertê-lo em instruções IL executáveis. Executar uma árvore de expressão pode retornar um valor ou apenas realizar uma ação, como chamar um método.

Somente árvores de expressão que representam expressões lambda podem ser executadas. Árvores de expressão que representam expressões lambda são do tipo [LambdaExpression](#) ou [Expression<TDelegate>](#). Para executar essas árvores de expressão, chame o método [Compile](#) para criar um delegado executável e, em seguida, invoque o delegado.

## ⓘ Observação

Se o tipo de delegado não for conhecido, ou seja, se a expressão lambda for do tipo [LambdaExpression](#) e não [Expression<TDelegate>](#), chame o método [DynamicInvoke](#) no delegado em vez de invocá-la diretamente.

Se uma árvore de expressão não representa uma expressão lambda, você pode criar uma expressão lambda que tenha a árvore de expressão original como corpo. Para isso, chame o método [Lambda<TDelegate>\(Expression, IEnumerable<ParameterExpression>\)](#). Em seguida, você pode executar a expressão lambda como descrito anteriormente nesta seção.

## Expressões lambda para funções

Você pode converter qualquer [LambdaExpression](#) ou qualquer tipo derivado de [LambdaExpression](#) em IL executável. Outros tipos de expressões não podem ser convertidos diretamente em código. Essa restrição tem pouco efeito na prática. As expressões lambda são os únicos tipos de expressões que você gostaria de executar convertendo em IL (linguagem intermediária) executável. (Pense no que significaria executar diretamente um [System.Linq.Expressions.ConstantExpression](#). Significaria algo útil?) Qualquer árvore de expressão, que seja um [System.Linq.Expressions.LambdaExpression](#) ou um tipo derivado de [LambdaExpression](#), pode ser convertida em IL. O tipo de expressão [System.Linq.Expressions.Expression<TDelegate>](#) é o único exemplo concreto nas

bibliotecas do .NET Core. Ele é usado para representar uma expressão que mapeia para qualquer tipo delegado. Como esse tipo mapeia para um tipo delegado, o .NET pode examinar a expressão e gerar a IL para um delegado apropriado que corresponda à assinatura da expressão lambda. O tipo delegado é baseado no tipo de expressão. Você deve conhecer o tipo de retorno e a lista de argumentos se quiser usar o objeto delegado de maneira fortemente tipada. O método `LambdaExpression.Compile()` retorna o tipo `Delegate`. Você precisará convertê-lo no tipo delegado correto para fazer com que as ferramentas de tempo de compilação verifiquem a lista de argumentos ou o tipo de retorno.

Na maioria dos casos, existe um mapeamento simples entre uma expressão e o delegado correspondente. Por exemplo, uma árvore de expressão representada por `Expression<Func<int>>` seria convertida em um delegado do tipo `Func<int>`. Para uma expressão lambda com qualquer tipo de retorno e lista de argumentos, existe um tipo delegado que é o tipo de destino para o código executável representado por essa expressão lambda.

O tipo `System.Linq.Expressions.LambdaExpression` contém membros `LambdaExpression.Compile` e `LambdaExpression.CompileToMethod` que você usaria para converter uma árvore de expressão em código executável. O método `Compile` cria um delegado. O método `CompileToMethod` atualiza um objeto `System.Reflection.Emit.MethodBuilder` com a IL que representa a saída compilada da árvore de expressão.

### ⓘ Importante

`CompileToMethod` só está disponível no .NET Framework, e não no .NET Core nem no .NET 5 e posterior.

Como opção, você também pode fornecer um `System.Runtime.CompilerServices.DebugInfoGenerator` que receberá as informações de depuração de símbolo para o objeto delegado gerado. O `DebugInfoGenerator` fornece informações completas de depuração sobre o delegado gerado.

Uma expressão seria convertida em um delegado usando o seguinte código:

C#

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

O exemplo de código a seguir demonstra os tipos concretos usados ao compilar e executar uma árvore de expressão.

C#

```
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

O exemplo de código a seguir demonstra como executar uma árvore de expressão que representa a elevação de um número a uma potência, criando uma expressão lambda e executando-a. O resultado, representado pelo número elevado à potência, é exibido.

C#

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2d),
                                         Expression.Constant(3d));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

## Execução e tempos de vida

O código é executado ao invocar o delegado que foi criado quando você chamou `LambdaExpression.Compile()`. O código anterior, `add.Compile()`, retorna um delegado. Você invoca esse delegado chamando `func()`, que executa o código.

Esse delegado representa o código na árvore de expressão. Você pode reter o identificador para esse delegado e invocá-lo mais tarde. Você não precisa compilar a árvore de expressão sempre que deseja executar o código que ela representa. (Lembre-se que as árvores de expressão são imutáveis e compilar a mesma árvore de expressão mais tarde, criará um delegado que executa o mesmo código.)

### ⊗ Cuidado

Não crie mecanismos de cache mais sofisticados para aumentar o desempenho evitando chamadas de compilação desnecessárias. Comparar duas árvores de expressão arbitrárias para determinar se elas representam o mesmo algoritmo é uma operação demorada. O tempo de computação que você economiza evitando chamadas extras a `LambdaExpression.Compile()` provavelmente é maior que o consumido pelo tempo de execução do código que determina se as duas árvores de expressão diferentes resultam no mesmo código executável.

## Advertências

A compilação de uma expressão lambda para um delegado e invocar esse delegado é uma das operações mais simples que você pode realizar com uma árvore de expressão. No entanto, mesmo com essa operação simples, há limitações que você deve estar ciente.

As expressões lambda criam fechamentos sobre todas as variáveis locais que são referenciadas na expressão. Você deve assegurar que todas as variáveis que farão parte do delegado são utilizáveis no local em que você chamar `Compile` e no momento em que você executar o delegado resultante. O compilador garante que as variáveis estejam no escopo. No entanto, se a sua expressão acessa uma variável que implementa `IDisposable`, é possível que o código descarte o objeto enquanto ele ainda é mantido pela árvore de expressão.

Por exemplo, esse código funciona bem, porque `int` não implementa `IDisposable`:

C#

```
private static Func<int, int> CreateBoundFunc()
{
```

```
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

O delegado capturou uma referência à variável local `constant`. Essa variável é acessada a qualquer momento mais tarde, quando a função retornada por `CreateBoundFunc` for executada.

No entanto, considere a seguinte classe (bastante artificial) que implementa `System.IDisposable`:

C#

```
public class Resource : IDisposable
{
    private bool _isDisposed = false;
    public int Argument
    {
        get
        {
            if (!_isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        _isDisposed = true;
    }
}
```

Se você a usar em uma expressão, conforme mostrado no seguinte código, obterá uma `System.ObjectDisposedException` ao executar o código referenciado pela propriedade `Resource.Argument`:

C#

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the
                                         // expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument +
        b;
        var rVal = expression.Compile();
        return rVal;
}
```

```
    }  
}
```

O delegado retornado desse método fechou sobre o objeto `constant`, que foi descartado. (Foi descartado, porque foi declarado em uma instrução `using`).

Agora, ao executar o delegado retornado por esse método, uma `ObjectDisposedException` será gerada no ponto de execução.

Parece estranho ter um erro de runtime representando um constructo de tempo de compilação, mas é isso que ocorre quando trabalha com árvores de expressão.

Há várias permutações desse problema, portanto é difícil oferecer diretrizes gerais para evitá-lo. Tenha cuidado ao acessar variáveis locais quando define expressões e ao acessar o estado no objeto atual (representado por `this`) quando cria uma árvore de expressão retornada por meio de uma API pública.

O código na sua expressão pode referenciar métodos ou propriedades em outros assemblies. Esse assembly deve estar acessível quando a expressão for definida, quando ela for compilada e quando o delegado resultante for invocado. Você é recebido com um `ReferencedAssemblyNotFoundException` quando ele não está presente.

## Resumo

As árvores de expressão que representam expressões lambda podem ser compiladas para criar um delegado que pode ser executado. As árvores de expressão fornecem um mecanismo para executar o código representado por uma árvore de expressão.

A árvore de expressão representa o código que seria executado para qualquer constructo específico que você criar. Contanto que o ambiente em que você compilar e executar o código corresponda ao ambiente em que você criar a expressão, tudo funcionará conforme o esperado. Quando isso não acontece, os erros são previsíveis e capturados nos primeiros testes de qualquer código que usam as árvores de expressão.

# Interpretar expressões

Artigo • 16/03/2023

O exemplo de código a seguir demonstra como a árvore de expressão que representa a expressão lambda `num => num < 5` pode ser decomposta em suas partes.

C#

```
// Add the following using directive to your code file:  
// using System.Linq.Expressions;  
  
// Create an expression tree.  
Expression<Func<int, bool>> exprTree = num => num < 5;  
  
// Decompose the expression tree.  
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];  
BinaryExpression operation = (BinaryExpression)exprTree.Body;  
ParameterExpression left = (ParameterExpression)operation.Left;  
ConstantExpression right = (ConstantExpression)operation.Right;  
  
Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",  
    param.Name, left.Name, operation.NodeType, right.Value);  
  
// This code produces the following output:  
  
// Decomposed expression: num => num LessThan 5
```

Agora, vamos escrever código para examinar a estrutura de um *árvore de expressão*. Cada nó em uma árvore de expressão é um objeto de uma classe derivada de `Expression`.

Esse design faz com que visitar todos os nós em uma árvore de expressão seja uma operação recursiva relativamente simples. A estratégia geral é iniciar no nó raiz e determine que tipo de nó ele é.

Se o tipo de nó tiver filhos, visite os filhos recursivamente. Em cada nó filho, repita o processo usado no nó raiz: determine o tipo e, se o tipo tiver filhos, visite cada um dos filhos.

## Examinar uma expressão sem filhos

Vamos começar visitando cada nó em uma árvore de expressão simples. Este é o código que cria uma expressão constante e, em seguida, examina suas propriedades:

C#

```
var constant = Expression.Constant(24, typeof(int));  
  
Console.WriteLine($"This is a/an {constant.NodeType} expression type");  
Console.WriteLine($"The type of the constant value is {constant.Type}");  
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

O código anterior gera a seguinte saída:

Saída

```
This is a/an Constant expression type  
The type of the constant value is System.Int32  
The value of the constant value is 24
```

Agora, vamos escrever o código que examinaria essa expressão e escrever algumas propriedades importantes sobre ele.

## Expressão de adição

Vamos começar com o exemplo de adição da introdução desta seção.

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

### ① Observação

Não use `var` para declarar essa árvore de expressão, pois o tipo natural do delegado é `Func<int>`, e não `Expression<Func<int>>`.

O nó raiz é um `LambdaExpression`. Para obter o código interessante no lado direito do operador `=>`, você precisa encontrar um dos filhos de `LambdaExpression`. Você fará isso com todas as expressões nesta seção. O nó pai nos ajudar a localizar o tipo de retorno do `LambdaExpression`.

Para examinar cada nó nesta expressão, você precisa visitar de maneira recorrente muitos nós. Esta é uma primeira implementação simples:

C#

```
Expression<Func<int, int, int>> addition = (a, b) => a + b;
```

```

Console.WriteLine($"This expression is a {addition.NodeType} expression
type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "
<null>" : addition.Name)}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count}
arguments. They are:");
foreach (var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"{"\tParameter Type:
{argumentExpression.Type.ToString()}, Name: {argumentExpression.Name}"});
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType}
expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"{"\tParameter Type: {left.Type.ToString()}, Name:
{left.Name}"});
Console.WriteLine($"The right side is a {additionBody.Right.NodeType}
expression");
var right = (ParameterExpression)additionBody.Right;
Console.WriteLine($"{"\tParameter Type: {right.Type.ToString()}, Name:
{right.Name}"});

```

Este exemplo imprime a seguinte saída:

Saída

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
    Parameter Type: System.Int32, Name: a
    Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
    Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
    Parameter Type: System.Int32, Name: b

```

É possível perceber muita repetição no exemplo de código anterior. Vamos limpar tudo isso e criar um visitante de nós de expressão com uma finalidade mais geral. Para isso, precisaremos escrever um algoritmo recursivo. Qualquer nó poderia ser de um tipo que pode ter filhos. Qualquer nó que tem filhos exige que nós visitemos esses filhos e determinemos o que é esse nó. Esta é a versão limpa que utiliza a recursão para visitar as operações de adição:

C#

```

using System.Linq.Expressions;

namespace Visitors;
// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node) => this.node = node;

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => node.NodeType;
    public static Visitor CreateFromExpression(Expression node) =>
        node.NodeType switch
        {
            ExpressionType.Constant => new
ConstantVisitor((ConstantExpression)node),
            ExpressionType.Lambda => new
LambdaVisitor((LambdaExpression)node),
            ExpressionType.Parameter => new
ParameterVisitor((ParameterExpression)node),
            ExpressionType.Add => new BinaryVisitor((BinaryExpression)node),
            _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
        };
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node) => this.node =
node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {{{(node.Name
== null) ? "<null>" : node.Name)}}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType}");
        Console.WriteLine($"{prefix}The expression has
{node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
    }
}

```

```

        }

    }

    // Binary Expression Visitor:
    public class BinaryVisitor : Visitor
    {
        private readonly BinaryExpression node;
        public BinaryVisitor(BinaryExpression node) : base(node) => this.node =
node;

        public override void Visit(string prefix)
        {
            Console.WriteLine($"{prefix}This binary expression is a {NodeType}
expression");
            var left = CreateFromExpression(node.Left);
            Console.WriteLine($"{prefix}The Left argument is:");
            left.Visit(prefix + "\t");
            var right = CreateFromExpression(node.Right);
            Console.WriteLine($"{prefix}The Right argument is:");
            right.Visit(prefix + "\t");
        }
    }

    // Parameter visitor:
    public class ParameterVisitor : Visitor
    {
        private readonly ParameterExpression node;
        public ParameterVisitor(ParameterExpression node) : base(node)
        {
            this.node = node;
        }

        public override void Visit(string prefix)
        {
            Console.WriteLine($"{prefix}This is an {NodeType} expression type");
            Console.WriteLine($"{prefix}Type: {node.Type}, Name: {node.Name},
ByRef: {node.IsByRef}");
        }
    }

    // Constant visitor:
    public class ConstantVisitor : Visitor
    {
        private readonly ConstantExpression node;
        public ConstantVisitor(ConstantExpression node) : base(node) =>
this.node = node;

        public override void Visit(string prefix)
        {
            Console.WriteLine($"{prefix}This is an {NodeType} expression type");
            Console.WriteLine($"{prefix}The type of the constant value is
{node.Type}");
            Console.WriteLine($"{prefix}The value of the constant value is
{node.Value}");
        }
    }
}

```

```
}
```

Esse algoritmo é a base de um algoritmo que visita qualquer `LambdaExpression` arbitrário. O código criado procura apenas uma pequena amostra dos possíveis conjuntos de nós de árvore de expressão que ele pode encontrar. No entanto, ainda é possível aprender bastante com o que ele produz. (O caso padrão no método `Visitor.CreateFromExpression` imprime uma mensagem no console de erro quando um novo tipo de nó é encontrado. Dessa forma, você sabe adicionar um novo tipo de expressão.)

Ao executar esse visitante na expressão de adição anterior, você obtém a seguinte saída:

Saída

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: a, ByRef: False
    This is an Parameter expression type
    Type: System.Int32, Name: b, ByRef: False
The expression body is:
    This binary expression is a Add expression
    The Left argument is:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
    The Right argument is:
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
```

Agora que criou uma implementação de visitante mais geral, você pode visitar e processar muitos tipos diferentes de expressões.

## Expressão de adição com mais operandos

Vamos testar um exemplo mais complicado, mas ainda limitar os tipos de nó somente à adição:

C#

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Antes de executar esses exemplos no algoritmo de visitante, tente pensar qual seria a saída. Lembre-se de que o operador `+` é um *operador binário*: ele deve ter dois filhos, que representam os operandos esquerdo e direito. Há várias maneiras possíveis de construir uma árvore que podem ser corretas:

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;
```

Você pode ver a separação em duas possíveis respostas para realçar a mais promissora. A primeira representa expressões *associativas à direita*. A segunda representa expressões *associativas à esquerda*. A vantagem desses dois formatos é que o formato pode ser dimensionado para qualquer número arbitrário de expressões de adição.

Se você executar essa expressão por meio do visitante, verá essa saída, verificando se a expressão de adição simples é *associativa à esquerda*.

Para executar esse exemplo e ver a árvore de expressão completa, é preciso fazer uma alteração na árvore de expressão de origem. Quando a árvore de expressão contém todas as constantes, a árvore resultante contém apenas o valor constante de `10`. O compilador executa toda a adição e reduz a expressão a sua forma mais simples. Simplesmente adicionar uma variável à expressão é suficiente para ver a árvore original:

C#

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Crie um visitante para essa soma e execute o visitante. Você verá esta saída:

Saída

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: a, ByRef: False
The expression body is:
    This binary expression is a Add expression
    The Left argument is:
        This binary expression is a Add expression
```

```
The Left argument is:  
    This binary expression is a Add expression  
    The Left argument is:  
        This is an Constant expression type  
        The type of the constant value is  
System.Int32  
        The value of the constant value is 1  
    The Right argument is:  
        This is an Parameter expression type  
        Type: System.Int32, Name: a, ByRef: False  
    The Right argument is:  
        This is an Constant expression type  
        The type of the constant value is System.Int32  
        The value of the constant value is 3  
The Right argument is:  
    This is an Constant expression type  
    The type of the constant value is System.Int32  
    The value of the constant value is 4
```

Você pode executar qualquer um dos outros exemplos pelo código visitante e ver que árvore ele representa. Veja um exemplo da expressão `sum3` anterior (com um parâmetro adicional para impedir que o compilador calcule a constante):

C#

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Esta é a saída do visitante:

Saída

```
This expression is a/an Lambda expression type  
The name of the lambda is <null>  
The return type is System.Int32  
The expression has 2 argument(s). They are:  
    This is an Parameter expression type  
    Type: System.Int32, Name: a, ByRef: False  
    This is an Parameter expression type  
    Type: System.Int32, Name: b, ByRef: False  
The expression body is:  
    This binary expression is a Add expression  
    The Left argument is:  
        This binary expression is a Add expression  
        The Left argument is:  
            This is an Constant expression type  
            The type of the constant value is System.Int32  
            The value of the constant value is 1  
        The Right argument is:  
            This is an Parameter expression type  
            Type: System.Int32, Name: a, ByRef: False  
    The Right argument is:
```

```
This binary expression is a Add expression
The Left argument is:
    This is an Constant expression type
    The type of the constant value is System.Int32
    The value of the constant value is 3
The Right argument is:
    This is an Parameter expression type
    Type: System.Int32, Name: b, ByRef: False
```

Observe que os parênteses não fazem parte da saída. Não há nenhum nó na árvore de expressão que representa os parênteses na expressão de entrada. A estrutura de árvore de expressão contém todas as informações necessárias para comunicar a precedência.

## Estendendo este exemplo

O exemplo lida apenas com as árvores de expressão mais rudimentares. O código que você viu nesta seção só lida com inteiros constantes e com o operador `+` binário. Como um exemplo final, vamos atualizar o visitante para lidar com uma expressão mais complicada. Vamos fazer com que funcione para a seguinte expressão fatorial:

C#

```
Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);
```

Este código representa uma possível implementação da função *fatorial* matemática. A maneira como você escreve o código destaca duas limitações da criação de árvores de expressão atribuindo expressões lambda a Expressões. Primeiro, lambdas de instrução não são permitidos. Isso significa que eu não posso usar loops, blocos, instruções if/else nem outras estruturas de controle comuns em C#. Você só pode usar expressões. Em segundo lugar, você não pode chamar a mesma expressão de maneira recorrente. Você poderia se ela já fosse um delegado, mas não pode chamá-la em sua forma de árvore de expressão. Na seção [criando árvores de expressão](#), você aprenderá técnicas para superar essas limitações.

Nesta expressão, você encontrará todos estes tipos de nós:

1. Igual (expressão binária)
2. Multiplicar (expressão binária)
3. Condisional (a expressão `? :`)
4. Expressão de chamada de método (chamar `Range()` e `Aggregate()`)

Uma maneira de modificar o algoritmo do visitante é continuar executando-o e escrever o tipo de nó toda vez que você atingir sua cláusula `default`. Após algumas iterações, você verá cada um dos possíveis nós. Então, você tem tudo de que você precisa. O resultado seria algo semelhante a:

C#

```
public static Visitor CreateFromExpression(Expression node) =>
    node.NodeType switch
    {
        ExpressionType.Constant      => new
        ConstantVisitor((ConstantExpression)node),
        ExpressionType.Lambda        => new
        LambdaVisitor((LambdaExpression)node),
        ExpressionType.Parameter     => new
        ParameterVisitor((ParameterExpression)node),
        ExpressionType.Add           => new
        BinaryVisitor((BinaryExpression)node),
        ExpressionType.Equal         => new
        BinaryVisitor((BinaryExpression)node),
        ExpressionType.Multiply      => new BinaryVisitor((BinaryExpression)
node),
        ExpressionType.Conditional   => new
        ConditionalVisitor((ConditionalExpression) node),
        ExpressionType.Call          => new
        MethodCallVisitor((MethodCallExpression) node),
        _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
    };
}
```

O `ConditionalVisitor` e o `MethodCallVisitor` processa esses dois nós:

C#

```
public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression
is {trueVisitor.Visit(prefix)}");
    }
}
```

```

        is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression
is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");

            var MethodInfo = node.Method;
            Console.WriteLine($"{prefix}The method name is
{MethodInfo.DeclaringType}.{MethodInfo.Name}");
            // There is more here, like generic arguments, and so on.
            Console.WriteLine($"{prefix}The Arguments are:");
            foreach (var arg in node.Arguments)
            {
                var argVisitor = Visitor.CreateFromExpression(arg);
                argVisitor.Visit(prefix + "\t");
            }
        }
    }
}

```

E a saída da árvore de expressão seria:

Saída

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: n, ByRef: False

```

The expression body is:  
This expression is a Conditional expression  
The Test for this expression is:  
    This binary expression is a Equal expression  
The Left argument is:  
    This is an Parameter expression type  
    Type: System.Int32, Name: n, ByRef: False  
The Right argument is:  
    This is an Constant expression type  
    The type of the constant value is System.Int32  
    The value of the constant value is 0  
The True clause for this expression is:  
    This is an Constant expression type  
    The type of the constant value is System.Int32  
    The value of the constant value is 1  
The False clause for this expression is:  
    This expression is a Call expression  
    This is a static method call  
    The method name is System.Linq.Enumerable.Aggregate  
The Arguments are:  
    This expression is a Call expression  
    This is a static method call  
    The method name is System.Linq.Enumerable.Range  
The Arguments are:  
    This is an Constant expression type  
    The type of the constant value is  
  
System.Int32  
    The value of the constant value is 1  
    This is an Parameter expression type  
    Type: System.Int32, Name: n, ByRef: False  
This expression is a Lambda expression type  
The name of the lambda is <null>  
The return type is System.Int32  
The expression has 2 arguments. They are:  
    This is an Parameter expression type  
    Type: System.Int32, Name: product, ByRef:  
  
False  
    This is an Parameter expression type  
    Type: System.Int32, Name: factor, ByRef:  
  
False  
The expression body is:  
    This binary expression is a Multiply  
expression  
The Left argument is:  
    This is an Parameter expression type  
    Type: System.Int32, Name: product,  
ByRef: False  
The Right argument is:  
    This is an Parameter expression type  
    Type: System.Int32, Name: factor,  
ByRef: False

# Estender a biblioteca de exemplos

Os exemplos nesta seção mostram as principais técnicas para visitar e examinar nós em uma árvore de expressão. Ela simplifica os tipos de nós que você encontrará para se concentrar nas tarefas principais de visitar e acessar nós em uma árvore de expressão.

Primeiro, os visitantes lidam somente com constantes que são números inteiros. Os valores das constantes podem ser de qualquer outro tipo numérico e a linguagem C# dá suporte a conversões e promoções entre esses tipos. Uma versão mais robusta desse código espelharia todos esses recursos.

Até o último exemplo reconhece um subconjunto dos tipos de nó possíveis. Você ainda pode alimentá-lo com muitas expressões que o fariam falhar. Uma implementação completa está incluída no .NET Standard com o nome [ExpressionVisitor](#) e pode lidar com todos os tipos de nó possíveis.

Por fim, a biblioteca usada neste artigo foi desenvolvida para demonstração e aprendizado. Ela não está otimizada. Ela deixa as estruturas claras e realça as técnicas usadas para visitar os nós e analisar o que está neles.

Mesmo com essas limitações, você deve estar bem no processo de escrever algoritmos que leem e entendem árvores de expressão.

# Criar árvores de expressão

Artigo • 16/03/2023

O compilador C# criou todas as árvores de expressão que você viu até agora. Você criou uma expressão lambda atribuída a uma variável digitada como um `Expression<Func<T>>` ou algum tipo semelhante. Para muitos cenários, você cria uma expressão na memória no tempo de execução.

As árvores de expressão são imutáveis. Ser imutável significa que você precisa criar a árvore de folhas até a raiz. As APIs que você usa para criar as árvores de expressão refletem esse fato: os métodos que você usa para criar um nó usam todos os filhos como argumentos. Vejamos alguns exemplos para mostrar as técnicas a você.

## Criar nós

Comece com a expressão de adição com a qual você tem trabalhado nestas seções:

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

Para construir essa árvore de expressão, você precisará criar os nós folha. Os nós folha são constantes. Use o método `Constant` para criar os nós:

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

Em seguida, crie a expressão de adição:

C#

```
var addition = Expression.Add(one, two);
```

Depois de criar a expressão de adição, crie a expressão lambda:

C#

```
var lambda = Expression.Lambda(addition);
```

Essa expressão lambda não contém argumentos. Mais adiante nesta seção, você verá como mapear argumentos para parâmetros e criar expressões mais complicadas.

Para expressões como essa, você pode combinar todas as chamadas em uma só instrução:

```
C#  
  
var lambda2 = Expression.Lambda(  
    Expression.Add(  
        Expression.Constant(1, typeof(int)),  
        Expression.Constant(2, typeof(int))  
    )  
);
```

## Criar uma árvore

A seção anterior mostrou os conceitos básicos da criação de uma árvore de expressão na memória. Árvores mais complexas geralmente significam mais tipos de nó e mais nós na árvore. Vamos percorrer mais um exemplo e mostrar dois outros tipos de nó que você normalmente criará quando criar árvores de expressão: os nós de argumento e os nós de chamada de método. Vamos criar uma árvore de expressão para criar esta expressão:

```
C#  
  
Expression<Func<double, double, double>> distanceCalc =  
    (x, y) => Math.Sqrt(x * x + y * y);
```

Você começará criando expressões de parâmetro para `x` e `y`:

```
C#  
  
var xParameter = Expression.Parameter(typeof(double), "x");  
var yParameter = Expression.Parameter(typeof(double), "y");
```

A criação de expressões de multiplicação e adição segue o padrão que você já viu:

```
C#  
  
var xSquared = Expression.Multiply(xParameter, xParameter);  
var ySquared = Expression.Multiply(yParameter, yParameter);  
var sum = Expression.Add(xSquared, ySquared);
```

Em seguida, você precisa criar uma expressão de chamada de método para a chamada para `Math.Sqrt`.

C#

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) }) ??
    throw new InvalidOperationException("Math.Sqrt not found!");
var distance = Expression.Call(sqrtMethod, sum);
```

A chamada `GetMethod` poderá retornar `null` se o método não for encontrado.

Provavelmente, isso ocorre porque você digitou incorretamente o nome do método. Caso contrário, isso pode significar que o assembly necessário não está carregado. Por fim, você colocará a chamada de método em uma expressão lambda e definirá os argumentos para a expressão lambda:

C#

```
var distanceLambda = Expression.Lambda(
    distance,
    xParameter,
    yParameter);
```

Neste exemplo mais complicado, é possível ver mais algumas técnicas que frequentemente serão necessárias para criar árvores de expressão.

Primeiro, você precisa criar os objetos que representam parâmetros ou variáveis locais antes de usá-los. Após ter criado esses objetos, você pode usá-los em sua árvore de expressão quando for necessário.

Depois, você precisa usar um subconjunto das APIs de reflexão para criar um objeto `System.Reflection.MethodInfo` para que possa criar uma árvore de expressão para acessar esse método. Você deve se limitar ao subconjunto das APIs de reflexão que estão disponíveis na plataforma do .NET Core. Mais uma vez, essas técnicas se estenderão a outras árvores de expressão.

## Criar código em profundidade

Você não fica limitado ao que pode criar usando essas APIs. No entanto, quanto mais complicada for a árvore de expressão que você quer criar, mais difícil será gerenciar e ler o código.

Vamos criar uma árvore de expressão que é o equivalente a este código:

C#

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

O código anterior não compilou a árvore de expressão, mas simplesmente o delegado. Usando a classe `Expression`, não é possível criar lambdas de instrução. Este é o código que é necessário para criar a mesma funcionalidade. Não há uma API para criar um loop `while`, em vez disso, você precisa criar um loop que contenha um teste condicional e um destino de rótulo para sair do loop.

C#

```
var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
            block,
            Expression.Break(label, result)
        ),
        label
    )
);
```

O código para criar a árvore de expressão para a função factorial é bem mais longo, mais complicado e está cheio de rótulos e instruções de interrupção, bem como outros elementos que você gostaria de evitar em nossas tarefas cotidianas de codificação.

Para esta seção, você gravou código para visitar todos os nós nesta árvore de expressão e gravar informações sobre os nós criados neste exemplo. Você pode [exibir ou baixar o código de exemplo](#) no repositório dotnet/docs do GitHub. Experimente por conta própria criando e executando os exemplos.

## Mapear constructos de código para expressões

O exemplo de código a seguir demonstra uma árvore de expressão que representa a expressão lambda `num => num < 5` usando a API.

C#

```
// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

A API de árvores de expressão também dá suporte a atribuições e expressões de fluxo de controle, como loops, blocos condicionais e blocos `try-catch`. Usando a API, você pode criar árvores de expressão mais complexas do que aquelas que podem ser criadas por meio de expressões lambda pelos compiladores do C#. O exemplo a seguir demonstra como criar uma árvore de expressão que calcula o fatorial de um número.

C#

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
```

```
// Assigning a constant to a local variable: result = 1
Expression.Assign(result, Expression.Constant(1)),
    // Adding a loop.
    Expression.Loop(
        // Adding a conditional block into the loop.
        Expression.IfThenElse(
            // Condition: value > 1
            Expression.GreaterThan(value, Expression.Constant(1)),
            // If true: result *= value --
            Expression.MultiplyAssign(result,
                Expression.PostDecrementAssign(value)),
            // If false, exit the loop and go to the label.
            Expression.Break(label, result)
        ),
        // Label to jump to.
        label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()
(5);

Console.WriteLine(factorial);
// Prints 120.
```

Para saber mais, confira [Gerar métodos dinâmicos com árvores de expressão no Visual Studio 2010](#), que também se aplica a versões mais recentes do Visual Studio.

# Mover árvores de expressão

Artigo • 16/03/2023

Neste artigo, você aprenderá a visitar cada nó em uma árvore de expressão enquanto estiver criando uma cópia modificada dessa árvore de expressão. Você converterá árvores de expressão para entender os algoritmos de modo que eles possam ser convertidos em outro ambiente. Você alterará o algoritmo criado. Você poderá adicionar registro em log, interceptar chamadas de método e monitorá-las ou realizar outras ações.

O código que você compila para mover uma árvore de expressão é uma extensão do que você já viu para visitar todos os nós em uma árvore. Quando você move uma árvore de expressão, visita todos os nós e ao visitá-los, cria a nova árvore. A nova árvore pode conter referências aos nós originais ou aos novos nós que você inseriu na árvore.

Vamos acessar uma árvore de expressão e criar uma árvore com alguns nós de substituição. Neste exemplo, substituiremos qualquer constante por uma constante dez vezes maior. Caso contrário, deixe a árvore de expressão intacta. Em vez de ler o valor da constante e substituí-la por uma nova constante, você fará essa substituição trocando o nó de constante por um novo nó que executa a multiplicação.

Aqui, quando encontrar um nó de constante, você criará um nó de multiplicação cujos filhos serão a constante original e a constante `10`:

C#

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

Crie uma árvore substituindo o nó original pelo substituto. Você verificará as alterações compilando e executando a árvore substituída.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

A criação de uma nova árvore é uma combinação da visita aos nós da árvore existentes e a criação de novos nós, inserindo-os na árvore. O anterior exemplo mostra a importância de as árvores de expressão serem imutáveis. Observe que a árvore criada no código anterior contém uma mistura de nós recém-criados e nós da árvore existente. Os nós podem ser usados em ambas as árvores porque os nós na árvore existente não podem ser modificados. Reutilizar nós resulta em eficiências significativas de memória. Os mesmos nós podem ser usados em toda a árvore ou em várias árvores de expressão. Como os nós não podem ser modificados, o mesmo nó pode ser reutilizado sempre que necessário.

## Percorrer e executar uma adição

Vamos verificar a nova árvore criando um segundo visitante que percorre a árvore de nós de adição e calcula o resultado. Faça algumas modificações no visitante que você viu até agora. Nessa nova versão, o visitante retorna a soma parcial da operação de adição até este ponto. Para uma expressão de constante, esse é simplesmente o valor da expressão de constante. Para uma expressão de adição, o resultado será a soma dos operandos esquerdos e direitos, uma vez que essas árvores forem percorridas.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three = Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so you can call it
// from itself recursively:
Func<Expression, int> aggregate = null!;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
```

```

aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
        (int)((ConstantExpression)exp).Value :
        aggregate(((BinaryExpression)exp).Left) +
        aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);

```

Tem bastante código nisso, mas os conceitos são acessíveis. Esse código visita filhos em uma pesquisa de profundidade inicial. Ao encontrar um nó constante, o visitante retorna o valor da constante. Depois de visitar ambos os filhos, o visitante terá a soma calculada para essa subárvore. Agora o nó de adição poderá computar sua soma. Uma vez que todos os nós da árvore de expressão forem visitados, a soma é calculada. Você pode executar o exemplo no depurador e rastrear a execução.

Vamos facilitar o rastreamento de como os nós são analisados e como a soma é calculada, percorrendo a árvore. Esta é uma versão atualizada do método de agregação que inclui bastante informação de rastreamento:

C#

```

private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        if (constantExp.Value is int value)
        {
            return value;
        }
        else
        {
            return 0;
        }
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
}

```

```
        else throw new NotSupportedException("Haven't written this yet");
    }
```

Executá-lo na expressão `sum` produz o seguinte resultado:

Saída

```
10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

Rastreie a saída e acompanhe no código anterior. Você será capaz de entender como o código visita cada nó e calcula a soma, à medida que percorre a árvore e localiza a soma.

Agora, vejamos uma execução diferente, com a expressão fornecida por `sum1`:

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

Aqui está a saída ao examinar essa expressão:

Saída

```
Found Addition Expression
Computing Left node
Found Constant: 1
```

```
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

Embora a resposta final seja a mesma, a forma de percorrer a árvore é diferente. Os nós são percorridos em uma ordem diferente, porque a árvore foi construída com operações diferentes que ocorrem primeiro.

## Criar uma cópia modificada

Crie um novo projeto de **Aplicativo de Console**. Adicione uma diretiva `using` ao arquivo para o namespace `System.Linq.Expressions`. Adicione a classe `AndAlsoModifier` ao seu projeto.

C#

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an
            // AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right,

```

```

        b.IsLiftedToNull, b.Method);
    }

    return base.VisitBinary(b);
}
}

```

Essa classe herda a classe [ExpressionVisitor](#) e é especializada para modificar expressões que representam operações `AND` condicionais. Ela muda essas operações de uma `AND` condicional para uma `OR` condicional. A classe substitui o método [VisitBinary](#) do tipo base, pois as expressões condicionais `AND` são representadas como expressões binárias. No método [VisitBinary](#), se a expressão que é passada a ele representa uma operação `AND` condicional, o código cria uma nova expressão que contém o operador `OR` condicional em vez do operador `AND` condicional. Se a expressão passada para o [VisitBinary](#) não representa uma operação `AND` condicional, o método adia para a implementação da classe base. Os métodos da classe base constroem nós semelhantes às árvores de expressão passadas, mas as subárvores dos nós são substituídas pelas árvores de expressão produzidas de maneira recorrente pelo visitante.

Adicione uma diretiva `using` ao arquivo para o namespace `System.Linq.Expressions`. Adicione código ao método `Main` no arquivo `Program.cs` para criar uma árvore de expressão e passá-la ao método que a modifica.

C#

```

Expression<Func<string, bool>> expr = name => name.Length > 10 &&
name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression)expr);

Console.WriteLine(modifiedExpr);

/* This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/

```

O código cria uma expressão que contém uma operação `AND` condicional. Em seguida, ele cria uma instância da classe `AndAlsoModifier` e passa a expressão ao método `Modify` dessa classe. A árvore de expressão original e a modificada são geradas para mostrar a alteração. Compile e execute o aplicativo.

## Saiba mais

Este exemplo mostra um pequeno subconjunto do código que você compilaria para percorrer e interpretar os algoritmos representados por uma árvore de expressão. Para obter informações sobre como criar uma biblioteca de uso geral que converte árvores de expressão em outra linguagem, leia [esta série](#) de Matt Warren. Ele entra em detalhes de como mover qualquer código que você pode encontrar em uma árvore de expressão.

Espero que você tenha visto o verdadeiro potencial das árvores de expressão. Você examina um conjunto de códigos, faz as alterações que deseja nesse código e executa a versão modificada. Como as árvores de expressão são imutáveis, você cria árvores usando os componentes de árvores existentes. A reutilização de nós minimiza a quantidade de memória necessária para criar árvores de expressão modificadas.

# Expression Trees

Article • 03/09/2023

*Expression trees* represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

If you have used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you aren't familiar with LINQ, you probably want to read [the LINQ tutorial](#) and the article about [lambda expressions](#) before this one.) Expression Trees provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

You've likely already written code that uses Expression trees. Entity Framework's LINQ APIs accept Expression trees as the arguments for the LINQ Query Expression Pattern. That enables [Entity Framework](#) to translate the query you wrote in C# into SQL that executes in the database engine. Another example is [Moq](#) ↗, which is a popular mocking framework for .NET.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you examine, modify, or execute. These tools give you the power to manipulate code during run time. You write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you modify running algorithms and even translate C# expressions into another form for execution in another environment.

You compile and run code represented by expression trees. Building and running expression trees enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see [How to use expression trees to build dynamic queries](#).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see [Dynamic Language Runtime Overview](#).

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the `System.Linq.Expressions` namespace.

When a lambda expression is assigned to a variable of type `Expression<TDelegate>`, the compiler emits code to build an expression tree that represents the lambda expression.

The C# compiler generates expression trees only from expression lambdas (or single-line lambdas). It can't parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see [Lambda Expressions](#).

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

C#

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

You create expression trees in your code. You build the tree by creating each node and attaching the nodes into a tree structure. You learn how to create expressions in the article on [building expression trees](#).

Expression trees are immutable. If you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You use an expression tree visitor to traverse the existing expression tree. For more information, see the article on [translating expression trees](#).

Once you build an expression tree, you [execute the code represented by the expression tree](#).

## Limitations

There are some newer C# language elements that don't translate well into expression trees. Expression trees can't contain `await` expressions, or `async` lambda expressions. Many of the features added in C# 6 and later don't appear exactly as written in expression trees. Instead, newer features are exposed in expression trees in the equivalent, earlier syntax, where possible. Other constructs aren't available. It means that code that interprets expression trees works the same when new language features are introduced. However, even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It enables rich libraries such as Entity Framework to accomplish what they do.

Expression trees won't support new expression node types. It would be a breaking change for all libraries interpreting expression trees to introduce new node types. The following list includes most C# language elements that can't be used:

- Conditional methods that have been removed
- base access
- Method group expressions, including *address-of* (&) a method group, and anonymous method expressions
- References to local functions
- Statements, including assignment (=) and statement bodied expressions
- Partial methods with only a defining declaration
- Unsafe pointer operations
- dynamic operations
- Coalescing operators with null or default literal left side, null coalescing assignment, and the null propagating operator (?)
- Multi-dimensional array initializers, indexed properties, and dictionary initializers
- throw expressions
- Accessing static virtual or abstract interface members
- Lambda expressions that have attributes
- Interpolated strings
- UTF-8 string conversions or UTF-8 string literals
- Method invocations using variable arguments, named arguments or optional arguments
- Expressions using System.Index or System.Range, index "from end" (^) operator or range expressions (..)
- async lambda expressions or await expressions, including await foreach and await using
- Tuple literals, tuple conversions, tuple == or !=, or with expressions
- Discards (\_), deconstructing assignment, pattern matching is operator or the pattern matching switch expression
- COM call with ref omitted on the arguments
- ref, in or out parameters, ref return values, out arguments, or any values of ref struct type

# Interoperability Overview

Article • 02/25/2023

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is *managed code*, and code that runs outside the CLR is *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the [System.Runtime.InteropServices](#) namespace, C++ interoperability, and COM interoperability (COM interop).

## Platform Invoke

*Platform invoke* is a service that enables managed code to call unmanaged functions implemented in dynamic link libraries (DLLs), such as the Microsoft Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see [Consuming Unmanaged DLL Functions](#) and [How to use platform invoke to play a WAV file](#).

### ⓘ Note

The **Common Language Runtime** (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see [Security in .NET](#).

## C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class. C++ interop enables code authored in C# or another .NET language to access it. You write C++ code to wrap a native DLL or COM component. Unlike other .NET languages, Visual C++ has interoperability support that enables managed and unmanaged code in the same application and even in the same file. You then build the C++ code by using the `/clr` compiler switch to produce a managed assembly. Finally, you add a reference to

the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

## Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use `regsvr32.exe` to register or un-register a COM DLL.
2. Add to the project a reference to the COM component or type library. When you add the reference, Visual Studio uses the [Tlbimp.exe \(Type Library Importer\)](#), which takes a type library as input, to output a .NET interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.
3. Create an instance of a class defined in the RCW. Creating an instance of that class creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see [Exposing COM Components to the .NET Framework](#).

## Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project. You can make an assembly COM visible by modifying C# project properties. For more information, see [Assembly Information Dialog Box](#).
2. Generate a COM type library and register it for COM usage. You can modify C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the [Regasm.exe \(Assembly Registration Tool\)](#), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see [Exposing .NET Framework Components to COM](#) and [Example COM Class](#).

## See also

- [Improving Interop Performance](#)
- [Introduction to Interoperability between COM and .NET](#)
- [Introduction to COM Interop in Visual Basic](#)
- [Marshaling between Managed and Unmanaged Code](#)
- [Interoperating with Unmanaged Code](#)

# Controle de versão em C#

Artigo • 09/05/2023

Neste tutorial, você aprenderá o que significa o controle de versão no .NET. Você também aprenderá sobre os fatores a serem considerados ao fazer o controle de versão de sua biblioteca, bem como ao atualizar para uma nova versão de uma biblioteca.

## Criando bibliotecas

Como um desenvolvedor que criou a bibliotecas .NET para uso público, provavelmente você esteve em situações em que precisa distribuir novas atualizações. Como você realiza esse processo é muito importante, pois você precisa garantir que haja uma transição suave do código existente para a nova versão da biblioteca. Aqui estão Vários aspectos a considerar ao criar uma nova versão:

### Controle de Versão Semântico

[Controle de versão semântico ↗](#) (SemVer, de forma abreviada) é uma convenção de nomenclatura aplicada a versões de sua biblioteca para indicar eventos com marcos específicos. Idealmente, as informações de versão que você fornece a sua biblioteca devem ajudar os desenvolvedores a determinar a compatibilidade com seus projetos que usam versões mais antigas da mesma biblioteca.

A abordagem mais básica ao SemVer é o formato de 3 componentes

MAJOR.MINOR.PATCH, em que:

- MAJOR é incrementado quando você faz alterações em APIs incompatíveis
- MINOR é incrementado quando você adiciona funcionalidades de maneira compatível com versões anteriores
- PATCH é incrementado quando você faz correções de bugs compatíveis com versões anteriores

Também há maneiras de especificar outros cenários, como versões de pré-lançamento, ao aplicar informações de versão à biblioteca do .NET.

### Compatibilidade com versões anteriores

Conforme você lança novas versões de sua biblioteca, a compatibilidade com versões anteriores provavelmente será uma de suas principais preocupações. Uma nova versão da biblioteca será compatível com a origem de uma versão anterior se o código que

depende da versão anterior puder, quando recompilado, trabalhar com a nova versão. Uma nova versão da biblioteca será compatível de forma binária se um aplicativo que dependia da versão anterior puder, sem recompilação, trabalhar com a nova versão.

Aqui estão algumas coisas a serem consideradas ao tentar manter a compatibilidade com versões mais antigas de sua biblioteca:

- Métodos virtuais: quando você torna um método em virtual não virtual na nova versão, significa que projetos que substituem esse método precisarão ser atualizados. Essa é uma alteração muito grande e significativa que é altamente desaconselhável.
- Assinaturas de método: quando atualizar o comportamento de um método exigir que você altere também sua assinatura, você deve criar uma sobrecarga para que o código que chamar esse método ainda funcione. Você sempre pode manipular a assinatura de método antiga para chamar a nova assinatura de método para que a implementação permaneça consistente.
- **Atributo obsoleto**: você pode usar esse atributo no seu código para especificar classes ou membros da classe que foram preteridos e provavelmente serão removidos em versões futuras. Isso garante que os desenvolvedores que utilizam sua biblioteca estarão melhor preparados para alterações significativas.
- Argumentos de método opcionais: quando você tornar argumentos de método que antes eram opcionais em compulsórios ou alterar seu valor padrão, todo código que não fornece esses argumentos precisará ser atualizado.

#### ① Observação

Tornar argumentos compulsórios em opcionais deve ter muito pouco efeito, especialmente se não alterar o comportamento do método.

Quanto mais fácil for para os usuários atualizarem para a nova versão da sua biblioteca, mais provável será que eles atualizem o quanto antes.

## Arquivo de Configuração do Aplicativo

Como um desenvolvedor de .NET, há uma chance muito grande de você já ter encontrado o [arquivo o app.config](#) na maioria dos tipos de projeto. Esse arquivo de configuração simples pode fazer muita diferença para melhorar a distribuição de novas atualizações. Em geral, você deve projetar suas bibliotecas de forma que as informações que provavelmente serão alteradas regularmente sejam armazenadas no arquivo `app.config`. Dessa forma, quando essas informações forem atualizadas, o arquivo de

configuração de versões mais antigas só precisa ser substituído pelo novo, sem a necessidade de recompilar a biblioteca.

## Consumindo bibliotecas

Como um desenvolvedor que consome bibliotecas .NET criadas por outros desenvolvedores, vocês provavelmente está ciente de que uma nova versão de uma biblioteca pode não ser totalmente compatível com seu projeto e pode acabar precisando atualizar seu código para trabalhar com essas alterações.

Para sua sorte, o ecossistema do C# e do .NET tem recursos e técnicas que permitem facilmente atualizar nosso aplicativo para trabalhar com novas versões das bibliotecas que podem introduzir alterações interruptivas.

### Redirecionamento de associação de assembly

Você pode usar o arquivo `app.config` para atualizar a versão de uma biblioteca que seu aplicativo usa. Adicionando o que é chamado de *redirecionamento de associação*, você pode usar a nova versão da biblioteca sem precisar recompilar seu aplicativo. O exemplo a seguir mostra como atualizar o arquivo `app.config` de seu aplicativo para uso com a versão de patch `1.0.1` de `ReferencedLibrary` em vez da versão `1.0.0` com que ele foi compilado originalmente.

XML

```
<dependentAssembly>
    <assemblyIdentity name="ReferencedLibrary"
publicToken="32ab4ba45e0a69a1" culture="en-us" />
    <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

#### ⓘ Observação

Essa abordagem só funcionará se a nova versão do `ReferencedLibrary` for compatível de forma binária com seu aplicativo. Consulte a seção **Compatibilidade com versões anteriores** acima para ver as alterações importantes ao determinar a compatibilidade.

novo

Você usa o modificador `new` para ocultar membros herdados de uma classe base. Essa é uma maneira das classes derivadas responderem a atualizações em classes base.

Veja o exemplo seguinte:

```
C#  
  
public class BaseClass  
{  
    public void MyMethod()  
    {  
        Console.WriteLine("A base method");  
    }  
}  
  
public class DerivedClass : BaseClass  
{  
    public new void MyMethod()  
    {  
        Console.WriteLine("A derived method");  
    }  
}  
  
public static void Main()  
{  
    BaseClass b = new BaseClass();  
    DerivedClass d = new DerivedClass();  
  
    b.MyMethod();  
    d.MyMethod();  
}
```

## Saída

```
Console  
  
A base method  
A derived method
```

No exemplo acima, você pode ver como `DerivedClass` oculta o método `MyMethod` presente em `BaseClass`. Isso significa que quando uma classe base na nova versão de uma biblioteca adiciona um membro que já existe em sua classe derivada, você pode simplesmente usar o modificador `new` no membro de sua classe derivada para ocultar o membro da classe base.

Quando nenhum modificador `new` é especificado, uma classe derivada ocultará por padrão membros conflitantes em uma classe base e, embora um aviso do compilador seja gerado, o código ainda será compilado. Isso significa que simplesmente adicionar

novos membros a uma classe existente torna a nova versão da biblioteca compatível com a origem e de forma binária com o código que depende dela.

## override

O modificador `override` significa que uma implementação derivada estende a implementação de um membro da classe base, em vez de ocultá-lo. O membro da classe base precisa ter o modificador `virtual` aplicado a ele.

C#

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine("Base Method One: {0}", b.MethodOne());
    Console.WriteLine("Derived Method One: {0}", d.MethodOne());
}
```

## Saída

Console

```
Base Method One: Method One
Derived Method One: Derived Method One
```

O modificador `override` é avaliado em tempo de compilação e o compilador gerará um erro se não encontrar um membro virtual para substituir.

Seu conhecimento sobre as técnicas discutidas e sua compreensão das situações em que usá-las, farão muita diferença para facilitar a transição entre versões de uma

biblioteca.

# Instruções (C#)

Artigo • 15/02/2023

Na seção de Instruções do Guia de C#, é possível encontrar respostas rápidas para perguntas comuns. Em alguns casos, os artigos podem ser listados em várias seções. Queremos facilitar que sejam localizados por vários caminhos de pesquisa.

## Conceitos gerais de C#

Há dicas e truques que são práticas comuns do desenvolvedor de C#:

- [Inicializar objetos usando um inicializador de objeto.](#)
- [Aprenda as diferenças entre passar um struct e uma classe para um método.](#)
- [Use a sobrecarga de operador.](#)
- [Implemente e chame um método de extensão personalizado.](#)
- [Crie um novo método para um tipo enum usando métodos de extensão.](#)

## Membros de classe, registro e struct

Crie classes, registros e structs para implementar seu programa. Essas técnicas são comumente usadas durante a gravação de classes, registros ou structs.

- [Declare propriedades implementadas automaticamente.](#)
- [Declare e use propriedades de leitura/gravação.](#)
- [Defina constantes.](#)
- [Substitua o método ToString para fornecer saída de cadeia de caracteres.](#)
- [Defina propriedades abstract.](#)
- [Use os recursos de documentação para documentar seu código.](#)
- [Implemente membros de interface explicitamente para manter a interface pública concisa.](#)
- [Implemente membros de duas interfaces explicitamente.](#)

## Trabalhando com coleções

Esses artigos ajudam você a trabalhar com coleções de dados.

- [Inicialize um dicionário com um inicializador de coleção.](#)

## Trabalhando com cadeias de caracteres

As cadeias de caracteres são o tipo de dados fundamental usado para exibir ou manipular texto. Esses artigos demonstram práticas comuns com cadeias de caracteres.

- Compare cadeias de caracteres.
- Modifique o conteúdo da cadeia de caracteres.
- Determine se uma cadeia de caracteres representa um número.
- Use `String.Split` para separar as cadeias de caracteres.
- Junte várias cadeias de caracteres em uma.
- Pesquise texto em uma cadeia de caracteres.

## Conversão entre tipos

Talvez seja necessário converter um objeto em um tipo diferente.

- Determine se uma cadeia de caracteres representa um número.
- Converta entre cadeias de caracteres que representam números hexadecimais e o número.
- Converta uma cadeia de caracteres para um `DateTime`.
- Converta uma matriz de bytes em um interno.
- Converta uma cadeia de caracteres em um número.
- Use a correspondência de padrões, os operadores `as` e `is` para converter para um tipo diferente com segurança.
- Define as conversões de tipo personalizado.
- Determine se um tipo é um tipo de valor anulável.
- Converta entre tipos de valor anuláveis e não anuláveis.

## Comparações de ordem e igualdade

É possível criar tipos que definem suas próprias regras de igualdade ou definem uma ordem natural entre os objetos desse tipo.

- Testar a igualdade com base em referência.
- Defina a igualdade com base em valor para um tipo.

## Tratamento de exceções

Programas .NET relatam que os métodos não concluíram seu trabalho com sucesso ao lançar exceções. Nesses artigos, você aprenderá a trabalhar com exceções.

- Trate exceções usando `try` e `catch`.
- Limpe recursos usando as `finally` cláusulas.

- Recupere com base em exceções não CLS (Common Language Specification).

## Representantes e eventos

Representantes e eventos fornecem uma capacidade para estratégias que envolve blocos de código acoplados livremente.

- Declare, crie uma instância e use delegados.
- Combine delegados multicast.

Os eventos fornecem um mecanismo para publicar ou assinar notificações.

- Assine e cancele a assinatura de eventos.
- Implemente eventos declarados nas interfaces.
- Esteja em conformidade com as diretrizes do .NET quando seu código publicar eventos.
- Gere eventos definidos nas classes de base de classes derivadas.
- Implemente acessadores de eventos personalizados.

## Práticas do LINQ

O LINQ permite que você grave códigos para consultar qualquer fonte de dados compatível com o padrão de expressão de consulta do LINQ. Esses artigos o ajudarão a entender o padrão e trabalhar com diferentes fontes de dados.

- Consulte uma coleção.
- Use var nas expressões de consulta.
- Retorne subconjuntos de propriedades de elementos em uma consulta.
- Grave consultas com filtragem complexa.
- Classifique os elementos de uma fonte de dados.
- Classifique os elementos em múltiplas chaves.
- Controle o tipo de uma projeção.
- Conte as ocorrências de um valor em uma sequência de origem.
- Calcule valores intermediários.
- Mescle dados de várias fontes.
- Encontre a diferença de conjunto entre duas sequências.
- Depure resultados de consultas vazios.
- Adicione métodos personalizados a consultas LINQ.

## Threads múltiplos e processamento assíncrono

Programas modernos geralmente usam operações assíncronas. Esses artigos o ajudarão a aprender a usar essas técnicas.

- Melhore o desempenho assíncrono usando [System.Threading.Tasks.Task.WhenAll](#).
- Faça várias solicitações da Web paralelamente usando `async` e `await`.
- Use um pool de thread.

## Argumentos da linha de comando para o programa

Geralmente, os programas de C# têm argumentos da linha de comando. Esses artigos o ensinam a acessar e processar esses argumentos da linha de comando.

- Recupere todos os argumentos da linha de comando com `for`.

# Como separar cadeias de caracteres usando String.Split em C#

Artigo • 17/08/2023

O método [String.Split](#) cria uma matriz de subcadeias, dividindo a cadeia de caracteres de entrada com base em um ou mais delimitadores. Esse método geralmente é a maneira mais fácil de separar uma cadeia de caracteres em limites de palavra. Ele também é usado para dividir cadeias de caracteres em outros caracteres específicos ou cadeias de caracteres.

## ⓘ Observação

Os exemplos de C# neste artigo são executados no executador de código embutido Try.NET  e no playground. Clique no botão **Executar** para executar um exemplo em uma janela interativa. Ao executar o código, é possível modificá-lo e executar o código modificado clicando em **Executar** novamente. O código modificado será executado na janela interativa ou, se a compilação falhar, a janela interativa exibirá todos as mensagens de erro do compilador C#.

O código a seguir divide uma frase comum em uma matriz de cadeias de caracteres para cada palavra.

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Cada instância de um caractere separador produz um valor na matriz retornada. Caracteres separadores consecutivos produzem a cadeia de caracteres vazia como um valor na matriz retornada. Você pode ver como uma cadeia de caracteres vazia é criada no exemplo a seguir, que usa o caractere de espaço como separador.

C#

```
string phrase = "The quick brown      fox      jumps over the lazy dog.";
string[] words = phrase.Split(' ');
```

```
foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Esse comportamento facilita para formatos como arquivos CSV (valores separados por vírgula) que representam dados de tabela. Vírgulas consecutivas representam uma coluna em branco.

Você pode passar um parâmetro `StringSplitOptions.RemoveEmptyEntries` opcional para excluir as cadeias de caracteres vazias da matriz retornada. Para um processamento mais complicado da coleção retornada, você pode usar o [LINQ](#) para manipular a sequência de resultado.

O `String.Split` pode usar vários caracteres separadores. O exemplo a seguir utiliza espaços, vírgulas, pontos, dois pontos e tabulações como caracteres de separação, que são passados para `Split` em uma matriz. O loop, na parte inferior do código, exibe cada uma das palavras na matriz retornada.

C#

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo three:four,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

As instâncias consecutivas de qualquer separador produzem a cadeia de caracteres vazia na matriz de saída:

C#

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo :,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
```

```
{  
    System.Console.WriteLine($"<{word}>");  
}
```

O [String.Split](#) pode receber uma matriz de cadeias de caracteres (sequências de caracteres que atuam como separadores para analisar a cadeia de caracteres de destino, em vez de um único caractere).

C#

```
string[] separatingStrings = { "<<", "..." };  
  
string text = "one<<two.....three<four";  
System.Console.WriteLine($"Original text: '{text}'");  
  
string[] words = text.Split(separatingStrings,  
System.StringSplitOptions.RemoveEmptyEntries);  
System.Console.WriteLine($"{words.Length} substrings in text:");  
  
foreach (var word in words)  
{  
    System.Console.WriteLine(word);  
}
```

## Confira também

- [Extrair elementos de uma cadeia de caracteres](#)
- [Guia de programação em C#](#)
- [Cadeias de caracteres](#)
- [Expressões regulares do .NET](#)

# Como concatenar várias cadeias de caracteres (Guia de C#)

Artigo • 10/05/2023

Concatenação é o processo de acrescentar uma cadeia de caracteres ao final de outra cadeia de caracteres. Você concatena cadeias de caracteres usando o operador `+`. Para literais de cadeia de caracteres e constantes de cadeia de caracteres, a concatenação ocorre em tempo de compilação; não ocorre nenhuma concatenação de tempo de execução. Para variáveis de cadeia de caracteres, a concatenação ocorre somente em tempo de execução.

## ⓘ Observação

Os exemplos de C# neste artigo são executados no executador de código embutido Try.NET ↗ e no playground. Clique no botão **Executar** para executar um exemplo em uma janela interativa. Ao executar o código, é possível modificá-lo e executar o código modificado clicando em **Executar** novamente. O código modificado será executado na janela interativa ou, se a compilação falhar, a janela interativa exibirá todos as mensagens de erro do compilador C#.

## Literais de cadeia de caracteres

O exemplo a seguir divide um literal de cadeia de caracteres longo em cadeias de caracteres menores para melhorar a legibilidade no código-fonte. O código concatena as cadeias de caracteres menores para criar o literal de cadeia de caracteres longa. As partes são concatenadas em uma única cadeia de caracteres em tempo de compilação. Não há custos de desempenho em tempo de execução, independentemente da quantidade de cadeias de caracteres envolvidas.

C#

```
// Concatenation of literals is performed at compile time, not run time.
string text = "Historically, the world of data and the world of objects " +
    "have not been well integrated. Programmers work in C# or Visual Basic " +
    "and also in SQL or XQuery. On the one side are concepts such as classes, "
    +
    "objects, fields, inheritance, and .NET Framework APIs. On the other side "
    +
    "are tables, columns, rows, nodes, and separate languages for dealing with "
    +
    "them. Data types often require translation between the two worlds; there
```

```
are " +
"different standard functions. Because the object world has no notion of
query, a " +
"query can only be represented as a string without compile-time type
checking or " +
"IntelliSense support in the IDE. Transferring data from SQL tables or XML
trees to " +
"objects in memory is often tedious and error-prone.";

System.Console.WriteLine(text);
```

## Operadores + e +=

Para concatenar variáveis de cadeia de caracteres, você pode usar os operadores `+` ou `+=`, a [interpolação de cadeia de caracteres](#) ou os métodos `String.Format`, `String.Concat`, `String.Join` ou `StringBuilder.Append`. O operador `+` é fácil de usar e torna o código intuitivo. Mesmo ao usar vários operadores `+` em uma instrução, o conteúdo da cadeia de caracteres será copiado apenas uma vez. O código a seguir mostra dois exemplos de como usar os operadores `+` e `+=` para concatenar cadeias de caracteres:

C#

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToString("dd/MM/yyyy");

// Use the + and += operators for one-time concatenations.
string str = "Hello " + userName + ". Today is " + dateString + ".";
System.Console.WriteLine(str);

str += " How are you today?";
System.Console.WriteLine(str);
```

## Interpolação de cadeia de caracteres

Em algumas expressões, é mais fácil concatenar cadeias de caracteres usando a interpolação de cadeia de caracteres, conforme mostra o seguinte código:

C#

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToString("dd/MM/yyyy");

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
```

```
str = $"{str} How are you today?";
System.Console.WriteLine(str);
```

### ① Observação

Em operações de concatenação de cadeia de caracteres, o compilador C# trata uma cadeia de caracteres nula da mesma maneira que uma cadeia de caracteres vazia.

A partir do C# 10, você pode usar a interpolação de cadeia de caracteres para inicializar uma cadeia de caracteres constante quando todas as expressões usadas para espaços reservados também são cadeias de caracteres constantes.

## String.Format

Outro método para concatenar cadeias de caracteres é o [String.Format](#). Esse método funciona bem quando você está criando uma cadeia de caracteres com base em um pequeno número de cadeias de caracteres de componente.

## StringBuilder

Em outros casos, você pode combinar cadeias de caracteres em um loop em que você não sabe quantas cadeias de caracteres de origem você está combinando, sendo que o número real de cadeias de caracteres de origem pode ser grande. A classe [StringBuilder](#) foi projetada para esses cenários. O código a seguir usa o método [Append](#) da classe [StringBuilder](#) para concatenar cadeias de caracteres.

C#

```
// Use StringBuilder for concatenation in tight loops.
var sb = new System.Text.StringBuilder();
for (int i = 0; i < 20; i++)
{
    sb.AppendLine(i.ToString());
}
System.Console.WriteLine(sb.ToString());
```

Você pode ler mais sobre os [motivos para escolher a concatenação de cadeia de caracteres ou a classe StringBuilder](#).

## String.Concat ou String.Join

Outra opção para unir cadeias de caracteres de uma coleção é usar o método [String.Concat](#). Use o método [String.Join](#) se desejar separar as cadeias de caracteres de origem por um delimitador. O código a seguir combina uma matriz de palavras usando os dois métodos:

C#

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the",
"lazy", "dog." };

var unreadablePhrase = string.Concat(words);
System.Console.WriteLine(unreadablePhrase);

var readablePhrase = string.Join(" ", words);
System.Console.WriteLine(readablePhrase);
```

## LINQ e [Enumerable.Aggregate](#)

Por fim, você pode usar [LINQ](#) e o método [Enumerable.Aggregate](#) para unir cadeias de caracteres de uma coleção. Esse método combina as cadeias de caracteres de origem usando uma expressão lambda. A expressão lambda faz o trabalho de adicionar cada cadeia de caracteres ao acúmulo existente. O exemplo a seguir combina uma matriz de palavras, adicionando um espaço entre cada palavra na matriz:

C#

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the",
"lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) => $"{partialPhrase}
{word}");
System.Console.WriteLine(phrase);
```

Essa opção pode causar mais alocações do que outros métodos para concatenar coleções, pois cria uma cadeia de caracteres intermediária para cada iteração. Se a otimização do desempenho for crítica, considere a classe [StringBuilder](#) ou o [String.Concat](#) ou o método [String.Join](#) para concatenar uma coleção, em vez de [Enumerable.Aggregate](#).

## Confira também

- [String](#)
- [StringBuilder](#)

- Guia de programação em C#
- Cadeias de caracteres

# Como pesquisar em cadeias de caracteres

Artigo • 10/05/2023

Você pode usar duas estratégias principais para pesquisar texto em cadeias de caracteres. Os métodos da classe [String](#) pesquisam por um texto específico. Expressões regulares pesquisam por padrões no texto.

## ⓘ Observação

Os exemplos de C# neste artigo são executados no executador de código embutido Try.NET ↗ e no playground. Clique no botão **Executar** para executar um exemplo em uma janela interativa. Ao executar o código, é possível modificá-lo e executar o código modificado clicando em **Executar** novamente. O código modificado será executado na janela interativa ou, se a compilação falhar, a janela interativa exibirá todos as mensagens de erro do compilador C#.

O tipo [string](#), que é um alias para a classe [System.String](#), fornece uma série de métodos úteis para pesquisar o conteúdo de uma cadeia de caracteres. Entre elas estão [Contains](#), [StartsWith](#), [EndsWith](#), [IndexOf](#) e [LastIndexOf](#). A classe [System.Text.RegularExpressions.Regex](#) fornece um vocabulário avançado para pesquisar por padrões de texto. Neste artigo, você aprenderá essas técnicas e como escolher o melhor método para suas necessidades.

## Uma cadeia de caracteres contém texto?

Os métodos [String.Contains](#), [String.StartsWith](#) e [String.EndsWith](#) pesquisam uma cadeia de caracteres em busca de um texto específico. O seguinte exemplo mostra cada um desses métodos e uma variação que usa uma pesquisa que não diferencia maiúsculas de minúsculas:

C#

```
string factMessage = "Extension methods have all the capabilities of regular
static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
```

```

Console.WriteLine($"Contains \"extension\"? {containsSearchResult}");

// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how
// to match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Starts with \"extension\"? {ignoreCaseSearchResult}
(ignoring case)");

bool endsWithSearchResult = factMessage.EndsWith(".", 
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");

```

O exemplo anterior demonstra um ponto importante para usar esses métodos. As pesquisas de texto diferenciam maiúsculas e minúsculas por padrão. Use o valor de enumeração `StringComparison.CurrentCultureIgnoreCase` para especificar uma pesquisa que não diferencia maiúsculas de minúsculas.

## Em que local de uma cadeia de caracteres o texto procurado ocorre?

Os métodos `IndexOf` e `LastIndexOf` também pesquisam texto em cadeias de caracteres. Esses métodos retornam o local do texto que está sendo procurado. Se o texto não for encontrado, elas retornarão `-1`. O exemplo a seguir mostra uma pesquisa para a primeira e a última ocorrência da palavra "métodos" e exibe o texto entre elas.

C#

```

string factMessage = "Extension methods have all the capabilities of regular
static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"Substring between \"methods\" and \"methods\":
'{str2}'");

```

## Localizar texto específico usando expressões regulares

A classe [System.Text.RegularExpressions.Regex](#) pode ser usada para pesquisar cadeias de caracteres. Essas pesquisas podem variar em complexidade, de padrões de texto simples até os complicados.

O exemplo de código a seguir procura a palavra "the" ou "their" em uma oração, sem diferenciar maiúsculas e minúsculas. O método estático [Regex.IsMatch](#) realiza a pesquisa. Você fornece a ele a cadeia de caracteres a pesquisar e um padrão de pesquisa. Nesse caso, um terceiro argumento especifica que a pesquisa não diferencia maiúsculas de minúsculas. Para obter mais informações, consulte [System.Text.RegularExpressions.RegexOptions](#).

O padrão de pesquisa descreve o texto pelo qual procurar. A tabela a seguir descreve cada elemento desse padrão de pesquisa. (A tabela abaixo usa a única \, que deve ser escapada como \\ em uma cadeia de caracteres C#).

Padrão	Significado
the	corresponder ao texto "the"
(eir)?	corresponder a 0 ou 1 ocorrência de "eir"
\s	corresponder a um caractere de espaço em branco

C#

```
string[] sentences =
{
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
};

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.WriteLine($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
        System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}
```

## Dica

Os métodos `string` são geralmente melhores opções quando você está procurando por uma cadeia de caracteres exata. Expressões regulares são melhores quando você está procurando por algum padrão em uma cadeia de caracteres de origem.

# Uma cadeia de caracteres segue um padrão?

O código a seguir usa expressões regulares para validar o formato de cada cadeia de caracteres em uma matriz. A validação requer que cada cadeia de caracteres tenha a forma de um número de telefone no qual os três grupos de dígitos são separados por traços, os dois primeiros grupos contêm três dígitos e o terceiro grupo contém quatro dígitos. O padrão de pesquisa usa a expressão regular `^\d{3}-\d{3}-\d{4}$`. Para obter mais informações, consulte [Linguagem de expressões regulares – referência rápida](#).

Padrão	Significado
<code>^</code>	corresponde ao início da cadeia de caracteres
<code>\d{3}</code>	corresponde a exatamente 3 caracteres de dígitos
<code>-</code>	corresponde ao caractere '-'
<code>\d{4}</code>	corresponde a exatamente 4 caracteres de dígitos
<code>\$</code>	corresponde ao final da cadeia de caracteres

C#

```
string[] numbers =
{
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
};
```

```
string sPattern = "^\d{3}-\d{3}-\d{4}$";

foreach (string s in numbers)
{
    Console.WriteLine($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
    else
    {
        Console.WriteLine(" - invalid");
    }
}
```

Este padrão de pesquisa único corresponde a várias cadeias de caracteres válidas. Expressões regulares são melhores para pesquisar por ou validar mediante um padrão, em vez de uma única cadeia de caracteres de texto.

## Confira também

- [Guia de programação em C#](#)
- [Cadeias de caracteres](#)
- [LINQ e cadeias de caracteres](#)
- [System.Text.RegularExpressions.Regex](#)
- [Expressões regulares do .NET](#)
- [Linguagem de expressão regular – referência rápida](#)
- [Práticas recomendadas para o uso de cadeias de caracteres no .NET](#)

# Como modificar o conteúdo de uma cadeia de caracteres em C#

Artigo • 09/05/2023

Este artigo demonstra várias técnicas para produzir um `string` modificando um `string` existente. Todas as técnicas demonstradas retornam o resultado das modificações como um novo objeto `string`. Para demonstrar que as cadeias de caracteres originais e modificadas são instâncias distintas, os exemplos armazenam o resultado em uma nova variável. Você pode examinar o original `string` e o `string` novo e modificado ao executar cada exemplo.

## ⓘ Observação

Os exemplos de C# neste artigo são executados no executador de código embutido Try.NET ↗ e no playground. Clique no botão **Executar** para executar um exemplo em uma janela interativa. Ao executar o código, é possível modificá-lo e executar o código modificado clicando em **Executar** novamente. O código modificado será executado na janela interativa ou, se a compilação falhar, a janela interativa exibirá todos as mensagens de erro do compilador C#.

Há várias técnicas demonstradas neste artigo. Você pode substituir o texto existente. Você pode procurar padrões e substituir o texto correspondente com outro texto. Você pode tratar uma cadeia de caracteres como uma sequência de caracteres. Você também pode usar métodos de conveniência que removem o espaço em branco. Escolha as técnicas que mais se aproximam do seu cenário.

## Substituir texto

O código a seguir cria uma nova cadeia de caracteres, substituindo o texto existente por um substituto.

C#

```
string source = "The mountains are behind the clouds today.";  
  
// Replace one substring with another with String.Replace.  
// Only exact matches are supported.  
var replacement = source.Replace("mountains", "peaks");  
Console.WriteLine($"The source string is <{source}>");  
Console.WriteLine($"The updated string is <{replacement}>");
```

O código anterior demonstra essa propriedade *immutable* de cadeias de caracteres. Você pode ver no exemplo anterior que a cadeia de caracteres original, `source`, não é modificada. O método `String.Replace` cria uma nova `string` contendo as modificações.

O método `Replace` pode substituir cadeias de caracteres ou caracteres únicos. Em ambos os casos, todas as ocorrências do texto pesquisado são substituídas. O exemplo a seguir substitui todos os caracteres ' ' com '\_':

C#

```
string source = "The mountains are behind the clouds today.";  
  
// Replace all occurrences of one char with another.  
var replacement = source.Replace(' ', '_');  
Console.WriteLine(source);  
Console.WriteLine(replacement);
```

A cadeia de caracteres de origem não é alterada e uma nova cadeia de caracteres é retornada com a substituição.

## Cortar espaço em branco

Você pode usar os métodos `String.Trim`, `String.TrimStart` e `String.TrimEnd` para remover espaços em branco à esquerda ou à direita. O código a seguir mostra um exemplo de cada um desses casos. A cadeia de caracteres de origem não é alterada; esses métodos retornam uma nova cadeia de caracteres com o conteúdo modificado.

C#

```
// Remove trailing and leading white space.  
string source = "    I'm wider than I need to be.      " ;  
// Store the results in a new string variable.  
var trimmedResult = source.Trim();  
var trimLeading = source.TrimStart();  
var trimTrailing = source.TrimEnd();  
Console.WriteLine($"<{source}>");  
Console.WriteLine($"<{trimmedResult}>");  
Console.WriteLine($"<{trimLeading}>");  
Console.WriteLine($"<{trimTrailing}>");
```

## Remover texto

Você pode remover texto de uma cadeia de caracteres usando o método `String.Remove`. Esse método remove um número de caracteres começando em um índice específico. O

exemplo a seguir mostra como usar `String.IndexOf` seguido por `Remove` para remover texto de uma cadeia de caracteres:

C#

```
string source = "Many mountains are behind many clouds today.";
// Remove a substring from the middle of the string.
string toRemove = "many ";
string result = string.Empty;
int i = source.IndexOf(toRemove);
if (i >= 0)
{
    result= source.Remove(i, toRemove.Length);
}
Console.WriteLine(source);
Console.WriteLine(result);
```

## Substituir padrões correspondentes

Você pode usar [expressões regulares](#) para substituir padrões correspondentes de texto com um novo texto, possivelmente definido por um padrão. O exemplo a seguir usa a classe [System.Text.RegularExpressions.Regex](#) para localizar um padrão em uma cadeia de caracteres de origem e substituí-lo pela capitalização correta. O método [Regex.Replace\(String, String, MatchEvaluator, RegexOptions\)](#) usa uma função que fornece a lógica de substituição como um de seus argumentos. Neste exemplo, essa função `LocalReplaceMatchCase` é uma **função local** declarada dentro do método de exemplo. `LocalReplaceMatchCase` usa a classe [System.Text.StringBuilder](#) para criar a cadeia de caracteres de substituição com a capitalização correta.

Expressões regulares são mais úteis para localizar e substituir texto que segue um padrão, em vez de texto conhecido. Saiba mais em [Como pesquisar cadeias de caracteres](#). O padrão de pesquisa "the\s" procura a palavra "the" seguida por um caractere de espaço em branco. Essa parte do padrão garante que isso não corresponda a "there" na cadeia de caracteres de origem. Para obter mais informações sobre elementos de linguagem de expressão regular, consulte [Linguagem de expressão regular – referência rápida](#).

C#

```
string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
```

```

source = System.Text.RegularExpressions.Regex.Replace(source, "the\\s",
LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match
matchExpression)
{
    // Test whether the match is capitalized
    if (Char.IsUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new
System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}

```

O método [StringBuilder.ToString](#) retorna uma cadeia de caracteres imutável com o conteúdo no objeto [StringBuilder](#).

## Modificar caracteres individuais

Você pode produzir uma matriz de caracteres de uma cadeia de caracteres, modificar o conteúdo da matriz e, em seguida, criar uma nova cadeia de caracteres com base no conteúdo modificado da matriz.

O exemplo a seguir mostra como substituir um conjunto de caracteres em uma cadeia de caracteres. Primeiro, ele usa o método [String.ToCharArray\(\)](#) para criar uma matriz de caracteres. Ele usa o método [IndexOf](#) para encontrar o índice inicial da palavra "fox". Os próximos três caracteres são substituídos por uma palavra diferente. Por fim, uma nova cadeia de caracteres é construída com a matriz de caracteres atualizada.

C#

```

string phrase = "The quick brown fox jumps over the fence";
Console.WriteLine(phrase);

char[] phraseAsChars = phrase.ToCharArray();
int animalIndex = phrase.IndexOf("fox");
if (animalIndex != -1)
{
    phraseAsChars[animalIndex++] = 'c';
    phraseAsChars[animalIndex++] = 'a';
}

```

```
    phraseAsChars[animalIndex] = 't';
}

string updatedPhrase = new string(phraseAsChars);
Console.WriteLine(updatedPhrase);
```

## Compilar programaticamente o conteúdo de cadeia de caracteres

Como as cadeias de caracteres são imutáveis, todos os exemplos anteriores criam cadeias de caracteres temporárias ou matrizes de caracteres. Em cenários de alto desempenho, pode ser desejável evitar essas alocações de heap. O .NET Core fornece um método [String.Create](#) que permite preencher programaticamente o conteúdo de caracteres de uma cadeia de caracteres por meio de um retorno de chamada, evitando as alocações de cadeias de caracteres temporárias intermediárias.

C#

```
// constructing a string from a char array, prefix it with some additional
// characters
char[] chars = { 'a', 'b', 'c', 'd', '\0' };
int length = chars.Length + 2;
string result = string.Create(length, chars, (Span<char> strContent, char[]
charArray) =>
{
    strContent[0] = '0';
    strContent[1] = '1';
    for (int i = 0; i < charArray.Length; i++)
    {
        strContent[i + 2] = charArray[i];
    }
});

Console.WriteLine(result);
```

Você pode modificar uma cadeia de caracteres em um bloco fixo com código não seguro, mas é **altamente** desaconselhável modificar o conteúdo da cadeia de caracteres depois que uma cadeia de caracteres é criada. Fazer isso pode quebrar as coisas de maneiras imprevisíveis. Por exemplo, se alguém internalizar uma cadeia de caracteres com o mesmo conteúdo que o seu, ele receberá a sua cópia e não esperará que você esteja modificando sua cadeia de caracteres.

## Confira também

- Expressões regulares do .NET
- Linguagem de expressão regular – referência rápida

# Como comparar cadeias de caracteres no C#

Artigo • 05/06/2023

Você compara cadeias de caracteres para responder a uma das duas perguntas: "Essas duas cadeias de caracteres são iguais?" ou "Em que ordem essas cadeias de caracteres devem ser colocadas ao classificá-las?"

Essas duas perguntas são complicadas devido a fatores que afetam as comparações de cadeia de caracteres:

- Você pode escolher uma comparação ordinal ou linguística.
- Você pode escolher se o uso de maiúsculas faz diferença.
- Você pode escolher comparações específicas de cultura.
- As comparações linguísticas dependem da plataforma e da cultura.

## ⓘ Observação

Os exemplos de C# neste artigo são executados no executador de código embutido Try.NET ↗ e no playground. Clique no botão **Executar** para executar um exemplo em uma janela interativa. Ao executar o código, é possível modificá-lo e executar o código modificado clicando em **Executar** novamente. O código modificado será executado na janela interativa ou, se a compilação falhar, a janela interativa exibirá todos as mensagens de erro do compilador C#.

Ao comparar cadeias de caracteres, você pode definir uma ordem entre elas. As comparações são usadas para classificar uma sequência de cadeias de caracteres. Com a sequência em uma ordem conhecida, fica mais fácil de pesquisar, tanto para softwares quanto para os usuários. Outras comparações podem verificar se as cadeias de caracteres são iguais. Essas verificações são semelhantes a igualdade, mas algumas diferenças, como diferenças entre maiúsculas e minúsculas, podem ser ignoradas.

## Comparações ordinárias padrão

Por padrão, as operações mais comuns:

- [String.Equals](#)
- [String.Equality](#) e [String.Inequality](#), ou seja, [operadores de igualdade == e !=](#), respectivamente,

realizem uma comparação ordinal que diferencie maiúsculas de minúsculas. No caso de [String.Equals](#), um argumento  [StringComparison](#) pode ser fornecido para alterar suas regras de classificação. O exemplo a seguir demonstra que:

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root ==
root2 ? "equal" : "not equal")});
```

A comparação ordinal padrão não leva em conta regras linguísticas ao comparar cadeias de caracteres. Ela compara o valor binário de cada objeto [Char](#) em duas cadeias de caracteres. Como resultado, a comparação ordinal padrão também diferencia maiúsculas de minúsculas.

Observe que o teste de igualdade com [String.Equals](#) e os operadores `==` e `!=` difere da comparação de cadeias de caracteres usando os métodos [String.CompareTo](#) e [Compare\(String, String\)](#). Todos eles executam uma comparação que diferencia maiúsculas de minúsculas. No entanto, enquanto os testes de igualdade executam uma comparação ordinal, os métodos [CompareTo](#) e [Compare](#) executam uma comparação linguística com reconhecimento de cultura usando a cultura atual. Como os métodos de comparação padrão diferem nas formas de comparar cadeias de caracteres, recomendamos que você sempre tenha a intenção de ter um código claro ao chamar uma sobrecarga que explicitamente especifique o tipo de comparação a ser realizada.

## Comparações ordinais que não diferenciam maiúsculas de minúsculas

O método [String.Equals\(String, StringComparison\)](#) permite que você especifique um valor  [StringComparison](#) de  [StringComparison.OrdinalIgnoreCaseIgnoreCase](#) para especificar uma comparação que não diferencia maiúsculas de minúsculas. Há também um método [String.Compare\(String, String, StringComparison\)](#) estático que fará uma comparação ordinal sem distinção entre maiúsculas e minúsculas se você especificar um valor de

`StringComparison.OrdinalIgnoreCase` para o argumento  `StringComparison`. Eles são mostrados no código a seguir:

```
C#  
  
string root = @"C:\users";  
string root2 = @"C:\Users";  
  
bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);  
bool areEqual = String.Equals(root, root2,  
StringComparison.OrdinalIgnoreCase);  
int comparison = String.Compare(root, root2, comparisonType:  
StringComparison.OrdinalIgnoreCase);  
  
Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result  
? "equal." : "not equal.")}");  
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are  
{(areEqual ? "equal." : "not equal.")}");  
if (comparison < 0)  
    Console.WriteLine($"<{root}> is less than <{root2}>");  
else if (comparison > 0)  
    Console.WriteLine($"<{root}> is greater than <{root2}>");  
else  
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");
```

Ao fazer uma comparação ordinal sem distinção entre maiúsculas e minúsculas, esses métodos usam as convenções de maiúsculas e minúsculas da [cultura invariável](#).

## Comparações linguísticas

Muitos métodos de comparação de cadeias de caracteres (como `String.StartsWith`) usam regras linguísticas para a *cultura atual* por padrão para ordenar suas entradas. Às vezes, isso é chamado de “ordem de classificação de palavras”. Quando você executa uma comparação linguística, alguns caracteres Unicode não alfanuméricos podem ter pesos especiais atribuídos. Por exemplo, um peso muito pequeno pode estar atribuído ao hífen “-”, de modo que “co-op” e “coop” apareçam próximos um do outro na ordem de classificação, enquanto alguns caracteres de controle não relacionados à impressão podem ser completamente ignorados. Além disso, alguns caracteres Unicode podem ser equivalentes a uma sequência de instâncias `Char`. O exemplo a seguir usa a frase “Eles dançam na rua” em alemão com os “ss” (U+0073 U+0073) em uma cadeia de caracteres e ‘ß’ (U+00DF) em outra. Linguisticamente (no Windows), “ss” é igual ao caractere ‘ß’ Esszet alemão nas culturas “en-US” e “de-DE”.

```
C#
```

```

string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second,
StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")}
equal.");
showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two,
StringComparison.InvariantCulture);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using invariant
culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant
culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using invariant culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using ordinal comparison");
}

```

No Windows, antes do .NET 5, a ordem de classificação de "cop", "coop" e "co-op" muda quando você muda de uma comparação linguística para uma comparação ordinal. As duas frases em alemão também são comparadas de forma diferente ao usar os tipos diferentes de comparação. Isso ocorre porque, antes do .NET 5, as APIs de globalização do .NET usavam bibliotecas [NLS \(National Language Support\)](#). No .NET 5 e versões posteriores, as APIs de globalização do .NET usam bibliotecas [ICU](#)

(Componentes Internacionais para Unicode) [🔗](#), que unificam o comportamento de globalização da .NET em todos os sistemas operacionais com suporte.

## Comparações usando culturas específicas

Este exemplo armazena objetos `CultureInfo` para as culturas en-US e de-DE. As comparações são feitas usando um objeto `CultureInfo` para garantir uma comparação específica da cultura.

A cultura usada afeta as comparações linguísticas. O exemplo a seguir mostra os resultados da comparação das duas frases em alemão usando a cultura "en-US" e a cultura "de-DE":

C#

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");

var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")} equal.");

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo
culture)
{
    int compareLinguistic = String.Compare(one, two, en,
System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.OrdinalIgnoreCase);
```

```

    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using en-US
culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US
culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using en-US culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using ordinal comparison");
}

```

As comparações que diferenciam cultura normalmente são usadas para comparar e classificar cadeias de caracteres inseridas por usuários com outras cadeias de caracteres inseridas por usuários. Os caracteres e as convenções de classificação dessas cadeias de caracteres podem variar de acordo com a localidade do computador do usuário. Até mesmo cadeias de caracteres que contêm caracteres idênticos podem ser classificadas de formas diferentes dependendo da cultura do thread atual.

## Classificação linguística e cadeias de caracteres de pesquisa em matrizes

Os exemplos a seguir mostram como classificar e pesquisar cadeias de caracteres em uma matriz usando uma comparação linguística que depende da cultura atual. Use os métodos [Array](#) estáticos que aceitam um parâmetro [System.StringComparer](#).

Este exemplo mostra como classificar uma matriz de cadeias de caracteres usando a cultura atual:

C#

```

string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

```

```

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}

Console.WriteLine("\n\nSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}

```

Depois que a matriz é classificada, você pode procurar as entradas usando uma pesquisa binária. Uma pesquisa binária é iniciada no meio da coleção para determinar qual metade da coleção contém a cadeia de caracteres procurada. Cada comparação subsequente subdivide a parte restante da coleção na metade. A matriz é classificada usando o [StringComparer.CurrentCulture](#). A função local `ShowWhere` exibe informações sobre o local em que a cadeia de caracteres foi encontrada. Se a cadeia de caracteres não for encontrada, o valor retornado indicará onde ela estaria se fosse encontrada.

C#

```

string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString,
StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")}{searchString}");

void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");
    }
}

```

```

    if (index == 0)
        Console.Write("beginning of sequence and ");
    else
        Console.WriteLine(${array[index - 1]} and ");

    if (index == array.Length)
        Console.WriteLine("end of sequence.");
    else
        Console.WriteLine(${array[index]}.");
}
else
{
    Console.WriteLine($"Found at index {index}.");
}
}

```

## Classificação ordinal e pesquisa em coleções

O código a seguir usa a classe de coleção `System.Collections.Generic.List<T>` para armazenar cadeias de caracteres. As cadeias de caracteres são classificadas usando o método `List<T>.Sort`. Esse método precisa de um delegado que compara e ordena as duas cadeias de caracteres. O método `String.CompareTo` fornece essa função de comparação. Execute o exemplo e observe a ordem. Essa operação de classificação usa uma classificação ordinal que diferencia maiúsculas e minúsculas. Você usaria os métodos `String.Compare` estáticos para especificar regras de comparação diferentes.

C#

```

List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}

Console.WriteLine("\n\nSorted order:");

lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{

```

```
        Console.WriteLine($"    {s}");  
    }
```

Uma vez classificada, a lista de cadeias de caracteres pode ser pesquisada usando uma pesquisa binária. O exemplo a seguir mostra como pesquisar a lista classificada usando a mesma função de comparação. A função local `ShowWhere` mostra o local em que o texto procurado está ou deveria estar:

C#

```
List<string> lines = new List<string>  
{  
    @"c:\public\textfile.txt",  
    @"c:\public\textFile.TXT",  
    @"c:\public\Text.txt",  
    @"c:\public\testfile2.txt"  
};  
lines.Sort((left, right) => left.CompareTo(right));  
  
string searchString = @"c:\public\TEXTFILE.TXT";  
Console.WriteLine($"Binary search for <{searchString}>");  
int result = lines.BinarySearch(searchString);  
ShowWhere<string>(lines, result);  
  
Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")}  
{searchString});  
  
void ShowWhere<T>(IList<T> collection, int index)  
{  
    if (index < 0)  
    {  
        index = ~index;  
  
        Console.Write("Not found. Sorts between: ");  
  
        if (index == 0)  
            Console.Write("beginning of sequence and ");  
        else  
            Console.Write($"{collection[index - 1]} and ");  
  
        if (index == collection.Count)  
            Console.WriteLine("end of sequence.");  
        else  
            Console.WriteLine($"{collection[index]}.");  
    }  
    else  
    {  
        Console.WriteLine($"Found at index {index}.");  
    }  
}
```

Use sempre o mesmo tipo de comparação para classificação e pesquisa. O uso de tipos diferentes de comparação para classificação e pesquisa produz resultados inesperados.

Classes de coleção como [System.Collections.Hashtable](#), [System.Collections.Generic.Dictionary<TKey,TValue>](#) e [System.Collections.Generic.List<T>](#) têm construtores que usam um parâmetro [System.StringComparer](#) quando o tipo dos elementos ou chaves é `string`. Em geral, você deve usar esses construtores sempre que possível e especificar [StringComparer.Ordinal](#) ou [StringComparer.OrdinalIgnoreCase](#).

## Confira também

- [System.Globalization.CultureInfo](#)
- [System.StringComparer](#)
- [Cadeias de caracteres](#)
- [Comparando cadeias de caracteres](#)
- [Globalizando e localizando aplicativos](#)

# Como capturar uma exceção não compatível com CLS

Artigo • 10/05/2023

Algumas linguagens .NET, incluindo o C++/CLI, permite que os objetos lancem exceções que não derivam de [Exception](#). Essas exceções são chamadas de *exceções não CLS* ou *não exceções*. Em C#, não é possível gerar exceções que não sejam do CLS, mas você pode capturá-las de duas maneiras:

- Em um bloco `catch (RuntimeWrappedException e)`.

Por padrão, um assembly do Visual C# captura exceções não CLS como exceções encapsuladas. Use este método se você precisar de acesso à exceção original, que pode ser acessada por meio da propriedade

[RuntimeWrappedException.WrappedException](#). O procedimento, mais adiante neste tópico, explica como capturar exceções dessa maneira.

- Em um bloco de captura geral (um bloco de captura sem um tipo de exceção especificado), que é colocado após todos os outros blocos `catch`.

Use esse método quando desejar realizar alguma ação (como gravar em um arquivo de log) em resposta a exceções não CLS e você não precisa de acesso às informações de exceção. Por padrão, o Common Language Runtime encapsula todas as exceções. Para desabilitar esse comportamento, adicione esse atributo de nível de assembly em seu código, geralmente no arquivo AssemblyInfo.cs:

```
[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)].
```

## Para capturar uma exceção não CLS

Em um bloco `catch(RuntimeWrappedException e)`, acesse a exceção original por meio da propriedade [RuntimeWrappedException.WrappedException](#).

## Exemplo

O exemplo a seguir mostra como capturar uma exceção que não é do CLS, que foi gerada por uma biblioteca de classes escrita em C++/CLI. Observe que, neste exemplo, o código cliente C# sabe com antecedência que o tipo da exceção que está sendo gerada é um [System.String](#). Você pode converter a propriedade [RuntimeWrappedException.WrappedException](#) de volta a seu tipo original, desde que o tipo seja acessível por meio do código.

C#

```
// Class library written in C++/CLI.  
var myClass = new ThrowNonCLS.Class1();  
  
try  
{  
    // throws gcnew System::String(  
    // "I do not derive from System.Exception!");  
    myClass.TestThrow();  
}  
catch (RuntimeWrappedException e)  
{  
    String s = e.WrappedException as String;  
    if (s != null)  
    {  
        Console.WriteLine(s);  
    }  
}
```

## Confira também

- [RuntimeWrappedException](#)
- [Exceções e manipulação de exceções](#)

# O SDK do .NET Compiler Platform

Artigo • 15/02/2023

Os compiladores criam um modelo detalhado do código do aplicativo conforme validam a sintaxe e a semântica do código. O uso desse modelo para criar a saída executável do código-fonte. O SDK do .NET Compiler Platform fornece acesso a esse modelo. Cada vez mais, contamos com recursos do IDE (ambiente de desenvolvimento integrado), como IntelliSense, refatoração, renomeação inteligente, "Localizar todas as referências" e "Ir para definição" para aumentar nossa produtividade. Contamos com ferramentas de análise de código para melhorar a qualidade e com geradores de código para ajudar na criação do aplicativo. À medida que essas ferramentas ficam mais inteligentes, elas precisam de acesso a cada vez mais do modelo que somente os compiladores podem criar conforme processam o código do aplicativo. Este é o objetivo principal das APIs do Roslyn: abrir as caixas opacas e permitir que as ferramentas e os usuários finais compartilhem a riqueza de informações que os compiladores têm sobre nosso código. Em vez de ser meros conversores de código-fonte e objeto-código-saída, por meio do Roslyn, os compiladores se tornam plataformas: as APIs que você pode usar para as tarefas relacionadas ao código em seus aplicativos e ferramentas.

## Conceitos do SDK do .NET Compiler Platform

O SDK do .NET Compiler Platform diminui drasticamente a barreira de entrada para a criação de aplicativos e ferramentas voltadas para o código. Ele cria várias oportunidades para inovação em áreas como metaprogramação, geração e transformação de código, uso interativo das linguagens C# e Visual Basic e incorporação de C# e Visual Basic a linguagens específicas de domínio.

O SDK do .NET Compiler Platform permite que você crie *analisadores* e *correções de código* que encontram e corrigem os erros de codificação. Os *analisadores* entendem a sintaxe (estrutura do código) e a semântica para detectar práticas que devem ser corrigidas. As *correções de código* fornecem uma ou mais correções sugeridas para tratar erros de codificação encontrados pelos analisadores ou diagnósticos do compilador. Normalmente, um analisador e as correções de código associadas são empacotados em um único projeto.

Os analisadores e as correções de código usam a análise estática para entender o código. Eles não executam o código ou fornecem outros benefícios de teste. No entanto, eles podem destacar práticas que frequentemente levam a erros, códigos de difícil manutenção ou violação de diretrizes padrão.

Além de analisadores e correções de código, o SDK do .NET Compiler Platform também permite criar *refatorações de código*. Ele também fornece um único conjunto de APIs que permitem que você examine e compreenda uma base de código C# ou Visual Basic. Uma vez que você pode usar essa base de código única, é possível escrever analisadores e correções de código com mais facilidade aproveitando as APIs de análise de sintaxe e de semântica fornecidas pelo SDK do .NET Compiler Platform. Liberado da enorme tarefa de replicar a análise feita pelo compilador, você pode se concentrar na tarefa de localizar e corrigir os erros de codificação comuns no projeto ou na biblioteca.

Um benefício menor é que os analisadores e as correções de código são menores e usam muito menos memória quando carregados no Visual Studio do que usariam se você tivesse escrito sua própria base de código para entender o código em um projeto. Aproveitando as mesmas classes usadas pelo compilador e o Visual Studio, você pode criar suas próprias ferramentas de análise estática. Isso significa que sua equipe poderá usar os analisadores e as correções de código sem um impacto significativo no desempenho do IDE.

Há três cenários principais para escrever analisadores e correções de código:

1. *Impor padrões de codificação à equipe*
2. *Fornecer diretrizes com pacotes de biblioteca*
3. *Fornecer diretrizes gerais*

## Impor padrões de codificação à equipe

Muitas equipes têm padrões de codificação aplicados por meio de revisões de código feitas com outros membros da equipe. Os analisadores e as correções de código podem tornar esse processo muito mais eficiente. As revisões de código ocorrem quando um desenvolvedor compartilha seu trabalho com outras pessoas da equipe. O desenvolvedor terá investido todo o tempo necessário para concluir um novo recurso antes de obter algum comentário. Semanas podem passar enquanto o desenvolvedor reforça hábitos que não correspondem às práticas da equipe.

Os analisadores são executados à medida que um desenvolvedor escreve o código. O desenvolvedor obtém comentários imediatos que o incentivam a seguir as diretrizes no mesmo instante. O desenvolvedor cria hábitos para gravar códigos compatíveis assim que começa a criar protótipos. Quando o recurso está pronto para que outras pessoas o examinem, todas as diretrizes padrão já terão sido impostas.

As equipes podem criar analisadores e correções de código que procurem as práticas mais comuns que violam as práticas de codificação em equipe. Eles podem ser instalados nos computadores de cada desenvolvedor para impor os padrões.

## Dica

Antes de criar seu próprio analisador, confira os analisadores internos. Para mais informações, confira [Regras de estilo de código](#).

## Fornecer diretrizes com pacotes de biblioteca

Há uma grande variedade de bibliotecas para desenvolvedores de .NET no NuGet. Algumas dessas provenientes da Microsoft, algumas de terceiros e outras de membros e de voluntários da comunidade. Essas bibliotecas obtêm mais adoção e análises mais positivas quando os desenvolvedores são bem-sucedidos com elas.

Além de fornecer a documentação, você pode fornecer analisadores e correções de código que encontram e corrigem os usos inadequados comuns da sua biblioteca. Essas correções imediatas ajudarão os desenvolvedores a obter êxito mais rapidamente.

Você pode empacotar analisadores e correções de código com sua biblioteca no NuGet. Nesse cenário, cada desenvolvedor que instalar o pacote do NuGet também instalará o pacote do analisador. Todos os desenvolvedores que estiverem usando a biblioteca obterão imediatamente as diretrizes da sua equipe na forma de comentários imediatos sobre erros e correções sugeridas.

## Fornecer diretrizes gerais

A comunidade de desenvolvedores do .NET descobriu, pela experiência, os padrões que funcionam bem e os padrões que devem ser evitados. Vários membros da comunidade criaram analisadores que impõem esses padrões recomendados. À medida que aprendemos mais, sempre haverá espaço para novas ideias.

Esses analisadores podem ser carregados no [Visual Studio Marketplace](#) e baixados por desenvolvedores que usam o Visual Studio. Quem ainda não tem experiência na linguagem e na plataforma aprende rapidamente as práticas aceitas e se torna produtivo mais cedo em sua jornada no .NET. Quando as práticas se tornam amplamente usadas, a comunidade as adota.

## Próximas etapas

O SDK do .NET Compiler Platform inclui os modelos de objeto de linguagem mais recentes para geração de código, análise e refatoração. Esta seção fornece uma visão

geral conceitual do SDK do .NET Compiler Platform. Mais detalhes podem ser encontrados nas seções de inícios rápidos, de exemplos e de tutoriais.

Você pode saber mais sobre os conceitos no SDK do .NET Compiler Platform nestes cinco tópicos:

- Explorar código com o visualizador de sintaxe
- Entender o modelo de API do compilador
- Trabalhar com sintaxe
- Trabalhar com semântica
- Trabalhar com um workspace

Para começar, será necessário instalar o **SDK do .NET Compiler Platform**:

## Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o **SDK da .NET Compiler Platform** no **Instalador do Visual Studio**:

### Instalar usando o Instalador do Visual Studio – exibição de cargas de trabalho

O SDK da .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento da extensão do Visual Studio. É necessário selecioná-lo como um componente opcional.

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Marque a carga de trabalho de **Desenvolvimento de extensão do Visual Studio**.
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará por último nos componentes opcionais.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

# Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará na parte superior, na seção **Compiladores, ferramentas de compilação e runtimes**.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Marque a caixa do **Editor DGML**. Você a encontrará na seção **Ferramentas de código**.

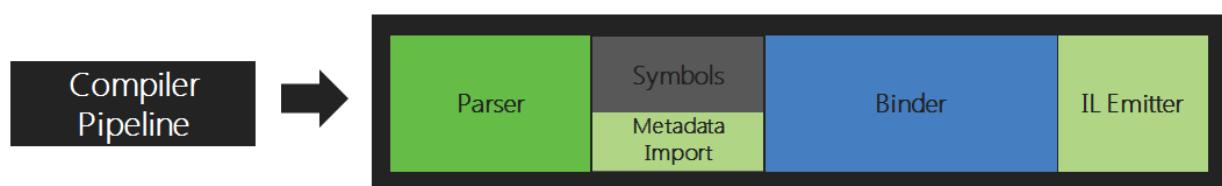
# Entender o modelo do SDK do .NET Compiler Platform

Artigo • 10/05/2023

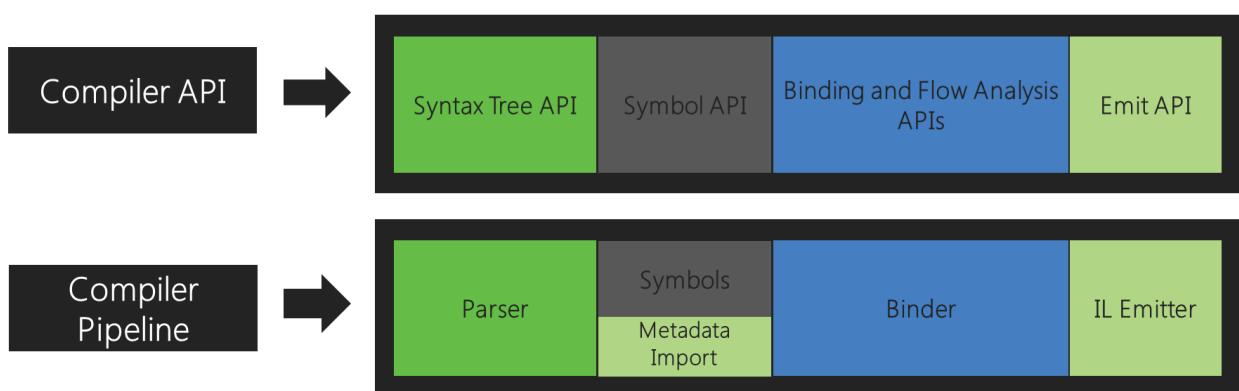
Os compiladores processam o código escrito seguindo regras estruturadas que geralmente diferem da forma como os humanos leem e entendem um código. Uma compreensão básica do modelo usado pelos compiladores é essencial para compreender as APIs usadas ao criar ferramentas baseadas no Roslyn.

## Áreas funcionais do pipeline do compilador

O SDK do .NET Compiler Platform expõe a análise de código dos compiladores C# e Visual Basic para você como um consumidor, fornecendo uma camada de API que espelha um pipeline de compilador tradicional.

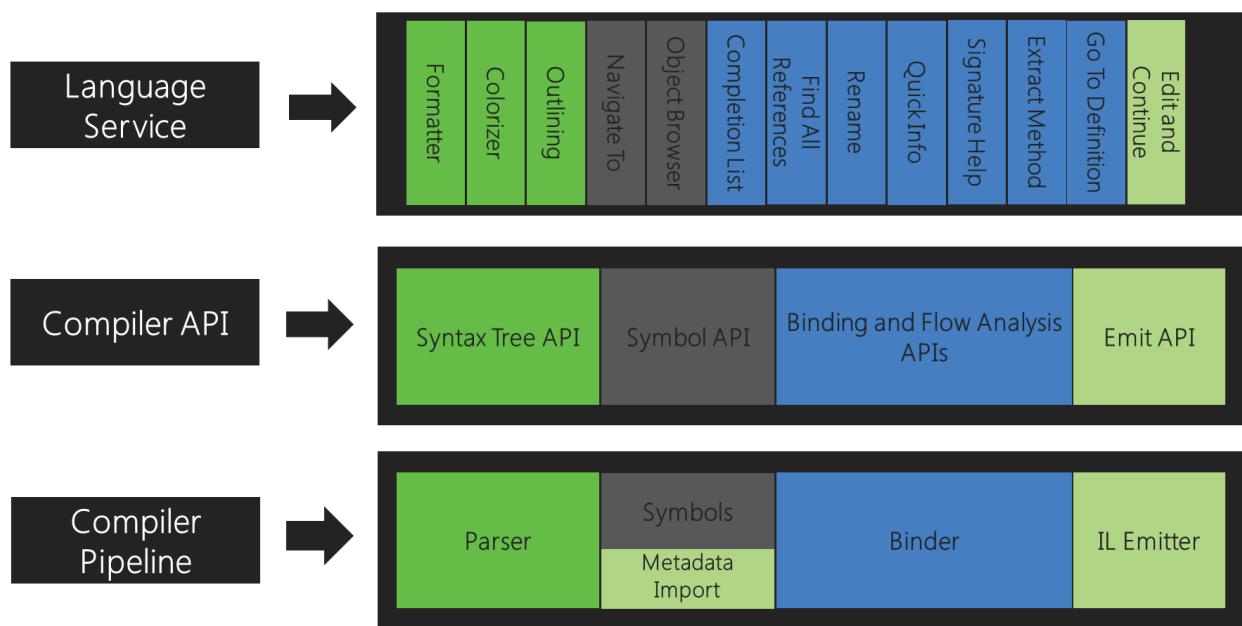


Cada fase desse pipeline é um componente separado. Primeiro, a fase de análise cria tokens do texto de origem e o analisa na sintaxe que segue a gramática da linguagem. Depois, a fase de declaração analisa os metadados de origem e importados para formar símbolos nomeados. Em seguida, a fase de associação corresponde os identificadores no código aos símbolos. Por fim, a fase de emissão emite um assembly com todas as informações criadas pelo compilador.



Correspondente a cada uma dessas fases, o SDK do .NET Compiler Platform expõe um modelo de objeto que permite o acesso às informações da fase. A fase de análise expõe uma árvore de sintaxe, a fase de declaração expõe uma tabela de símbolos hierárquica,

a fase de associação expõe o resultado da análise semântica do compilador e a fase de emissão é uma API que gera códigos de bytes de IL.



Cada compilador combina esses componentes como um único inteiro de ponta a ponta.

Essas APIs são as mesmas usadas pelo Visual Studio. Por exemplo, os recursos de formatação e estrutura de tópicos do código usam as árvores de sintaxe, o **Pesquisador de Objetos** e os recursos de navegação usam a tabela de símbolos, as refatorações e o recurso **Ir para Definição** usam o modelo semântico e o recurso **Editar e Continuar** usa todos eles, inclusive a API de Emissão.

## Camadas de API

O SDK do compilador .NET consiste em várias camadas de APIs, ou seja: APIs de compilador, de diagnóstico, de script e de espaços de trabalho.

## APIs do compilador

A camada do compilador contém os modelos de objeto que correspondem às informações expostas em cada fase do pipeline do compilador, tanto sintáticas quanto semânticas. A camada do compilador também contém um instantâneo imutável de uma única invocação de um compilador, incluindo referências de assembly, opções do compilador e arquivos de código-fonte. Há duas APIs distintas que representam a linguagem C# e a linguagem Visual Basic. Essas duas APIs são semelhantes na forma, mas adaptadas para alta fidelidade a cada linguagem individual. Essa camada não tem dependências em componentes do Visual Studio.

## APIs de diagnóstico

Como parte da análise, o compilador pode produzir um conjunto de diagnósticos que abrangem tudo, desde sintaxe, semântica e erros de atribuição definida a vários diagnósticos de avisos e informativos. A camada de API do Compilador expõe o diagnóstico por meio de uma API extensível que permite que os analisadores definidos pelo usuário sejam conectados ao processo de compilação. Ela possibilita que o diagnóstico definido pelo usuário, como aqueles gerados por ferramentas como o StyleCop, seja produzido junto com o diagnóstico definido pelo compilador. Essa forma de produção de diagnóstico tem o benefício da integração natural a ferramentas como o MSBuild e o Visual Studio, que dependem do diagnóstico para experiências como interrupção de um build com base na política, exibição de textos sublinhados em tempo real no editor e sugestão de correções de código.

## APIs de script

APIs de hospedagem e script fazem parte da camada do compilador. Você pode usá-las para execução de snippets de código e acúmulo de um contexto de execução em runtime. O REPL (Loop de Leitura-Avaliação-Impressão) interativo do C# usa essas APIs. O REPL permite usar o C# como a linguagem de scripts, executando o código de forma interativa à medida que ele é escrito.

## APIs dos workspaces

A camada Workspaces contém a API de Workspace, que é o ponto de partida para fazer a análise de código e refatoração em soluções inteiras. Ela ajuda a organizar todas as informações sobre os projetos de uma solução em um único modelo de objeto, oferecendo acesso direto aos modelos de objeto da camada do compilador, sem a necessidade de analisar arquivos, configurar opções ou gerenciar dependências projeto a projeto.

Além disso, a camada Workspaces expõe um conjunto de APIs usado ao implementar ferramentas de análise de código e refatoração que funcionam em um ambiente de host como o IDE do Visual Studio. Exemplos incluem as APIs Localizar Todas as Referências, Formatação e Geração de Código.

Essa camada não tem dependências em componentes do Visual Studio.

# Trabalhar com sintaxe

Artigo • 10/05/2023

A árvore de sintaxe é uma estrutura de dados imutável e fundamental exposta pelas APIs do compilador. Essas árvores representam a estrutura lexical e sintática do código-fonte. Elas servem duas finalidades importantes:

- Permitir que ferramentas – como um IDE, suplementos, ferramentas de análise de código e refatorações – vejam e processem a estrutura sintática do código-fonte no projeto do usuário.
- Permitir que ferramentas – como refatorações e um IDE – criem, modifiquem e reorganizem o código-fonte de uma maneira natural sem a necessidade de uso de edições de texto diretas. Criando e manipulando árvores, as ferramentas podem criar e reorganizar o código-fonte com facilidade.

## Árvores de sintaxe

Árvores de sintaxe são a estrutura principal usada para compilação, análise de código, associação, refatoração, recursos de IDE e geração de código. Nenhuma parte do código-fonte é entendida sem primeiro ser identificada e categorizada em um dos muitos elementos de linguagem estrutural conhecidos.

As árvores de sintaxe têm três atributos-chave:

- Elas têm todas as informações de origem com fidelidade total. A fidelidade total significa que a árvore de sintaxe contém cada informação encontrada no texto de origem, cada constructo gramatical, cada token lexical e todo o resto, incluindo espaço em branco, comentários e diretivas do pré-processador. Por exemplo, cada literal mencionado na fonte é representado exatamente como foi digitado. Através da representação de tokens ignorados ou ausentes, as árvores de sintaxe também capturam erros no código-fonte quando o programa está incompleto ou mal-formado.
- Elas podem produzir o texto exato do qual foram analisados. Em qualquer nó de sintaxe, é possível obter a representação de texto da subárvore com raiz nesse nó. Essa habilidade significa que as árvores de sintaxe podem ser usadas como uma maneira de construir e editar o texto de origem. Ao criar uma árvore, por implicação, você criou o texto equivalente e, ao criar uma nova árvore com base nas alterações de uma árvore existente, você editou o texto efetivamente.
- Eles são imutáveis e thread-safe. Depois que uma árvore é obtida, ela é um instantâneo do estado atual do código e nunca é alterada. Isso permite que vários

usuários interajam com a mesma árvore de sintaxe ao mesmo tempo em threads diferentes sem bloqueio nem duplicação. Como as árvores são imutáveis e nenhuma modificação pode ser feita diretamente em uma árvore, os métodos de fábrica ajudam a criar e modificar árvores de sintaxe criando instantâneos adicionais da árvore. As árvores são eficientes no modo como reutilizam os nós subjacentes, de forma que uma nova versão possa ser recompilada rapidamente e com pouca memória extra.

Uma árvore de sintaxe é literalmente uma estrutura de dados de árvore, em que os elementos estruturais não terminais são pais de outros elementos. Cada árvore de sintaxe é composta por nós, tokens e desafios.

## Nós de sintaxe

Nós de sintaxe são um dos elementos principais das árvores de sintaxe. Esses nós representam os constructos sintáticos como declarações, instruções, cláusulas e expressões. Cada categoria de nós de sintaxe é representada por uma classe separada derivada de [Microsoft.CodeAnalysis.SyntaxNode](#). O conjunto de classes de nó não é extensível.

Todos os nós de sintaxe são nós não terminais na árvore de sintaxe, o que significa que eles sempre têm outros nós e tokens como filhos. Como filho de outro nó, cada nó tem um nó pai que pode ser acessado por meio da propriedade [SyntaxNode.Parent](#). Como os nós e as árvores são imutáveis, o pai de um nó nunca é alterado. A raiz da árvore tem um pai nulo.

Cada nó tem um método [SyntaxNode.ChildNodes\(\)](#), que retorna uma lista de nós filho em ordem sequencial com base em sua posição no texto de origem. Essa lista não contém tokens. Cada nó também tem métodos para examinar os Descendentes, como [DescendantNodes](#), [DescendantTokens](#) ou [DescendantTrivia](#) – que representam uma lista de todos os nós, tokens ou desafios, que existem na subárvore com raiz nesse nó.

Além disso, cada subclasse de nó de sintaxe expõe os mesmos filhos por meio de propriedades fortemente tipadas. Por exemplo, uma classe de nó [BinaryExpressionSyntax](#) tem três propriedades adicionais específicas aos operadores binários: [Left](#), [OperatorToken](#) e [Right](#). O tipo de [Left](#) e [Right](#) é [ExpressionSyntax](#) e o tipo de [OperatorToken](#) é [SyntaxToken](#).

Alguns nós de sintaxe têm filhos opcionais. Por exemplo, um [IfStatementSyntax](#) tem um [ElseClauseSyntax](#) opcional. Se o filho não estiver presente, a propriedade retornará nulo.

# Tokens de sintaxe

Os tokens de sintaxe são os terminais da gramática da linguagem, que representam os menores fragmentos sintáticos do código. Eles nunca são os pais de outros nós ou tokens. Os tokens de sintaxe consistem em palavras-chave, identificadores, literais e pontuação.

Para fins de eficiência, o tipo [SyntaxToken](#) é um tipo de valor CLR. Portanto, ao contrário dos nós de sintaxe, há apenas uma estrutura para todos os tipos de tokens com uma combinação de propriedades que têm significado, dependendo do tipo de token que está sendo representado.

Por exemplo, um token literal inteiro representa um valor numérico. Além do texto de origem não processado abrangido pelo token, o token literal tem uma propriedade [Value](#) que informa o valor inteiro decodificado exato. Essa propriedade é tipada como [Object](#) porque pode ser um dos muitos tipos primitivos.

A propriedade [ValueText](#) indica as mesmas informações que a propriedade [Value](#); no entanto, essa propriedade sempre é tipada como [String](#). Um identificador no texto de origem C# pode incluir caracteres de escape Unicode, embora a sintaxe da sequência de escape em si não seja considerada parte do nome do identificador. Portanto, embora o texto não processado abrangido pelo token inclua a sequência de escape, isso não ocorre com a propriedade [ValueText](#). Em vez disso, ela inclui os caracteres Unicode identificados pelo escape. Por exemplo, se o texto de origem contiver um identificador gravado como `\u03c0`, a propriedade [ValueText](#) desse token retornará `π`.

# Desafios de sintaxe

Os desafios de sintaxe representam as partes do texto de origem que são amplamente insignificantes para o reconhecimento normal do código, como espaço em branco, comentários e diretivas do pré-processador. Assim como os tokens de sintaxe, os desafios são tipos de valor. O único tipo [Microsoft.CodeAnalysis.SyntaxTrivia](#) é usado para descrever todos os tipos de desafios.

Como os desafios não fazem parte da sintaxe de linguagem normal e podem aparecer em qualquer lugar entre dois tokens quaisquer, eles não são incluídos na árvore de sintaxe como um filho de um nó. Apesar disso, como eles são importantes ao implementar um recurso como refatoração e para manter fidelidade total com o texto de origem, eles existem como parte da árvore de sintaxe.

Acesse os desafios inspecionando as coleções [SyntaxToken.LeadingTrivia](#) ou [SyntaxToken.TrailingTrivia](#) de um token. Quando o texto de origem é analisado,

sequências de desafios são associadas aos tokens. Em geral, um token possui qualquer desafio após ele na mesma linha até o próximo token. Qualquer desafio após essa linha é associado ao próximo token. O primeiro token no arquivo de origem obtém todos os desafios iniciais e a última sequência de desafios no arquivo é anexada ao token de fim do arquivo, que, de outro modo, tem largura zero.

Ao contrário dos nós e tokens de sintaxe, os desafios de sintaxe não têm pais. Apesar disso, como eles fazem parte da árvore e cada um deles é associado um único token, você poderá acessar o token ao qual ele está associado usando a propriedade [SyntaxTrivia.Token](#).

## Intervalos

Cada nó, token ou desafio conhece sua posição dentro do texto de origem e o número de caracteres no qual ele consiste. Uma posição de texto é representada como um inteiro de 32 bits, que é um índice `char` baseado em zero. Um objeto [TextSpan](#) é a posição inicial e uma contagem de caracteres, ambas representadas como inteiros. Se [TextSpan](#) tem comprimento zero, ele se refere a um local entre dois caracteres.

Cada nó tem duas propriedades [TextSpan](#): [Span](#) e [FullSpan](#).

A propriedade [Span](#) é o intervalo de texto do início do primeiro token na subárvore do nó ao final do último token. Esse intervalo não inclui nenhum desafio à esquerda ou à direita.

A propriedade [FullSpan](#) é o intervalo de texto que inclui o intervalo normal do nó mais o intervalo de qualquer desafio à esquerda ou à direita.

Por exemplo:

C#

```
if (x > 3)
{
||   // this is bad
|throw new Exception("Not right.");| // better exception? ||
}
```

O nó de instrução dentro do bloco tem um intervalo indicado pelas barras verticais simples (`|`). Ele inclui os caracteres `throw new Exception("Not right.");`. O intervalo total é indicado pelas barras verticais duplas (`||`). Ele inclui os mesmos caracteres do intervalo e os caracteres associados ao desafio à esquerda e à direita.

# Variantes

Cada nó, token ou desafio tem uma propriedade `SyntaxNode.RawKind`, do tipo `System.Int32`, que identifica o elemento de sintaxe exato representado. Esse valor pode ser convertido em uma enumeração específica a idioma. Cada linguagem, C# ou Visual Basic, tem uma única enumeração `SyntaxKind` ([Microsoft.CodeAnalysis.CSharp.SyntaxKind](#) e [Microsoft.CodeAnalysis.VisualBasic.SyntaxKind](#), respectivamente), que lista todos os possíveis nós, tokens e elementos de trívia na gramática. Esta conversão pode ser feita automaticamente acessando os métodos de extensão `CSharpExtensions.Kind` ou `VisualBasicExtensions.Kind`.

A propriedade `RawKind` permite a desambiguidade fácil de tipos de nó de sintaxe que compartilham a mesma classe de nó. Para tokens e desafios, essa propriedade é a única maneira de diferenciar um tipo de elemento de outro.

Por exemplo, uma única classe `BinaryExpressionSyntax` tem `Left`, `OperatorToken` e `Right` como filhos. A propriedade `Kind` distingue se ela é um tipo `AddExpression`, `SubtractExpression` ou `MultiplyExpression` de nó de sintaxe.

## 💡 Dica

É recomendado verificar os tipos usando os métodos de extensão `IsKind` (para C#) ou `IsKind` (para VB).

# Errors

Mesmo quando o texto de origem contém erros de sintaxe, uma árvore de sintaxe completa com ida e volta para a origem é exposta. Quando o analisador encontra um código que não está em conformidade com a sintaxe definida da linguagem, ele usa uma das duas técnicas para criar uma árvore de sintaxe:

- Se o analisador espera determinado tipo de token, mas não o encontra, ele pode inserir um token ausente na árvore de sintaxe no local em que o token era esperado. Um token ausente representa o token real que era esperado, mas tem um intervalo vazio e sua propriedade `SyntaxNode.IsMissing` retorna `true`.
- O analisador pode ignorar tokens até encontrar um no qual pode continuar a análise. Nesse caso, os tokens ignorados são anexados como um nó de desafio com o tipo `SkippedTokensTrivia`.

# Trabalhar com semântica

Artigo • 10/05/2023

As [árvores de sintaxe](#) representam a estrutura lexical e sintática do código-fonte. Embora essas informações apenas sejam suficientes para descrever todas as declarações e a lógica na fonte, não são informações suficientes para identificar o que está sendo referenciado. Um nome pode representar:

- um tipo
- um campo
- um método
- uma variável local

Embora cada um deles seja exclusivamente diferente, determinar a qual deles um identificador, de fato, se refere exige uma compreensão profunda das regras da linguagem.

Há elementos do programa representados no código-fonte e os programas também podem se referir a bibliotecas compiladas anteriormente, empacotadas em arquivos do assembly. Embora nenhum código-fonte e, portanto, nenhum nó ou árvore de sintaxe, esteja disponível para assemblies, os programas ainda podem se referir a elementos contidos neles.

Para essas tarefas, é necessário usar o **Modelo semântico**.

Além de um modelo sintático do código-fonte, um modelo semântico encapsula as regras da linguagem, fornecendo uma maneira fácil para fazer a correspondência correta de identificadores com o elemento de programa correto que está sendo referenciado.

## Compilação

Uma compilação é uma representação de tudo o que é necessário para compilar um programa do C# ou Visual Basic, que inclui todas as referências de assembly, opções do compilador e arquivos de origem.

Como todas essas informações estão em um só lugar, os elementos contidos no código-fonte podem ser descritos mais detalhadamente. A compilação representa cada tipo, membro ou variável declarada como um símbolo. A compilação contém uma variedade de métodos que ajudam você encontrar e identificar os símbolos que foram declarados no código-fonte ou importados como metadados de um assembly.

Semelhantes às árvores de sintaxe, as compilações são imutáveis. Depois de criar uma compilação, ela não pode ser alterada por você ou por outra pessoa com quem você pode está compartilhando. No entanto, é possível criar uma nova compilação com base em uma compilação existente, especificando uma alteração conforme ela é feita. Por exemplo, é possível criar uma compilação que é igual em todos os aspectos a uma compilação existente, com exceção de que ela pode incluir um arquivo de origem ou uma referência de assembly adicional.

## Símbolos

Um símbolo representa um elemento distinto declarado pelo código-fonte ou importado de um assembly como metadados. Cada namespace, tipo, método, propriedade, campo, evento, parâmetro ou variável local é representado por um símbolo.

Uma variedade de métodos e propriedades no tipo [Compilation](#) ajudam você a encontrar símbolos. Por exemplo, encontre um símbolo para um tipo declarado pelo seu nome comum de metadados. Também acesse a tabela inteira de símbolos como uma árvore de símbolos com raiz no namespace global.

Os símbolos também contêm informações adicionais que o compilador determina da fonte ou dos metadados, como outros símbolos referenciados. Cada tipo de símbolo é representado por uma interface separada derivada de [ISymbol](#), cada um com seus próprios métodos e propriedades que fornecem detalhes das informações reunidas pelo compilador. Muitas dessas propriedades referenciam outros símbolos diretamente. Por exemplo, a propriedade [IMethodSymbol.ReturnType](#) indica o símbolo de tipo real que o método retorna.

Os símbolos apresentam uma representação comum de namespaces, tipos e membros, entre o código-fonte e os metadados. Por exemplo, um método que foi declarado no código-fonte e um método que foi importado dos metadados são representados por um [IMethodSymbol](#) com as mesmas propriedades.

Símbolos são semelhantes em conceito ao sistema de tipos CLR, conforme representado pela API [System.Reflection](#), mas são mais sofisticados pois modelam mais do que apenas tipos. Namespaces, variáveis locais e rótulos são todos símbolos. Além disso, os símbolos são uma representação dos conceitos da linguagem, não dos conceitos do CLR. Há muita sobreposição, mas há muitas diferenças significativas também. Por exemplo, um método iterador no C# ou Visual Basic é um único símbolo. No entanto, quando o método iterador é convertido em metadados CLR, ele é um tipo e vários métodos.

# Modelo semântico

Um modelo semântico representa todas as informações semânticas de um único arquivo de origem. Use-o para descobrir o seguinte:

- Os símbolos referenciados em um local específico na fonte.
- O tipo resultante de qualquer expressão.
- Todo o diagnóstico, que são erros e avisos.
- Como as variáveis fluem bidirecionalmente entre as regiões de origem.
- As respostas a perguntas mais especulativas.

# Trabalhar com um workspace

Artigo • 10/05/2023

A camada **Workspaces** é o ponto inicial para fazer a análise de código e refatoração em soluções inteiras. Nessa camada, a API do Workspace ajudará você a organizar todas as informações sobre os projetos de uma solução em um único modelo de objeto, oferecendo acesso direto aos modelos de objeto da camada do compilador como texto de origem, árvores de sintaxe, modelos semânticos e compilações, sem a necessidade de analisar arquivos, configurar opções ou gerenciar dependências entre projetos.

Ambientes de host, como um IDE, fornecem um workspace para você correspondente à solução aberta. Também é possível usar esse modelo fora de um IDE apenas carregando um arquivo de solução.

## Workspace

Um workspace é uma representação ativa da solução como uma coleção de projetos, cada uma com uma coleção de documentos. Um workspace normalmente é vinculado a um ambiente de host que está sendo modificado constantemente conforme um usuário digita ou manipula propriedades.

O [Workspace](#) fornece acesso ao modelo atual da solução. Quando ocorre uma alteração no ambiente de host, o workspace aciona eventos correspondentes e a propriedade [Workspace.CurrentSolution](#) é atualizada. Por exemplo, quando o usuário digita em um editor de texto algo correspondente a um dos documentos de origem, o workspace usa um evento para sinalizar que o modelo geral da solução foi alterado e qual documento foi modificado. Em seguida, você pode reagir a essas alterações analisando o novo modelo quanto à exatidão, realçando áreas de significância ou fazendo uma sugestão de alteração de código.

Também pode criar workspaces independentes desconectados do ambiente de host ou usados em um aplicativo que não tem nenhum ambiente de host.

## Soluções, projetos e documentos

Embora um workspace possa ser alterado sempre que uma tecla é pressionada, você pode trabalhar com o modelo da solução de forma isolada.

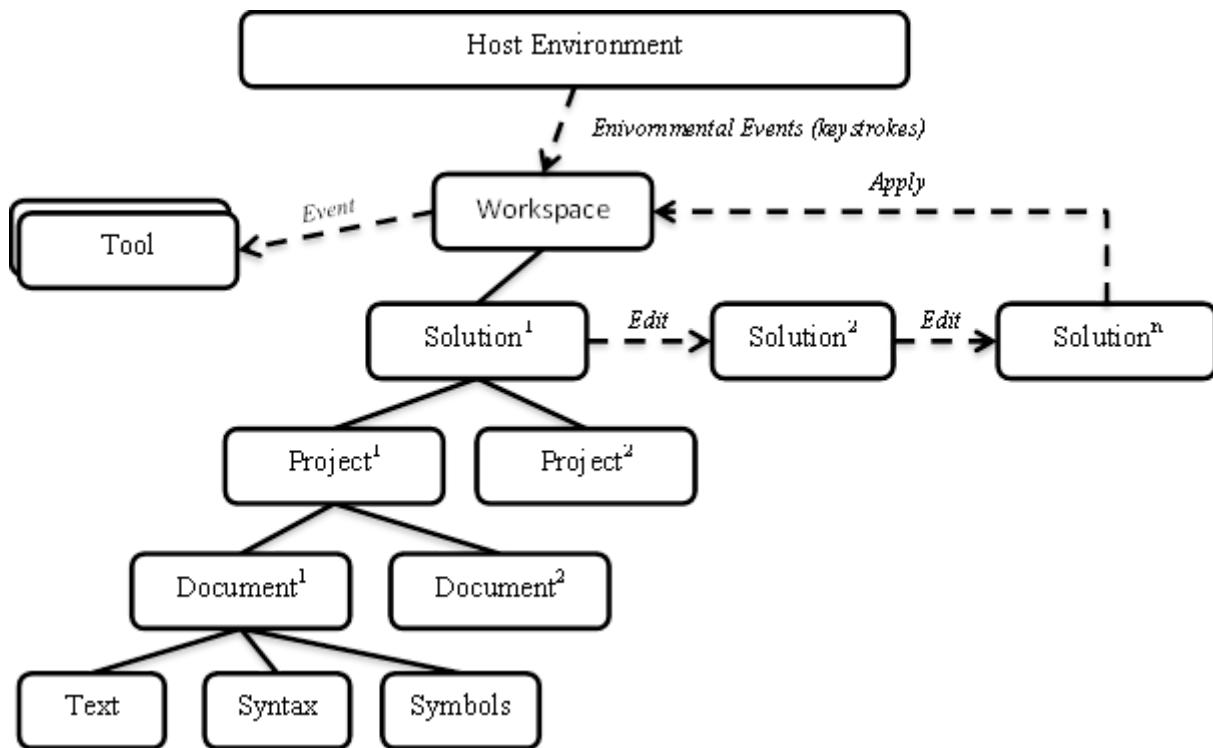
Uma solução é um modelo imutável dos projetos e documentos. Isso significa que o modelo pode ser compartilhado sem bloqueio nem duplicação. Depois de obter uma

instância da solução da propriedade `Workspace.CurrentSolution`, essa instância nunca será alterada. No entanto, assim como árvores de sintaxe e compilações, você pode modificar soluções construindo novas instâncias com base em soluções existentes e alterações específicas. Para fazer com que o workspace reflita as alterações, é necessário aplicar explicitamente a solução alterada novamente ao workspace.

Um projeto é uma parte do modelo de solução imutável geral. Ele representa todos os documentos do código-fonte, opções de análise e compilação e referências de assembly e projeto a projeto. Em um projeto, você pode acessar a compilação correspondente sem a necessidade de determinar as dependências de projeto nem analisar arquivos de origem.

Um documento também é uma parte do modelo de solução imutável geral. Um documento representa um único arquivo de origem do qual você pode acessar o texto do arquivo, a árvore de sintaxe e o modelo semântico.

O diagrama a seguir é uma representação de como o Workspace se relaciona ao ambiente de host, às ferramentas e ao modo como as edições são feitas.



## Resumo

O Roslyn expõe um conjunto de APIs do compilador e APIs dos Workspaces que fornecem informações detalhadas sobre o código-fonte e que têm fidelidade total com as linguagens C# e Visual Basic. O SDK do .NET Compiler Platform diminui drasticamente a barreira de entrada para a criação de aplicativos e ferramentas voltadas para o código. Ele cria várias oportunidades para inovação em áreas como

metaprogramação, geração e transformação de código, uso interativo das linguagens C# e Visual Basic e incorporação de C# e Visual Basic em linguagens específicas de domínio.

# Explorar código com o visualizador de sintaxe Roslyn no Visual Studio

Artigo • 10/05/2023

Este artigo oferece uma visão geral sobre a ferramenta de Visualizador de sintaxe que é fornecida como parte do SDK do .NET Compiler Platform ("Roslyn"). O Visualizador de sintaxe é uma janela de ferramentas que ajuda a inspecionar e explorar árvores de sintaxe. É uma ferramenta essencial para compreender os modelos do código que você deseja analisar. Também é um auxílio para depuração ao desenvolver seus próprios aplicativos usando o SDK do .NET Compiler Platform ("Roslyn"). Abra essa ferramenta ao criar seus primeiros analisadores. O visualizador ajuda você a entender os modelos usados pelas APIs. Você também pode usar ferramentas como [SharpLab](#) ou [LINQPad](#) para inspecionar o código e entender as árvores de sintaxe.

## Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o **SDK da .NET Compiler Platform** no **Instalador do Visual Studio**:

### Instalar usando o Instalador do Visual Studio – exibição de cargas de trabalho

O SDK da .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento da extensão do Visual Studio. É necessário selecioná-lo como um componente opcional.

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Marque a carga de trabalho de **Desenvolvimento de extensão do Visual Studio**.
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará por último nos componentes opcionais.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

# Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará na parte superior, na seção **Compiladores, ferramentas de compilação e runtimes**.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

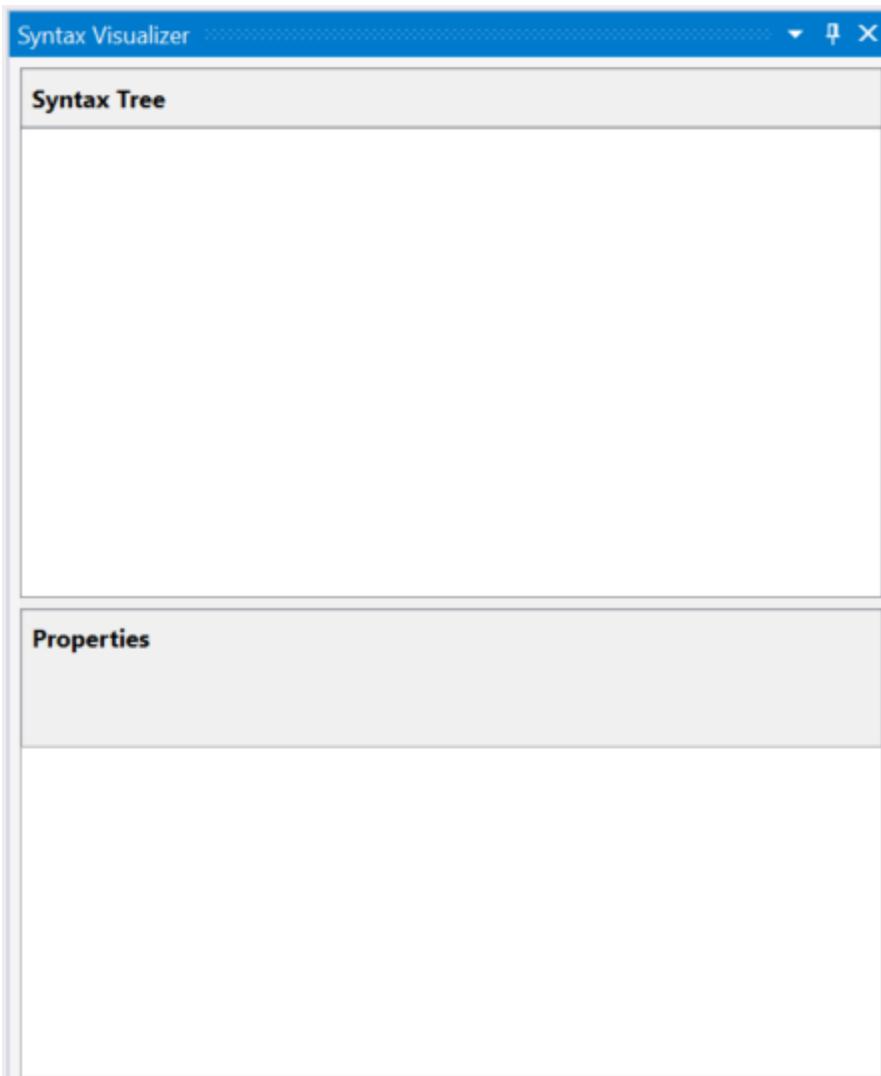
1. Marque a caixa do **Editor DGML**. Você a encontrará na seção **Ferramentas de código**.

Familiarize-se com os conceitos usados no SDK do.NET Compiler Platform lendo o artigo de [visão geral](#). Ele fornece uma introdução a árvores de sintaxe, nós, tokens e trívia.

## Visualizador de sintaxe

O **Visualizador de sintaxe** permite a inspeção da árvore de sintaxe de arquivo de código C# ou Visual Basic na janela do editor atualmente ativa dentro do IDE do Visual Studio. O visualizador pode ser iniciado ao clicar em **Exibir>Outras Janelas>Visualizador de sintaxe**. Você também pode usar a barra de ferramentas de **Início Rápido** no canto superior direito. Digite "sintaxe" e o comando para abrir o **Visualizador de sintaxe** deverá aparecer.

Este comando abre o Visualizador de sintaxe como uma janela de ferramentas flutuante. Se você não tiver uma janela de editor de código aberta, a exibição ficará em branco, conforme mostrado na figura a seguir.

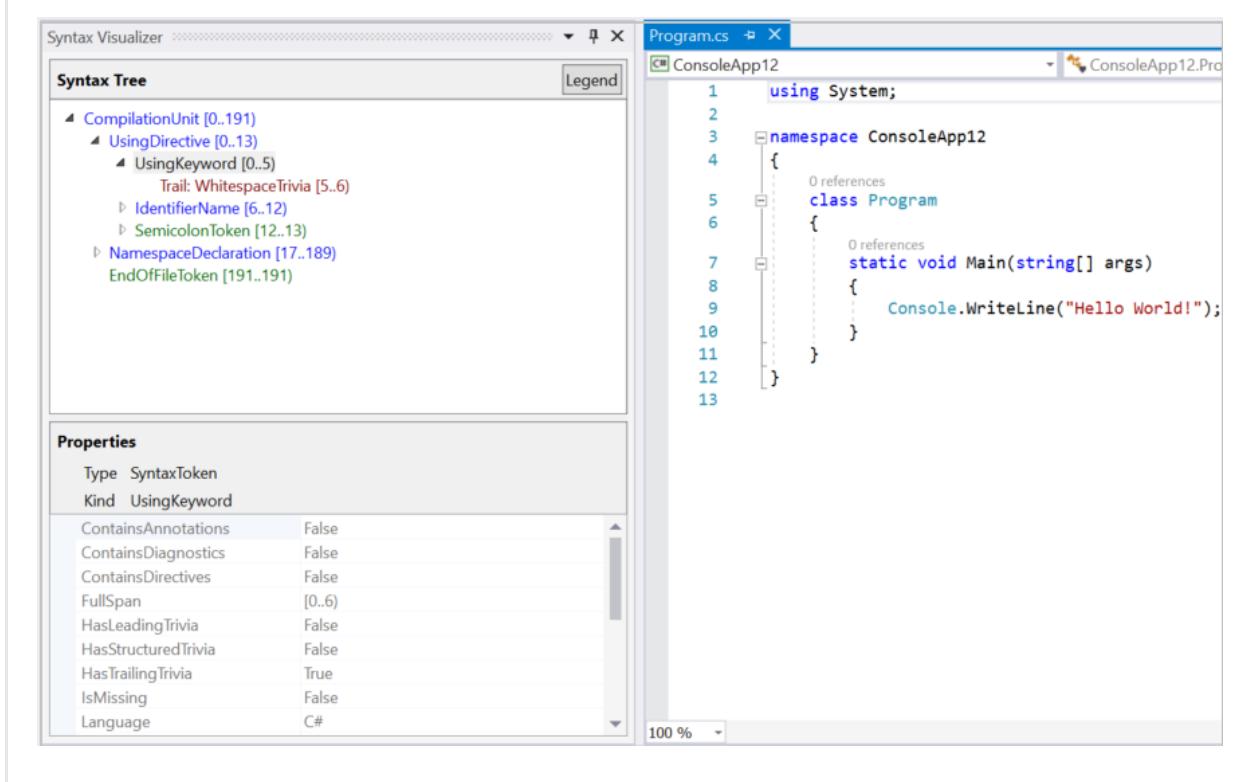


Encaixe esta janela de ferramentas em um local conveniente dentro do Visual Studio, como o lado esquerdo. O Visualizador mostra informações sobre o arquivo de código atual.

Crie um novo projeto usando o comando **Arquivo > Novo Projeto**. Você pode criar um projeto Visual Basic ou C#. Quando o Visual Studio abre o arquivo de código principal deste projeto, o visualizador exibe a árvore de sintaxe dele. Você pode abrir qualquer arquivo de C# ou Visual Basic existente nesta instância do Visual Studio e o visualizador exibirá a árvore de sintaxe do arquivo correspondente. Se você tiver vários arquivos de código abertos no Visual Studio, o visualizador exibirá a árvore de sintaxe do arquivo de código atualmente ativo, (o arquivo de código que tem o foco do teclado).

---

C#



Conforme mostrado nas imagens acima, a janela de ferramentas do visualizador exibe a árvore de sintaxe na parte superior e uma grade de propriedade na parte inferior. A grade de propriedade exibe as propriedades do item atualmente selecionado na árvore, incluindo *Type* e *Kind* (SyntaxKind) do .NET do item.

As árvores de sintaxe incluem três tipos de itens: *nós*, *tokens* e *trívia*. Você pode ler mais sobre esses tipos no artigo [Trabalhar com sintaxe](#). Os itens de cada tipo estão representados com cores diferentes. Clique no botão "Legenda" para uma visão geral das cores usadas.

Cada item da árvore também exibe sua própria **extensão**. A extensão é composta pelos índices (a posição inicial e final) do nó no arquivo de texto. No exemplo de C# anterior, o token "UsingKeyword [0..5)" selecionado tem uma **abrangência** de cinco caracteres de largura, [0..5). A notação "[..)" significa que o índice inicial faz parte da extensão, mas o índice final não.

Há duas maneiras de navegar na árvore:

- Expandir ou clicar em itens na árvore. O visualizador seleciona automaticamente o texto correspondente à extensão do item no editor de código.
- Clicar ou selecionar texto no editor de código. No exemplo anterior do Visual Basic, se você seleciona a linha que contém "Module Module1" no editor de código, o visualizador navega automaticamente até o nó `ModuleStatement` correspondente na árvore.

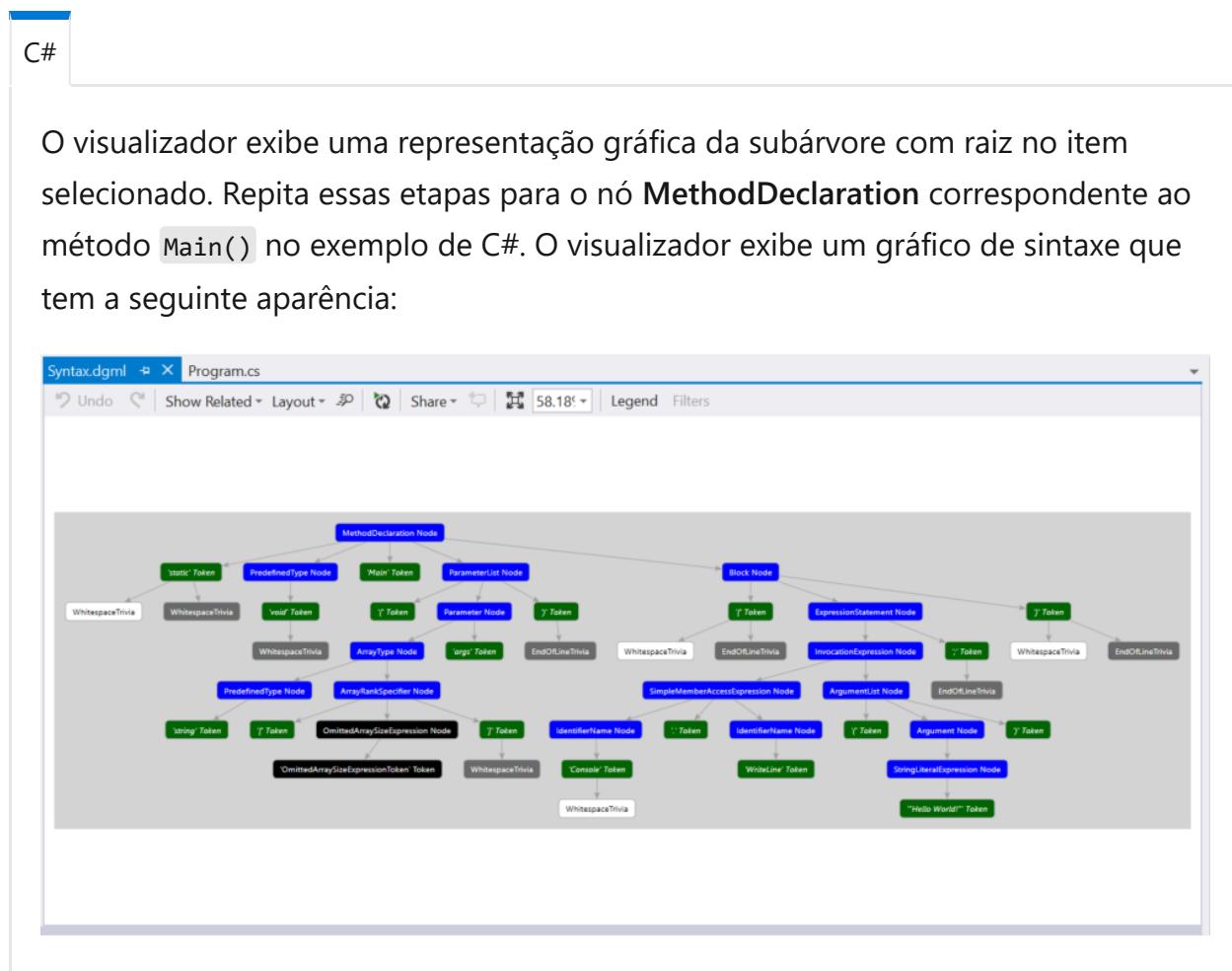
O visualizador realça o item da árvore cuja extensão melhor corresponda com a extensão do texto selecionado no editor.

O visualizador atualiza a árvore para corresponder às modificações no arquivo de código ativo. Adicione uma chamada a `Console.WriteLine()` dentro de `Main()`. Conforme você digita, o visualizador atualiza a árvore.

Pare de digitar quando já tiver digitado `Console..`. A árvore tem alguns itens coloridos em rosa. Neste ponto, há erros (também conhecidos como "Diagnóstico") no código digitado. Esses erros são anexados a nós, tokens e trávia na árvore de sintaxe. O visualizador mostra quais itens têm erros anexados a eles, realçando a tela de fundo em rosa. Você pode inspecionar os erros em qualquer item colorido em rosa passando o mouse sobre o item. O visualizador exibe somente erros sintáticos (os erros relacionados à sintaxe do código digitado); erros semânticos não são exibidos.

## Gráficos de sintaxe

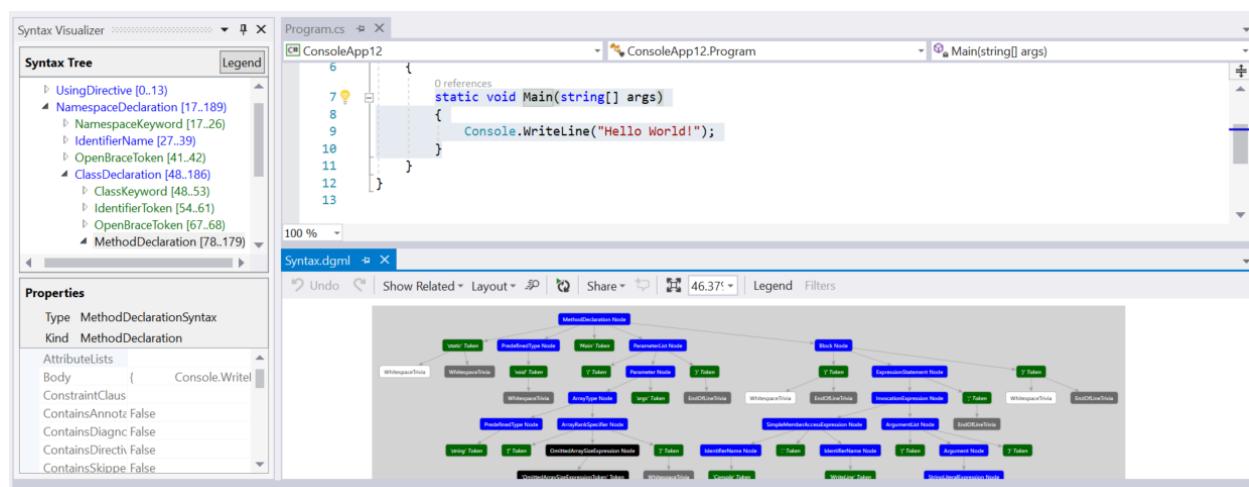
Clique com o botão direito do mouse em qualquer item da árvore e clique em **Exibir gráfico de sintaxe direcionado**.



O visualizador de gráfico de sintaxe tem uma opção para exibir uma legenda para seu esquema de cores. Você também pode passar o mouse sobre itens específicos no gráfico de sintaxe para exibir as respectivas propriedades.

Você pode exibir gráficos de sintaxe de itens diferentes da árvore repetidamente e os gráficos serão sempre exibidos na mesma janela no Visual Studio. Você pode encaixar essa janela em um local conveniente no Visual Studio para não precisar alternar entre as guias para exibir um novo gráfico de sintaxe. A parte inferior, abaixo das janelas do editor de código, geralmente é conveniente.

Veja o layout de encaixe para usar com a janela de ferramentas do visualizador e a janela do gráfico de sintaxe:



Outra opção é colocar a janela de gráfico de sintaxe em um segundo monitor, em uma configuração de dois monitores.

## Inspecionando semântica

O Visualizador de sintaxe possibilita uma inspeção rudimentar de símbolos e informações semânticas. Digite `double x = 1 + 1;` dentro de `Main()` no exemplo de C#. Em seguida, selecione a expressão `1 + 1` na janela do editor de código. O visualizador realça o nó **AddExpression** no visualizador. Clique com o botão direito do mouse nesse **AddExpression** e clique em **Exibir Symbol (se houver)**. Observe que a maioria dos itens de menu tem o qualificador "se houver". O Visualizador de sintaxe inspeciona as propriedades de um Nó, incluindo propriedades que podem não estar presentes em todos os nós.

A grade de propriedade do visualizador é atualizada conforme mostrado na figura a seguir: o símbolo da expressão é um **SynthesizedIntrinsicOperatorSymbol** com **Kind = Method**.

The screenshot shows the Syntax Visualizer window with the following details:

- Syntax Tree:** Shows the hierarchical structure of the C# code. The `AddExpression` node is selected.
- Properties:**
  - Type: `SynthesizedIntrinsicOperatorSymbol`
  - Kind: `Method`
  - Name: `op_Addition`
  - OriginalDefinition: `int.operator +(int, int)`
  - OverriddenMethod: None
  - Parameters: `(Collection)`
  - PartialDefinitionPart: None
  - PartialImplementation: None
  - ReceiverType: `int`

Red arrows point from the text "Exiba o TypeSymbol para o nó AddExpression" to the "Type", "Kind", and "Name" fields in the Properties panel.

```

static void Main(string[] args)
{
    double x = 1 + 1;
}

```

Experimente Exibir TypeSymbol (se houver) para o mesmo nó AddExpression. A grade de propriedade no visualizador é atualizada, conforme mostrado na figura a seguir, indicando que o tipo da expressão selecionada é `Int32`.

The screenshot shows the Syntax Visualizer window with the following details:

- Syntax Tree:** Shows the hierarchical structure of the C# code. The `AddExpression` node is selected.
- Properties:**
  - Type: `PENamedTypeSymbolNonGeneric`
  - Kind: `NamedType`
  - Name: `Int32`
  - OriginalDefinition: `int`
  - SpecialType: `System_Int32`
  - StaticConstructors: `(Collection)`
  - TupleElementName: `(Collection)`
  - TupleElements: `(Collection)`
  - TupleElementTypes: `(Collection)`

Red arrows point from the text "Exiba o TypeSymbol Convertido (se houver) para o mesmo nó AddExpression" to the "Type", "Kind", and "Name" fields in the Properties panel.

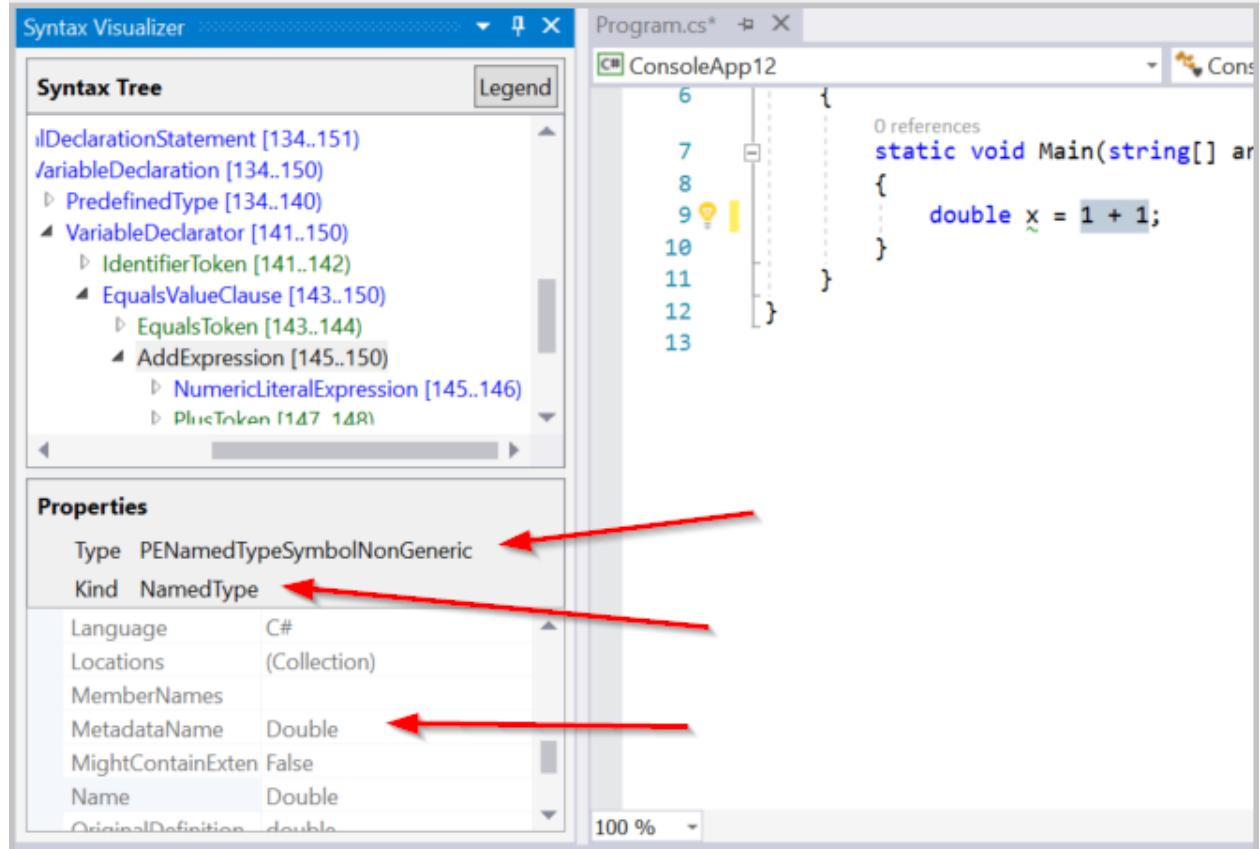
```

static void Main(string[] args)
{
    double x = 1 + 1;
}

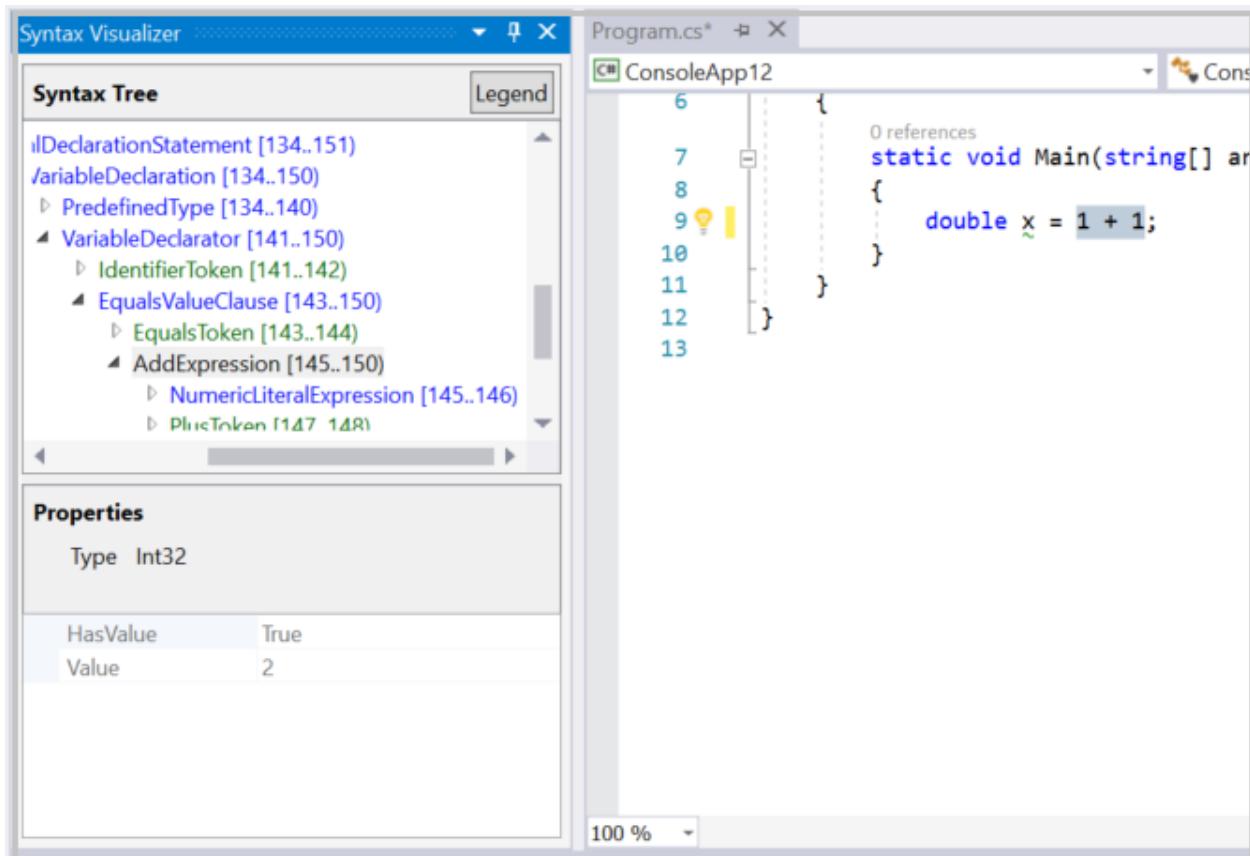
```

Experimente Exibir TypeSymbol Convertido (se houver) para o mesmo nó AddExpression. A grade de propriedade é atualizada, indicando que, embora o tipo da

expressão seja `Int32`, o tipo convertido da expressão é `Double`, conforme mostrado na figura a seguir. Esse nó inclui informações de símbolo de tipo convertido porque a expressão `Int32` ocorre em um contexto em que deve ser convertida em um `Double`. Essa conversão satisfaz o tipo `Double` especificado para a variável `x` no lado esquerdo do operador de atribuição.



Por fim, experimente **Exibir Valor Constante (se houver)** para o mesmo nó **AddExpression**. A grade de propriedade mostra que o valor da expressão é uma constante de tempo de compilação com o valor `2`.



O exemplo anterior também pode ser replicado no Visual Basic. Digite `Dim x As Double = 1 + 1` em um arquivo do Visual Basic. Selecione a expressão `1 + 1` na janela do editor de código. O visualizador realça o nó **AddExpression** correspondente no visualizador. Repita as etapas anteriores para esta **AddExpression** e você verá resultados idênticos.

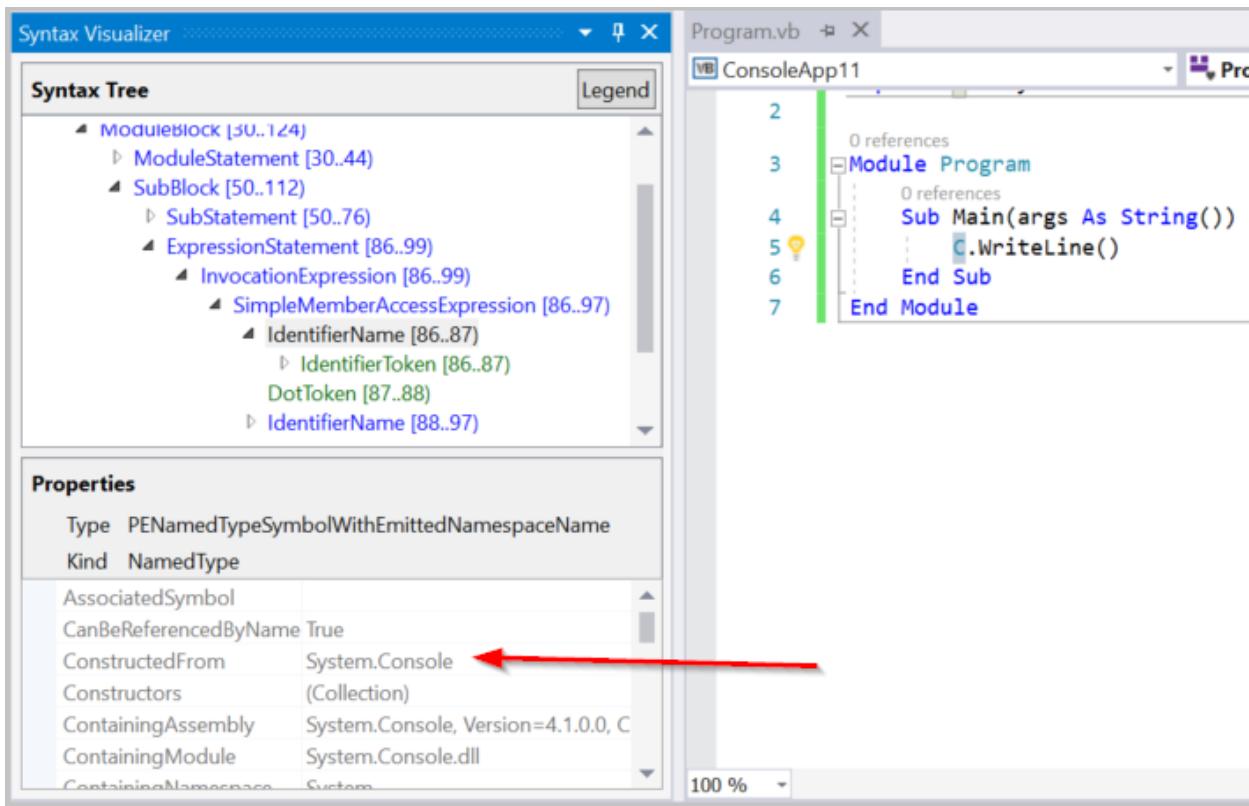
Examine mais código no Visual Basic. Atualize seu arquivo principal do Visual Basic com o seguinte código:

```
VB

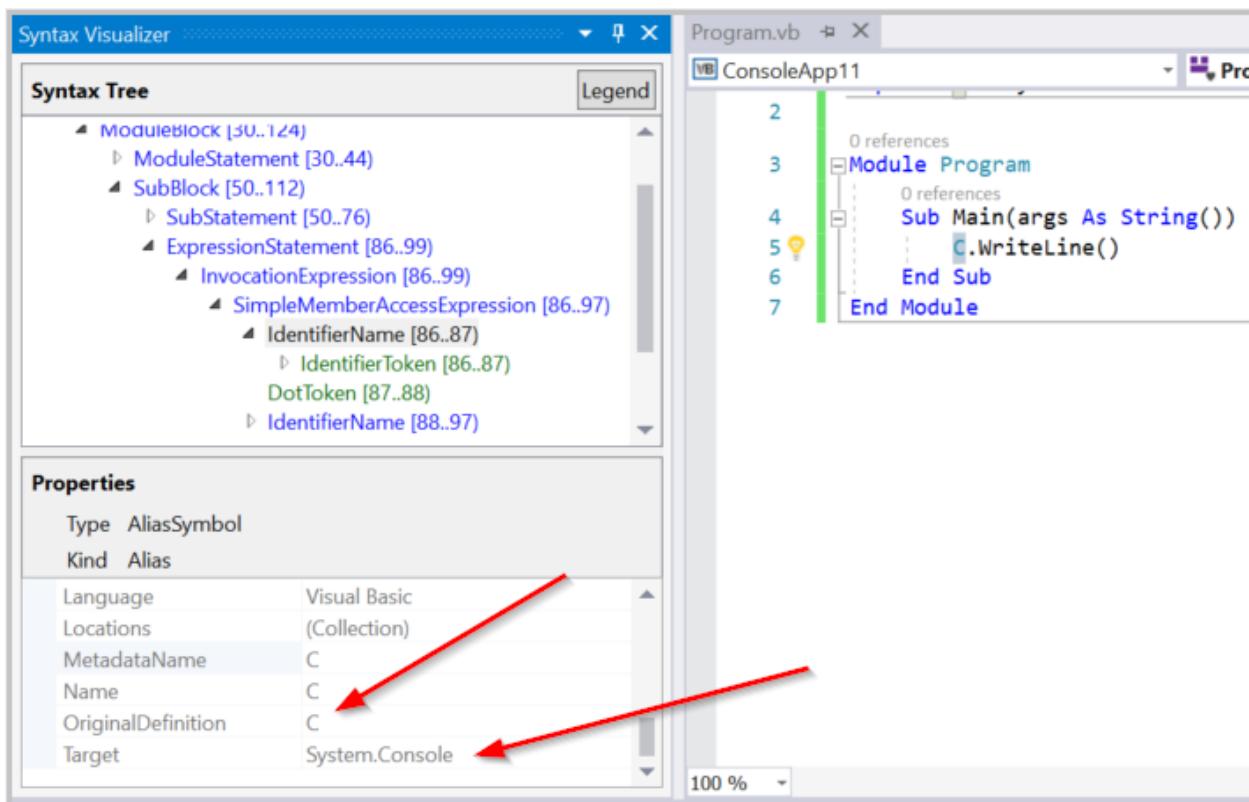
Imports C = System.Console

Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

Esse código introduz um alias chamado `C` que mapeia para o tipo `System.Console` na parte superior do arquivo e usa esse alias em `Main()`. Selecione o uso desse alias, o `C` em `C.WriteLine()`, dentro do método `Main()`. O visualizador seleciona o nó **IdentifierName** correspondente no visualizador. Clique com botão direito do mouse nesse nó e clique em **Exibir Symbol** (se houver). A grade de propriedade mostra que esse identificador é associado com o tipo `System.Console`, conforme mostrado na figura a seguir:

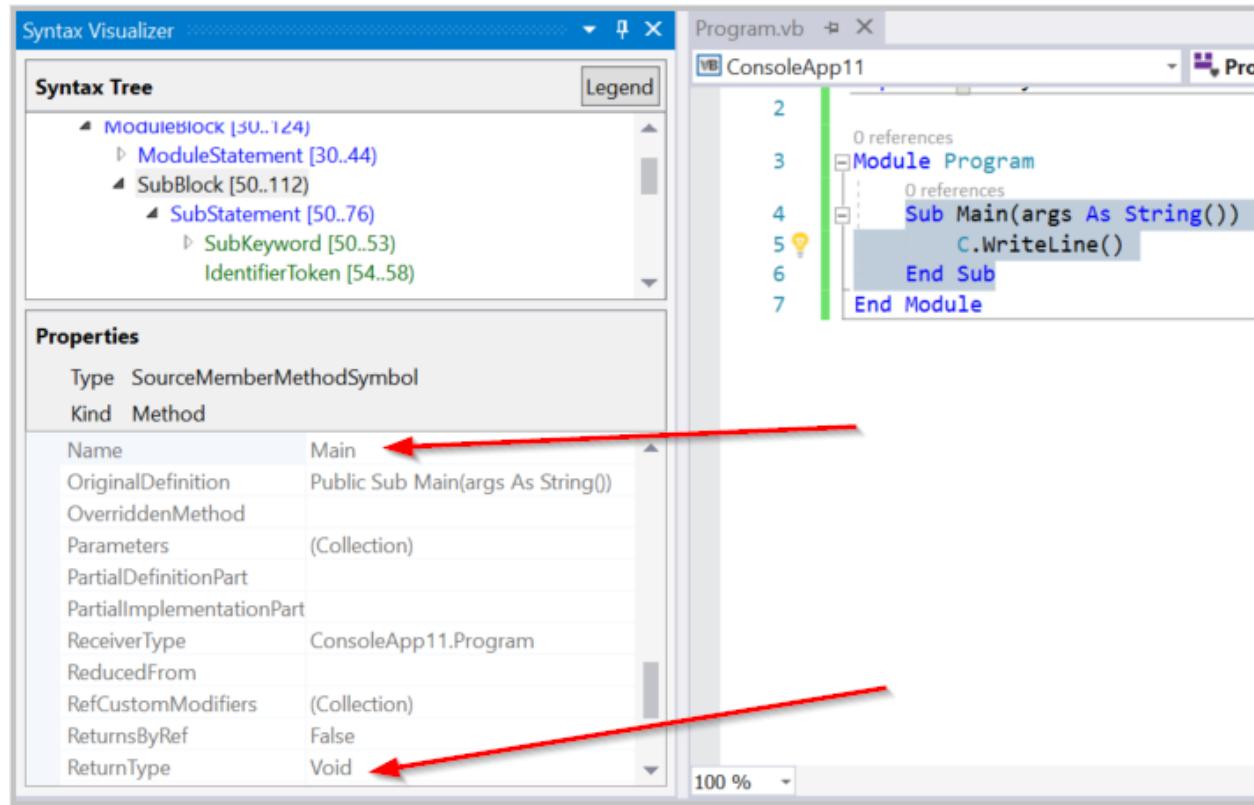


Experimente Exibir AliasSymbol (se houver) para o mesmo nó IdentifierName. A grade de propriedade mostra que o identificador é um alias com nome `C`, que é associado com o destino `System.Console`. Em outras palavras, a grade de propriedade fornece informações sobre o AliasSymbol correspondente ao identificador `C`.



Inspecione o símbolo correspondente a qualquer tipo, método e propriedade declarados. Selecione o nó correspondente no visualizador e clique em Exibir Symbol

(se houver). Selecione o método `Sub Main()`, incluindo o corpo do método. Clique em **Exibir Symbol (se houver)** para o nó **SubBlock** correspondente no visualizador. A grade de propriedade mostra que o **MethodSymbol** deste **SubBlock** tem nome `Main` com o tipo de retorno `Void`.



Os exemplos de Visual Basic acima podem ser facilmente replicados em C#. Digite `using C = System.Console;` no lugar de `Imports C = System.Console` para o alias. As etapas anteriores em C# geram resultados idênticos na janela do visualizador.

As operações de inspeção semântica estão disponíveis somente em nós. Elas não estão disponíveis em tokens nem trávia. Nem todos os nós têm informações semânticas interessantes para inspecionar. Quando um nó não tem informações semânticas interessantes, clicar em **Exibir \* Symbol (se houver)** mostrará uma grade de propriedade em branco.

Você pode ler mais sobre as APIs para executar análise semântica no documento de visão geral [Trabalhar com semântica](#).

## Fechando o visualizador de sintaxe

Você pode fechar a janela do visualizador quando não a estiver usando para examinar o código-fonte. O visualizador de sintaxe atualiza a exibição enquanto você navega pelo código, editando e alterando a origem. Ele poderá se tornar uma distração quando você não estiver usando.

# Geradores de origem

Artigo • 25/06/2023

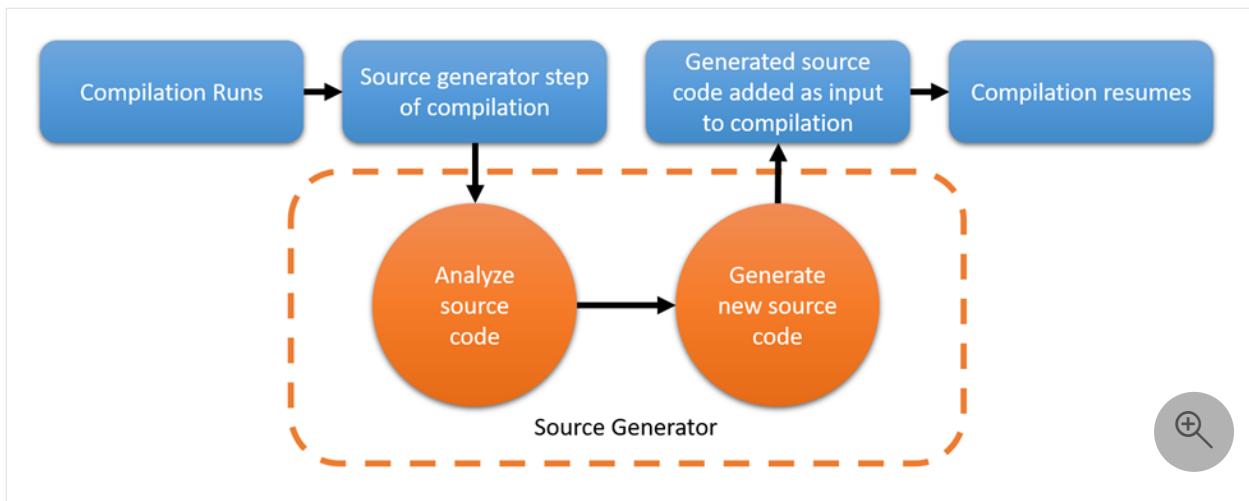
Este artigo oferece uma visão geral sobre os Geradores de Origem que são fornecidos como parte do SDK do .NET Compiler Platform ("Roslyn"). Os Geradores de Origem permitem que os desenvolvedores do C# inspecionem o código do usuário enquanto ele está sendo compilado. O gerador pode criar novos arquivos de origem C# em tempo real que são adicionados à compilação do usuário. Dessa forma, você tem um código que é executado durante a compilação. Ele inspeciona seu programa para produzir arquivos de origem adicionais que são compilados junto com o restante do código.

Um Gerador de Origem é um novo tipo de componente que os desenvolvedores do C# podem gravar que permite que você faça duas coisas importantes:

1. Recupere um *objeto de compilação* que representa todo o código do usuário que está sendo compilado. Esse objeto pode ser inspecionado e você pode gravar um código que funciona com a sintaxe e modelos semânticos para o código que está sendo compilado, assim como nos analisadores de hoje.
2. Gere arquivos de origem C# que podem ser adicionados a um objeto de compilação durante a compilação. Em outras palavras, você pode fornecer o código-fonte adicional como entrada para uma compilação enquanto o código está sendo compilado.

Quando combinadas, essas duas coisas são o que tornam os Geradores de Origem tão úteis. Você pode inspecionar o código do usuário com todos os metadados avançados que o compilador acumula durante a compilação. Em seguida, o gerador emite o código C# de volta na mesma compilação baseada nos dados analisados. Se você estiver familiarizado com os Analisadores Roslyn, poderá considerar os Geradores de Origem como analisadores que podem emitir código-fonte C#.

Os geradores de origem são executados como uma fase de compilação visualizada abaixo:



Um Gerador de Origem é um assembly .NET Standard 2.0 que é carregado pelo compilador junto com todos os analisadores. Pode ser usado em ambientes em que os componentes do .NET Standard podem ser carregados e executados.

### ⓘ Importante

Atualmente, somente assemblies .NET Standard 2.0 podem ser usados como Geradores de Origem.

## Cenários comuns

Há três abordagens gerais para inspecionar o código do usuário e gerar informações ou código com base nessa análise usada por tecnologias atualmente:

- Reflexão de runtime.
- Fazendo malabarismo com tarefas do MSBuild.
- Introdução de IL (Linguagem Intermediária) (não discutida neste artigo).

Os Geradores de Origem podem ser uma melhoria em cada abordagem.

## Reflexão de runtime

A reflexão de runtime é uma tecnologia poderosa que foi adicionada ao .NET há muito tempo. Há inúmeros cenários para usá-lo. Um cenário comum é executar algumas análises do código do usuário quando um aplicativo for iniciado e usar esses dados para gerar coisas.

Por exemplo, o ASP.NET Core usa a reflexão quando seu serviço Web é executado pela primeira vez para descobrir construções que você definiu para que ele possa "conectar" coisas como controladores e páginas do Razor. Embora isso permita que você grave um

código simples com abstrações poderosas, ele vem com uma penalidade de desempenho no runtime: quando seu serviço Web ou aplicativo é iniciado pela primeira vez, ele não pode aceitar nenhuma solicitação até que todo o código de reflexão de runtime que descubra informações sobre seu código seja concluído em execução. Embora essa penalidade de desempenho não seja enorme, é um custo fixo que você não pode melhorar em seu próprio aplicativo.

Com um Gerador de Origem, a fase de descoberta do controlador da inicialização pode ocorrer no tempo de compilação. Um gerador pode analisar o código-fonte e emitir o código necessário para "conectar" seu aplicativo. O uso de geradores de origem pode resultar em alguns tempos de inicialização mais rápidos, uma vez que uma ação que acontece no tempo de execução hoje pode ser enviada por push para o tempo de compilação.

## Fazendo malabarismo com tarefas do MSBuild

Os Geradores de Origem também podem melhorar o desempenho de maneiras que não se limitam à reflexão no tempo de execução para descobrir tipos. Alguns cenários envolvem chamar a tarefa MSBuild C# (chamada CSC) várias vezes para que eles possam inspecionar os dados de uma compilação. Como você pode imaginar, chamar o compilador mais de uma vez afeta o tempo total necessário para criar seu aplicativo. Estamos investigando como os Geradores de Origem podem ser usados para evitar a necessidade de fazer malabarismo com tarefas do MSBuild como esta, já que os geradores de origem não oferecem apenas alguns benefícios de desempenho, mas também permitem que as ferramentas operem no nível certo de abstração.

Outro recurso que os Geradores de Origem podem oferecer é evitar o uso de algumas APIs "tipadas com cadeia de caracteres", como a forma como o roteamento do ASP.NET Core entre controladores e as páginas do Razor funcionam. Com um Gerador de Origem, o roteamento pode ser fortemente tipado com as cadeias de caracteres necessárias sendo geradas como um detalhe de tempo de compilação. Isso reduziria o número de vezes que um literal de cadeia de caracteres digitado incorretamente leva a uma solicitação que não atinge o controlador correto.

## Introdução aos geradores de origem

Neste guia, você explorará a criação de um gerador de origem usando a API [IIncrementalGenerator](#).

1. Crie um aplicativo de console .NET Core. Este exemplo usa o .NET 7.

2. Substitua a classe `Program` pelo código a seguir. O código a seguir não usa instruções de nível superior. O formulário clássico é necessário porque este primeiro gerador de origem grava um método parcial nessa classe `Program`:

```
C#  
  
namespace ConsoleApp;  
  
partial class Program  
{  
    static void Main(string[] args)  
    {  
        HelloFrom("Generated Code");  
    }  
  
    static partial void HelloFrom(string name);  
}
```

#### ⚠ Observação

Você pode executar este exemplo no estado em que se encontra, mas nada acontecerá ainda.

3. Em seguida, criaremos um projeto de gerador de origem que implementará o equivalente do método `partial void HelloFrom`.
4. Crie um projeto de biblioteca padrão do .NET direcionado ao TFM (Moniker da Estrutura de Destino) `netstandard2.0`. Adicione os pacotes NuGet `Microsoft.CodeAnalysis.Analyzers` e `Microsoft.CodeAnalysis.CSharp`:

```
XML  
  
<Project Sdk="Microsoft.NET.Sdk">  
  
    <PropertyGroup>  
        <TargetFramework>netstandard2.0</TargetFramework>  
        <LangVersion>11.0</LangVersion>  
        <EnforceExtendedAnalyzerRules>true</EnforceExtendedAnalyzerRules>  
    </PropertyGroup>  
  
    <ItemGroup>  
        <PackageReference Include="Microsoft.CodeAnalysis.CSharp"  
Version="4.6.0" PrivateAssets="all" />  
        <PackageReference Include="Microsoft.CodeAnalysis.Analyzers"  
Version="3.3.4" PrivateAssets="all" />  
    </ItemGroup>  
  
</Project>
```

### Dica

O projeto do gerador de origem precisa ser direcionado ao TFM `netstandard2.0`, caso contrário, ele não funcionará.

5. Crie um novo arquivo C# chamado `HelloSourceGenerator.cs` que especifica seu próprio Gerador de Origem da seguinte maneira:

C#

```
using Microsoft.CodeAnalysis;

namespace SourceGenerator;

public sealed class HelloSourceGenerator : IIncrementalGenerator
{
    public void Initialize(IncrementalGeneratorInitializationContext
context)
    {
        var compilationProvider = context.CompilationProvider;

        context.RegisterSourceOutput(
            compilationProvider,
            static (context, compilation) =>
        {
            // Code generation goes here
        });
    }
}
```

Um gerador de origem precisa implementar a interface `Microsoft.CodeAnalysis.IIncrementalGenerator` e ter o `Microsoft.CodeAnalysis.GeneratorAttribute`.

1. Substitua o conteúdo do método `IIncrementalGenerator.Initialize` pelo código a seguir:

C#

```
using Microsoft.CodeAnalysis;

namespace SourceGenerator;

[Generator]
public sealed class HelloSourceGenerator : IIncrementalGenerator
{
```

```

        public void Initialize(IncrementalGeneratorInitializationContext
context)
{
    var compilationIncrementalValue = context.CompilationProvider;

    context.RegisterSourceOutput(
        compilationIncrementalValue,
        static (context, compilation) =>
    {
        // Get the entry point method
        var mainMethod =
compilation.GetEntryPoint(context.CancellationToken);
        var typeName = mainMethod.ContainingType.Name;

        string source = $$"""
            // Auto-generated code
            namespace
{{mainMethod.ContainingNamespace.ToString()}};

        public static partial class {{typeName}}
{
            static partial void HelloFrom(string name) =>
                Console.WriteLine($"Generator says: Hi from
'{name}'");
        }
        """;

        // Add the source code to the compilation
        context.AddSource($"{typeName}.g.cs", source);
    });
}
}

```

A variável `compilationIncrementalValue` é usada para armazenar o valor de modelo incremental da compilação que é passada para o método `RegisterSourceOutput` de `context`. Com base no parâmetro `compilation`, podemos acessar o ponto de entrada das compilações ou o método `Main`. A instância `mainMethod` é um `IMethodSymbol`, e representa um método ou símbolo semelhante a um método (incluindo construtor, destruidor, operador ou acessador de propriedade/evento). O método `Microsoft.CodeAnalysis.Compilation.GetEntryPoint` retorna o `IMethodSymbol` do ponto de entrada do programa. Outros métodos permitem que você encontre qualquer símbolo de método em um projeto. Nesse objeto, podemos justificar o namespace contido (se houver um) e o tipo. O `source` neste exemplo é uma cadeia de caracteres interpolada que define o código-fonte a ser gerado, onde os espaços interpolados são preenchidos com o namespace contido e as informações de tipo. O `source` é adicionado ao `context` com o nome da dica. Neste exemplo, o gerador cria um novo arquivo de origem gerado que contém

uma implementação do método `partial` no aplicativo de console. Você pode gravar geradores de origem para adicionar qualquer fonte desejada.

### Dica

O parâmetro `hintName` do método `SourceProductionContext.AddSource` pode ser qualquer nome exclusivo. É comum fornecer uma extensão de arquivo C# explícita, como `".g.cs"` ou `".generated.cs"` para o nome. O nome do arquivo ajuda a identificar o arquivo como sendo gerado pela fonte.

2. Agora temos um gerador em operação, mas precisamos conectá-lo ao nosso aplicativo de console. Edite o projeto do aplicativo de console original e adicione o seguinte, substituindo o caminho do projeto pelo do projeto .NET Standard criado acima:

#### XML

```
<!-- Add this as a new ItemGroup, replacing paths and names  
appropriately -->  
<ItemGroup>  
    <ProjectReference Include=".\\PathTo\\SourceGenerator.csproj"  
        OutputItemType="Analyzer"  
        ReferenceOutputAssembly="false" />  
</ItemGroup>
```

Essa nova referência não é uma referência de projeto tradicional e precisa ser editada manualmente para incluir os atributos `OutputItemType` e `ReferenceOutputAssembly`. Para obter mais informações sobre os atributos `OutputItemType` e `ReferenceOutputAssembly` de `ProjectReference`, consulte [Itens comuns de projeto do MSBuild: ProjectReference](#).

3. Agora, ao executar o aplicativo de console, você deverá ver que o código gerado é executado e impresso na tela. O aplicativo de console em si não implementa o método `HelloFrom`, em vez disso, é a fonte gerada durante a compilação do projeto Gerador de Origem. O texto a seguir é um exemplo de saída do aplicativo:

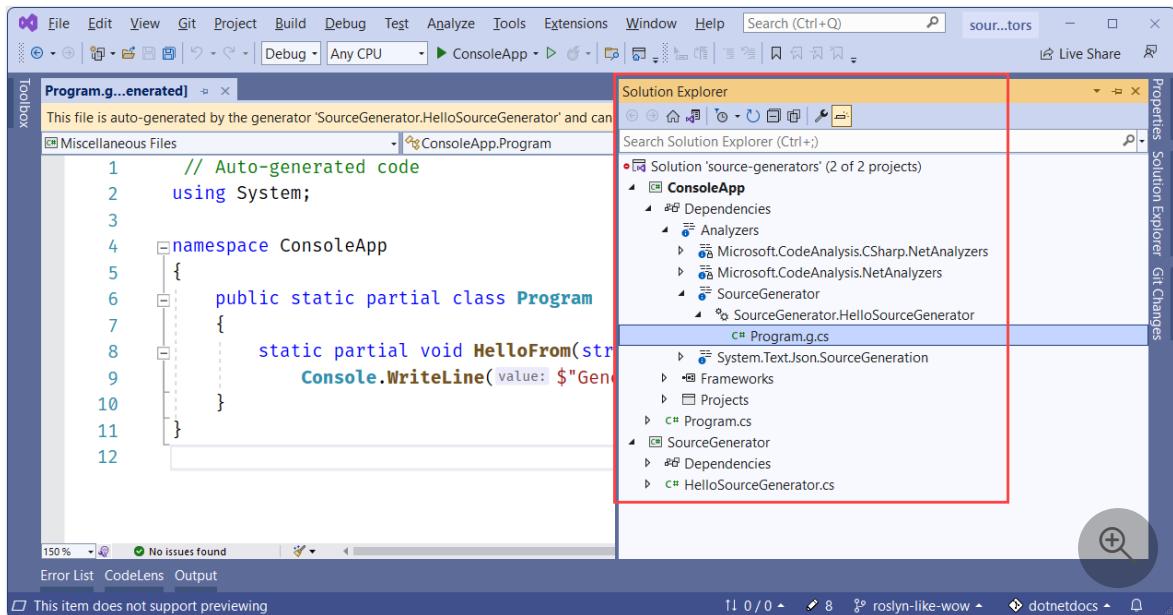
#### Console

```
Generator says: Hi from 'Generated Code'
```

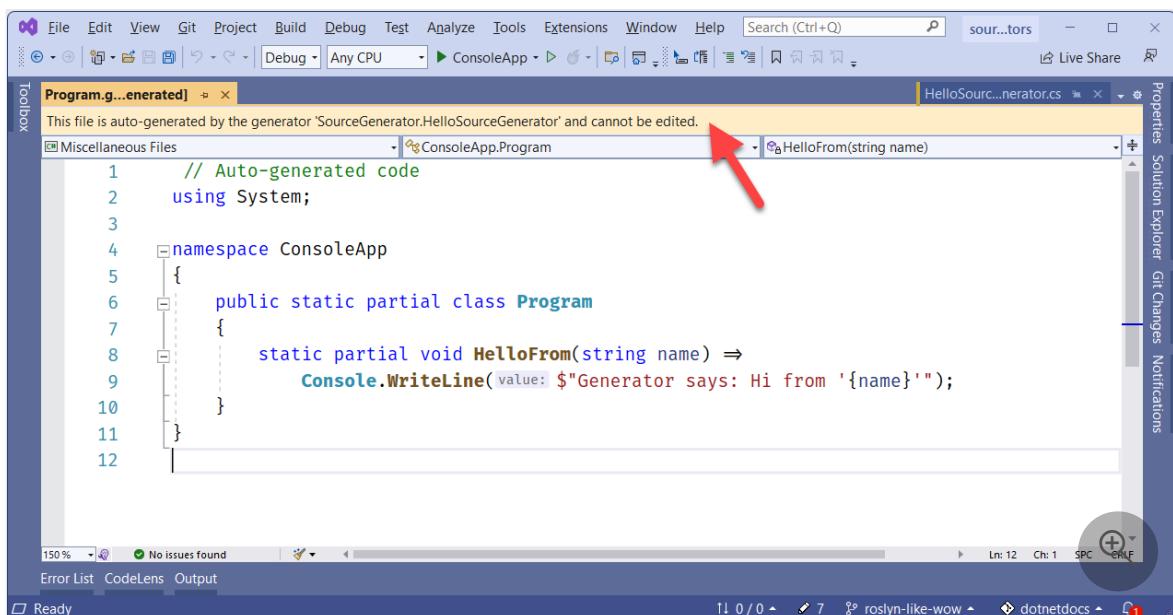
### Observação

Talvez seja necessário reiniciar o Visual Studio para ver o IntelliSense e se livrar de erros, pois a experiência de ferramentas está sendo aprimorada ativamente.

4. Se você estiver usando o Visual Studio, poderá ver os arquivos gerados pela origem. Na janela Gerenciador de Soluções, expanda **Dependência>Analisadores>SourceGenerator>SourceGenerator.HelloSourceGenerator** e clique duas vezes no arquivo *Program.g.cs*.



Quando você abrir esse arquivo gerado, o Visual Studio indicará que o arquivo é gerado automaticamente e que não pode ser editado.



5. Você também pode definir propriedades de build para salvar o arquivo gerado e controlar onde os arquivos gerados são armazenados. No arquivo de projeto do aplicativo de console, adicione o elemento `<EmitCompilerGeneratedFiles>` a um

`<PropertyGroup>` e defina seu valor como `true`. Compile o projeto novamente.

Agora, os arquivos gerados são criados em `obj/Debug/net7.0/generated/SourceGenerator/SourceGenerator.HelloSourceGenerator`. Os componentes do mapa de caminho para a configuração de build, a estrutura de destino, o nome do projeto do gerador de origem e o nome de tipo totalmente qualificado do gerador. Você pode escolher uma pasta de saída mais conveniente adicionando o elemento `<CompilerGeneratedFilesOutputPath>` ao arquivo de projeto do aplicativo.

## Próximas etapas

O [Livro de Receitas de Geradores de Origem](#) analisa alguns desses exemplos com algumas abordagens recomendadas para resolvê-los. Além disso, temos um [conjunto de exemplos disponíveis no GitHub](#) que você pode experimentar por conta própria.

Saiba mais sobre os Geradores de Origem nesses artigos:

- [Documento de design dos Geradores de Origem](#)
- [Livro de receitas dos Geradores de Origem](#)

# Introdução à análise de sintaxe

Artigo • 09/05/2023

Neste tutorial, você explorará a **API de sintaxe**. A API de sintaxe fornece acesso às estruturas de dados que descrevem um programa C# ou Visual Basic. Essas estruturas de dados têm detalhes suficientes para que possam representar qualquer programa, de qualquer tamanho. Essas estruturas podem descrever programas completos que compilam e executam corretamente. Elas também podem descrever programas incompletos, enquanto você os escreve no editor.

Para habilitar essa expressão avançada, as estruturas de dados e as APIs que compõem a API de sintaxe são necessariamente complexas. Começaremos com a aparência da estrutura de dados para o programa "Olá, Mundo" típico:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Veja o texto do programa anterior. Você reconhece elementos familiares. O texto inteiro representa um único arquivo de origem, ou uma **unidade de compilação**. As três primeiras linhas do arquivo de origem são **diretivas de uso**. A origem restante está contida em uma **declaração de namespace**. A declaração de namespace contém uma **declaração de classe** filha. A declaração de classe contém uma **declaração de método**.

A API de sintaxe cria uma estrutura de árvore com a raiz que representa a unidade de compilação. Nós da árvore representam as diretivas using, a declaração de namespace e todos os outros elementos do programa. A estrutura da árvore continua até os níveis mais baixos: a cadeia de caracteres "Olá, Mundo!" é um **token literal de cadeia de caracteres** descendente de um **argumento**. A API de sintaxe fornece acesso à estrutura do programa. Você pode consultar as práticas recomendadas de código específico,

percorrer a árvore inteira para entender o código e criar novas árvores ao modificar a árvore existente.

Essa breve descrição fornece uma visão geral do tipo de informações acessíveis usando a API de sintaxe. A API de sintaxe não é nada mais de uma API formal que descreve os constructos de código familiares que você conhece do C#. As funcionalidades completas incluem informações sobre como o código é formatado, incluindo quebras de linha, espaço em branco e recuo. Usando essas informações, você pode representar totalmente o código como escrito e lido por programadores humanos ou pelo compilador. Usar essa estrutura permite que você interaja com o código-fonte em um nível muito significativo. Não se trata mais de cadeias de caracteres de texto, mas de dados que representam a estrutura de um programa C#.

Para começar, será necessário instalar o [SDK do .NET Compiler Platform](#):

## Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o SDK da .NET Compiler Platform no [Instalador do Visual Studio](#):

### Instalar usando o Instalador do Visual Studio – exibição de cargas de trabalho

O SDK da .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento da extensão do Visual Studio. É necessário selecioná-lo como um componente opcional.

1. Execute o [Instalador do Visual Studio](#)
2. Escolha **Mudar**
3. Marque a carga de trabalho de **Desenvolvimento de extensão do Visual Studio**.
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará por último nos componentes opcionais.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

# Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará na parte superior, na seção **Compiladores, ferramentas de compilação e runtimes**.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Marque a caixa do **Editor DGML**. Você a encontrará na seção **Ferramentas de código**.

## Noções básicas sobre árvores de sintaxe

Você pode usar a API de sintaxe para uma análise da estrutura do código C#. A **API de sintaxe** expõe os analisadores, as árvores de sintaxe e os utilitários para analisar e criar árvores de sintaxe. Trata-se do modo como você pesquisa o código em busca de elementos de sintaxe específicos ou lê o código para um programa.

Uma árvore de sintaxe é uma estrutura de dados usada pelos compiladores C# e Visual Basic para entender programas nessas linguagens. Árvores de sintaxe são produzidas pelo mesmo analisador que é executado quando um projeto é compilado ou quando um desenvolvedor pressiona F5. As árvores de sintaxe têm fidelidade total com à linguagem de programação; cada bit de informações em um arquivo de código é representado na árvore. Gravar uma árvore de sintaxe em texto reproduz o texto original exato que foi analisado. As árvores de sintaxe também são **imutáveis**; uma vez criada, uma árvore de sintaxe nunca pode ser alterada. Os consumidores de árvores podem analisar as árvores de vários threads, sem bloqueios ou outras medidas de simultaneidade, sabendo que os dados nunca são alterados. Você pode usar APIs para criar novas árvores que são o resultado da modificação de uma árvore existente.

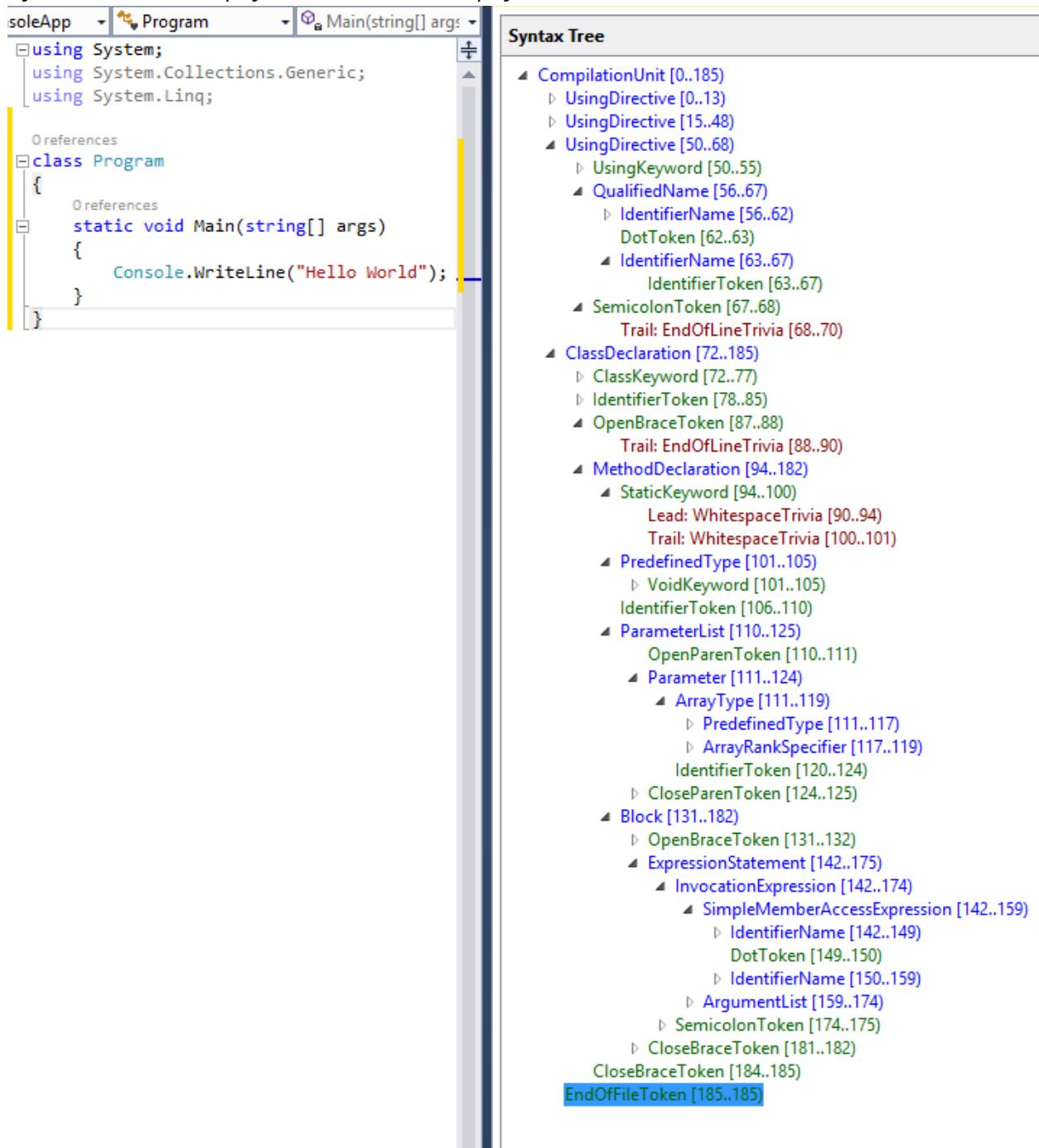
Os quatro principais blocos de construção de árvores de sintaxe são:

- A classe [Microsoft.CodeAnalysis.SyntaxTree](#), uma instância da qual representa uma árvore de análise inteira. [SyntaxTree](#) é uma classe abstrata que tem derivativos específicos a um idioma. Você usa os métodos de análise da classe [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree](#) (ou [Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree](#)) para analisar o texto em C# (ou em Visual Basic).

- A classe [Microsoft.CodeAnalysis.SyntaxNode](#), instâncias da qual representam constructos sintáticos como declarações, instruções, cláusulas e expressões.
- A estrutura [Microsoft.CodeAnalysis.SyntaxToken](#), que representa uma pontuação, operador, identificador ou palavra-chave individual.
- Finalmente, a estrutura [Microsoft.CodeAnalysis.SyntaxTrivia](#), que representa os bits de informação sem significância sintática, tais como o espaço em branco entre tokens, diretivas de pré-processamento e comentários.

Trívia, tokens e nós são compostos hierarquicamente para formar uma árvore que representa completamente tudo em um fragmento de código do Visual Basic ou do C#. Você pode ver essa estrutura usando a janela **Visualizador de Sintaxe**. No Visual Studio, escolha **Exibição>Outras Janelas>Visualizador de Sintaxe**. Por exemplo, o arquivo de origem C# anterior examinado usando o **Visualizador de Sintaxe** se parecerá com a figura a seguir:

SyntaxNode: Azul | SyntaxToken: Verde | SyntaxTrivia: Vermelho



Ao navegar nessa estrutura de árvore, você pode encontrar qualquer instrução, expressão, token ou bit de espaço em branco em um arquivo de código.

Embora você possa encontrar tudo em um arquivo de código usando as APIs de sintaxe, a maioria dos cenários envolvem o exame de pequenos snippets de código ou a pesquisa por instruções ou fragmentos específicos. Os dois exemplos a seguir mostram usos típicos para navegar pela estrutura de códigos ou pesquisar por instruções individuais.

## Percorrendo árvores

Você pode examinar os nós em uma árvore de sintaxe de duas maneiras. Você pode percorrer a árvore para examinar cada nó, ou então consultar elementos ou nós específicos.

## Passagem manual

Você pode ver o código concluído para essa amostra no [nossa repositório do GitHub](#).

### Observação

Os tipos de árvore de sintaxe usam a herança para descrever os elementos de sintaxe diferentes que são válidos em locais diferentes no programa. Usar essas APIs geralmente significa converter propriedades ou membros da coleção em tipos derivados específicos. Nos exemplos a seguir, a atribuição e as conversões são instruções separadas, usando variáveis explicitamente tipadas. Você pode ler o código para ver os tipos de retorno da API e o tipo de runtime dos objetos retornados. Na prática, é mais comum usar variáveis implicitamente tipadas e depender de nomes de API para descrever o tipo de objeto que está sendo examinado.

Criar um novo projeto de **Ferramenta de Análise de Código Autônoma** do C#:

- No Visual Studio, escolha **Arquivo > Novo > Projeto** para exibir a caixa de diálogo **Novo Projeto**.
- Em **Visual C# > Extensibilidade**, escolha **Ferramenta de Análise de Código Autônoma**.
- Nomeie o projeto "**SyntaxTreeManualTraversal**" e clique em **OK**.

Você vai analisar o programa básico "Olá, Mundo!" mostrado anteriormente. Adicione o texto ao programa Olá, Mundo como uma constante em sua classe **Program**:

C#

```
const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            Console.WriteLine("""Hello, World!""");
        }
    }";
}
```

Em seguida, adicione o código a seguir para criar a **árvore de sintaxe** para o texto do código na constante `programText`. Adicione a seguinte linha ao seu método `Main`:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Essas duas linhas criam a árvore e recuperam o nó raiz dessa árvore. Agora você pode examinar os nós na árvore. Adicione essas linhas ao seu método `Main` para exibir algumas das propriedades do nó raiz na árvore:

C#

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"{element.Name}");
```

Execute o aplicativo para ver o que seu código descobriu sobre o nó raiz nessa árvore.

Normalmente, percorreria a árvore para saber mais sobre o código. Neste exemplo, você está analisando código que você conhece para explorar as APIs. Adicione o código a seguir para examinar o primeiro membro do nó `root`:

C#

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

Esse membro é um [Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax](#). Ele representa tudo no escopo da declaração `namespace HelloWorld`. Adicione o seguinte código para examinar quais nós são declarados dentro do namespace `HelloWorld`:

C#

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared  
in this namespace.");  
WriteLine($"The first member is a  
{helloWorldDeclaration.Members[0].Kind()}.");
```

Execute o programa para ver o que você aprendeu.

Agora que você sabe que a declaração é um [Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax](#), declare uma nova variável de tipo para examinar a declaração de classe. Essa classe contém somente um membro: o método `Main`. Adicione o código a seguir para localizar o método `Main` e convertê-lo em um [Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax](#).

C#

```
var programDeclaration =  
(ClassDeclarationSyntax)helloWorldDeclaration.Members[0];  
WriteLine($"There are {programDeclaration.Members.Count} members declared in  
the {programDeclaration.Identifier} class.");  
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");  
var mainDeclaration =  
(MethodDeclarationSyntax)programDeclaration.Members[0];
```

O nó de declaração do método contém todas as informações de sintaxe sobre o método. Permite exibir o tipo de retorno do método `Main`, o número e os tipos dos argumentos e o texto do corpo do método. Adicione os códigos a seguir:

C#

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is  
{mainDeclaration.ReturnType}.");  
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count}  
parameters.");  
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)  
    WriteLine($"The type of the {item.Identifier} parameter is  
{item.Type}.");  
WriteLine($"The body text of the {mainDeclaration.Identifier} method  
follows:");  
WriteLine(mainDeclaration.Body?.ToFullString());  
  
var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

Execute o programa para ver todas as informações que você descobriu sobre este programa:

text

```
The tree is a CompilationUnit node.  
The tree has 1 elements in it.  
The tree has 4 using statements. They are:  
    System  
    System.Collections  
    System.Linq  
    System.Text  
The first member is a NamespaceDeclaration.  
There are 1 members declared in this namespace.  
The first member is a ClassDeclaration.  
There are 1 members declared in the Program class.  
The first member is a MethodDeclaration.  
The return type of the Main method is void.  
The method has 1 parameters.  
The type of the args parameter is string[].  
The body text of the Main method follows:  
{  
    Console.WriteLine("Hello, World!");  
}
```

## Métodos de consulta

Além de percorrer árvores, você também pode explorar a árvore de sintaxe usando os métodos de consulta definidos em [Microsoft.CodeAnalysis.SyntaxNode](#). Esses métodos devem ser imediatamente familiares a qualquer pessoa familiarizada com o XPath. Você pode usar esses métodos com o LINQ para localizar itens rapidamente em uma árvore. O [SyntaxNode](#) tem métodos de consulta como [DescendantNodes](#), [AncestorsAndSelf](#) e [ChildNodes](#).

Você pode usar esses métodos de consulta para localizar o argumento para o método `Main` como uma alternativa a navegar pela árvore. Adicione o seguinte código à parte inferior do método `Main`:

C#

```
var firstParameters = from methodDeclaration in root.DescendantNodes()  
                      .OfType<MethodDeclarationSyntax>()  
                      where methodDeclaration.Identifier.ValueText == "Main"  
                      select  
    methodDeclaration.ParameterList.Parameters.First();  
  
var argsParameter2 = firstParameters.Single();  
  
WriteLine(argsParameter == argsParameter2);
```

A primeira instrução usa uma expressão LINQ e o método [DescendantNodes](#) para localizar o mesmo parâmetro do exemplo anterior.

Execute o programa e você poderá ver que a expressão LINQ encontrou o mesmo parâmetro encontrado ao navegar manualmente pela árvore.

O exemplo usa instruções `WriteLine` para exibir informações sobre as árvores de sintaxe conforme elas são percorridas. Você também pode aprender mais executando o programa concluído no depurador. Você pode examinar mais das propriedades e métodos que fazem parte da árvore de sintaxe criada para o programa Olá, Mundo.

## Caminhadores de sintaxe

Muitas vezes, você deseja localizar todos os nós de um tipo específico em uma árvore de sintaxe, por exemplo, cada declaração de propriedade em um arquivo. Ao estender a classe `Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalker` e substituir o método `VisitPropertyDeclaration(PropertyDeclarationSyntax)`, você processa cada declaração de propriedade em uma árvore de sintaxe sem conhecer a estrutura dele com antecedência. `CSharpSyntaxWalker` é um tipo específico de `CSharpSyntaxVisitor`, que visita recursivamente um nó e cada um dos filhos desse nó.

Este exemplo implementa um `CSharpSyntaxWalker` que examina uma árvore de sintaxe. Ele coleta diretivas `using` que ele constata que não estão importando um namespace `System`.

Crie um novo projeto de **Ferramenta de Análise de Código Autônoma** do C#; nomeie-o "SyntaxWalker".

Você pode ver o código concluído para essa amostra no [nossa repositório do GitHub](#). A amostra no GitHub contém os dois projetos descritos neste tutorial.

Assim como no exemplo anterior, você pode definir uma constante de cadeia de caracteres para conter o texto do programa que você pretende analisar:

```
C#  
  
const string programText =  
@"
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using Microsoft.CodeAnalysis;  
using Microsoft.CodeAnalysis.CSharp;  
  
namespace TopLevel  
{  
    using Microsoft;  
    using System.ComponentModel;
```

```
namespace Child1
{
    using Microsoft.Win32;
    using System.Runtime.InteropServices;

    class Foo { }
}

namespace Child2
{
    using System.CodeDom;
    using Microsoft.CSharp;

    class Bar { }
}
}";


```

Este texto de origem contém diretivas `using` espalhadas em quatro locais diferentes: o nível de arquivo, no namespace de nível superior e nos dois namespaces aninhados. Este exemplo destaca um cenário principal para usar a classe [CSharpSyntaxWalker](#) para consultar código. Seria complicado visitar cada nó na árvore de sintaxe de raiz para encontrar declarações `using`. Em vez disso, você pode criar uma classe derivada e substituir o método chamado apenas quando o nó atual na árvore é uma diretiva `using`. O visitante não realiza nenhum trabalho em nenhum outro tipo de nó. Esse método único examina cada uma das instruções `using` e compila uma coleção de namespaces que não estão no namespace `System`. Você compila um [CSharpSyntaxWalker](#) que examina todas as instruções `using`, mas apenas as instruções `using`.

Agora que você definiu o texto do programa, você precisa criar um `SyntaxTree` e obter a raiz dessa árvore:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Em seguida, crie uma nova classe. No Visual Studio, escolha **Projeto > Adicionar Novo Item**. Na caixa de diálogo **Adicionar Novo Item**, digite `UsingCollector.cs` como o nome do arquivo.

Você implementa a funcionalidade de visitante `using` na classe `UsingCollector`. Para começar, crie a classe `UsingCollector` derivada de [CSharpSyntaxWalker](#).

C#

```
class UsingCollector : CSharpSyntaxWalker
```

Você precisa de armazenamento para conter os nós de namespace que você está coletando. Declare uma propriedade pública somente leitura na classe `UsingCollector`; use essa variável para armazenar os nós `UsingDirectiveSyntax` que você encontrar:

C#

```
public ICollection<UsingDirectiveSyntax> Usings { get; } = new  
List<UsingDirectiveSyntax>();
```

A classe base `CSharpSyntaxWalker` implementa a lógica para visitar cada nó na árvore de sintaxe. A classe derivada substitui os métodos chamados para os nós específicos nos quais você está interessado. Nesse caso, você está interessado em qualquer diretiva `using`. Isso significa que você deve substituir o método

`VisitUsingDirective(UsingDirectiveSyntax)`. Um argumento para esse método é um objeto `Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax`. Essa é uma importante vantagem de se usar os visitantes: eles chamam os métodos substituídos com argumentos que já foram convertidos para o tipo de nó específico. A classe `Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax` tem uma propriedade `Name` que armazena o nome do namespace que está sendo importado. É um `Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax`. Adicione o código a seguir na substituição `VisitUsingDirective(UsingDirectiveSyntax)`:

C#

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)  
{  
    WriteLine($"\\tVisitUsingDirective called with {node.Name}.");  
    if (node.Name.ToString() != "System" &&  
        !node.Name.ToString().StartsWith("System."))  
    {  
        WriteLine($"\\t\\tSuccess. Adding {node.Name}.");  
        this.Usings.Add(node);  
    }  
}
```

Assim como no exemplo anterior, você adicionou uma variedade de instruções `WriteLine` para ajudar na compreensão do método. Você pode ver quando ele é chamado e quais argumentos são passados para ele a cada vez.

Por fim, você precisa adicionar duas linhas de código para criar o `UsingCollector` e fazer com que ele acesse o nó raiz, coletando todas as instruções `using`. Em seguida, adicione um loop `foreach` para exibir todas as instruções `using` encontradas pelo seu coletores:

C#

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

Compile e execute o programa. Você deve ver o seguinte resultado:

Console

```
VisitUsingDirective called with System.
VisitUsingDirective called with System.Collections.Generic.
VisitUsingDirective called with System.Linq.
VisitUsingDirective called with System.Text.
VisitUsingDirective called with Microsoft.CodeAnalysis.
    Success. Adding Microsoft.CodeAnalysis.
VisitUsingDirective called with Microsoft.CodeAnalysis.CSharp.
    Success. Adding Microsoft.CodeAnalysis.CSharp.
VisitUsingDirective called with Microsoft.
    Success. Adding Microsoft.
VisitUsingDirective called with System.ComponentModel.
VisitUsingDirective called with Microsoft.Win32.
    Success. Adding Microsoft.Win32.
VisitUsingDirective called with System.Runtime.InteropServices.
VisitUsingDirective called with System.CodeDom.
VisitUsingDirective called with Microsoft.CSharp.
    Success. Adding Microsoft.CSharp.

Microsoft.CodeAnalysis
Microsoft.CodeAnalysis.CSharp
Microsoft
Microsoft.Win32
Microsoft.CSharp
Press any key to continue . . .
```

Parabéns! Você usou a **API de sintaxe** para localizar tipos específicos de instruções C# e declarações em código-fonte C#.

# Introdução à análise semântica

Artigo • 10/05/2023

Este tutorial presume que você está familiarizado com a API de sintaxe. O artigo [Introdução à a análise de sintaxe](#) fornece uma introdução suficiente.

Neste tutorial, você explora as APIs de **Símbolo** e de **Associação**. Essas APIs fornecem informações sobre o *significado semântico* de um programa. Elas permitem fazer e responder perguntas sobre os tipos representados por qualquer símbolo em seu programa.

Você deverá instalar o **SDK do .NET Compiler Platform**:

## Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o **SDK da .NET Compiler Platform** no **Instalador do Visual Studio**:

### Instalar usando o Instalador do Visual Studio – exibição de cargas de trabalho

O SDK da .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento da extensão do Visual Studio. É necessário selecioná-lo como um componente opcional.

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Marque a carga de trabalho de **Desenvolvimento de extensão do Visual Studio**.
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará por último nos componentes opcionais.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

# Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o [Instalador do Visual Studio](#)
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará na parte superior, na seção **Compiladores, ferramentas de compilação e runtimes**.

Opcionalmente, você também poderá fazer o [Editor DGML](#) exibir gráficos no visualizador:

1. Marque a caixa do **Editor DGML**. Você a encontrará na seção **Ferramentas de código**.

## Noções básicas sobre compilações e símbolos

Conforme você trabalha mais com o SDK do .NET Compiler, você se familiariza com as distinções entre a API de Sintaxe e a API de Semântica. A **API de Sintaxe** permite que você examine a *estrutura* de um programa. Muitas vezes, no entanto, você deseja as informações sobre a semântica ou *significado* de um programa. Embora um snippet ou arquivo de código livre do Visual Basic ou C# possa ser analisado sintaticamente de modo isolado, não faz sentido fazer a esmo perguntas como "qual é o tipo dessa variável". O significado de um nome de tipo pode ser dependente de referências de assembly, importações de namespace ou outros arquivos de código. Essas perguntas são respondidas usando-se a **API de Semântica**, especificamente a classe [Microsoft.CodeAnalysis.Compilation](#).

Uma instância de [Compilation](#) é análoga a um único projeto conforme visto pelo compilador e representa tudo o que é necessário para compilar um programa Visual Basic ou C#. A **compilação** inclui o conjunto de arquivos de origem a serem compilados, referências de assembly e opções de compilador. Você pode avaliar o significado do código usando todas as outras informações neste contexto. Um [Compilation](#) permite que você encontre **símbolos** – entidades como tipos, namespaces, membros e variáveis aos quais os nomes e outras expressões se referem. O processo de associar nomes e expressões com **símbolos** é chamado de **associação**.

Assim como [Microsoft.CodeAnalysis.SyntaxTree](#), [Compilation](#) é uma classe abstrata com derivativos específicos a um idioma. Ao criar uma instância de compilação, você deve invocar um método de fábrica na classe [Microsoft.CodeAnalysis.CSharp.CSharpCompilation](#) (ou [Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation](#)).

# Consultar símbolos

Neste tutorial, você analisa novamente o programa "Olá, Mundo". Dessa vez, você consulta os símbolos no programa para compreender quais tipos esses símbolos representam. Você consulta os tipos em um namespace e aprende a localizar os métodos disponíveis em um tipo.

Você pode ver o código concluído para essa amostra no [nossa repositório do GitHub](#).

## ① Observação

Os tipos de árvore de sintaxe usam a herança para descrever os elementos de sintaxe diferentes que são válidos em locais diferentes no programa. Usar essas APIs geralmente significa converter propriedades ou membros da coleção em tipos derivados específicos. Nos exemplos a seguir, a atribuição e as conversões são instruções separadas, usando variáveis explicitamente tipadas. Você pode ler o código para ver os tipos de retorno da API e o tipo de runtime dos objetos retornados. Na prática, é mais comum usar variáveis implicitamente tipadas e depender de nomes de API para descrever o tipo de objeto que está sendo examinado.

Criar um novo projeto de **Ferramenta de Análise de Código Autônoma** do C#:

- No Visual Studio, escolha **Arquivo > Novo > Projeto** para exibir a caixa de diálogo **Novo Projeto**.
- Em **Visual C# > Extensibilidade**, escolha **Ferramenta de Análise de Código Autônoma**.
- Nomeie o projeto "**SemanticQuickStart**" e clique em **OK**.

Você vai analisar o programa básico "Olá, Mundo!" mostrado anteriormente. Adicione o texto ao programa Olá, Mundo como uma constante em sua classe **Program**:

C#

```
const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
        Console.WriteLine("Hello, World!");
    }
}
";
```

Em seguida, adicione o código a seguir para criar a árvore de sintaxe para o texto do código na constante `programText`. Adicione a seguinte linha ao seu método `Main`:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Em seguida, compile uma `CSharpCompilation` da árvore que você já criou. A amostra "Olá, Mundo" depende dos tipos `String` e `Console`. Você precisa fazer referência ao assembly que declara esses dois tipos em sua compilação. Adicione a seguinte linha ao seu método `Main` para criar uma compilação de sua árvore de sintaxe, incluindo a referência ao assembly apropriado:

C#

```
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

O método `CSharpCompilation.AddReferences` adiciona referências à compilação. O método `MetadataReference.CreateFromFile` carrega um assembly como uma referência.

## Consultar o modelo semântico

Assim que você tiver uma `Compilation`, você poderá solicitar a ela um `SemanticModel` para qualquer `SyntaxTree` contida nessa `Compilation`. Você pode pensar no modelo semântico como a origem de todas as informações normalmente obtidas do IntelliSense. Um `SemanticModel` pode responder a perguntas como "O que são nomes no escopo nesse local?", "Quais membros são acessíveis deste método?", "Quais variáveis são usadas neste bloco de texto?" e "A que este nome/expressão se refere?" Adicione esta instrução para criar o modelo semântico:

C#

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

# Associar um nome

A [Compilation](#) cria o [SemanticModel](#) da [SyntaxTree](#). Depois de criar o modelo, você pode consultar para localizar a primeira diretiva `using` e recuperar as informações de símbolo para o namespace `System`. Adicione estas duas linhas a seu método `Main` para criar o modelo semântico e recuperar o símbolo para a primeira instrução `using`:

C#

```
// Use the syntax tree to find "using System;"  
UsingDirectiveSyntax usingSystem = root.Usings[0];  
NameSyntax systemName = usingSystem.Name;  
  
// Use the semantic model for symbol information:  
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

O código anterior mostra como associar o nome na primeira diretiva `using` para recuperar um [Microsoft.CodeAnalysis.SymbolInfo](#) para o namespace `System`. O código anterior também ilustra o uso da **sintaxe de modelo** para localizar a estrutura do código; você usa o **modelo semântico** para entender seu significado. A **sintaxe de modelo** localiza a cadeia de caracteres `System` na instrução `using`. O **modelo semântico** tem todas as informações sobre os tipos definidos no namespace `System`.

Do objeto [SymbolInfo](#), você pode obter o [Microsoft.CodeAnalysis.ISymbol](#) usando a propriedade [SymbolInfo.Symbol](#). Essa propriedade retorna o símbolo a que essa expressão se refere. Para expressões que não se referem a nada (como literais numéricos), essa propriedade é `null`. Quando o [SymbolInfo.Symbol](#) não for `null`, o [ISymbol.Kind](#) denotará o tipo do símbolo. Nesse exemplo, a propriedade [ISymbol.Kind](#) é um [SymbolKind.Namespace](#). Adicione o código a seguir ao método `Main`. Ele recupera o símbolo para o namespace `System` e, em seguida, exibe todos os namespaces filho declarados no namespace `System`:

C#

```
var systemSymbol = (INamespaceSymbol?)nameInfo.Symbol;  
if (systemSymbol?.GetNamespaceMembers() is not null)  
{  
    foreach (INamespaceSymbol ns in systemSymbol?.GetNamespaceMembers()!)  
    {  
        Console.WriteLine(ns);  
    }  
}
```

Execute o programa e você deverá ver a seguinte saída:

Saída

```
System.Collections
System.Configuration
System.Deployment
System.Diagnostics
System.Globalization
System.IO
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.StubHelpers
System.Text
System.Threading
Press any key to continue . . .
```

### ⓘ Observação

A saída não inclui todos os namespaces que são namespaces filhos do namespace `System`. El exibe cada namespace presente nessa compilação, que só referencia o assembly em que `System.String` é declarada. Quaisquer outros namespaces declarados em outros assemblies não são conhecidos desta compilação

## Associar uma expressão

O código anterior mostra como encontrar um símbolo associando-o a um nome. Há outras expressões em um programa C# que podem ser associadas que não são nomes. Para demonstrar essa capacidade, acessaremos a associação a um único literal de cadeia de caracteres.

O programa "Olá, Mundo" contém

[Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax](#), uma cadeia de caracteres "Olá, Mundo!" exibida no console.

Você encontra a cadeia de caracteres "Olá, Mundo!" localizando o único literal de cadeia de caracteres no programa. Em seguida, depois de localizar o nó de sintaxe, você obtém as informações de tipo para esse nó do modelo semântico. Adicione o código a seguir ao método `Main`:

C#

```
// Use the syntax model to find the literal string:
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()
```

```
.OfType<LiteralExpressionSyntax>()
.Single();

// Use the semantic model for type information:
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);
```

O struct [Microsoft.CodeAnalysis.TypeInfo](#) inclui uma propriedade [TypeInfo.Type](#) que permite o acesso às informações semânticas sobre o tipo do literal. Neste exemplo, ele é do tipo `string`. Adicione uma declaração que atribui essa propriedade a uma variável local:

C#

```
var stringTypeSymbol = (INamedTypeSymbol?)literalInfo.Type;
```

Para concluir este tutorial, criaremos uma consulta LINQ que criará uma sequência de todos os métodos públicos declarados no tipo `string` que retorna um `string`. Essa consulta torna-se complexa, então a compilaremos linha a linha e então a reconstruiremos como uma única consulta. A ordem desta consulta é a sequência de todos os membros declarados no tipo `string`:

C#

```
var allMembers = stringTypeSymbol?.GetMembers();
```

Essa sequência de origem contém todos os membros, incluindo propriedades e campos, portanto, filtre-a usando o método [ImmutableArray<T>.OfType](#) para localizar elementos que são objetos [Microsoft.CodeAnalysis.IMethodSymbol](#):

C#

```
var methods = allMembers?.OfType<IMethodSymbol>();
```

Em seguida, adicione outro filtro para retornar somente os métodos que são públicos e retornam um `string`:

C#

```
var publicStringReturningMethods = methods?
    .Where(m => SymbolEqualityComparer.Default.Equals(m.ReturnType,
stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

Selecione apenas a propriedade de nome e somente os nomes distintos, removendo quaisquer sobrecargas:

C#

```
var distinctMethods = publicStringReturningMethods?.Select(m =>
    m.Name).Distinct();
```

Você pode também compilar a consulta completa usando a sintaxe de consulta LINQ e, em seguida, exibir todos os nomes de método no console:

C#

```
foreach (string name in (from method in stringTypeSymbol?
                           .GetMembers().OfType<IMethodSymbol>()
                           where
                               SymbolEqualityComparer.Default.Equals(method.ReturnType, stringTypeSymbol)
                               &&
                               method.DeclaredAccessibility ==
                               Accessibility.Public
                           select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Compile e execute o programa. Você deve ver o seguinte resultado:

Saída

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
```

IsInterned

Press any key to continue . . .

Você usou a API de semântica para localizar e exibir informações sobre os símbolos que fazem parte deste programa.

# Introdução à transformação de sintaxe

Artigo • 09/05/2023

Este tutorial baseia-se nos conceitos e técnicas explorados nos guias de início rápido [Introdução à análise de sintaxe](#) e [Introdução à análise semântica](#). Se ainda não o fez, você deve concluir as etapas rápidas antes de começar esta.

Neste início rápido, você explora técnicas para criar e transformar árvores de sintaxe. Em combinação com as técnicas que você aprendeu em guias de início rápido anteriores, você cria sua primeira refatoração de linha de comando!

## Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o [SDK da .NET Compiler Platform](#) no [Instalador do Visual Studio](#):

### Instalar usando o Instalador do Visual Studio – exibição de cargas de trabalho

O SDK da .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento da extensão do Visual Studio. É necessário selecioná-lo como um componente opcional.

1. Execute o [Instalador do Visual Studio](#)
2. Escolha **Mudar**
3. Marque a carga de trabalho de **Desenvolvimento de extensão do Visual Studio**.
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará por último nos componentes opcionais.

Opcionalmente, você também poderá fazer o [Editor DGML](#) exibir gráficos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

### Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará na parte superior, na seção **Compiladores, ferramentas de compilação e runtimes**.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Marque a caixa do **Editor DGML**. Você a encontrará na seção **Ferramentas de código**.

## Imutabilidade e a plataforma de compiladores .NET

**Imutabilidade** é um princípio fundamental da plataforma de compiladores .NET. Estruturas de dados imutáveis não podem ser alteradas depois de criadas. Estruturas de dados imutáveis podem ser compartilhadas com segurança e analisadas por vários consumidores simultaneamente. Não há perigo de que um consumidor afete o outro de maneiras imprevisíveis. Seu analisador não precisa de bloqueios ou outras medidas de simultaneidade. Essa regra se aplica a árvores de sintaxe, compilações, símbolos, modelos semânticos e todas as outras estruturas de dados que você encontrar. Em vez de modificar as estruturas existentes, as APIs criam novos objetos com base nas diferenças especificadas para os antigos. Você aplica esse conceito a árvores de sintaxe para criar novas árvores usando transformações.

## Criar e transformar árvores

Você escolhe uma das duas estratégias para transformações de sintaxe. Os **métodos de fábrica** são melhor usados quando você está procurando por nós específicos para substituir ou locais específicos onde deseja inserir um novo código. **Regravadores** são a melhor opção quando você deseja examinar um projeto inteiro em busca dos padrões de código que deseja substituir.

## Criar nós com métodos de fábrica

A primeira transformação de sintaxe demonstra os métodos de fábrica. Substitua uma instrução `using System.Collections;` por uma instrução `using System.Collections.Generic;`. Este exemplo demonstra como você cria objetos `Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode` usando os métodos de fábrica

[Microsoft.CodeAnalysis.CSharp.SyntaxFactory](#). Para cada tipo de **nó**, **token** ou **trívia**, há um método de fábrica que cria uma instância desse tipo. Você cria árvores de sintaxe compondo os nós hierarquicamente de baixa para cima. Em seguida, você transformará o programa existente substituindo nós existentes pela nova árvore criada.

Inicie o Visual Studio e crie um novo projeto de **Ferramenta de Análise de Código Autônoma** do C#. No Visual Studio, escolha **Arquivo > Novo > Projeto** para exibir a caixa de diálogo **Novo Projeto**. Em **Visual C# > Extensibilidade**, escolha uma **Ferramenta de Análise de Código Autônoma**. Este guia de início rápido tem dois projetos de exemplo, portanto, nomeie a solução **SyntaxTransformationQuickStart** e nomeie o projeto **ConstructionCS**. Clique em **OK**.

Este projeto usa os métodos de classe [Microsoft.CodeAnalysis.CSharp.SyntaxFactory](#) para construir um [Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax](#) representando o namespace `System.Collections.Generic`.

Adicione a seguinte diretiva de uso à parte superior do `Program.cs`.

```
C#
```

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

Você criará **nós de sintaxe de nome** para construir a árvore que representa a instrução `using System.Collections.Generic;`. [NameSyntax](#) é a classe base para quatro tipos de nomes que aparecem no C#. Você compõe esses quatro tipos de nomes para criar qualquer nome que possa aparecer na linguagem C#:

- [Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax](#), que representa nomes simples de identificadores únicos como `System` e `Microsoft`.
- [Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax](#), que representa um tipo genérico ou nome de método, como `List<int>`.
- [Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax](#), que representa um nome qualificado do formulário `<left-name>.<right-identifier-or-generic-name>`, como `System.IO`.
- [Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax](#), que representa um nome usando um alias externo de assembly como `LibraryV2::Foo`.

Você usa o método [IdentifierName\(String\)](#) para criar um nó [NameSyntax](#). Adicione o seguinte código no seu método `Main` no `Program.cs`:

```
C#
```

```
NameSyntax name = IdentifierName("System");
WriteLine($"\\tCreated the identifier {name}");
```

O código anterior cria um objeto [IdentifierNameSyntax](#) e o atribui à variável `name`. Muitas das APIs Roslyn retornam classes básicas para facilitar o trabalho com tipos relacionados. A variável `name`, um [NameSyntax](#), pode ser reutilizada conforme você constrói o [QualifiedNameSyntax](#). Não use inferência de tipo ao criar a amostra. Você automatizará essa etapa neste projeto.

Você criou o nome. Agora, é hora de criar mais nós na árvore criando um [QualifiedNameSyntax](#). A nova árvore usa `name` como a esquerda do nome e um novo [IdentifierNameSyntax](#) para o namespace `Collections` como o lado direito do [QualifiedNameSyntax](#). Adicione o seguinte código a `program.cs`:

C#

```
name = QualifiedName(name, IdentifierName("Collections"));
WriteLine(name.ToString());
```

Execute o código novamente e confira os resultados. Você está construindo uma árvore de nós que representa o código. Você continuará este padrão para construir o [QualifiedNameSyntax](#) para o namespace `System.Collections.Generic`. Adicione o seguinte código a `Program.cs`:

C#

```
name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

Execute o programa novamente para ver se você construiu a árvore para o código a ser adicionado.

## Criar uma árvore modificada

Você criou uma pequena árvore de sintaxe que contém uma instrução. As APIs para criar novos nós são a escolha certa para criar instruções únicas ou outros pequenos blocos de código. No entanto, para construir blocos maiores de código, você deve usar métodos que substituem nós ou inserem nós em uma árvore existente. Lembre-se de que as árvores de sintaxe são imutáveis. A [API de Sintaxe](#) não fornece nenhum mecanismo para modificar uma árvore de sintaxe existente após a construção. Em vez disso, fornece métodos que produzem novas árvores com base nas alterações

existentes. Os métodos `With*` são definidos em classes concretas que derivam de [SyntaxNode](#) ou em métodos de extensão declarados na classe [SyntaxNodeExtensions](#). Esses métodos criam um novo nó aplicando alterações nas propriedades filho de um nó existente. Além disso, o método de extensão `ReplaceNode` pode ser usado para substituir um nó descendente em uma subárvore. Esse método também atualiza o pai para apontar para o filho recém-criado e repete esse processo até a árvore inteira — um processo conhecido como *re-spinning* da árvore.

O próximo passo é criar uma árvore que represente um programa inteiro (pequeno) e depois modificá-la. Adicione o seguinte código ao início da classe `Program`:

C#

```
private const string sampleCode =  
@"using System;  
using System.Collections;  
using System.Linq;  
using System.Text;  
  
namespace HelloWorld  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Hello, World!");  
        }  
    }  
};
```

### ⓘ Observação

O código de exemplo usa o namespace `System.Collections` e não o namespace `System.Collections.Generic`.

Em seguida, adicione o seguinte código à parte inferior do método `Main` para analisar o texto e criar uma árvore:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);  
var root = (CompilationUnitSyntax)tree.GetRoot();
```

Este exemplo usa o método `WithName(NameSyntax)` para substituir o nome em um nó `UsingDirectiveSyntax` pelo que foi construído no código anterior.

Crie um novo nó [UsingDirectiveSyntax](#) usando o método [WithName\(NameSyntax\)](#) para atualizar o nome `System.Collections` com o nome criado no código anterior. Adicione o seguinte código à parte inferior do método `Main`:

```
C#
```

```
var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
WriteLine(root.ToString());
```

Execute o programa e observe atentamente a saída. O `newUsing` não foi colocado na árvore raiz. A árvore original não foi alterada.

Adicione o seguinte código usando o método de extensão [ReplaceNode](#) para criar uma nova árvore. A nova árvore é o resultado da substituição da importação existente pelo nó `newUsing` atualizado. Você atribui essa nova árvore à variável `root` existente:

```
C#
```

```
root = root.ReplaceNode(oldUsing, newUsing);
WriteLine(root.ToString());
```

Execute o programa novamente. Desta vez, a árvore importa corretamente o namespace `System.Collections.Generic`.

## Transformar árvores usando [SyntaxRewriters](#)

Os métodos `With*` e [ReplaceNode](#) fornecem meios convenientes para transformar ramos individuais de uma árvore de sintaxe. A classe [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter](#) realiza várias transformações em uma árvore de sintaxe. A classe [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter](#) é uma subclasse de [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult>](#). O [CSharpSyntaxRewriter](#) aplica uma transformação a um tipo específico de [SyntaxNode](#). Você pode aplicar transformações a vários tipos de objetos [SyntaxNode](#) sempre que eles aparecerem em uma árvore de sintaxe. O segundo projeto neste guia de início rápido cria uma refatoração de linha de comando que remove tipos explícitos em declarações de variáveis locais em qualquer lugar em que a inferência de tipo possa ser usada.

Crie um novo projeto de **Ferramenta de Análise de Código Autônoma** do C#. No Visual Studio, clique com o botão direito do mouse no nó da solução `SyntaxTransformationQuickStart`. Escolha **Adicionar > Novo Projeto** para exibir o diálogo

**Novo Projeto.** Em Visual C#>**Extensibilidade**, escolha **Ferramenta de Análise de Código Autônoma**. Nomeie seu projeto como `TransformationCS` e clique em OK.

A primeira etapa é criar uma classe que deriva de `CSharpSyntaxRewriter` para executar as transformações. Adicione um novo arquivo de classe ao projeto. No Visual Studio, escolha **Projeto>Adicionar Classes....** Na caixa de diálogo **Adicionar Novo Item**, digite `TypeInferenceRewriter.cs` como nome do arquivo.

Adicione o seguinte usando diretivas ao arquivo `TypeInferenceRewriter.cs`:

```
C#  
  
using Microsoft.CodeAnalysis;  
using Microsoft.CodeAnalysis.CSharp;  
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

Em seguida, faça a classe `TypeInferenceRewriter` se estender à classe `CSharpSyntaxRewriter`:

```
C#  
  
public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

Adicione o seguinte código para declarar um campo somente leitura privado para conter um `SemanticModel` e inicializá-lo no construtor. Você precisará deste campo posteriormente para determinar onde a inferência de tipos pode ser usada:

```
C#  
  
private readonly SemanticModel SemanticModel;  
  
public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel = semanticModel;
```

Substitua o método `VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax)`:

```
C#  
  
public override SyntaxNode  
VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)  
{  
  
}
```

## ⓘ Observação

Muitas das APIs Roslyn declaram os tipos de retorno que são classes base dos tipos de runtime reais retornados. Em muitos cenários, um tipo de nó pode ser substituído inteiramente por outro tipo de nó, ou até mesmo removido. Neste exemplo, o método

`VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax)` retorna `SyntaxNode`, em vez do tipo derivado de `LocalDeclarationStatementSyntax`. Este regravador retorna um novo nó `LocalDeclarationStatementSyntax` baseado no existente.

Este guia de início rápido lida com declarações de variáveis locais. Você poderia estendê-lo para outras declarações, como loops `foreach`, loops `for`, expressões LINQ e expressões lambda. Além disso, este regravador só irá transformar as declarações da forma mais simples:

C#

```
Type variable = expression;
```

Se você quiser explorar por conta própria, considere estender a amostra finalizada para esses tipos de declarações de variáveis:

C#

```
// Multiple variables in a single declaration.  
Type variable1 = expression1,  
      variable2 = expression2;  
// No initializer.  
Type variable;
```

Adicione o seguinte código ao corpo do método `visitLocalDeclarationStatement` para ignorar a reescrita dessas formas de declaração:

C#

```
if (node.Declaration.Variables.Count > 1)  
{  
    return node;  
}  
if (node.Declaration.Variables[0].Initializer == null)  
{  
    return node;  
}
```

O método indica que nenhuma reescrita ocorre retornando o parâmetro `node` não modificado. Se nenhuma dessas expressões `if` for verdadeira, o nó representa uma possível declaração com inicialização. Adicione estas instruções para extrair o nome do tipo especificado na declaração e vinculá-lo usando o campo [SemanticModel](#) para obter um símbolo de tipo:

C#

```
var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

Agora, adicione esta instrução para associar a expressão inicializadora:

C#

```
var initializerInfo =
SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

Por fim, inclua a seguinte instrução `if` para substituir o nome do tipo existente pela palavra-chave `var`, se o tipo de expressão do inicializador corresponder ao tipo especificado:

C#

```
if (SymbolEqualityComparer.Default.Equals(variableType,
initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

    return node.ReplaceNode(variableTypeName, varTypeName);
}
else
{
    return node;
}
```

A condicional é necessária porque a declaração pode converter a expressão inicializadora em uma classe ou interface base. Se este for o caso, os tipos à esquerda e à direita da atribuição não coincidem. Remover o tipo explícito nesses casos alteraria a

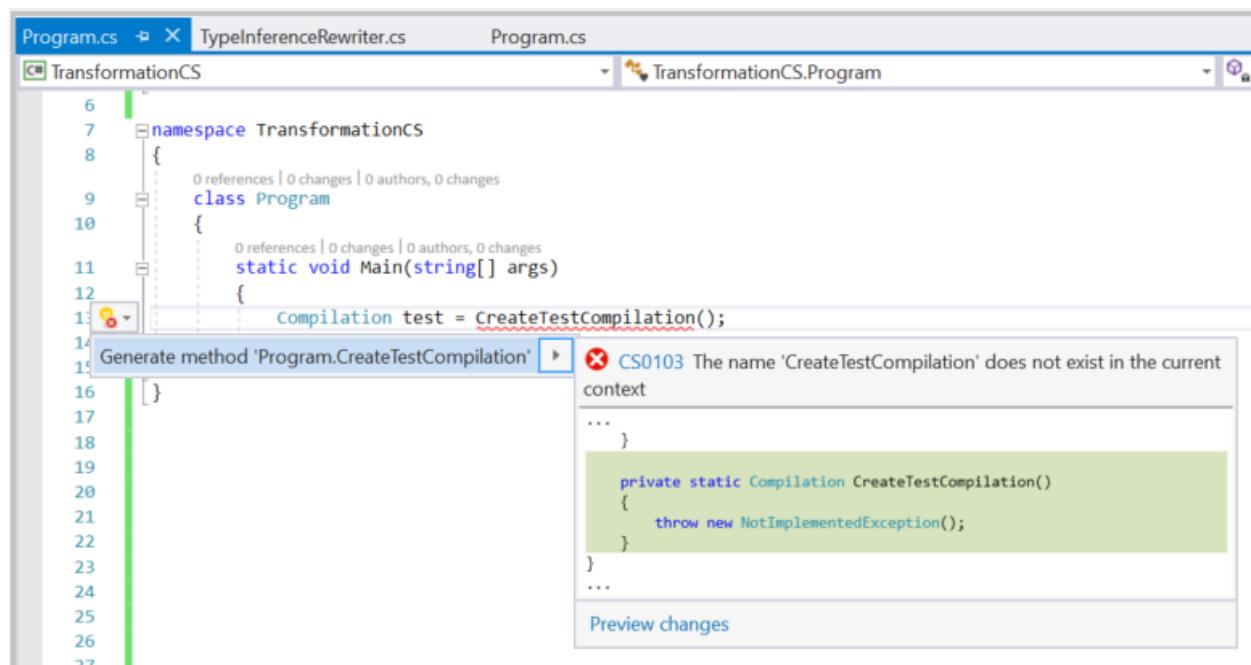
semântica de um programa. `var` é especificado como um identificador em vez de uma palavra-chave porque `var` é uma palavra-chave contextual. As trivialidades inicial e final (espaço em branco) são transferidas do nome do tipo antigo para a palavra-chave `var` para manter espaço em branco vertical e recuo. É mais simples usar `ReplaceNode` em vez de `With*` para transformar o `LocalDeclarationStatementSyntax`, pois o nome do tipo é na verdade o neto da declaração.

Você concluiu o `TypeInferenceRewriter`. Agora, retorne ao arquivo `Program.cs` para finalizar o exemplo. Crie um teste `Compilation` e obtenha o `SemanticModel` dele. Use esse `SemanticModel` para testar seu `TypeInferenceRewriter`. Você realizará esta etapa por último. Enquanto isso, declare uma variável de espaço reservado representando sua compilação de teste:

C#

```
Compilation test = CreateTestCompilation();
```

Após uma pausa, um erro será exibido informando que não existe método `CreateTestCompilation`. Pressione **Ctrl + Ponto** para abrir a lâmpada e pressione **Enter** para invocar o comando **Gerar Stub de Método**. Este comando irá gerar um stub de método para o método `CreateTestCompilation` na classe `Program`. Você voltará a preencher este método mais tarde:



Grave o seguinte código para iterar sobre cada `SyntaxTree` no teste `Compilation`. Para cada um, initialize um novo `TypeInferenceRewriter` com o `SemanticModel` para essa árvore:

C#

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

Dentro da instrução `foreach` criada, adicione o seguinte código para executar a transformação em cada árvore de origem. Este código registra condicionalmente a nova árvore transformada se alguma edição tiver sido feita. Seu regravador só deve modificar uma árvore se encontrar uma ou mais declarações de variáveis locais que poderiam ser simplificadas usando a inferência de tipos:

C#

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

Você deve ver rabiscos abaixo do código `File.WriteAllText`. Selecione a lâmpada e adicione a instrução `using System.IO;` necessária.

Você está quase lá! Resta somente uma etapa: criar um teste [Compilation](#). Como você não usou a inferência de tipos durante este guia de início rápido, este seria um caso de teste perfeito. Infelizmente, criar uma compilação a partir de um arquivo de projeto C# está além do escopo desta explicação passo a passo. Mas, felizmente, se você está seguindo as instruções cuidadosamente, há esperança. Substitua o conteúdo do método `CreateTestCompilation` pelo código a seguir. Ele cria uma compilação de teste que combina coincidentemente com o projeto descrito neste guia de início rápido:

C#

```
String programPath = @"..\..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
```

```

        CSharpSyntaxTree.ParseText(programText)
            .WithFilePath(programPath);

String rewriterPath = @"..\..\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
    CSharpSyntaxTree.ParseText(rewriterText)
        .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };

MetadataReference mscorelib =
    MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =
    MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =
    MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location)
;

MetadataReference[] references = { mscorelib, codeAnalysis,
csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));

```

Cruze os dedos e execute o projeto. No Visual Studio, escolha **Depurar>Iniciar Depuração**. O Visual Studio exibirá um aviso dizendo que os arquivos em seu projeto foram alterados. Clique em "**Sim para Todos**" para recarregar os arquivos modificados. Examine-os para observar sua grandiosidade. Observe como o código parece mais limpo sem todos os especificadores de tipo explícitos e redundantes.

Parabéns! Você usou as **APIs do compilador** para criar sua própria refatoração que pesquisa todos os arquivos em um projeto C# para determinados padrões sintáticos, analisa a semântica do código-fonte que corresponde a esses padrões e os transforma. Agora você é oficialmente um autor de refatoração!

# Tutorial: escrever seu primeiro analisador e correção de código

Artigo • 10/05/2023

O SDK do .NET Compiler Platform fornece as ferramentas necessárias para criar diagnósticos personalizados (analisadores), correções de código, refatoração de código e supressores de diagnóstico destinados a códigos C# ou Visual Basic. Um **analisador** contém código que reconhece violações às suas regras. Sua **correção de código** contém o código que corrige a violação. As regras que você implementar podem ser qualquer coisa, incluindo estrutura do código, estilo de codificação, convenções de nomenclatura e muito mais. O .NET Compiler Platform fornece a estrutura para executar análise conforme os desenvolvedores escrevem o código, bem como todos os recursos de interface do usuário do Visual Studio para corrigir o código: mostrar rabiscos no editor, popular a Lista de Erros do Visual Studio, criar as sugestões da "lâmpada" e mostrar a visualização avançada das correções sugeridas.

Neste tutorial, você explorará a criação de um **analisador** e uma **correção de código** que o acompanha, usando as APIs do Roslyn. Um analisador é uma maneira de executar a análise de código-fonte e relatar um problema para o usuário. Opcionalmente, uma correção de código pode ser associada ao analisador para representar uma modificação no código-fonte do usuário. Este tutorial cria um analisador que localiza as declarações de variável local que poderiam ser declaradas usando o modificador `const`, mas não o são. A correção de código anexa modifica essas declarações para adicionar o modificador `const`.

## Pré-requisitos

- [Visual Studio 2019](#), versão 16.8 ou posterior

Será necessário instalar o **SDK do .NET Compiler Platform** por meio do Instalador do Visual Studio:

## Instruções de instalação – Instalador do Visual Studio

Há duas maneiras diferentes de encontrar o **SDK da .NET Compiler Platform** no **Instalador do Visual Studio**:

## Instalar usando o Instalador do Visual Studio – exibição de cargas de trabalho

O SDK da .NET Compiler Platform não é selecionado automaticamente como parte da carga de trabalho de desenvolvimento da extensão do Visual Studio. É necessário selecioná-lo como um componente opcional.

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Marque a carga de trabalho de **Desenvolvimento de extensão do Visual Studio**.
4. Abra o nó **Desenvolvimento de extensão do Visual Studio** na árvore de resumo.
5. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará por último nos componentes opcionais.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Abra o nó **Componentes individuais** na árvore de resumo.
2. Marque a caixa do **Editor DGML**

## Instalar usando o Instalador do Visual Studio – guia Componentes individuais

1. Execute o **Instalador do Visual Studio**
2. Escolha **Mudar**
3. Selecione a guia **Componentes individuais**
4. Marque a caixa do **SDK da .NET Compiler Platform**. Você a encontrará na parte superior, na seção **Compiladores, ferramentas de compilação e runtimes**.

Opcionalmente, você também poderá fazer o **Editor DGML** exibir gráficos no visualizador:

1. Marque a caixa do **Editor DGML**. Você a encontrará na seção **Ferramentas de código**.

Há várias etapas para criar e validar o analisador:

1. Crie a solução.
2. Registre o nome e a descrição do analisador.
3. Relate os avisos e recomendações do analisador.
4. Implemente a correção de código para aceitar as recomendações.
5. Melhore a análise por meio de testes de unidade.

# Criar a solução

- No Visual Studio, escolha **Arquivo > Novo > Projeto...** para exibir o diálogo Novo Projeto.
- Em **Visual C# > Extensibilidade**, escolha **Analisador com correção de código (.NET Standard)**.
- Nomeie seu projeto como "**MakeConst**" e clique em **OK**.

## ⓘ Observação

Você pode receber um erro de compilação (*MSB4062: não foi possível carregar a tarefa "CompareBuildTaskVersion"*). Para corrigir isso, atualize os pacotes NuGet na solução com o Gerenciador de Pacotes NuGet ou use `Update-Package` na janela do console do Gerenciador de Pacotes.

# Explorar o modelo do analisador

O analisador com o modelo de correção de código cria cinco projetos:

- **MakeConst**, que contém o analisador.
- **MakeConst.CodeFixes**, que contém a correção de código.
- **MakeConst.Package**, que é usado para produzir o pacote NuGet para o analisador e a correção de código.
- **MakeConst.Test**, que é um projeto de teste de unidade.
- **MakeConst.Vsix**, que é o projeto de inicialização padrão que carrega uma segunda instância do Visual Studio com seu novo analisador. Pressione `F5` para iniciar o projeto VSIX.

## ⓘ Observação

Os analisadores devem ter como destino o .NET Standard 2.0 porque podem ser executados nos ambientes do .NET Core (builds de linha de comando) e do .NET Framework (Visual Studio).

## 💡 Dica

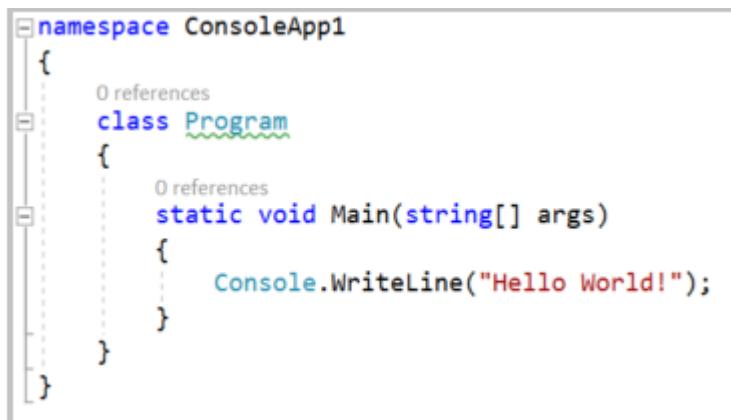
Quando você executa seu analisador, você pode iniciar uma segunda cópia do Visual Studio. Essa segunda cópia usa um hive do Registro diferente para armazenar configurações. Isso lhe permite diferenciar as configurações visuais em

duas cópias do Visual Studio. Você pode escolher um tema diferente para a execução experimental do Visual Studio. Além disso, não use perfil móvel de suas configurações nem faça logon na conta do Visual Studio usando a execução experimental do Visual Studio. Isso mantém as diferenças entre as configurações.

O hive inclui não apenas o analisador em desenvolvimento, mas também todos os analisadores anteriores abertos. Para redefinir o hive da Roslyn, você precisa excluí-lo manualmente de %LocalAppData%\Microsoft\VisualStudio. O nome da pasta do hive da Roslyn terminará em `Roslyn`, por exemplo, `16.0_9ae182f9Roslyn`. Observe que talvez seja necessário limpar a solução e compilar novamente após excluir o hive.

Na segunda instância do Visual Studio que você acabou de iniciar, crie um projeto de Aplicativo de Console do C# (qualquer estrutura de destino funcionará – os analisadores funcionam no nível de origem). Passe o mouse sobre o token com um sublinhado ondulado e o texto de aviso fornecido por um analisador será exibido.

O modelo cria um analisador que relata um aviso em cada declaração de tipo em que o nome do tipo contém letras minúsculas, conforme mostrado na figura a seguir:



The screenshot shows the code editor with the following C# code:

```
namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

The word "Program" is underlined with a wavy green line, indicating a diagnostic from the analyzer.

O modelo também fornece uma correção de código que altera qualquer nome de tipo que contenha caracteres de letras minúsculas, deixando-o com todas as letras maiúsculas. Você pode clicar na lâmpada exibida com o aviso para ver as alterações sugeridas. Aceitar as alterações sugeridas atualiza o nome do tipo e todas as referências para esse tipo na solução. Agora que você já viu o analisador inicial em ação, feche a segunda instância do Visual Studio e retorne ao projeto do analisador.

Você não precisa iniciar uma segunda cópia do Visual Studio e criar um novo código para testar cada alteração em seu analisador. O modelo também cria um projeto de teste de unidade para você. Esse projeto contém dois testes. `TestMethod1` mostra o formato típico de um teste que analisa o código sem disparar um diagnóstico.

`TestMethod2` mostra o formato de um teste que dispara um diagnóstico e, em seguida,

aplica uma correção de código sugerida. Conforme você cria o analisador e a correção de código, você escreve testes para estruturas de código diferentes para verificar seu trabalho. Testes de unidade para os analisadores são muito mais rápidos do que testá-los de forma interativa com o Visual Studio.

### Dica

Testes de unidade de analisador são uma excelente ferramenta quando você sabe quais constructos de código devem e não devem disparar seu analisador. Carregar o analisador em outra cópia do Visual Studio é uma excelente ferramenta para explorar e encontrar constructos nos quais você talvez não tenha pensado ainda.

Neste tutorial, você grava um analisador que relata ao usuário qualquer declaração de variável local que possa ser convertida em constante local. Por exemplo, considere o seguinte código:

```
C#  
  
int x = 0;  
Console.WriteLine(x);
```

No código acima, um valor constante é atribuído a `x`, que nunca é modificado. Ele pode ser declarado usando o modificador `const`:

```
C#  
  
const int x = 0;  
Console.WriteLine(x);
```

A análise para determinar se uma variável pode ser tornada constante está envolvida, exigindo análise sintática, análise constante da expressão de inicializador e também análise de fluxo de dados, para garantir que nunca ocorram gravações na variável. O .NET Compiler Platform fornece APIs que facilitam essa análise.

## Criar registros do analisador

O modelo cria a classe inicial `DiagnosticAnalyzer`, no arquivo `MakeConstAnalyzer.cs`. Esse analisador inicial mostra duas propriedades importantes de cada analisador.

- Cada analisador de diagnóstico deve fornecer um atributo `[DiagnosticAnalyzer]` que descreve a linguagem em que opera.

- Cada analisador de diagnóstico deve derivar (direta ou indiretamente) da classe [DiagnosticAnalyzer](#).

O modelo também mostra os recursos básicos que fazem parte de qualquer analisador:

1. Registrar ações. As ações representam alterações de código que devem disparar o analisador para examinar se há violações de código. Quando o Visual Studio detecta as edições de código que correspondem a uma ação registrada, ele chama o método registrado do analisador.
2. Criar diagnósticos. Quando o analisador detecta uma violação, ele cria um objeto de diagnóstico que o Visual Studio usa para notificar o usuário sobre a violação.

Registrar ações na substituição do método

[DiagnosticAnalyzer.Initialize\(AnalysisContext\)](#). Neste tutorial, você visitará **nós de sintaxe** em busca de declarações locais e verá quais delas têm valores constantes. Se houver possibilidade de uma declaração ser constante, seu analisador criará e relatará um diagnóstico.

A primeira etapa é atualizar as constantes de registro e o método `Initialize`, de modo que essas constantes indiquem seu analisador "Make Const". A maioria das constantes de cadeia de caracteres é definida no arquivo de recurso de cadeia de caracteres. Você deve seguir essa prática para uma localização mais fácil. Abra o arquivo *Resources.resx* para o projeto do analisador **MakeConst**. Isso exibe o editor de recursos. Atualize os recursos de cadeia de caracteres da seguinte maneira:

- Altere `AnalyzerDescription` para "Variables that are not modified should be made constants.".
- Altere `AnalyzerMessageFormat` para "Variable '{0}' can be made constant".
- Altere `AnalyzerTitle` para "Variable can be made constant".

Quando você terminar, o editor de recursos deverá aparecer conforme mostrado na seguinte figura:

	Name	Value	Comment
	AnalyzerDescription	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
	AnalyzerMessageFormat	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
▶	AnalyzerTitle	Variable can be made constant	The title of the diagnostic.
*			

As alterações restantes estão no arquivo do analisador. Abra *MakeConstAnalyzer.cs* no Visual Studio. Altere a ação registrada de uma que age em símbolos para uma que age sobre a sintaxe. No método `MakeConstAnalyzer.Initialize`, localize a linha que registra a ação em símbolos:

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

Substitua-a com a seguinte linha:

C#

```
context.RegisterSyntaxNodeAction(AnalyzeNode,  
SyntaxKind.LocalDeclarationStatement);
```

Após essa alteração, você poderá excluir o método `AnalyzeSymbol`. Este analisador examina `SyntaxKind.LocalDeclarationStatement`, e não instruções `SymbolKind.NamedType`. Observe que `AnalyzeNode` tem rabiscos vermelhos sob ele. O código apenas que você acaba de adicionar referencia um método `AnalyzeNode` que não foi declarado. Declare esse método usando o seguinte código:

C#

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)  
{  
}
```

Altere `Category` para "Usage" em `MakeConstAnalyzer.cs`, conforme mostrado no seguinte código:

C#

```
private const string Category = "Usage";
```

## Localize as declarações locais que podem ser constantes

É hora de escrever a primeira versão do método `AnalyzeNode`. Ele deve procurar uma única declaração local que poderia ser `const` mas não é, algo semelhante ao seguinte código:

C#

```
int x = 0;  
Console.WriteLine(x);
```

A primeira etapa é encontrar declarações locais. Adicione o seguinte código a `AnalyzeNode` em `MakeConstAnalyzer.cs`:

```
C#
```

```
var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

Essa conversão sempre terá êxito porque seu analisador fez o registro para alterações unicamente a declarações locais. Nenhum outro tipo de nó dispara uma chamada para seu método `AnalyzeNode`. Em seguida, verifique a declaração para quaisquer modificadores `const`. Se você encontrá-los, retorne imediatamente. O código a seguir procura por quaisquer modificadores `const` na declaração local:

```
C#
```

```
// make sure the declaration isn't already const:  
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))  
{  
    return;  
}
```

Por fim, você precisa verificar que a variável pode ser `const`. Isso significa assegurar que ela nunca seja atribuída após ser inicializada.

Você executará alguma análise semântica usando o `SyntaxNodeAnalysisContext`. Você usa o argumento `context` para determinar se a declaração de variável local pode ser tornada `const`. Um `Microsoft.CodeAnalysis.SemanticModel` representa todas as informações semânticas em apenas um arquivo de origem. Você pode aprender mais no artigo que aborda [modelos semânticos](#). Você usará o `Microsoft.CodeAnalysis.SemanticModel` para realizar a análise de fluxo de dados na instrução de declaração local. Em seguida, você usa os resultados dessa análise de fluxo de dados para garantir que a variável local não seja escrita com um novo valor em nenhum outro lugar. Chame o método de extensão `GetDeclaredSymbol` para recuperar o `ILocalSymbol` para a variável e verifique se ele não está contido na coleção `DataFlowAnalysis.WrittenOutside` da análise de fluxo de dados. Adicione o seguinte código ao final do método `AnalyzeNode`:

```
C#
```

```
// Perform data flow analysis on the local declaration.  
DataFlowAnalysis dataFlowAnalysis =  
    context.SemanticModel.AnalyzeDataFlow(localDeclaration);  
  
// Retrieve the local symbol for each variable in the local declaration
```

```
// and ensure that it is not written outside of the data flow analysis
region.

VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

O código recém-adicionado garante que a variável não seja modificada e pode, portanto, ser tornada `const`. É hora de gerar o diagnóstico. Adicione o código a seguir como a última linha em `AnalyzeNode`:

C#

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation(),
localDeclaration.Declaration.Variables.First().Identifier.ValueText));
```

Você pode verificar seu andamento pressionando `F5` para executar o analisador. Você pode carregar o aplicativo de console que você criou anteriormente e, em seguida, adicionar o seguinte código de teste:

C#

```
int x = 0;
Console.WriteLine(x);
```

A lâmpada deve aparecer e o analisador deve relatar um diagnóstico. No entanto, dependendo da sua versão do Visual Studio, você verá:

- A lâmpada, que ainda usa a correção de código gerada por modelo e dirá a você que ela pode ser colocada em letras maiúsculas.
- Uma mensagem na barra de notificações, na parte superior do editor, informando que o "MakeConstCodeFixProvider" encontrou um erro e foi desabilitado. Isso ocorre porque o provedor de correção de código ainda não foi alterado e espera encontrar `TypeDeclarationSyntax` elementos em vez de `LocalDeclarationStatementSyntax`.

A próxima seção explica como escrever a correção de código.

## Escrever a correção de código

Um analisador pode fornecer uma ou mais correções de código. Uma correção de código define uma edição que resolve o problema relatado. Para o analisador que você criou, você pode fornecer uma correção de código que insere a palavra-chave `const`:

```
diff
```

```
- int x = 0;
+ const int x = 0;
Console.WriteLine(x);
```

O usuário escolhe-a da lâmpada da interface do usuário no editor e do Visual Studio altera o código.

Abra o arquivo `CodeFixResources.resx` e altere `CodeFixTitle` para "Make constant".

Abra o arquivo `MakeConstCodeFixProvider.cs` adicionado pelo modelo. Essa correção de código já está conectada à ID de Diagnóstico produzida pelo analisador de diagnóstico, mas ela ainda não implementa a transformação de código correta.

Em seguida, exclua o método `MakeUppercaseAsync`. Ele não se aplica mais.

Todos os provedores de correção de código derivam de `CodeFixProvider`. Todos eles substituem `CodeFixProvider.RegisterCodeFixesAsync(CodeFixContext)` para relatar as correções de código disponíveis. Em `RegisterCodeFixesAsync`, altere o tipo de nó ancestral pelo qual você está pesquisando para um `LocalDeclarationStatementSyntax` para corresponder ao diagnóstico:

```
C#
```

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarationStatementSyntax>().First();
```

Em seguida, altere a última linha para registrar uma correção de código. A correção criará um novo documento resultante da adição do modificador `const` para uma declaração existente:

```
C#
```

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: CodeFixResources.CodeFixTitle,
        createChangedDocument: c => MakeConstAsync(context.Document,
declaration, c),
```

```
equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),  
diagnostic);
```

Você observará rabiscos vermelhos no código que você acabou de adicionar no símbolo `MakeConstAsync`. Adicione uma declaração para `MakeConstAsync` semelhante ao seguinte código:

C#

```
private static async Task<Document> MakeConstAsync(Document document,  
    LocalDeclarationStatementSyntax localDeclaration,  
    CancellationToken cancellationToken)  
{  
}
```

Seu novo método `MakeConstAsync` transformará o `Document` que representa o arquivo de origem do usuário em um novo `Document` que agora contém uma declaração `const`.

Você cria um novo token de palavra-chave `const` a ser inserido no início da instrução de declaração. Tenha cuidado para remover qualquer desafio à esquerda do primeiro token de instrução de declaração e anexe-o ao token `const`. Adicione o seguinte código ao método `MakeConstAsync`:

C#

```
// Remove the leading trivia from the local declaration.  
SyntaxToken firstToken = localDeclaration.GetFirstToken();  
SyntaxTriviaList leadingTrivia = firstToken.LeadingTrivia;  
LocalDeclarationStatementSyntax trimmedLocal =  
localDeclaration.ReplaceToken(  
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));  
  
// Create a const token with the leading trivia.  
SyntaxToken constToken = SyntaxFactory.Token(leadingTrivia,  
SyntaxKind.ConstKeyword,  
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

Em seguida, adicione o token `const` à declaração usando o seguinte código:

C#

```
// Insert the const token into the modifiers list, creating a new modifiers  
list.  
SyntaxTokenList newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);  
// Produce the new local declaration.  
LocalDeclarationStatementSyntax newLocal = trimmedLocal
```

```
.WithModifiers(newModifiers)
.WithDeclaration(localDeclaration.Declaration);
```

Em seguida, formate a nova declaração de acordo com regras de formatação de C#. Formatar de suas alterações para corresponderem ao código existente cria uma experiência melhor. Adicione a instrução a seguir imediatamente após o código existente:

C#

```
// Add an annotation to format the new local declaration.
LocalDeclarationStatementSyntax formattedLocal =
newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

Um novo namespace é necessário para esse código. Adicione a seguinte diretiva `using` para a parte superior do arquivo:

C#

```
using Microsoft.CodeAnalysis.Formatting;
```

A etapa final é fazer a edição. Há três etapas para esse processo:

1. Obter um identificador para o documento existente.
2. Criar um novo documento, substituindo a declaração existente pela nova declaração.
3. Retornar o novo documento.

Adicione o seguinte código ao final do método `MakeConstAsync`:

C#

```
// Replace the old local declaration with the new local declaration.
SyntaxNode oldRoot = await
document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
SyntaxNode newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
```

A correção de código está pronta para ser experimentada. Pressione `F5` para executar o projeto do analisador em uma segunda instância do Visual Studio. Na segunda instância do Visual Studio, crie um novo projeto de Aplicativo de Console em C# e adicione algumas declarações de variável local inicializadas com valores constantes para o

método Main. Você verá que elas são relatadas como avisos, conforme mostrado a seguir.

```
static void Main(string[] args)
{
    int i = 1;
    int j = 2i;
    int k = i + j;
}
```

Você fez muito progresso. Há rabiscos sob as declarações que podem ser tornados `const`. Mas ainda há trabalho a fazer. Isso funciona bem se você adicionar `const` às declarações começando com `i`, depois `j` e, por fim, `k`. Mas se você adicionar o modificador `const` em uma ordem diferente, começando com `k`, seu analisador criará erros: `k` não pode ser declarado como `const`, a menos que `i` e `j` já sejam ambos `const`. Você tem que fazer mais análise para assegurar que lida com as diferentes maneiras em que variáveis podem ser declaradas e inicializadas.

## Criar testes de unidade

Seu analisador e correção de código trabalham em um caso simples de uma única declaração que pode ser tornada `const`. Há várias instruções de declaração possíveis em que essa implementação comete erros. Você tratará desses casos trabalhando com a biblioteca de teste de unidade gravada pelo modelo. Isso é muito mais rápido do que abrir repetidamente uma segunda cópia do Visual Studio.

Abra o arquivo `MakeConstUnitTests.cs` no projeto de teste de unidade. O modelo criou dois testes que seguem os dois padrões comuns para um analisador e o teste de unidade de correção de código. `TestMethod1` mostra o padrão para um teste que garante que o analisador não relata um diagnóstico quando não deve fazê-lo. `TestMethod2` mostra o padrão para relatar um diagnóstico e executar a correção de código.

O modelo usa pacotes [Microsoft.CodeAnalysis.Testing](#) para testes de unidade.

### 💡 Dica

A biblioteca de testes dá suporte a uma sintaxe de marcação especial, incluindo o seguinte:

- `[|text|]`: indica que um diagnóstico é relatado para `text`. Por padrão, esse formulário só pode ser usado para testar analisadores com exatamente um

`DiagnosticDescriptor` fornecido por  
`DiagnosticAnalyzer.SupportedDiagnostics`.

- `{|ExpectedDiagnosticId:text|}`: indica que um diagnóstico com `Id` `ExpectedDiagnosticId` é relatado para `text`.

Substitua os testes de modelo na classe `MakeConstUnitTest` pelo seguinte método de teste:

C#

```
[TestMethod]
public async Task LocalIntCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@""
using System;

class Program
{
    static void Main()
    {
        [|int i = 0;|]
        Console.WriteLine(i);
    }
}
", @""
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
}
```

Execute este teste para garantir que ele passa. No Visual Studio, abra o **Gerenciador de Testes** selecionando **Teste>Windows>Gerenciador de Testes**. Então selecione **Executar tudo**.

## Criar testes para declarações válidas

Como regra geral, os analisadores devem sair assim que possível, fazendo o mínimo de trabalho. O Visual Studio chama analisadores registrados conforme o usuário edita o

código. A capacidade de resposta é um requisito fundamental. Há vários casos de teste para o código que não deverão gerar o diagnóstico. O analisador já gerencia um desses testes, o caso em que uma variável é atribuída após ser inicializada. Adicione o seguinte método de teste para representar esse caso:

```
C#  
  
[TestMethod]  
public async Task VariableIsAssigned_NoDiagnostic()  
{  
    await VerifyCS.VerifyAnalyzerAsync(@"  
using System;  
  
class Program  
{  
    static void Main()  
{  
        int i = 0;  
        Console.WriteLine(i++);  
    }  
}  
");  
}
```

Esse teste é aprovado também. Em seguida, adicione métodos de teste para condições que você ainda não gerenciou:

- Declarações que já são `const`, porque elas já são `const`:

```
C#  
  
[TestMethod]  
public async Task VariableIsAlreadyConst_NoDiagnostic()  
{  
    await VerifyCS.VerifyAnalyzerAsync(@"  
using System;  
  
class Program  
{  
    static void Main()  
{  
        const int i = 0;  
        Console.WriteLine(i);  
    }  
}  
");  
}
```

- Declarações que não têm nenhum inicializador, porque não há nenhum valor a ser usado:

```
C#  
  
[TestMethod]  
public async Task NoInitializer_NoDiagnostic()  
{  
    await VerifyCS.VerifyAnalyzerAsync(@"  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        int i;  
        i = 0;  
        Console.WriteLine(i);  
    }  
}  
");  
}
```

- Declarações em que o inicializador não é uma constante, porque elas não podem ser constantes de tempo de compilação:

```
C#  
  
[TestMethod]  
public async Task InitializerIsNotConstant_NoDiagnostic()  
{  
    await VerifyCS.VerifyAnalyzerAsync(@"  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        int i = DateTime.Now.DayOfYear;  
        Console.WriteLine(i);  
    }  
}  
");  
}
```

Isso pode ser ainda mais complicado, porque o C# permite várias declarações como uma instrução. Considere a seguinte constante de cadeia de caracteres de caso de teste:

```
C#
```

```

[TestMethod]
public async Task MultipleInitializers_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0, j = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
        Console.WriteLine(j);
    }
}
");
}

```

A variável `i` pode ser tornada constante, mas o mesmo não se aplica à variável `j`. Portanto, essa instrução não pode ser tornada uma declaração `const`.

Execute os testes novamente e você verá esses novos casos de teste falharem.

## Atualize seu analisador para ignorar as declarações corretas

Você precisa de algumas melhorias no método `AnalyzeNode` do analisador para filtrar o código que corresponde a essas condições. Elas são todas condições relacionadas, portanto, alterações semelhantes corrigirão todas essas condições. Faça as alterações a seguir em `AnalyzeNode`:

- A análise semântica examinou uma única declaração de variável. Esse código deve estar em um loop de `foreach` que examina todas as variáveis declaradas na mesma instrução.
- Cada variável declarada precisa ter um inicializador.
- O inicializador de cada variável declarada precisa ser uma constante de tempo de compilação.

No seu método `AnalyzeNode`, substitua a análise semântica original:

C#

```

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

```

```
// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis
// region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

com o snippet de código a seguir:

```
C#  
  
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    EqualsValueClauseSyntax initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    Optional<object> constantValue =
context.SemanticModel.GetConstantValue(initializer.Value,
context.CancellationToken);
    if (!constantValue.HasValue)
    {
        return;
    }
}  
  
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    // Retrieve the local symbol for each variable in the local declaration
    // and ensure that it is not written outside of the data flow analysis
    // region.
    ISymbol variableSymbol =
context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
    {
        return;
    }
}
```

```
    }  
}
```

O primeiro loop de `foreach` examina cada declaração de variável usando a análise sintática. A primeira verificação garante que a variável tenha um inicializador. A segunda verificação garante que o inicializador seja uma constante. O segundo loop tem a análise semântica original. As verificações semânticas estão em um loop separado porque ele tem um impacto maior no desempenho. Execute os testes novamente e você deverá ver todos eles serem aprovados.

## Adicionar o final polonês

Você está quase lá. Há mais algumas condições com as quais o seu analisador deve lidar. Enquanto o usuário está escrevendo código, o Visual Studio chama os analisadores. Muitas vezes o analisador será chamado para código que não é compilado. O método `AnalyzeNode` do analisador de diagnóstico não verifica para ver se o valor da constante é conversível para o tipo de variável. Assim, a implementação atual converterá facilmente uma declaração incorreta, tal como `int i = "abc"`, em uma constante local. Adicione um método de teste para este caso:

```
C#
```

```
[TestMethod]  
public async Task DeclarationIsInvalid_NoDiagnostic()  
{  
    await VerifyCS.VerifyAnalyzerAsync(@"  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        int x = {|CS0029: ""abc""|};  
    }  
}  
");  
}
```

Além disso, os tipos de referência não são tratados corretamente. O único valor de constante permitido para um tipo de referência é `null`, exceto no caso de `System.String`, que permite literais de cadeia de caracteres. Em outras palavras, `const string s = "abc"` é legal, mas `const object s = "abc"` não é. Este snippet de código verifica essa condição:

C#

```
[TestMethod]
public async Task DeclarationIsNotString_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        object s = ""abc"";;
    }
}
");
}
```

Para ser criterioso, você precisará adicionar outro teste para verificar se pode criar uma declaração de constante para uma cadeia de caracteres. O snippet de código a seguir define o código que gera o diagnóstico e o código após a aplicação da correção:

C#

```
[TestMethod]
public async Task StringCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|string s = ""abc"";|]
    }
},
@", @"
using System;

class Program
{
    static void Main()
    {
        const string s = ""abc"";;
    }
}
");
}
```

Por fim, se uma variável é declarada com a palavra-chave `var`, a correção de código faz a coisa errada e gera uma declaração `const var`, que não é compatível com a linguagem C#. Para corrigir esse bug, a correção de código deve substituir a palavra-chave `var` pelo nome do tipo inferido:

```
C#  
  
[TestMethod]  
public async Task VarIntDeclarationCouldBeConstant_Diagnostic()  
{  
    await VerifyCS.VerifyCodeFixAsync(@"  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        [|var item = 4;|]  
    }  
}  
", @"  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        const int item = 4;  
    }  
}  
");  
}  
  
[TestMethod]  
public async Task VarStringDeclarationCouldBeConstant_Diagnostic()  
{  
    await VerifyCS.VerifyCodeFixAsync(@"  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        [|var item = ""abc"";|]  
    }  
}  
", @"  
using System;  
  
class Program  
{  
    static void Main()  
    {
```

```
    const string item = ""abc"";  
}  
}  
");  
}
```

Felizmente, todos os erros acima podem ser resolvidos usando as mesmas técnicas que você acabou de aprender.

Para corrigir o primeiro bug, primeiro abra *DiagnosticAnalyzer.cs* e localize o loop `foreach` em que cada um dos inicializadores de declaração local é verificado para garantir que valores constantes sejam atribuídos a eles. Imediatamente *antes* do primeiro loop `foreach`, chame `context.SemanticModel.GetTypeInfo()` para recuperar informações detalhadas sobre o tipo declarado da declaração local:

C#

```
TypeSyntax variableTypeName = localDeclaration.Declaration.Type;  
ITypeSymbol variableType =  
context.SemanticModel.GetTypeInfo(variableTypeName,  
context.CancellationToken).ConvertedType;
```

Em seguida, dentro do loop `foreach`, verifique cada inicializador para garantir que ele pode ser convertido no tipo de variável. Adicione a seguinte verificação depois de garantir que o inicializador é uma constante:

C#

```
// Ensure that the initializer value can be converted to the type of the  
// local declaration without a user-defined conversion.  
Conversion conversion =  
context.SemanticModel.ClassifyConversion(initializer.Value, variableType);  
if (!conversion.Exists || conversion.IsUserDefined)  
{  
    return;  
}
```

A próxima alteração é realizada com base na última. Antes da chave de fechamento do primeiro loop `foreach`, adicione o código a seguir para verificar o tipo da declaração de local quando a constante é uma cadeia de caracteres ou valor nulo.

C#

```
// Special cases:  
// * If the constant value is a string, the type of the local declaration  
//   must be System.String.  
// * If the constant value is null, the type of the local declaration must
```

```

//      be a reference type.
if (constantValue.Value is string)
{
    if (variableType.SpecialType != SpecialType.System_String)
    {
        return;
    }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}

```

Você precisa escrever um pouco mais de código no seu provedor de correção de código para substituir a palavra-chave `var` pelo nome do tipo correto. Retorne para `MakeConstCodeFixProvider.cs`. O código que você adicionará realizará as seguintes etapas:

- Verifique se a declaração é uma declaração `var` e se afirmativo:
- Crie um novo tipo para o tipo inferido.
- Certifique-se de que a declaração de tipo não é um alias. Em caso afirmativo, é legal declarar `const var`.
- Certifique-se de que `var` não é um nome de tipo neste programa. (Em caso afirmativo, `const var` é legal).
- Simplificar o nome completo do tipo

Isso soa como muito código. Mas não é. Substitua a linha que declara e inicializa `newLocal` com o código a seguir. Ele é colocado imediatamente após a inicialização de `newModifiers`:

C#

```

// If the type of the declaration is 'var', create a new type name
// for the inferred type.
VariableDeclarationSyntax variableDeclaration =
localDeclaration.Declaration;
TypeSyntax variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    SemanticModel semanticModel = await
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

    // Special case: Ensure that 'var' isn't actually an alias to another
    // type
    // (e.g. using var = System.String).
    IAliasSymbol aliasInfo = semanticModel.GetAliasInfo(variableTypeName,
cancellationToken);
    if (aliasInfo == null)

```

```

{
    // Retrieve the type inferred for var.
    ITypeSymbol type = semanticModel.GetTypeInfo(variableTypeName,
cancellationToken).ConvertedType;

    // Special case: Ensure that 'var' isn't actually a type named
    'var'.
    if (type.Name != "var")
    {
        // Create a new TypeSyntax for the inferred type. Be careful
        // to keep any leading and trailing trivia from the var keyword.
        TypeSyntax typeName =
SyntaxFactory.ParseTypeName(type.ToString())
            .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
            .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

        // Add an annotation to simplify the type name.
        TypeSyntax simplifiedTypeName =
typeName.WithAdditionalAnnotations(Simplifier.Annotation);

        // Replace the type in the variable declaration.
        variableDeclaration =
variableDeclarationWithType(simplifiedTypeName);
    }
}
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal =
trimmedLocal.WithModifiers(newModifiers)
            .WithDeclaration(variableDeclaration);

```

Você precisará adicionar uma diretiva `using` para usar o tipo [Simplifier](#):

C#

```
using Microsoft.CodeAnalysis.Simplification;
```

Execute seus testes, que devem todos ser aprovados. Dê parabéns a si mesmo, executando seu analisador concluído. Pressione `ctrl + F5` para executar o projeto do analisador em uma segunda instância do Visual Studio com a extensão de versão prévia da Roslyn carregada.

- Na segunda instância do Visual Studio, crie um novo projeto de Aplicativo de Console de C# e adicione `int x = "abc";` ao método Main. Graças à primeira correção de bug, nenhum aviso deve ser relatado para esta declaração de variável local (embora haja um erro do compilador, conforme esperado).
- Em seguida, adicione `object s = "abc";` ao método Main. Devido à segunda correção de bug, nenhum aviso deve ser relatado.

- Por fim, adicione outra variável local que usa a palavra-chave `var`. Você verá que um aviso é relatado e uma sugestão é exibida abaixo e a esquerda.
- Mova o cursor do editor sobre o sublinhado ondulado e pressione `ctrl + .` para exibir a correção de código sugerida. Ao selecionar a correção de código, observe que a palavra-chave `var` agora é gerenciada corretamente.

Por fim, adicione o seguinte código:

C#

```
int i = 2;  
int j = 32;  
int k = i + j;
```

Após essas alterações, você obtém linhas onduladas vermelhas apenas nas duas primeiras variáveis. Adicione `const` para ambos `i` e `j`, e você receberá um novo aviso em `k` porque ele agora pode ser `const`.

Parabéns! Você criou sua primeira extensão do .NET Compiler Platform que executa análise de código com o sistema em funcionamento para detectar um problema e fornece uma correção rápida para corrigi-lo. Ao longo do caminho, você aprendeu muitas das APIs de código que fazem parte do SDK do .NET Compiler Platform (APIs do Roslyn). Você pode verificar seu trabalho comparando-o à [amostra concluída](#) em nosso repositório GitHub de exemplos.

## Outros recursos

- [Introdução à análise de sintaxe](#)
- [Introdução à análise semântica](#)

# Guia de programação em C#

Artigo • 15/02/2023

Esta seção fornece informações detalhadas sobre os principais recursos da linguagem C# e recursos acessíveis ao C# por meio do .NET.

Grande parte dessa seção pressupõe que você já sabe algo sobre o C# e conceitos gerais de programação. Se você é totalmente iniciante em programação ou no C#, talvez seja útil visitar os [Tutoriais de Introdução ao C#](#) e o [Tutorial de Introdução ao .NET no Navegador](#), que não exigem conhecimento de programação.

Para obter informações sobre palavras-chave específicas, operadores e diretivas de pré-processador, consulte [Referência de C#](#). Para obter mais informações sobre as especificações da linguagem C#, consulte [Especificações da linguagem C#](#).

## Seções de programa

[Por dentro de um programa em C#](#)

[Main\(\) e argumentos de linha de comando](#)

## Seções da linguagem

[Instruções](#) [Operadores e expressões](#) [Membros aptos para expressão](#) [Comparações de Igualdade](#)

[Types](#)

[Programação orientada a objeto](#)

[Interfaces](#)

[Representantes](#)

[matrizes](#)

[Cadeias de caracteres](#)

[Propriedades](#)

[Indexadores](#)

[Eventos](#)

[Genéricos](#)

[Iteradores](#)

[Expressões de Consulta LINQ](#)

[Namespaces](#)

[Código não seguro e ponteiros](#)

[Comentários da documentação XML](#)

## Seções da plataforma

[Domínios do aplicativo](#)

[Assemblies no .NET](#)

[Atributos](#)

[Coleções](#)

[Exceções e manipulação de exceções](#)

[Sistema de arquivos e o Registro \(Guia de Programação em C#\)](#)

[Interoperabilidade](#)

[Reflexão](#)

## Confira também

- [Referência de C#](#)

# Conceitos de programação (C#)

Artigo • 15/02/2023

Esta seção explica conceitos de programação na linguagem C#.

## Nesta seção

Title	Descrição
<a href="#">Assemblies no .NET</a>	Descreve como criar e usar um assemblies.
<a href="#">Programação assíncrona com async e await (C#)</a>	Descreve como criar soluções assíncronas usando as palavras-chave <code>async</code> e <code>await</code> no C#. Inclui um passo a passo.
<a href="#">Atributos (C#)</a>	Discute como fornecer informações adicionais sobre como programar elementos como tipos, campos, métodos e propriedades por meio de atributos.
<a href="#">Coleções (C#)</a>	Descreve alguns dos tipos de coleções fornecidos pelo .NET. Demonstra como usar coleções simples e coleções de pares chave/valor.
<a href="#">Covariância e contravariância (C#)</a>	Mostra como habilitar a conversão implícita de parâmetros de tipo genérico em interfaces e delegados.
<a href="#">Árvores de expressão (C#)</a>	Explica como você pode usar árvores de expressão para habilitar a modificação dinâmica de código executável.
<a href="#">Iteradores (C#)</a>	Descreve os iteradores, que são usados para percorrer coleções e retornar elementos um por vez.
<a href="#">LINQ (Consulta Integrada à Linguagem) (C#)</a>	Discute os recursos avançados de consulta na sintaxe de linguagem do C# e o modelo para consultar bancos de dados relacionais, documentos XML, conjuntos de dados e coleções na memória.
<a href="#">Reflexão (C#)</a>	Explica como usar a reflexão para criar dinamicamente uma instância de um tipo, associar o tipo a um objeto existente ou obter o tipo de um objeto existente e invocar seus métodos ou acessar suas propriedades e campos.
<a href="#">Serialização (C#)</a>	Descreve os principais conceitos em binário, XML e serialização SOAP.

## Seções relacionadas

- Dicas de desempenho

Discute várias regras básicas que podem ajudá-lo a aumentar o desempenho do seu aplicativo.

# Programação assíncrona

Artigo • 10/05/2023

Se você tiver qualquer necessidade vinculada à E/S (como a solicitação de dados de uma rede, o acesso a um banco de dados ou a leitura e gravação em um sistema de arquivos), você pode querer usar a programação assíncrona. Você também pode ter código vinculado à CPU, como a execução de um cálculo dispendioso, que também é um bom cenário para escrever código assíncrono.

O C# tem um modelo de programação assíncrona em nível de linguagem que permite escrever facilmente código assíncrono sem precisar manipular retornos de chamada ou estar em conformidade com uma biblioteca que dê suporte à assincronia. Ele segue o que é conhecido como [TAP \(Padrão assíncrono baseado em tarefa\)](#).

## Visão geral do modelo assíncrono

O núcleo da programação assíncrona são os objetos `Task` e `Task<T>`, que modelam as operações assíncronas. Eles têm suporte das palavras-chave `async` e `await`. O modelo é bastante simples na maioria dos casos:

- Para código vinculado à E/S, você aguarda uma operação que retorna um `Task` ou `Task<T>` dentro de um método `async`.
- Para o código vinculado à CPU, você aguarda uma operação iniciada em um thread em segundo plano com o método `Task.Run`.

É na palavra-chave `await` que a mágica acontece. Ela cede o controle para o chamador do método que executou `await` e, em última instância, permite que uma interface do usuário tenha capacidade de resposta ou que um serviço seja elástico. Embora [existam maneiras](#) de abordar o código assíncrono diferentes de `async` e `await`, este artigo se concentra nos constructos no nível da linguagem.

## Exemplo vinculado à E/S: baixar dados de um serviço Web

Talvez você queira baixar alguns dados de um serviço Web quando um botão for pressionado, mas não deseja bloquear o thread da interface do usuário. Isso pode ser feito assim, usando a classe [System.Net.Http.HttpClient](#):

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

O código expressa a intenção (baixar alguns dados de forma assíncrona) sem se prender à interação com objetos `Task`.

## Exemplo vinculado à CPU: executar um cálculo para um jogo

Digamos que você está escrevendo um jogo para dispositivo móvel em que, ao pressionar um botão, poderá causar danos a muitos inimigos na tela. A realização do cálculo de dano pode ser dispendiosa e fazê-lo no thread da interface do usuário faria com que o jogo parecesse pausar durante a realização do cálculo!

A melhor maneira de lidar com isso é iniciar um thread em segundo plano que faz o trabalho usando `Task.Run`, e aguardar o resultado usando `await`. Isso permitirá que a interface do usuário funcione de maneira suave enquanto o trabalho está sendo feito.

C#

```
private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
}

calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

Esse código expressa claramente a intenção do evento de clique do botão. Ele não requer o gerenciamento manual de um thread em segundo plano, e faz isso sem bloqueios.

## O que acontece nos bastidores

No lado C# das coisas, o compilador transforma seu código em uma máquina de estado que mantém o controle de situações como transferir a execução quando uma `await` é alcançada e retomar a execução quando um trabalho em segundo plano for concluído.

Para os que gostam da teoria, essa é uma implementação do [Modelo Promise de assincronia ↗](#).

## Informações importantes para entender

- O código assíncrono pode ser usado tanto para o código vinculado à E/S quanto vinculado à CPU, mas de maneira diferente para cada cenário.
- O código assíncrono usa `Task<T>` e `Task`, que são constructos usados para modelar o trabalho que está sendo feito em segundo plano.
- A palavra-chave `async` transforma um método em um método assíncrono, o que permite que você use a palavra-chave `await` em seu corpo.
- Quando a palavra-chave `await` é aplicada, ela suspende o método de chamada e transfere o controle de volta ao seu chamador até que a tarefa em espera seja concluída.
- A `await` só pode ser usada dentro de um método assíncrono.

## Reconhecer trabalho vinculado à CPU e vinculado à E/S

Os primeiros dois exemplos deste guia mostraram como você pode usar `async` e `await` para trabalho vinculado à E/S e vinculado à CPU. É fundamental que você saiba identificar quando um trabalho que você precisa fazer é vinculado à E/S ou vinculado à CPU, porque isso pode afetar significativamente o desempenho do seu código e poderia potencialmente levar ao uso indevido de determinados constructos.

Aqui estão duas perguntas que devem ser feitas antes de escrever qualquer código:

1. Seu código ficará em "espera" por alguma coisa, como dados de um banco de dados?

Se a resposta é "sim", seu trabalho é **vinculado à E/S**.

2. Seu código executará uma computação dispendiosa?

Se você respondeu "sim", seu trabalho é **vinculado à CPU**.

Se o seu trabalho for **vinculado à E/S**, use `async` e `await sem Task.Run`. Você *não deve* usar a biblioteca de paralelismo de tarefas.

Se o seu trabalho for **vinculado à CPU** e você se importa com a capacidade de resposta, use `async` e `await`, mas gere o trabalho em outro thread com `Task.Run`. Se o trabalho for adequado para a simultaneidade e paralelismo, você também deverá considerar o uso da [Biblioteca de paralelismo de tarefas](#).

Além disso, você sempre deve medir a execução do seu código. Por exemplo, talvez você tenha uma situação em que seu trabalho vinculado à CPU não é caro o suficiente em comparação com os custos gerais das trocas de contexto ao realizar o multithreading. Cada opção tem vantagens e desvantagens e você deve escolher o que é correto para a sua situação.

## Mais exemplos

Os exemplos a seguir demonstram várias maneiras para escrever código assíncrono no C#. Elas abordam alguns cenários diferentes que você pode encontrar.

## Extrair dados de uma rede

Este snippet de código baixa o HTML da página inicial em <https://dotnetfoundation.org> e conta o número de vezes que a cadeia de caracteres ".NET" ocorre no HTML. Ele usa o ASP.NET para definir um método do controlador da API Web que realiza essa tarefa, retornando o número.

### ⓘ Observação

Se você pretende fazer análise de HTML no código de produção, não use expressões regulares. Use uma biblioteca de análise.

C#

```
private readonly HttpClient _httpClient = new HttpClient();

[HttpGet, Route("DotNetCount")]
public async Task<int> GetDotNetCount()
```

```

{
    // Suspends GetDotNetCount() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await
    _httpClient.GetStringAsync("https://dotnetfoundation.org");

    return Regex.Matches(html, @"\.\NET").Count;
}

```

Aqui está o mesmo cenário escrito para um aplicativo universal do Windows, que executa a mesma tarefa quando um botão for pressionado:

C#

```

private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task
    // later.
    var getDotNetFoundationHtmlTask =
    _httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a
    // Progress Bar.
    // This is important to do here, before the "await" call, so that the
    // user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends OnSeeTheDotNetsButtonClick(), returning
    // control to its caller.
    // This is what allows the app to be responsive and not block the UI
    // thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.\NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org:
{count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}

```

## Aguardar a conclusão de várias tarefas

Você pode encontrar em uma situação em que precisa recuperar várias partes de dados simultaneamente. A API `Task` contém dois métodos, `Task.WhenAll` e `Task.WhenAny`, que

permitem escrever um código assíncrono que realiza uma espera sem bloqueio em vários trabalhos em segundo plano.

Este exemplo mostra como você pode obter os dados `User` para um conjunto de `userId`s.

C#

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int>
userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
```

Aqui está outro jeito de escrever isso de forma mais sucinta usando LINQ:

C#

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
    return await Task.WhenAll(getUserTasks);
}
```

Embora seja menos código, tome cuidado ao misturar LINQ com código assíncrono. Como o LINQ utiliza a execução adiada (lenta), as chamadas assíncronas não acontecerão imediatamente como em um loop `foreach`, a menos que você force a

sequência gerada a iterar com uma chamada a `.ToList()` ou `.ToArray()`. O exemplo acima usa `Enumerable.ToArray` para executar a consulta ansiosamente e armazenar os resultados em uma matriz. Isso força o código `id => GetUserAsync(id)` a executar e iniciar a tarefa.

## Conselhos e informações importantes

Com a programação assíncrona, há alguns detalhes para ter em mente que podem impedir um comportamento inesperado.

- `async` Os métodos precisam ter uma palavra-chave `await` no corpo ou eles nunca transferirão!

É importante ter isso em mente. Se `await` não for usado no corpo de um método `async`, o compilador do C# gerará um aviso, mas o código será compilado e executado como se fosse um método normal. Isso também seria extremamente ineficiente, pois a máquina de estado gerada pelo compilador do C# para o método assíncrono não realizaria nada.

- Você deve adicionar "Async" como o sufixo de cada nome de método assíncrono que escrever.

Essa é a convenção usada no .NET para diferenciar mais facilmente os métodos síncronos e assíncronos. Isso não se aplica, necessariamente, a alguns métodos que não são explicitamente chamados pelo seu código (como manipuladores de eventos ou métodos do controlador da Web). Como eles não são chamados explicitamente pelo seu código, não é tão importante ser explícito em relação à nomenclatura.

- `async void` O só deve ser usado para manipuladores de eventos.

O `async void` é a única maneira de permitir que os manipuladores de eventos assíncronos trabalhem, pois os eventos não têm tipos de retorno (portanto, não podem fazer uso de `Task` e `Task<T>`). Qualquer outro uso de `async void` não segue o modelo TAP e pode ser um desafio utilizá-lo, como:

- As exceções lançadas em um método `async void` não podem ser capturadas fora desse método.
- Métodos `async void` são difíceis de testar.
- Métodos `async void` poderão causar efeitos colaterais indesejados se o chamador não estiver esperando que eles sejam assíncronos.

- Vá com cuidado ao usar lambdas assíncronas em expressões LINQ

As expressões lambda em LINQ usam a execução adiada, o que significa que o código poderia acabar executando em um momento que você não está esperando. A introdução de tarefas de bloqueio no meio disso poderia facilmente resultar em um deadlock, se não estivessem escritas corretamente. Além disso, o aninhamento de código assíncrono dessa maneira também pode dificultar a ponderação a respeito da execução do código. A assíncrona e a LINQ são poderosas, mas devem ser usadas de uma maneira mais cuidadosa e clara possível.

- **Escrever código que aguarda tarefas de uma maneira sem bloqueio**

Bloquear o thread atual como um meio de aguardar a conclusão de uma `Task` pode resultar em deadlocks e threads de contexto bloqueados e pode exigir tratamento de erros significativamente mais complexo. A tabela a seguir fornece diretrizes de como lidar com a espera de tarefas de uma forma sem bloqueio:

Use isto...	Em vez disto...	Quando desejar fazer isso...
<code>await</code>	<code>Task.Wait</code> ou <code>Task.Result</code>	Recuperação do resultado de uma tarefa em segundo plano
<code>await</code> <code>Task.WhenAny</code>	<code>Task.WaitAny</code>	Aguardar a conclusão de qualquer tarefa
<code>await</code> <code>Task.WhenAll</code>	<code>Task.WaitAll</code>	Aguardar a conclusão de todas as tarefas
<code>await Task.Delay</code>	<code>Thread.Sleep</code>	Aguardar por um período de tempo

- **Considere usar `ValueTask` sempre que possível**

Retornar um objeto `Task` de métodos assíncronos pode introduzir gargalos de desempenho em determinados caminhos. `Task` é um tipo de referência, portanto, usá-lo significa alocar um objeto. Em casos em que um método declarado com o modificador `async` retorna um resultado armazenado em cache ou é concluído de forma síncrona, as alocações extras podem se tornar um custo de tempo significativo em seções críticas de desempenho de código. Isso pode se tornar caro se essas alocações ocorrem em loops rígidos. Para obter mais informações, consulte [Tipos de retorno assíncronos generalizados](#).

- **Considere usar `ConfigureAwait(false)`**

Uma pergunta comum é: "quando devo usar o método `Task.ConfigureAwait(Boolean?)`?". O método permite que uma instância `Task` configure seu awaiter. Essa é uma consideração importante, e defini-la

incorretamente pode potencialmente ter implicações de desempenho e até deadlocks. Para obter mais informações sobre `ConfigureAwait`, consulte [FAQ do ConfigureAwait](#).

- **Escrever código com menos monitoração de estado**

Não depender do estado de objetos globais ou da execução de determinados métodos. Em vez disso, depender apenas dos valores retornados dos métodos. Por quê?

- Será mais fácil raciocinar sobre o código.
- O código será mais fácil de testar.
- Misturar código assíncrono e síncrono será muito mais simples.
- As condições de corrida poderão, normalmente, ser completamente evitadas.
- Dependendo dos valores retornados, a coordenação de código assíncrono se tornará simples.
- (Bônus) funciona muito bem com a injeção de dependência.

Uma meta recomendada é alcançar a [Transparência referencial](#) completa ou quase completa em seu código. Isso resultará em uma base de código previsível, testável e de fácil manutenção.

## Outros recursos

- [Modelo de programação assíncrono de tarefas \(C#\)](#).

# Asynchronous programming with `async` and `await`

Article • 02/13/2023

The [Task asynchronous programming model \(TAP\)](#) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs many transformations because some of those statements may start work and return a [Task](#) that represents the ongoing work.

That's the goal of this syntax: enable code that reads like a sequence of statements, but executes in a much more complicated order based on external resource allocation and when tasks are complete. It's analogous to how people give instructions for processes that include asynchronous tasks. Throughout this article, you'll use an example of instructions for making breakfast to see how the `async` and `await` keywords make it easier to reason about code that includes a series of asynchronous instructions. You'd write the instructions something like the following list to explain how to make a breakfast:

1. Pour a cup of coffee.
2. Heat a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

If you have experience with cooking, you'd execute those instructions **asynchronously**. You'd start warming the pan for eggs, then start the bacon. You'd put the bread in the toaster, then start the eggs. At each step of the process, you'd start a task, then turn your attention to tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all these tasks. Continuing the breakfast analogy, one person can make breakfast asynchronously by starting the next task before the first task completes. The cooking progresses whether or not someone is watching it. As soon as you start warming the pan for the eggs, you can begin frying the bacon. Once the bacon starts, you can put the bread into the toaster.

For a parallel algorithm, you'd need multiple cooks (or threads). One would make the eggs, one the bacon, and so on. Each one would be focused on just that one task. Each

cook (or thread) would be blocked synchronously waiting for the bacon to be ready to flip, or the toast to pop.

Now, consider those same instructions written as C# statements:

C#

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
    // example. They are simply marker classes for the purpose of demonstration,
    // contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
        static void Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            Egg eggs = FryEggs(2);
            Console.WriteLine("eggs are ready");

            Bacon bacon = FryBacon(3);
            Console.WriteLine("bacon is ready");

            Toast toast = ToastBread(2);
            ApplyButter(toast);
            ApplyJam(toast);
            Console.WriteLine("toast is ready");

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");
    }
}
```

```

    private static void ApplyButter(Toast toast) =>
        Console.WriteLine("Putting butter on the toast");

    private static Toast ToastBread(int slices)
    {
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("Putting a slice of bread in the
toaster");
        }
        Console.WriteLine("Start toasting...");
        Task.Delay(3000).Wait();
        Console.WriteLine("Remove toast from toaster");

        return new Toast();
    }

    private static Bacon FryBacon(int slices)
    {
        Console.WriteLine($"putting {slices} slices of bacon in the
pan");
        Console.WriteLine("cooking first side of bacon...");
        Task.Delay(3000).Wait();
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("flipping a slice of bacon");
        }
        Console.WriteLine("cooking the second side of bacon...");
        Task.Delay(3000).Wait();
        Console.WriteLine("Put bacon on plate");

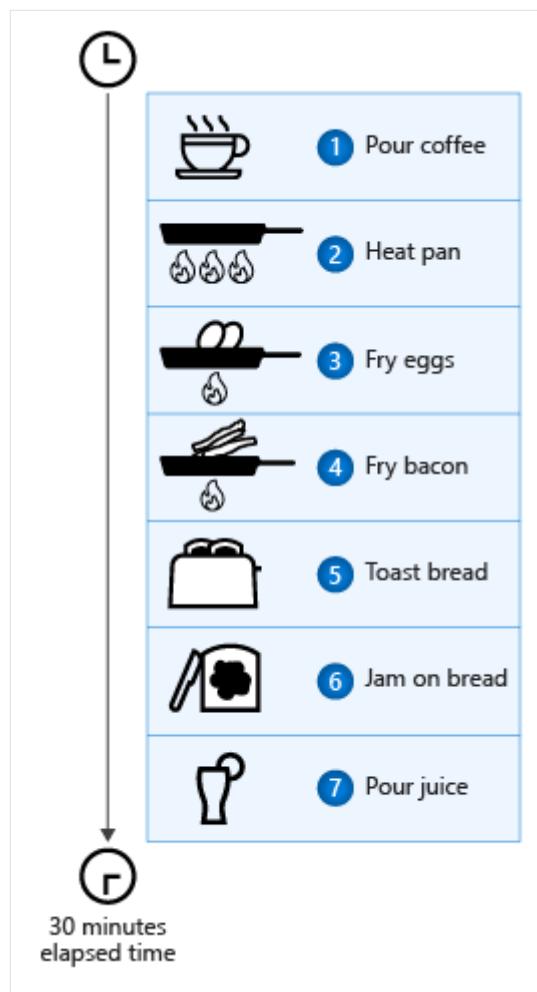
        return new Bacon();
    }

    private static Egg FryEggs(int howMany)
    {
        Console.WriteLine("Warming the egg pan...");
        Task.Delay(3000).Wait();
        Console.WriteLine($"cracking {howMany} eggs");
        Console.WriteLine("cooking the eggs ...");
        Task.Delay(3000).Wait();
        Console.WriteLine("Put eggs on plate");

        return new Egg();
    }

    private static Coffee PourCoffee()
    {
        Console.WriteLine("Pouring coffee");
        return new Coffee();
    }
}

```



The synchronously prepared breakfast took roughly 30 minutes because the total is the sum of each task.

Computers don't interpret those instructions the same way people do. The computer will block on each statement until the work is complete before moving on to the next statement. That creates an unsatisfying breakfast. The later tasks wouldn't be started until the earlier tasks had been completed. It would take much longer to create the breakfast, and some items would have gotten cold before being served.

If you want the computer to execute the above instructions asynchronously, you must write asynchronous code.

These concerns are important for the programs you write today. When you write client programs, you want the UI to be responsive to user input. Your application shouldn't make a phone appear frozen while it's downloading data from the web. When you write server programs, you don't want threads blocked. Those threads could be serving other requests. Using synchronous code when asynchronous alternatives exist hurts your ability to scale out less expensively. You pay for those blocked threads.

Successful modern applications require asynchronous code. Without language support, writing asynchronous code required callbacks, completion events, or other means that obscured the original intent of the code. The advantage of the synchronous code is that

its step-by-step actions make it easy to scan and understand. Traditional asynchronous models forced you to focus on the asynchronous nature of the code, not on the fundamental actions of the code.

## Don't block, await instead

The preceding code demonstrates a bad practice: constructing synchronous code to perform asynchronous operations. As written, this code blocks the thread executing it from doing any other work. It won't be interrupted while any of the tasks are in progress. It would be as though you stared at the toaster after putting the bread in. You'd ignore anyone talking to you until the toast popped.

Let's start by updating this code so that the thread doesn't block while tasks are running. The `await` keyword provides a non-blocking way to start a task, then continue execution when that task completes. A simple asynchronous version of the make a breakfast code would look like the following snippet:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

### ⓘ Important

The total elapsed time is roughly the same as the initial synchronous version. The code has yet to take advantage of some of the key features of asynchronous programming.

## 💡 Tip

The method bodies of the `FryEggsAsync`, `FryBaconAsync`, and `ToastBreadAsync` have all been updated to return `Task<Egg>`, `Task<Bacon>`, and `Task<Toast>` respectively. The methods are renamed from their original version to include the "Async" suffix. Their implementations are shown as part of the **final version** later in this article.

## ⓘ Note

The `Main` method returns `Task`, despite not having a `return` expression—this is by design. For more information, see [Evaluation of a void-returning async function](#).

This code doesn't block while the eggs or the bacon are cooking. This code won't start any other tasks though. You'd still put the toast in the toaster and stare at it until it pops. But at least, you'd respond to anyone that wanted your attention. In a restaurant where multiple orders are placed, the cook could start another breakfast while the first is cooking.

Now, the thread working on the breakfast isn't blocked while awaiting any started task that hasn't yet finished. For some applications, this change is all that's needed. A GUI application still responds to the user with just this change. However, for this scenario, you want more. You don't want each of the component tasks to be executed sequentially. It's better to start each of the component tasks before awaiting the previous task's completion.

## Start tasks concurrently

In many scenarios, you want to start several independent tasks immediately. Then, as each task finishes, you can continue other work that's ready. In the breakfast analogy, that's how you get breakfast done more quickly. You also get everything done close to the same time. You'll get a hot breakfast.

The `System.Threading.Tasks.Task` and related types are classes you can use to reason about tasks that are in progress. That enables you to write code that more closely resembles the way you'd create breakfast. You'd start cooking the eggs, bacon, and toast at the same time. As each requires action, you'd turn your attention to that task, take care of the next action, then wait for something else that requires your attention.

You start a task and hold on to the `Task` object that represents the work. You'll `await` each task before working with its result.

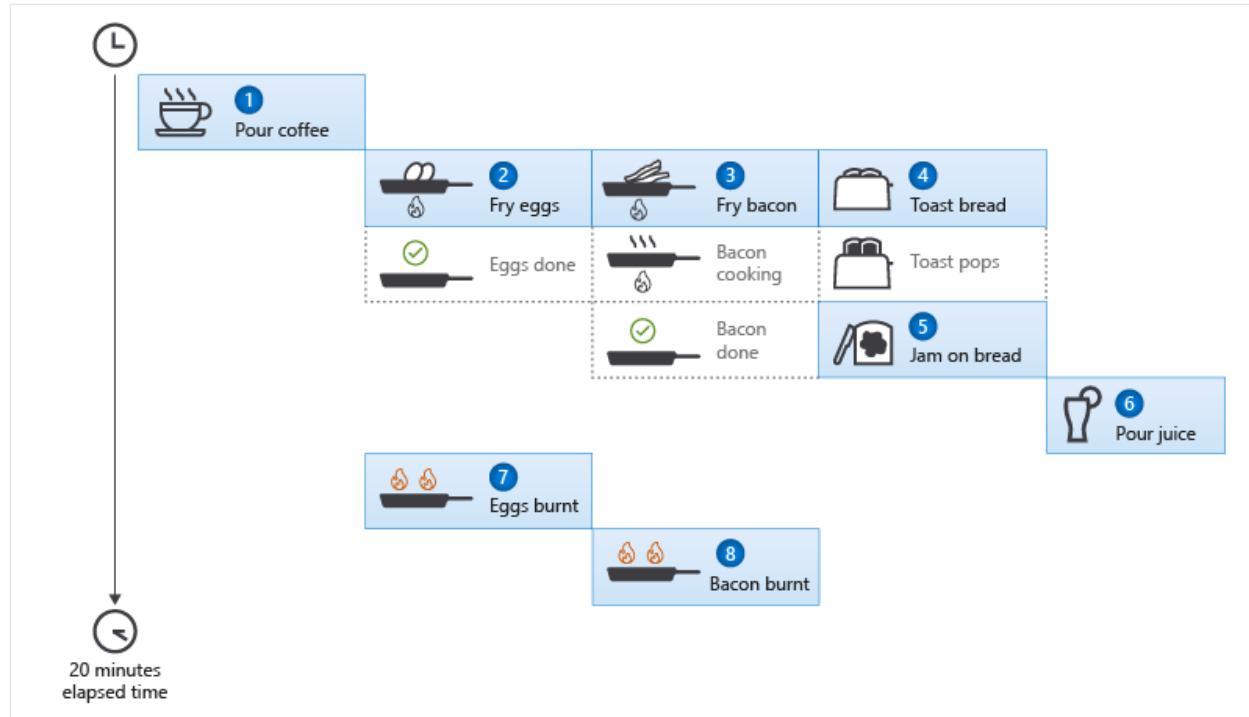
Let's make these changes to the breakfast code. The first step is to store the tasks for operations when they start, rather than awaiting them:

```
C#  
  
Coffee cup = PourCoffee();  
Console.WriteLine("Coffee is ready");  
  
Task<Egg> eggsTask = FryEggsAsync(2);  
Egg eggs = await eggsTask;  
Console.WriteLine("Eggs are ready");  
  
Task<Bacon> baconTask = FryBaconAsync(3);  
Bacon bacon = await baconTask;  
Console.WriteLine("Bacon is ready");  
  
Task<Toast> toastTask = ToastBreadAsync(2);  
Toast toast = await toastTask;  
ApplyButter(toast);  
ApplyJam(toast);  
Console.WriteLine("Toast is ready");  
  
Juice oj = PourOJ();  
Console.WriteLine("Oj is ready");  
Console.WriteLine("Breakfast is ready!");
```

The preceding code won't get your breakfast ready any faster. The tasks are all awaited as soon as they are started. Next, you can move the `await` statements for the bacon and eggs to the end of the method, before serving breakfast:

```
C#  
  
Coffee cup = PourCoffee();  
Console.WriteLine("Coffee is ready");  
  
Task<Egg> eggsTask = FryEggsAsync(2);  
Task<Bacon> baconTask = FryBaconAsync(3);  
Task<Toast> toastTask = ToastBreadAsync(2);  
  
Toast toast = await toastTask;  
ApplyButter(toast);  
ApplyJam(toast);  
Console.WriteLine("Toast is ready");  
Juice oj = PourOJ();  
Console.WriteLine("Oj is ready");  
  
Egg eggs = await eggsTask;  
Console.WriteLine("Eggs are ready");  
Bacon bacon = await baconTask;  
Console.WriteLine("Bacon is ready");
```

```
Console.WriteLine("Breakfast is ready!");
```



The asynchronously prepared breakfast took roughly 20 minutes, this time savings is because some tasks ran concurrently.

The preceding code works better. You start all the asynchronous tasks at once. You await each task only when you need the results. The preceding code may be similar to code in a web application that makes requests to different microservices, then combines the results into a single page. You'll make all the requests immediately, then `await` all those tasks and compose the web page.

## Composition with tasks

You have everything ready for breakfast at the same time except the toast. Making the toast is the composition of an asynchronous operation (toasting the bread), and synchronous operations (adding the butter and the jam). Updating this code illustrates an important concept:

### ⓘ Important

The composition of an asynchronous operation followed by synchronous work is an asynchronous operation. Stated another way, if any portion of an operation is asynchronous, the entire operation is asynchronous.

The preceding code showed you that you can use `Task` or `Task<TResult>` objects to hold running tasks. You `await` each task before using its result. The next step is to create methods that represent the combination of other work. Before serving breakfast, you want to await the task that represents toasting the bread before adding butter and jam. You can represent that work with the following code:

C#

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

The preceding method has the `async` modifier in its signature. That signals to the compiler that this method contains an `await` statement; it contains asynchronous operations. This method represents the task that toasts the bread, then adds butter and jam. This method returns a `Task<TResult>` that represents the composition of those three operations. The main block of code now becomes:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

The previous change illustrated an important technique for working with asynchronous code. You compose tasks by separating the operations into a new method that returns a task. You can choose when to await that task. You can start other tasks concurrently.

## Asynchronous exceptions

Up to this point, you've implicitly assumed that all these tasks complete successfully. Asynchronous methods throw exceptions, just like their synchronous counterparts. Asynchronous support for exceptions and error handling strives for the same goals as asynchronous support in general: You should write code that reads like a series of synchronous statements. Tasks throw exceptions when they can't complete successfully. The client code can catch those exceptions when a started task is awaited. For example, let's assume that the toaster catches fire while making the toast. You can simulate that by modifying the `ToastBreadAsync` method to match the following code:

C#

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

### ⓘ Note

You'll get a warning when you compile the preceding code regarding unreachable code. That's intentional, because once the toaster catches fire, operations won't proceed normally.

Run the application after making these changes, and you'll output similar to the following text:

Console

```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
Cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a slice of bacon
Flipping a slice of bacon
Flipping a slice of bacon
Cooking the second side of bacon...
Cracking 2 eggs
Cooking the eggs ...
Put bacon on plate
Put eggs on plate
Eggs are ready
Bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on
fire
    at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in
Program.cs:line 65
    at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in
Program.cs:line 36
    at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
    at AsyncBreakfast.Program.<Main>(String[] args)
```

You'll notice quite a few tasks are completed between when the toaster catches fire and the exception is observed. When a task that runs asynchronously throws an exception, that Task is *faulted*. The Task object holds the exception thrown in the [Task.Exception](#) property. Faulted tasks throw an exception when they're awaited.

There are two important mechanisms to understand: how an exception is stored in a faulted task, and how an exception is unpackaged and rethrown when code awaits a faulted task.

When code running asynchronously throws an exception, that exception is stored in the [Task](#). The [Task.Exception](#) property is a [System.AggregateException](#) because more than one exception may be thrown during asynchronous work. Any exception thrown is added to the [AggregateException.InnerExceptions](#) collection. If that [Exception](#) property is null, a new [AggregateException](#) is created and the thrown exception is the first item in the collection.

The most common scenario for a faulted task is that the [Exception](#) property contains exactly one exception. When code [awaits](#) a faulted task, the first exception in the [AggregateException.InnerExceptions](#) collection is rethrown. That's why the output from

this example shows an `InvalidOperationException` instead of an `AggregateException`. Extracting the first inner exception makes working with asynchronous methods as similar as possible to working with their synchronous counterparts. You can examine the `Exception` property in your code when your scenario may generate multiple exceptions.

Before going on, comment out these two lines in your `ToastBreadAsync` method. You don't want to start another fire:

```
C#
```

```
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

## Await tasks efficiently

The series of `await` statements at the end of the preceding code can be improved by using methods of the `Task` class. One of those APIs is `WhenAll`, which returns a `Task` that completes when all the tasks in its argument list have completed, as shown in the following code:

```
C#
```

```
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("Eggs are ready");
Console.WriteLine("Bacon is ready");
Console.WriteLine("Toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Another option is to use `WhenAny`, which returns a `Task<Task>` that completes when any of its arguments complete. You can await the returned task, knowing that it has already finished. The following code shows how you could use `WhenAny` to await the first task to finish and then process its result. After processing the result from the completed task, you remove that completed task from the list of tasks passed to `WhenAny`.

```
C#
```

```
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("Eggs are ready");
```

```
        }
        else if (finishedTask == baconTask)
        {
            Console.WriteLine("Bacon is ready");
        }
        else if (finishedTask == toastTask)
        {
            Console.WriteLine("Toast is ready");
        }
        await finishedTask;
        breakfastTasks.Remove(finishedTask);
    }
```

Near the end, you see the line `await finishedTask;`. The line `await Task.WhenAny` doesn't await the finished task. It `awaits` the `Task` returned by `Task.WhenAny`. The result of `Task.WhenAny` is the task that has completed (or faulted). You should `await` that task again, even though you know it's finished running. That's how you retrieve its result, or ensure that the exception causing it to fault gets thrown.

After all those changes, the final version of the code looks like this:

C#

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
    // example. They are simply marker classes for the purpose of demonstration,
    // contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
```

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var breakfastTasks = new List<Task> { eggsTask, baconTask,
toastTask };
    while (breakfastTasks.Count > 0)
    {
        Task finishedTask = await Task.WhenAny(breakfastTasks);
        if (finishedTask == eggsTask)
        {
            Console.WriteLine("eggs are ready");
        }
        else if (finishedTask == baconTask)
        {
            Console.WriteLine("bacon is ready");
        }
        else if (finishedTask == toastTask)
        {
            Console.WriteLine("toast is ready");
        }
        await finishedTask;
        breakfastTasks.Remove(finishedTask);
    }

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}

static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}

private static Juice PourOJ()
{
    Console.WriteLine("Pouring orange juice");
    return new Juice();
}

private static void ApplyJam(Toast toast) =>
    Console.WriteLine("Putting jam on the toast");

private static void ApplyButter(Toast toast) =>
    Console.WriteLine("Putting butter on the toast");
```

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the
toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(3000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}

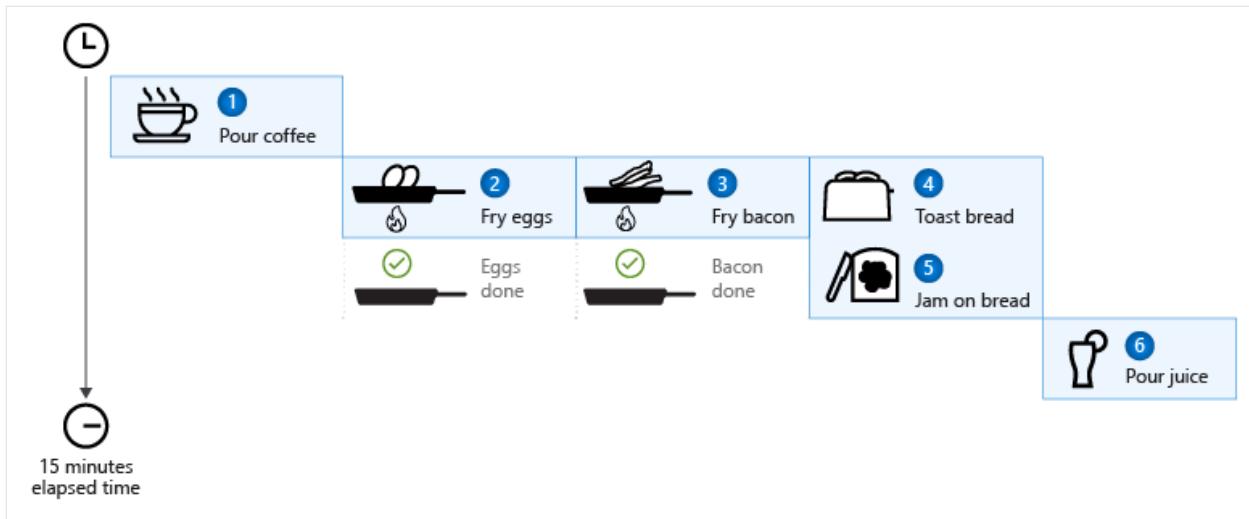
private static async Task<Bacon> FryBaconAsync(int slices)
{
    Console.WriteLine($"putting {slices} slices of bacon in the
pan");
    Console.WriteLine("cooking first side of bacon...");
    await Task.Delay(3000);
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("flipping a slice of bacon");
    }
    Console.WriteLine("cooking the second side of bacon...");
    await Task.Delay(3000);
    Console.WriteLine("Put bacon on plate");

    return new Bacon();
}

private static async Task<Egg> FryEggsAsync(int howMany)
{
    Console.WriteLine("Warming the egg pan...");
    await Task.Delay(3000);
    Console.WriteLine($"cracking {howMany} eggs");
    Console.WriteLine("cooking the eggs ...");
    await Task.Delay(3000);
    Console.WriteLine("Put eggs on plate");

    return new Egg();
}

private static Coffee PourCoffee()
{
    Console.WriteLine("Pouring coffee");
    return new Coffee();
}
}
```



The final version of the asynchronously prepared breakfast took roughly 6 minutes because some tasks ran concurrently, and the code monitored multiple tasks at once and only took action when it was needed.

This final code is asynchronous. It more accurately reflects how a person would cook a breakfast. Compare the preceding code with the first code sample in this article. The core actions are still clear from reading the code. You can read this code the same way you'd read those instructions for making a breakfast at the beginning of this article. The language features for `async` and `await` provide the translation every person makes to follow those written instructions: start tasks as you can and don't block waiting for tasks to complete.

## Next steps

[Explore real world scenarios for asynchronous programs](#)

# Modelo de programação assíncrona de tarefa

Artigo • 28/03/2023

É possível evitar gargalos de desempenho e aprimorar a resposta geral do seu aplicativo usando a programação assíncrona. No entanto, as técnicas tradicionais para escrever aplicativos assíncronos podem ser complicadas, dificultando sua escrita, depuração e manutenção.

O C# apresenta uma abordagem simplificada à programação assíncrona que faz uso do suporte assíncrono no runtime do .NET. O compilador faz o trabalho difícil que o desenvolvedor costumava fazer, e seu aplicativo mantém a estrutura lógica que se assemelha ao código síncrono. Como resultado, você obtém todas as vantagens da programação assíncrona com uma fração do esforço.

Este tópico oferece uma visão geral de quando e como usar a programação assíncrona e inclui links para tópicos de suporte que contêm detalhes e exemplos.

## A assincronia melhora a capacidade de resposta

A assincronia é essencial para atividades que são potencialmente de bloqueio, como o acesso via Web. O acesso a um recurso da Web às vezes é lento ou atrasado. Se tal atividade for bloqueada em um processo síncrono, todo o aplicativo deverá esperar. Em um processo assíncrono, o aplicativo poderá prosseguir com outro trabalho que não dependa do recurso da Web até a tarefa potencialmente causadora do bloqueio terminar.

A tabela a seguir mostra as áreas típicas onde a programação assíncrona melhora a resposta. As APIs listadas do .NET e do Windows Runtime contêm métodos que dão suporte à programação assíncrona.

Área do aplicativo	Tipos .NET com métodos assíncronos	Tipos Windows Runtime com métodos assíncronos
Acesso à Web	<a href="#">HttpClient</a>	<a href="#">Windows.Web.Http.HttpClient</a> <a href="#">SyndicationClient</a>
Trabalhando com arquivos	<a href="#">JsonSerializer</a> <a href="#">StreamReader</a> <a href="#">StreamWriter</a>	<a href="#">StorageFile</a>

Área do aplicativo	Tipos .NET com métodos assíncronos	Tipos Windows Runtime com métodos assíncronos
	<a href="#">XmlReader</a> <a href="#">XmlWriter</a>	
Trabalhando com imagens		<a href="#">MediaCapture</a> <a href="#">BitmapEncoder</a> <a href="#">BitmapDecoder</a>
Programação WCF	<a href="#">Operações síncronas e assíncronas</a>	

A assincronia é especialmente importante para aplicativos que acessam o thread de interface de usuário porque todas as atividades relacionadas à interface do usuário normalmente compartilham um único thread. Se um processo for bloqueado em um aplicativo síncrono, todos serão bloqueados. Seu aplicativo para de responder, o que poderia levar você a concluir que ele falhou quando, na verdade, está apenas aguardando.

Quando você usa métodos assíncronos, o aplicativo continua a responder à interface do usuário. Você poderá redimensionar ou minimizar uma janela, por exemplo, ou fechar o aplicativo se você não desejar aguardar sua conclusão.

A abordagem baseada em assincronia adiciona o equivalente de uma transmissão automática à lista de opções disponíveis para escolha ao criar operações assíncronas. Ou seja, você obtém todos os benefícios da programação assíncrona tradicional, mas com muito menos esforço do desenvolvedor.

## Métodos assíncronos são mais fáceis de escrever

As palavras-chave `async` e `await` em C# são a parte central da programação assíncrona. Ao usar essas duas palavras-chave, você poderá usar recursos do .NET Framework, do .NET Core ou do Windows Runtime para criar um método assíncrono quase que tão facilmente quanto criar um método síncrono. Os métodos assíncronos que você define usando a palavra-chave `async` são chamados de *métodos assíncronos*.

O exemplo a seguir mostra um método assíncrono. Você deve estar familiarizado com quase tudo no código.

Você pode encontrar um exemplo completo de WPF (Windows Presentation Foundation) disponível para download na [Programação assíncrona com `async` e `await` em C#](#).

C#

```
public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://learn.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

Você pode aprender várias práticas com a amostra anterior. Comece com a assinatura do método. Ele inclui o modificador `async`. O tipo de retorno é `Task<int>` (confira a seção "Tipos de retorno" para obter mais opções). O nome do método termina com `Async`. No corpo do método, `GetStringAsync` retorna uma `Task<string>`. Isso significa que, quando você executar `await` em uma tarefa, obterá uma `string` (`contents`). Antes de aguardar a tarefa, você poderá fazer um trabalho que não dependa da `string` em `GetStringAsync`.

Preste muita atenção no operador `await`. Ele suspende `GetUrlContentLengthAsync`:

- `GetUrlContentLengthAsync` não poderá continuar enquanto `getStringTask` não for concluída.
- Enquanto isso, o controle é retornado ao chamador de `GetUrlContentLengthAsync`.
- O controle será retomado aqui quando a `getStringTask` for concluída.
- Em seguida, o operador `await` recupera o resultado `string` de `getStringTask`.

A instrução de retorno especifica um resultado inteiro. Os métodos que estão aguardando `GetUrlContentLengthAsync` recuperar o valor de comprimento.

Se `GetUrlContentLengthAsync` não tiver nenhum trabalho que possa fazer entre chamar `GetStringAsync` e aguardar a conclusão, você poderá simplificar o código ao chamar e esperar na instrução única a seguir.

C#

```
string contents = await  
client.GetStringAsync("https://learn.microsoft.com/dotnet");
```

As seguintes características resumem o que transforma o exemplo anterior em um método assíncrono:

- A assinatura do método inclui um modificador `async`.
- O nome de um método assíncrono, por convenção, termina com um sufixo "Async".
- O tipo de retorno é um dos seguintes tipos:
  - `Task<TResult>` se o método possui uma instrução de retorno em que o operando tem o tipo `TResult`.
  - `Task` se o método não possui instrução de retorno alguma ou se ele possui uma instrução de retorno sem operando.
  - `void` se você estiver escrevendo um manipulador de eventos assíncronos.
  - Qualquer outro tipo que tenha um método `GetAwaiter`.

Para obter mais informações, confira a seção [Tipos e parâmetros de retorno](#).

- O método geralmente inclui, pelo menos, uma expressão `await`, que marca um ponto em que o método não poderá continuar enquanto a operação assíncrona aguardada não for concluída. Enquanto isso, o método é suspenso e o controle retorna para o chamador do método. A próxima seção deste tópico ilustra o que acontece no ponto de suspensão.

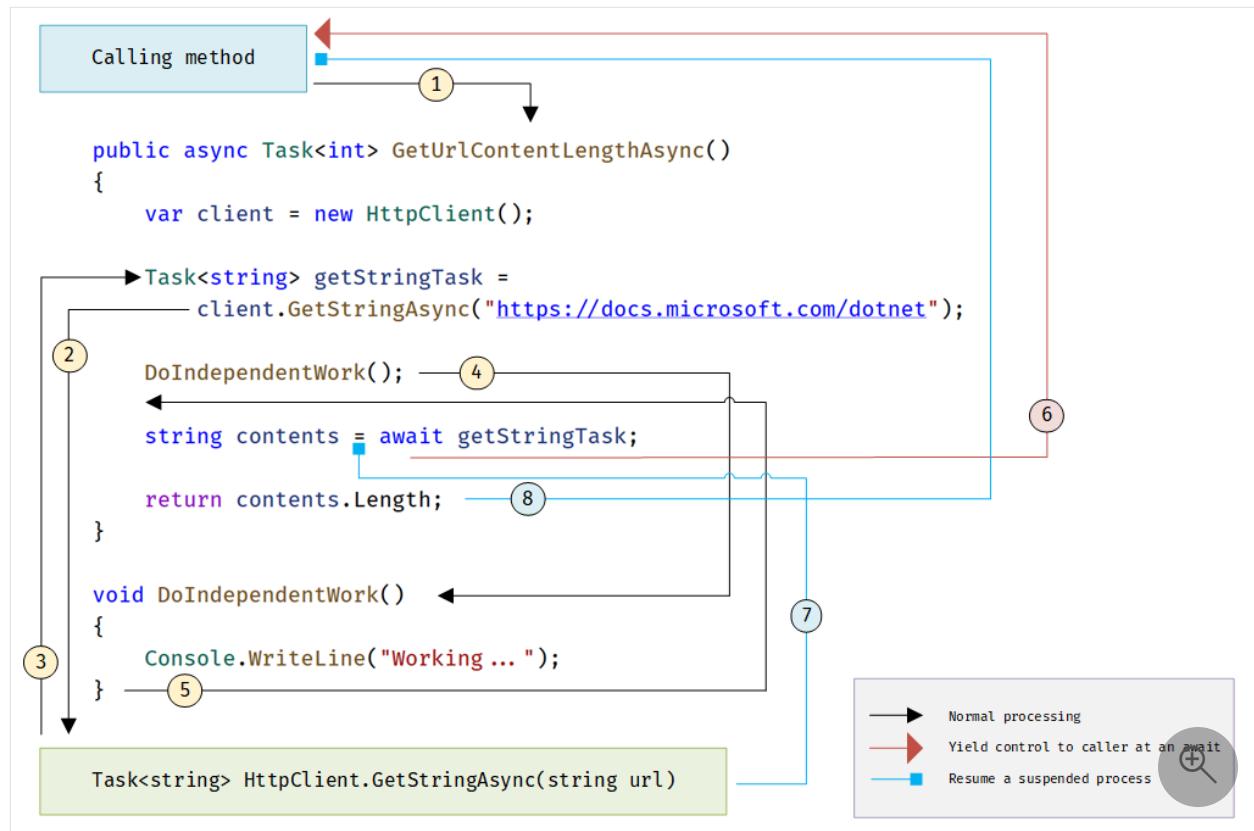
Em métodos assíncronos, você usa as palavras-chave e os tipos fornecidos para indicar o que deseja fazer, e o compilador faz o resto, inclusive acompanhar o que deve acontecer quando o controle retorna a um ponto de espera em um método suspenso. Alguns processos de rotina, como loops e a manipulação de exceções, podem ser difíceis de manipular em um código assíncrono tradicional. Em um método assíncrono, você escreve esses elementos da mesma forma que faria em uma solução síncrona, e o problema é resolvido.

Para obter mais informações sobre assincronia nas versões anteriores do .NET Framework, confira [Programação assíncrona do .NET Framework tradicional e do TPL](#).

## O que acontece em um método assíncrono

O mais importante que você deve compreender na programação assíncrona é a forma como o fluxo de controle avança de um método para outro. O diagrama a seguir pode

ser usado para conduzi-lo pelo processo:



Os números do diagrama correspondem às etapas a seguir, iniciadas quando um método de chamada chama o método assíncrono.

1. Um método de chamada chama e aguarda o método assíncrono

`GetUrlContentLengthAsync`.

2. `GetUrlContentLengthAsync` cria uma instância de `HttpClient` e chama o método assíncrono `GetStringAsync` para baixar o conteúdo de um site como uma cadeia de caracteres.

3. Algo acontece em `GetStringAsync` que suspende o andamento. Talvez ele deva aguardar o download de um site ou alguma outra atividade causadora de bloqueio. Para evitar o bloqueio de recursos, `GetStringAsync` transfere o controle para seu chamador, `GetUrlContentLengthAsync`.

`GetStringAsync` retorna um `Task<TResult>`, em que `TResult` é uma cadeia de caracteres, e `GetUrlContentLengthAsync` atribui a tarefa à variável `getStringTask`. A tarefa representa o processo contínuo para a chamada a `GetStringAsync`, com um compromisso de produzir um valor de cadeia de caracteres real quando o trabalho estiver concluído.

4. Como o `getStringTask` ainda não foi esperado, `GetUrlContentLengthAsync` pode continuar com outro trabalho que não depende do resultado final de

`GetStringAsync`. O trabalho é representado por uma chamada ao método síncrono `DoIndependentWork`.

5. `DoIndependentWork` é um método síncrono que faz seu trabalho e retorna ao seu chamador.
6. `GetUrlContentLengthAsync` está sem trabalho que ele possa executar sem um resultado de `getStringTask`. Em seguida, `GetUrlContentLengthAsync` deseja calcular e retornar o comprimento da cadeia de caracteres baixada, mas o método não poderá calcular o valor enquanto o método tiver a cadeia de caracteres.

Portanto, `GetUrlContentLengthAsync` usa um operador `await` para suspender seu andamento e para transferir o controle para o método que chamou `GetUrlContentLengthAsync`. `GetUrlContentLengthAsync` retorna um `Task<int>` ao chamador. A tarefa representa uma promessa de produzir um resultado inteiro que é o comprimento da cadeia de caracteres baixada.

#### ⓘ Observação

Se `GetStringAsync` (e, portanto, `getStringTask`) for concluído antes que `GetUrlContentLengthAsync` o aguarde, o controle permanecerá em `GetUrlContentLengthAsync`. A despesa de suspender e depois retornar para `GetUrlContentLengthAsync` seria desperdiçada caso o processo assíncrono chamado `getStringTask` já tivesse sido concluído e `GetUrlContentLengthAsync` não tivesse que aguardar o resultado final.

Dentro do método de chamada, o padrão de processamento continua. O chamador pode fazer outro trabalho que não dependa do resultado de `GetUrlContentLengthAsync` antes de aguardar o resultado, ou o chamador pode aguardar imediatamente. O método de chamada está aguardando `GetUrlContentLengthAsync` e `GetUrlContentLengthAsync` está aguardando `GetStringAsync`.

7. `GetStringAsync` completa e produz um resultado de cadeia de caracteres. O resultado da cadeia de caracteres não é retornado pela chamada para `GetStringAsync` da maneira que você poderia esperar. (Lembre-se que o método já retornou uma tarefa na etapa 3.) Em vez disso, o resultado da cadeia de caracteres é armazenado na tarefa que representa a conclusão do método, `getStringTask`. O operador `await` recupera o resultado de `getStringTask`. A instrução de atribuição atribui o resultado retornado a `contents`.

8. Quando `GetUrlContentLengthAsync` tem o resultado da cadeia de caracteres, o método pode calcular o comprimento da cadeia de caracteres. Em seguida, o trabalho de `GetUrlContentLengthAsync` também é concluído e o manipulador de eventos de espera poderá retomar. No exemplo completo no final do tópico, é possível confirmar que o manipulador de eventos recuperou e imprimiu o valor do comprimento do resultado. Se você não tiver experiência em programação assíncrona, considere por um minuto a diferença entre o comportamento síncrono e o assíncrono. Um método síncrono retorna quando seu trabalho é concluído (etapa 5), mas um método assíncrono retorna um valor de tarefa quando seu trabalho está suspenso (etapas 3 e 6). Quando o método assíncrono eventualmente concluir seu trabalho, a tarefa será marcada como concluída e o resultado, se houver, será armazenado na tarefa.

## Métodos assíncronos da API

Você pode estar curioso para saber onde encontrar métodos como `GetStringAsync` que oferecem suporte à programação assíncrona. O .NET Framework 4.5 ou versão superior e o .NET Core contêm muitos membros que funcionam com `async` e com `await`. É possível reconhecê-los pelo sufixo “`Async`” que é acrescentado ao nome do membro e pelo tipo de retorno de `Task` ou de `Task<TResult>`. Por exemplo, a classe `System.IO.Stream` contém métodos como `CopyToAsync`, `ReadAsync` e `WriteAsync`, juntamente com os métodos síncronos `CopyTo`, `Read` e `Write`.

O Windows Runtime também contém vários métodos que você pode usar com `async` e `await` em aplicativos do Windows. Para obter mais informações, veja [Threading e programação assíncrona](#) para o desenvolvimento da UWP e [Programação assíncrona \(aplicativos da Windows Store\)](#) e [Início Rápido: chamando APIs assíncronas em C# ou Visual Basic](#) se você usa versões anteriores do Windows Runtime.

## Threads

Os métodos assíncronos destinam-se a ser operações não causadoras de bloqueios. Uma expressão `await` em um método assíncrono não bloqueia o thread atual enquanto a tarefa aguardada está em execução. Em vez disso, a expressão anterior assina o restante do método como uma continuação e retorna o controle para o chamador do método assíncrono.

As palavras-chave `async` e `await` não fazem com que threads adicionais sejam criados. Os métodos assíncronos não exigem multithreading, pois um método assíncrono não executa em seu próprio thread. O método é executado no contexto de sincronização

atual e usa tempo no thread somente quando o método está ativo. É possível usar [Task.Run](#) para mover o trabalho de CPU associado a um thread em segundo plano, mas um thread em segundo plano não ajuda com um processo que está apenas aguardando que os resultados tornem-se disponíveis.

A abordagem baseada em `async` para a programação assíncrona é preferível às abordagens existentes em quase todos os casos. Essa abordagem é especialmente mais eficiente do que a classe [BackgroundWorker](#) para operações de entrada e saída, porque o código é mais simples e você não precisa se proteger contra condições de corrida. Em combinação com o método [Task.Run](#), a programação assíncrona é melhor que [BackgroundWorker](#) para operações associadas à CPU, porque a programação assíncrona separa os detalhes de coordenação da execução do código de trabalho que `Task.Run` transfere ao pool de threads.

## async e await

Se você especificar que um método é assíncrono usando um modificador `async`, você habilitará os dois recursos a seguir.

- O método assíncrono marcado pode usar `await` para designar pontos de suspensão. O operador `await` informa ao compilador que o método assíncrono não poderá continuar além daquele ponto até que o processo assíncrono aguardado seja concluído. Enquanto isso, o controle retorna para o chamador do método assíncrono.

A suspensão de um método assíncrono em uma expressão `await` não constitui uma saída de método e os blocos `finally` não são executados.

- O método assíncrono marcado pode ele próprio ser aguardado por métodos que o chamam.

Um método assíncrono normalmente contém uma ou mais ocorrências do operador `await`, mas a ausência de expressões `await` não causa erro de compilação. Se um método assíncrono não usa o operador `await` para marcar um ponto de suspensão, o método é executado da mesma forma que um método síncrono, independentemente do modificador `async`. O compilador emite um aviso para esses métodos.

`async` e `await` são palavras-chave contextuais. Para obter mais informações e exemplos, consulte os seguintes tópicos:

- [async](#)
- [await](#)

# Tipos e parâmetros de retorno

Um método assíncrono normalmente retorna `Task` ou `Task<TResult>`. Dentro de um método assíncrono, um operador `await` é aplicado a uma tarefa que é retornada de uma chamada para outro método assíncrono.

Você especificará `Task<TResult>` como o tipo de retorno se o método contiver uma instrução `return` que especifica um operando do tipo `TResult`.

Você usará `Task` como o tipo de retorno caso o método não possua nenhuma instrução `return` ou tenha uma instrução `return` que não retorna um operando.

Você também pode especificar qualquer outro tipo de retorno, desde que o tipo inclua um método `GetAwaiter`. `ValueTask<TResult>` é um exemplo de tal tipo. Ele está disponível no pacote NuGet [System.Threading.Tasks.Extensions](#).

O seguinte exemplo mostra como você declara e chama um método que retorna `Task<TResult>` ou `Task`:

C#

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);

    return hours;
}

Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
await returnedTask;
// Single line
await GetTaskAsync();
```

Cada tarefa retornada representa um trabalho em andamento. Uma tarefa encapsula informações sobre o estado do processo assíncrono e, consequentemente, o resultado

final do processo ou a exceção que o processo apresenta quando não é bem-sucedido.

Um método assíncrono também pode ter um tipo de retorno `void`. Esse tipo de retorno é usado principalmente para definir manipuladores de eventos, onde o tipo de retorno `void` é necessário. Os manipuladores de eventos assíncronos geralmente servem como o ponto de partida para programas assíncronos.

Um método assíncrono que tem o tipo de retorno `void` não pode ser esperado, e o chamador de um método que retorna nulo não pode capturar nenhuma exceção lançada pelo método.

O método não pode declarar nenhum parâmetro `in`, `ref` ou `out`, mas pode chamar métodos com tais parâmetros. Da mesma forma, um método assíncrono não pode retornar um valor por referência, embora possa chamar métodos com valores retornados `ref`.

Para obter mais informações e exemplos, confira [Tipos de retorno assíncronos \(C#\)](#).

As APIs assíncronas na programação do Windows Runtime têm um dos seguintes tipos de retorno, que são semelhantes às tarefas:

- `IAsyncOperation<TResult>`, que corresponde a `Task<TResult>`
- `IAsyncAction`, que corresponde a `Task`
- `IAsyncActionWithProgress<TProgress>`
- `IAsyncOperationWithProgress<TResult,TProgress>`

## Convenção de nomenclatura

Por convenção, os métodos que geralmente retornam tipos esperáveis (por exemplo, `Task`, `Task<T>`, `ValueTask` e `ValueTask<T>`) devem ter nomes que terminam com "Async". Os métodos que iniciam uma operação assíncrona, mas não retornam um tipo aguardável não devem ter nomes que terminam com "Async", mas podem começar com "Begin", "Start" ou algum outro verbo que indique que esse método não retorna nem gera o resultado da operação.

É possível ignorar a convenção quando um evento, uma classe base ou um contrato de interface sugere um nome diferente. Por exemplo, você não deve renomear manipuladores de eventos comuns, como `OnButtonClick`.

## Artigos relacionados (Visual Studio)

Title	Descrição
<a href="#">Como fazer várias solicitações da Web em paralelo usando async e await (C#)</a>	Demonstra como iniciar várias tarefas ao mesmo tempo.
<a href="#">Tipos de retorno assíncronos (C#)</a>	Ilustra os tipos que os métodos assíncronos podem retornar e explica quando cada tipo é apropriado.
Cancelar tarefas com um token de cancelamento como um mecanismo de sinalização.	<p>Mostra como adicionar a seguinte funcionalidade à sua solução assíncrona:</p> <ul style="list-style-type: none"> <li>- <a href="#">Cancelar uma lista de tarefas (C#)</a></li> <li>- <a href="#">Cancelar tarefas após um período (C#)</a></li> <li>- <a href="#">Processar tarefas assíncronas conforme elas forem concluídas (C#)</a></li> </ul>
<a href="#">Como usar o Async para acessar arquivos (C#)</a>	Lista e demonstra as vantagens de usar async e await para acessar arquivos.
<a href="#">TAP (padrão assíncrono baseado em tarefas)</a>	Descreve um padrão assíncrono, o padrão é baseado nos tipos <a href="#">Task</a> e <a href="#">Task&lt;TResult&gt;</a> .
<a href="#">Vídeos sobre assincronia no Channel 9</a>	Fornece links para uma variedade de vídeos sobre programação assíncrona.

## Confira também

- [Programação assíncrona com async e await](#)
- [async](#)
- [await](#)

# Tipos de retorno assíncronos (C#)

Artigo • 10/05/2023

Métodos assíncronos podem conter os seguintes tipos de retorno:

- [Task](#), para um método assíncrono que executa uma operação, mas não retorna nenhum valor.
- [Task<TResult>](#), para um método assíncrono que retorna um valor.
- [void](#), para um manipulador de eventos.
- Qualquer tipo que tenha um método `GetAwaiter` acessível. O objeto retornado pelo método `GetAwaiter` deve implementar a interface [System.Runtime.CompilerServices.ICriticalNotifyCompletion](#).
- [IAsyncEnumerable<T>](#), para um método assíncrono que retorna um *fluxo assíncrono*.

Para obter mais informações sobre os métodos assíncronos, confira [Programação assíncrona com async e await \(C#\)](#).

Também existem vários outros tipos específicos para cargas de trabalho do Windows:

- [DispatcherOperation](#), para operações assíncronas limitadas ao Windows.
- [IAsyncAction](#), para ações assíncronas na UWP que não retornam um valor.
- [IAsyncActionWithProgress<TProgress>](#), para ações assíncronas na UWP que relatam o progresso, mas não retornam um valor.
- [IAsyncOperation<TResult>](#), para operações assíncronas na UWP que retornam um valor.
- [IAsyncOperationWithProgress<TResult,TProgress>](#), para operações assíncronas na UWP que relatam progresso e retornam um valor.

## Tipo de retorno da tarefa

Os métodos assíncronos que não contêm uma instrução `return` ou que contêm uma instrução `return` que não retorna um operando, normalmente têm um tipo de retorno de [Task](#). Esses métodos retornam `void` se eles são executados de forma síncrona. Se você usar um tipo de retorno [Task](#) para um método assíncrono, um método de chamada poderá usar um operador `await` para suspender a conclusão do chamador até que o método assíncrono chamado seja concluído.

No exemplo a seguir, o método `WaitAndApologizeAsync` não contém uma instrução `return`, então o método retorna um objeto [Task](#). Retornar `Task` habilita a espera por

`WaitAndApologizeAsync`. O tipo `Task` não inclui uma propriedade `Result` porque ele não tem nenhum valor retornado.

C#

```
public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}
// Example output:
//     Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.
```

O `WaitAndApologizeAsync` é aguardado, usando uma instrução `await`, em vez de uma expressão `await`, semelhante à instrução de chamada a um método síncrono de retorno `void`. A aplicação de um operador `await`, nesse caso, não produz um valor. Quando o operando direito de `await` é `Task<TResult>`, a expressão `await` produz o resultado `T`. Quando o operando direito de `await` é `Task`, `await` e o operando dele são uma instrução.

Você pode separar a chamada a `WaitAndApologizeAsync` da aplicação de um operador `await`, como mostrado no código a seguir. No entanto, lembre-se que uma `Task` não tem uma propriedade `Result` e que nenhum valor será produzido quando um operador `await` for aplicado a uma `Task`.

O código a seguir separa a chamada ao método `WaitAndApologizeAsync` da espera pela tarefa que o método retorna.

C#

```
Task waitAndApologizeTask = WaitAndApologizeAsync();

string output =
    $"Today is {DateTime.Now:D}\n" +
```

```
$"The current time is {DateTime.Now.TimeOfDay:t}\n" +
"The current temperature is 76 degrees.\n";

await waitAndApologizeTask;
Console.WriteLine(output);
```

## Tipo de retorno Task<TResult>

O tipo de retorno `Task<TResult>` é usado para um método assíncrono que contém uma instrução `return` em que o operando é `TResult`.

No exemplo a seguir, o método `GetLeisureHoursAsync` contém uma instrução `return` que retorna um número inteiro. A declaração do método deve especificar um tipo de retorno igual a `Task<int>`. O método assíncrono `FromResult` é um espaço reservado para uma operação que retorna `DayOfWeek`.

C#

```
public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
//   Today is Wednesday, May 24, 2017
//   Today's hours of leisure: 5
```

Quando `GetLeisureHoursAsync` é chamado de dentro de uma expressão `await` no método `ShowTodaysInfo`, a expressão `await` recupera o valor inteiro (o valor de `leisureHours`) que está armazenado na tarefa que é retornada pelo método `GetLeisureHours`. Para obter mais informações sobre expressões `await`, consulte [await](#).

Você pode entender melhor como `await` recupera o resultado de `Task<T>` separando a chamada a `GetLeisureHoursAsync` da aplicação de `await`, como mostra o código a seguir. Uma chamada ao método `GetLeisureHoursAsync` que não é aguardada imediatamente, retorna um `Task<int>`, como você esperaria da declaração do método. A tarefa é atribuída à variável `getLeisureHoursTask` no exemplo. Já que `getLeisureHoursTask` é um `Task<TResult>`, ele contém uma propriedade `Result` do tipo `TResult`. Nesse caso, `TResult` representa um tipo inteiro. Quando `await` é aplicado à `getLeisureHoursTask`, a expressão `await` é avaliada como o conteúdo da propriedade `Result` de `getLeisureHoursTask`. O valor é atribuído à variável `ret`.

### ⓘ Importante

A propriedade `Result` é uma propriedade de bloqueio. Se você tentar acessá-la antes que sua tarefa seja concluída, o thread que está ativo no momento será bloqueado até que a tarefa seja concluída e o valor esteja disponível. Na maioria dos casos, você deve acessar o valor usando `await` em vez de acessar a propriedade diretamente.

O exemplo anterior recuperou o valor da propriedade `Result` para bloquear o thread principal, de modo que o método `Main` pudesse imprimir `message` no console antes do encerramento do aplicativo.

C#

```
var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);
```

## Tipo de retorno nulo

O tipo de retorno `void` é usado em manipuladores de eventos assíncronos, que exigem um tipo de retorno `void`. Para métodos diferentes de manipuladores de eventos que não retornam um valor, você deve retornar um `Task`, porque não é possível esperar por um método assíncrono que retorna `void`. Qualquer chamador desse método deve continuar a conclusão sem esperar que o método assíncrono chamado seja concluído. O

chamador deve ser independente de quaisquer valores ou exceções que o método assíncrono gere.

O chamador de um método assíncrono de retorno nulo não pode capturar exceções geradas por meio do método. Essas exceções sem tratamento provavelmente causarão falha no aplicativo. Se um método que retorna [Task](#) ou [Task<TResult>](#) gera uma exceção, ela é armazenada na tarefa retornada. A exceção é gerada novamente quando a tarefa é aguardada. Verifique se qualquer método assíncrono que pode produzir uma exceção tem o tipo de retorno [Task](#) ou [Task<TResult>](#) e se as chamadas ao método são aguardadas.

O exemplo a seguir mostra o comportamento de um manipulador de eventos assíncrono. No exemplo de código, um manipulador de eventos assíncrono deve informar o thread principal quando for concluído. Em seguida, o thread principal pode aguardar um manipulador de eventos assíncronos ser concluído antes de sair do programa.

C#

```
public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the
event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
    static readonly TaskCompletionSource<bool> s_tcs = new
    TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;

        var button = new NaiveButton();

        button.Clicked += OnButtonClicked1;
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;

        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");
    }
}
```

```

        await secondHandlerFinished;
    }

    private static void OnButtonClicked1(object? sender, EventArgs e)
    {
        Console.WriteLine("  Handler 1 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("  Handler 1 is done.");
    }

    private static async void OnButtonClicked2Async(object? sender,
EventArgs e)
    {
        Console.WriteLine("  Handler 2 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("  Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("  Handler 2 is done.");
        s_tcs.SetResult(true);
    }

    private static void OnButtonClicked3(object? sender, EventArgs e)
    {
        Console.WriteLine("  Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("  Handler 3 is done.");
    }
}

// Example output:
//
// Before button.Click() is called...
// Somebody has clicked a button. Let's raise the event...
//   Handler 1 is starting...
//   Handler 1 is done.
//   Handler 2 is starting...
//   Handler 2 is about to go async...
//   Handler 3 is starting...
//   Handler 3 is done.
// All listeners are notified.
// After button.Click() is called...
//   Handler 2 is done.

```

## Tipos de retorno assíncronos generalizados e ValueTask<TResult>

Um método assíncrono pode retornar qualquer tipo que tenha um método `GetAwaiter` acessível que retorne uma instância de um *tipo de aguardador*. Além disso, o tipo retornado do método `GetAwaiter` deve ter o atributo `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute`. Saiba mais no artigo

sobre [Atributos lidos pelo compilador](#) ou na especificação de C# para o [Padrão de construtor de tipo de tarefa](#).

Esse recurso é o complemento para [expressões aguardáveis](#), que descreve os requisitos para o operando `await`. Tipos de retorno assíncronos generalizados permitem que o compilador gere métodos `async` que retornam tipos diferentes. Tipos de retorno assíncronos generalizados habilitaram melhorias de desempenho nas bibliotecas .NET. Já que `Task` e `Task<TResult>` são tipos de referência, a alocação de memória em caminhos críticos para o desempenho, especialmente quando alocações ocorrerem em loops estreitos, podem afetar o desempenho. Suporte para tipos de retorno generalizados significa que você pode retornar um tipo de valor leve em vez de um tipo de referência para evitar as alocações de memória adicionais.

O .NET fornece a estrutura `System.Threading.Tasks.ValueTask<TResult>` como uma implementação leve de um valor de retorno de tarefa generalizado. O exemplo a seguir usa a estrutura `ValueTask<TResult>` para recuperar o valor de dois lançamentos de dados.

C#

```
class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()}");

    static async ValueTask<int> GetDiceRollAsync()
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync();
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }

    static async ValueTask<int> RollAsync()
    {
        await Task.Delay(500);

        int diceRoll = s_rnd.Next(1, 7);
        return diceRoll;
    }
}

// Example output:
//   Shaking dice...
//   You rolled 8
```

Escrever um tipo de retorno assíncrono generalizado é um cenário avançado, direcionado para uso em ambientes especializados. Considere o uso dos tipos `Task`, `Task<T>` e `ValueTask<T>`, que abrangem a maioria dos cenários para código assíncrono.

No C# 10 e versões posteriores, você pode aplicar o atributo `AsyncMethodBuilder` a um método assíncrono (em vez da declaração de tipo de retorno assíncrono) para substituir o construtor daquele tipo. Normalmente, você aplicaria esse atributo para usar um construtor diferente fornecido no runtime do .NET.

## Fluxos assíncronos com `IAsyncEnumerable<T>`

Um método assíncrono pode retornar um *fluxo assíncrono*, representado por `IAsyncEnumerable<T>`. Um fluxo assíncrono fornece uma forma de enumerar itens lidos de um fluxo quando os elementos são gerados em partes com chamadas assíncronas repetidas. O seguinte exemplo mostra um método assíncrono que gera um fluxo assíncrono:

C#

```
static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.
        Here is the second line of text.
        And there is one more for good measure.
        Wait, that was the penultimate line.";

    using var readStream = new StringReader(data);

    string? line = await readStream.ReadLineAsync();
    while (line != null)
    {
        foreach (string word in line.Split(' ',
StringSplitOptions.RemoveEmptyEntries))
        {
            yield return word;
        }

        line = await readStream.ReadLineAsync();
    }
}
```

O exemplo anterior lê linhas de uma cadeia de caracteres de modo assíncrono. Depois que cada linha é lida, o código enumera cada palavra na cadeia de caracteres. Os chamadores enumerariam cada palavra usando a instrução `await foreach`. O método

aguarda quando precisa ler de modo assíncrono a próxima linha da cadeia de caracteres de origem.

## Confira também

- [FromResult](#)
- [Processar tarefas assíncronas conforme elas são concluídas](#)
- [Programação assíncrona com async e await \(C#\)](#)
- [async](#)
- [await](#)

# Cancelar uma lista de tarefas

Artigo • 28/03/2023

Você pode cancelar um aplicativo de console assíncrono se não quiser esperar que ele seja concluído. Seguindo o exemplo neste tópico, você pode adicionar um cancelamento a um aplicativo que baixa o conteúdo de uma lista de sites. Você pode cancelar várias tarefas associando a instância [CancellationTokenSource](#) a cada tarefa. Se você selecionar a tecla `Enter`, você cancele todas as tarefas que ainda não foram concluídas.

Este tutorial abrange:

- ✓ Criando um aplicativo de console .NET
- ✓ Escrever um aplicativo assíncrono que dá suporte a cancelamento
- ✓ Demonstrando cancelamento de sinalização

## Pré-requisitos

Este tutorial exige o seguinte:

- [.NET 5 ou SDK posterior](#)
- IDE (ambiente de desenvolvimento integrado)
  - [Recomendamos o Visual Studio, Visual Studio Code ou Visual Studio para Mac](#)

## Criar aplicativo de exemplo

Criar um novo aplicativo de console do .NET Core. Você pode criar um usando o comando ou do [dotnet new consoleVisual Studio](#). Abra o arquivo *Program.cs* em seu editor de código favorito.

## Substituir usando instruções

Substitua as instruções de uso existentes por estas declarações:

C#

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
```

```
using System.Threading;
using System.Threading.Tasks;
```

## Adicionar campos

Na definição de classe `Program`, adicione estes três campos:

C#

```
static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};
```

O `CancellationTokenSource` é usado para sinalizar um cancelamento solicitado para um `CancellationToken`. O `HttpClient` expõe a capacidade de enviar solicitações HTTP e receber respostas HTTP. O `s_urlList` contém todas as URLs que o aplicativo planeja processar.

## Atualize o ponto de entrada do aplicativo

O principal ponto de entrada no aplicativo de console é o método `Main`. Substitua o método existente pelo seguinte:

```
C#  
  
static async Task Main()  
{  
    Console.WriteLine("Application started.");  
    Console.WriteLine("Press the ENTER key to cancel...\n");  
  
    Task cancelTask = Task.Run(() =>  
    {  
        while (Console.ReadKey().Key != ConsoleKey.Enter)  
        {  
            Console.WriteLine("Press the ENTER key to cancel...");  
        }  
  
        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");  
        s_cts.Cancel();  
    });  
  
    Task sumPageSizesTask = SumPageSizesAsync();  
  
    Task finishedTask = await Task.WhenAny(new[] { cancelTask,  
sumPageSizesTask });  
    if (finishedTask == cancelTask)  
    {  
        // wait for the cancellation to take place:  
        try  
        {  
            await sumPageSizesTask;  
            Console.WriteLine("Download task completed before cancel request  
was processed.");  
        }  
        catch (TaskCanceledException)  
        {  
            Console.WriteLine("Download task has been cancelled.");  
        }  
    }  
  
    Console.WriteLine("Application ending.");  
}
```

O método atualizado `Main` agora é considerado um [Async main](#), que permite um ponto de entrada assíncrono no executável. Ele grava algumas mensagens instrutivas no console e, em seguida, declara uma instância `Task` chamada `cancelTask`, que lerá os traços de chave do console. Se a tecla `Enter` for pressionada, será feita uma chamada `CancellationTokenSource.Cancel()`. Isso sinalizará o cancelamento. Em seguida, a variável `sumPageSizesTask` é atribuída do método `SumPageSizesAsync`. Ambas as tarefas são

então passadas para `Task.WhenAny(Task[])`, o que continuará quando qualquer uma das duas tarefas tiver sido concluída.

O próximo bloco de código garante que o aplicativo não saia até que o cancelamento seja processado. Se a primeira tarefa a ser concluída for `cancelTask`, o `sumPageSizeTask` será aguardado. Se ele foi cancelado, quando aguardado, ele lança um `System.Threading.Tasks.TaskCanceledException`. O bloco captura essa exceção e imprime uma mensagem.

## Criar o método de tamanhos de página de soma assíncrona

Abaixo do método `Main`, adicione o método `SumPageSizesAsync`:

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}
```

O método começa instanciando e iniciando um `Stopwatch`. Em seguida, ele faz loops por cada URL no `s_urlList` e chama `ProcessUrlAsync`. A cada iteração, o método `s_cts.Token` é passado para o método `ProcessUrlAsync` e o código retorna um `Task<TResult>`, que que `TResult` é um inteiro:

C#

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
```

```
    total += contentLength;  
}
```

## Adicionar método de processo

Adicione o seguinte método `ProcessUrlAsync` abaixo do método `SumPageSizesAsync`:

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client,  
CancellationToken token)  
{  
    HttpResponseMessage response = await client.GetAsync(url, token);  
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);  
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");  
  
    return content.Length;  
}
```

Para qualquer URL fornecida, o método usará a instância `client` fornecida para obter a resposta como um `byte[]`. A instância `CancellationToken` é passada para os métodos `HttpClient.GetAsync(String, CancellationToken)` e `HttpContent.ReadAsByteArrayAsync()`. O `token` é usado para registrar-se para cancelamento solicitado. O comprimento é retornado depois que a URL e o comprimento são gravados no console.

## Exemplo de saída de aplicativo

Console

```
Application started.  
Press the ENTER key to cancel...
```

https://learn.microsoft.com	37,357
https://learn.microsoft.com/aspnet/core	85,589
https://learn.microsoft.com/azure	398,939
https://learn.microsoft.com/azure/devops	73,663
https://learn.microsoft.com/dotnet	67,452
https://learn.microsoft.com/dynamics365	48,582
https://learn.microsoft.com/education	22,924

```
ENTER key pressed: cancelling downloads.
```

```
Application ending.
```

# Exemplo completo

O código a seguir é o texto completo do arquivo *Program.cs* para o exemplo.

C#

```
using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
    CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
        "https://learn.microsoft.com",
        "https://learn.microsoft.com/aspnet/core",
        "https://learn.microsoft.com/azure",
        "https://learn.microsoft.com/azure/devops",
        "https://learn.microsoft.com/dotnet",
        "https://learn.microsoft.com/dynamics365",
        "https://learn.microsoft.com/education",
        "https://learn.microsoft.com/enterprise-mobility-security",
        "https://learn.microsoft.com/gaming",
        "https://learn.microsoft.com/graph",
        "https://learn.microsoft.com/microsoft-365",
        "https://learn.microsoft.com/office",
        "https://learn.microsoft.com/powershell",
        "https://learn.microsoft.com/sql",
        "https://learn.microsoft.com/surface",
        "https://learn.microsoft.com/system-center",
        "https://learn.microsoft.com/visualstudio",
        "https://learn.microsoft.com/windows",
        "https://learn.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
        Console.WriteLine("Press the ENTER key to cancel...\n");

        Task cancelTask = Task.Run(() =>
        {
            while (Console.ReadKey().Key != ConsoleKey.Enter)
            {
                Console.WriteLine("Press the ENTER key to cancel...");
            }
        });
    }
}
```

```

        Console.WriteLine("\nENTER key pressed: cancelling
downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
    if (finishedTask == cancelTask)
    {
        // wait for the cancellation to take place:
        try
        {
            await sumPageSizesTask;
            Console.WriteLine("Download task completed before cancel
request was processed.");
        }
        catch (TaskCanceledException)
        {
            Console.WriteLine("Download task has been cancelled.");
        }
    }

    Console.WriteLine("Application ending.");
}

static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,##}");

    return content.Length;
}
}

```

## Confira também

- [CancellationToken](#)
- [CancellationTokenSource](#)
- [Programação assíncrona com async e await \(C#\)](#)

## Próximas etapas

[Cancelar tarefas assíncronas após um período \(C#\)](#)

# Cancelar tarefas assíncronas após um período

Artigo • 25/03/2023

Você pode cancelar uma operação assíncrona após um período usando o método [CancellationTokenSource.CancelAfter](#) se não quiser aguardar a conclusão da operação. Esse método agenda o cancelamento de quaisquer tarefas associadas que não são concluídas dentro do período designado pela expressão `CancelAfter`.

Este exemplo adiciona o código desenvolvido em [Cancelar uma tarefa assíncrona ou uma lista de tarefas \(C#\)](#) para baixar uma lista de sites e para exibir o tamanho dos conteúdos de cada um.

Este tutorial abrange:

- ✓ Atualizando um aplicativo de console .NET existente
- ✓ Agendando um cancelamento

## Pré-requisitos

Este tutorial exige o seguinte:

- Espera-se que você tenha criado um aplicativo no tutorial [Cancelar uma lista de tarefas \(C#\)](#)
- [.NET 5 ou SDK posterior](#)
- IDE (ambiente de desenvolvimento integrado)
  - [Recomendamos o Visual Studio, Visual Studio Code ou Visual Studio para Mac](#)

## Atualize o ponto de entrada do aplicativo.

Substitua o método `Main` existente pelo seguinte:

```
C#  
  
static async Task Main()  
{  
    Console.WriteLine("Application started.");  
  
    try  
    {  
        s_cts.CancelAfter(3500);  
    }  
}
```

```

        await SumPageSizesAsync();
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }
    finally
    {
        s_cts.Dispose();
    }

    Console.WriteLine("Application ending.");
}

```

O método `Main` atualizado grava algumas mensagens instrutivas no console. Dentro do `try-catch`, uma chamada para `CancellationTokenSource.CancelAfter(Int32)` agenda um cancelamento. Isso sinalizará o cancelamento após um período.

Em seguida, o método `SumPageSizesAsync` é aguardado. Se o processamento de todas as URLs ocorrer mais rápido do que o cancelamento agendado, o aplicativo é encerrado. No entanto, se o cancelamento agendado for disparado antes que todas as URLs sejam processadas, um `OperationCanceledException` será gerado.

## Exemplo de saída de aplicativo

```

Console

Application started.

https://learn.microsoft.com                                         37,357
https://learn.microsoft.com/aspnet/core                           85,589
https://learn.microsoft.com/azure                                398,939
https://learn.microsoft.com/azure/devops                         73,663

Tasks cancelled: timed out.

Application ending.

```

## Exemplo completo

O código a seguir é o texto completo do arquivo `Program.cs` para o exemplo.

```

C#

using System.Diagnostics;

class Program

```

```
{  
    static readonly CancellationTokenSource s_cts = new  
    CancellationTokenSource();  
  
    static readonly HttpClient s_client = new HttpClient  
{  
        MaxResponseContentBufferSize = 1_000_000  
    };  
  
    static readonly IEnumerable<string> s_urlList = new string[]  
    {  
        "https://learn.microsoft.com",  
        "https://learn.microsoft.com/aspnet/core",  
        "https://learn.microsoft.com/azure",  
        "https://learn.microsoft.com/azure/devops",  
        "https://learn.microsoft.com/dotnet",  
        "https://learn.microsoft.com/dynamics365",  
        "https://learn.microsoft.com/education",  
        "https://learn.microsoft.com/enterprise-mobility-security",  
        "https://learn.microsoft.com/gaming",  
        "https://learn.microsoft.com/graph",  
        "https://learn.microsoft.com/microsoft-365",  
        "https://learn.microsoft.com/office",  
        "https://learn.microsoft.com/powershell",  
        "https://learn.microsoft.com/sql",  
        "https://learn.microsoft.com/surface",  
        "https://learn.microsoft.com/system-center",  
        "https://learn.microsoft.com/visualstudio",  
        "https://learn.microsoft.com/windows",  
        "https://learn.microsoft.com/xamarin"  
    };  
  
    static async Task Main()  
    {  
        Console.WriteLine("Application started.");  
  
        try  
        {  
            s_cts.CancelAfter(3500);  
  
            await SumPageSizesAsync();  
        }  
        catch (OperationCanceledException)  
        {  
            Console.WriteLine("\nTasks cancelled: timed out.\n");  
        }  
        finally  
        {  
            s_cts.Dispose();  
        }  
  
        Console.WriteLine("Application ending.");  
    }  
  
    static async Task SumPageSizesAsync()
```

```

{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
}

```

## Confira também

- [CancellationToken](#)
- [CancellationTokenSource](#)
- [Programação assíncrona com async e await \(C#\)](#)
- [Cancelar uma lista de tarefas \(C#\)](#)

# Processar tarefas assíncronas conforme elas são concluídas (C#)

Artigo • 23/05/2023

Usando [Task.WhenAny](#), você pode iniciar várias tarefas ao mesmo tempo e processá-las individualmente conforme elas forem concluídas, em vez de processá-las na ordem em que foram iniciadas.

O exemplo a seguir usa uma consulta para criar uma coleção de tarefas. Cada tarefa baixa o conteúdo de um site especificado. Em cada iteração de um loop "while", uma chamada esperada para [WhenAny](#) retorna a tarefa na coleção de tarefas que concluir o download primeiro. Essa tarefa é removida da coleção e processada. O loop é repetido até que a coleção não contenha mais tarefas.

## Pré-requisitos

Você pode seguir este tutorial usando uma das seguintes opções:

- [Visual Studio 2022](#) com a carga de trabalho **Desenvolvimento de área de trabalho do .NET** instalada. O SDK do .NET é instalado automaticamente quando você seleciona essa carga de trabalho.
- O [SDK do .NET](#) com um editor de código de sua escolha, como [Visual Studio Code](#).

## Criar aplicativo de exemplo

Criar um novo aplicativo de console do .NET Core. Você pode criar um usando o comando [dotnet new console](#) ou do Visual Studio.

Abra o arquivo *Program.cs* no editor de código e substitua o código existente por este:

```
C#  
  
using System.Diagnostics;  
  
namespace ProcessTasksAsTheyFinish;  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        Console.WriteLine("Hello World!");  
    }  
}
```

```
    }  
}
```

## Adicionar campos

Na definição de classe `Program`, adicione os dois seguintes campos:

C#

```
static readonly HttpClient s_client = new HttpClient  
{  
    MaxResponseContentBufferSize = 1_000_000  
};  
  
static readonly IEnumerable<string> s_urlList = new string[]  
{  
    "https://learn.microsoft.com",  
    "https://learn.microsoft.com/aspnet/core",  
    "https://learn.microsoft.com/azure",  
    "https://learn.microsoft.com/azure/devops",  
    "https://learn.microsoft.com/dotnet",  
    "https://learn.microsoft.com/dynamics365",  
    "https://learn.microsoft.com/education",  
    "https://learn.microsoft.com/enterprise-mobility-security",  
    "https://learn.microsoft.com/gaming",  
    "https://learn.microsoft.com/graph",  
    "https://learn.microsoft.com/microsoft-365",  
    "https://learn.microsoft.com/office",  
    "https://learn.microsoft.com/powershell",  
    "https://learn.microsoft.com/sql",  
    "https://learn.microsoft.com/surface",  
    "https://learn.microsoft.com/system-center",  
    "https://learn.microsoft.com/visualstudio",  
    "https://learn.microsoft.com/windows",  
    "https://learn.microsoft.com/xamarin"  
};
```

O `HttpClient` expõe a capacidade de enviar solicitações HTTP e receber respostas HTTP. O `s_urlList` contém todas as URLs que o aplicativo planeja processar.

## Atualize o ponto de entrada do aplicativo

O principal ponto de entrada no aplicativo de console é o método `Main`. Substitua o método existente pelo seguinte:

C#

```
static Task Main() => SumPageSizesAsync();
```

O método atualizado `Main` agora é considerado um [Async main](#), que permite um ponto de entrada assíncrono no executável. Ele é expresso como uma chamada para `SumPageSizesAsync`.

## Criar o método de tamanhos de página de soma assíncrona

Abaixo do método `Main`, adicione o método `SumPageSizesAsync`:

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}
```

O loop `while` remove uma das tarefas em cada iteração. Depois que todas as tarefas forem concluídas, o loop terminará. O método começa instanciando e iniciando um [Stopwatch](#). Em seguida, ele inclui uma consulta que, quando executada, cria uma coleção de tarefas. Cada chamada para `ProcessUrlAsync` no código a seguir retorna um `Task<TResult>`, em que `TResult` é um inteiro:

C#

```
IEnumerable<Task<int>> downloadTasksQuery =  
    from url in s_urlList  
    select ProcessUrlAsync(url, s_client);
```

Devido à [execução adiada](#) com o LINQ, você chama `Enumerable.ToList` para iniciar cada tarefa.

C#

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

O loop `while` executa as seguintes etapas para cada tarefa na coleção:

1. Espera uma chamada para `WhenAny`, com o objetivo de identificar a primeira tarefa na coleção que concluiu o download.

C#

```
Task<int> finishedTask = await Task.WhenAny(downloadTasks);
```

2. Remove a tarefa da coleção.

C#

```
downloadTasks.Remove(finishedTask);
```

3. Espera `finishedTask`, que é retornado por uma chamada para `ProcessUrlAsync`. A variável `finishedTask` é uma `Task<TResult>` em que `TResult` é um inteiro. A tarefa já foi concluída, mas você espera para recuperar o tamanho do site baixado, como mostra o exemplo a seguir. Se a tarefa tiver falha, `await` gerará a primeira exceção filho armazenada no `AggregateException`, ao contrário da leitura da propriedade `Task<TResult>.Result`, que lançaria o `AggregateException`.

C#

```
total += await finishedTask;
```

## Adicionar método de processo

Adicione o seguinte método `ProcessUrlAsync` abaixo do método `SumPageSizesAsync`:

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Para qualquer URL fornecida, o método usará a instância `client` fornecida para obter a resposta como um `byte[]`. O comprimento é retornado depois que a URL e o comprimento são gravados no console.

Execute o programa várias vezes para verificar se os tamanhos baixados não aparecem sempre na mesma ordem.

### ⊗ Cuidado

Você pode usar `WhenAny` em um loop, conforme descrito no exemplo, para resolver problemas que envolvem um número pequeno de tarefas. No entanto, outras abordagens são mais eficientes se você tiver um número grande de tarefas para processar. Para obter mais informações e exemplos, consulte [Processando tarefas quando elas são concluídas](#).

## Exemplo completo

O código a seguir é o texto completo do arquivo *Program.cs* para o exemplo.

C#

```
using System.Diagnostics;

HttpClient s_client = new()
{
    MaxResponseContentBufferSize = 1_000_000
};

IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
```

```

"https://learn.microsoft.com/education",
"https://learn.microsoft.com/enterprise-mobility-security",
"https://learn.microsoft.com/gaming",
"https://learn.microsoft.com/graph",
"https://learn.microsoft.com/microsoft-365",
"https://learn.microsoft.com/office",
"https://learn.microsoft.com/powershell",
"https://learn.microsoft.com/sql",
"https://learn.microsoft.com/surface",
"https://learn.microsoft.com/system-center",
"https://learn.microsoft.com/visualstudio",
"https://learn.microsoft.com/windows",
"https://learn.microsoft.com/xamarin"
};

await SumPageSizesAsync();

async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"Total bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}

// Example output:
// https://learn.microsoft.com                                         132,517
// https://learn.microsoft.com/powershell                           57,375
// https://learn.microsoft.com/gaming                             33,549
// https://learn.microsoft.com/aspnet/core                         88,714
// https://learn.microsoft.com/surface                            39,840

```

```
// https://learn.microsoft.com/enterprise-mobility-security      30,903
// https://learn.microsoft.com/microsoft-365                  67,867
// https://learn.microsoft.com/windows                      26,816
// https://learn.microsoft.com/xamarin                     57,958
// https://learn.microsoft.com/dotnet                      78,706
// https://learn.microsoft.com/graph                      48,277
// https://learn.microsoft.com/dynamics365                49,042
// https://learn.microsoft.com/office                     67,867
// https://learn.microsoft.com/system-center              42,887
// https://learn.microsoft.com/education                 38,636
// https://learn.microsoft.com/azure                      421,663
// https://learn.microsoft.com/visualstudio               30,925
// https://learn.microsoft.com/sql                        54,608
// https://learn.microsoft.com/azure/devops             86,034

// Total bytes returned:    1,454,184
// Elapsed time:           00:00:01.1290403
```

## Confira também

- [WhenAny](#)
- [Programação assíncrona com async e await \(C#\)](#)

# Acesso a arquivos assíncronos (C#)

Artigo • 10/05/2023

Você pode usar o recurso `async` para acessar arquivos. Usando o recurso `async`, você pode chamar os métodos assíncronos sem usar retornos de chamada ou dividir seu código em vários métodos ou expressões lambda. Para tornar síncrono um código assíncrono, basta chamar um método assíncrono em vez de um método síncrono e adicionar algumas palavras-chave ao código.

Você pode considerar seguintes motivos para adicionar a assincronia às chamadas de acesso ao arquivo:

- ✓ A assincronia torna os aplicativos de interface do usuário mais responsivos porque o thread de interface do usuário que inicia a operação pode executar outro trabalho. Se o thread de interface do usuário precisar executar o código que leva muito tempo (por exemplo, mais de 50 milissegundos), a interface do usuário poderá congelar até que a E/S seja concluída e o thread da interface do usuário possa processar entradas do mouse e do teclado e outros eventos.
- ✓ A assincronia melhora a escalabilidade do ASP.NET e outros aplicativos baseados em servidor reduzindo a necessidade de threads. Se o aplicativo usar um thread dedicado por resposta e mil solicitações forem tratadas simultaneamente, serão necessários mil threads. As operações assíncronas normalmente não precisam usar um thread durante a espera. Elas podem usar o thread de conclusão de E/S existente rapidamente no final.
- ✓ A latência de uma operação de acesso de arquivo pode ser muito baixa nas condições atuais, mas a latência pode aumentar consideravelmente no futuro. Por exemplo, um arquivo pode ser movido para um servidor que está do outro lado do mundo.
- ✓ A sobrecarga adicional de usar o recurso `async` é pequena.
- ✓ As tarefas assíncronas podem facilmente ser executadas em paralelo.

## Usar classes apropriadas

Os exemplos simples neste tópico demonstram `File.WriteAllTextAsync` e `File.ReadAllTextAsync`. Para controle fino sobre as operações de E/S do arquivo, use a classe `FileStream`, que tem uma opção que faz com que a E/S assíncrona ocorra no nível do sistema operacional. Usando essa opção, você pode evitar o bloqueio de um thread de pool de threads em muitos casos. Para habilitar essa opção, você deve especificar o argumento `useAsync=true` ou `options=FileOptions.Asynchronous` na chamada do construtor.

Você não pode usar essa opção com `StreamReader` e `StreamWriter` se você os abrir diretamente especificando um caminho de arquivo. No entanto, você poderá usar essa opção se fornecer um `Stream` que a classe `FileStream` abriu. Chamadas assíncronas serão mais rápidas em aplicativos de interface do usuário mesmo se um thread de pool de threads estiver bloqueado, porque o thread de interface do usuário não é bloqueado durante a espera.

## Gravar texto

Os exemplos a seguir gravam texto em um arquivo. A cada instrução `await`, o método sai imediatamente. Quando o arquivo de E/S for concluído, o método continuará na instrução após a instrução `await`. O modificador `async` é a definição de métodos que usam a instrução `await`.

### Exemplo simples

C#

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

### Exemplo de controle finito

C#

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using var sourceStream =
        new FileStream(
            filePath,
```

```
 FileMode.Create, FileAccess.Write, FileShare.None,  
 bufferSize: 4096, useAsync: true);  
  
 await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);  
}
```

O exemplo original tem a instrução `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);`, que é uma contração das duas instruções a seguir:

C#

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);  
await theTask;
```

A primeira instrução retorna uma tarefa e faz com que o processamento do arquivo seja iniciado. A segunda instrução com o `await` faz com que o método saia imediatamente e retorne uma tarefa diferente. Quando o processamento do arquivo é concluído posteriormente, a execução retorna para a instrução após a `await`.

## Ler texto

Os exemplos a seguir leem texto de um arquivo.

### Exemplo simples

C#

```
public async Task SimpleReadAsync()  
{  
    string filePath = "simple.txt";  
    string text = await File.ReadAllTextAsync(filePath);  
  
    Console.WriteLine(text);  
}
```

### Exemplo de controle finito

O texto é armazenado em buffer e, nesse caso, colocado em um [StringBuilder](#). Diferentemente do exemplo anterior, a avaliação de `await` produz um valor. O método [ReadAsync](#) retorna um [Task<Int32>](#), portanto, a avaliação do `await` produz um [Int32](#) valor `numRead` após a conclusão da operação. Para obter mais informações, consulte [Tipos de retorno assíncronos \(C#\)](#).

C#

```
public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine($"file not found: {filePath}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var sb = new StringBuilder();

    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await sourceStream.ReadAsync(buffer, 0,
buffer.Length)) != 0)
    {
        string text = Encoding.Unicode.GetString(buffer, 0, numRead);
        sb.Append(text);
    }

    return sb.ToString();
}
```

## E/S assíncrona paralela

Os exemplos a seguir demonstram o processamento paralelo gravando dez arquivos de texto.

## Exemplo simples

```
C#  
  
public async Task SimpleParallelWriteAsync()  
{  
    string folder = Directory.CreateDirectory("tempfolder").Name;  
    IList<Task> writeTaskList = new List<Task>();  
  
    for (int index = 11; index <= 20; ++ index)  
    {  
        string fileName = $"file-{index:00}.txt";  
        string filePath = $"{folder}/{fileName}";  
        string text = $"In file {index}{Environment.NewLine}";  
  
        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));  
    }  
  
    await Task.WhenAll(writeTaskList);  
}
```

## Exemplo de controle finito

Para cada arquivo, o método `WriteAsync` retorna uma tarefa que é então adicionada a uma lista de tarefas. A instrução `await Task.WhenAll(tasks);` sai do método e retoma no método quando o processamento do arquivo é concluído para todas as tarefas.

O exemplo fecha todas as instâncias de `FileStream` em um bloco `finally` após as tarefas serem concluídas. Se cada `FileStream` foi criado em uma instrução `using`, o `FileStream` pode ter sido descartado antes de a tarefa ter sido concluída.

Qualquer aumento de desempenho é quase que totalmente do processamento paralelo e não o processamento assíncrono. As vantagens da assincronia são que ela não bloqueia vários threads e que ela não bloqueia o thread da interface do usuário.

```
C#  
  
public async Task ProcessMultipleWritesAsync()  
{  
    IList<FileStream> sourceStreams = new List<FileStream>();  
  
    try  
    {  
        string folder = Directory.CreateDirectory("tempfolder").Name;  
        IList<Task> writeTaskList = new List<Task>();  
  
        for (int index = 1; index <= 10; ++ index)  
        {
```

```

        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";

        string text = $"In file {index}{Environment.NewLine}";
        byte[] encodedText = Encoding.Unicode.GetBytes(text);

        var sourceStream =
            new FileStream(
                filePath,
                FileMode.Create, FileAccess.Write, FileShare.None,
                bufferSize: 4096, useAsync: true);

        Task writeTask = sourceStream.WriteAsync(encodedText, 0,
encodedText.Length);
        sourceStreams.Add(sourceStream);

        writeTaskList.Add(writeTask);
    }

    await Task.WhenAll(writeTaskList);
}
finally
{
    foreach (FileStream sourceStream in sourceStreams)
    {
        sourceStream.Close();
    }
}
}

```

Ao usar os métodos `WriteAsync` e `ReadAsync`, você pode especificar um `CancellationToken`, que pode ser usado para cancelar o fluxo intermediário da operação. Para saber mais, confira [Cancelamento em threads gerenciados](#).

## Confira também

- Programação assíncrona com `async` e `await` (C#)
- Tipos de retorno assíncronos (C#)

# Attributes

Article • 03/15/2023

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*.

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required.
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection.

Reflection provides objects (of type [Type](#)) that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you're using attributes in your code, reflection enables you to access them. For more information, see [Attributes](#).

Here's a simple example of reflection using the [GetType\(\)](#) method - inherited by all types from the `Object` base class - to obtain the type of a variable:

## ① Note

Make sure you add `using System;` and `using System.Reflection;` at the top of your .cs file.

C#

```
// Using GetType to obtain type information:  
int i = 42;  
Type type = i.GetType();  
Console.WriteLine(type);
```

The output is: `System.Int32`.

The following example uses reflection to obtain the full name of the loaded assembly.

C#

```
// Using Reflection to get information of an Assembly:  
Assembly info = typeof(int).Assembly;  
Console.WriteLine(info);
```

The output is something like: `System.Private.CoreLib, Version=7.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`.

### ⓘ Note

The C# keywords `protected` and `internal` have no meaning in Intermediate Language (IL) and are not used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. To identify an `internal` method using reflection, use the `IsAssembly` property. To identify a `protected internal` method, use the `IsFamilyOrAssembly`.

## Using attributes

Attributes can be placed on almost any declaration, though a specific attribute might restrict the types of declarations on which it's valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets (`[]`) above the declaration of the entity to which it applies.

In this example, the `SerializableAttribute` attribute is used to apply a specific characteristic to a class:

C#

```
[Serializable]  
public class SampleClass  
{  
    // Objects of this type can be serialized.  
}
```

A method with the attribute `DllImportAttribute` is declared like the following example:

C#

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

More than one attribute can be placed on a declaration as the following example shows:

C#

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is [ConditionalAttribute](#):

C#

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

### ① Note

By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Class Library.

## Attribute parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and can't be omitted. Named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

C#

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Positional parameters correspond to the parameters of the attribute constructor. Named or optional parameters correspond to either properties or fields of the attribute. Refer to the individual attribute's documentation for information on default parameter values.

For more information on allowed parameter types, see the [Attributes](#) section of the [C# language specification](#)

## Attribute targets

The *target* of an attribute is the entity that the attribute applies to. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that follows it. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
C#  
[target : attribute-list]
```

The list of possible `target` values is shown in the following table.

Target value	Applies to
<code>assembly</code>	Entire assembly
<code>module</code>	Current assembly module
<code>field</code>	Field in a class or a struct
<code>event</code>	Event
<code>method</code>	Method or <code>get</code> and <code>set</code> property accessors
<code>param</code>	Method parameters or <code>set</code> property accessor parameters
<code>property</code>	Property
<code>return</code>	Return value of a method, property indexer, or <code>get</code> property accessor
<code>type</code>	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an [auto-implemented property](#).

The following example shows how to apply attributes to assemblies and modules. For more information, see [Common Attributes \(C#\)](#).

C#

```
using System;
using System.Reflection;
[assembly: AssemblyTitle("Production assembly 4")]
[module: CLSCompliant(true)]
```

The following example shows how to apply attributes to methods, method parameters, and method return values in C#.

C#

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

### ⓘ Note

Regardless of the targets on which `ValidatedContract` is defined to be valid, the `return` target has to be specified, even if `ValidatedContract` were defined to apply only to return values. In other words, the compiler will not use `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see [AttributeUsage](#).

## Common uses for attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see [WebMethodAttribute](#).
- Describing how to marshal method parameters when interoperating with native code. For more information, see [MarshalAsAttribute](#).
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the `DllImportAttribute` class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

## Reflection overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see [Retrieving Information Stored in Attributes](#).
- For examining and instantiating types in an assembly.
- For building new types at run time. Use classes in [System.Reflection.Emit](#).
- For performing late binding, accessing methods on types created at run time. See the article [Dynamically Loading and Using Types](#).

## Related sections

For more information:

- [Common Attributes \(C#\)](#)
- [Caller Information \(C#\)](#)
- [Attributes](#)
- [Reflection](#)
- [Viewing Type Information](#)
- [Reflection and Generic Types](#)
- [System.Reflection.Emit](#)
- [Retrieving Information Stored in Attributes](#)

# Criar atributos personalizados

Artigo • 20/03/2023

Você pode criar seus próprios atributos personalizados definindo uma classe de atributos, uma classe que deriva direta ou indiretamente de [Attribute](#), o que faz com que a identificação das definições de atributo nos metadados seja rápida e fácil.

Suponha que você queira marcar tipos com o nome do programador que escreveu o tipo. Você pode definir uma classe de atributos [Author](#) personalizada:

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |  
                      System.AttributeTargets.Struct)  
]  
public class AuthorAttribute : System.Attribute  
{  
    private string Name;  
    public double Version;  
  
    public AuthorAttribute(string name)  
    {  
        Name = name;  
        Version = 1.0;  
    }  
}
```

O nome de classe [AuthorAttribute](#) é o nome do atributo, [Author](#), acrescido do sufixo [Attribute](#). Ela é derivada de [System.Attribute](#), portanto, é uma classe de atributo personalizado. Os parâmetros do construtor são parâmetros posicionais do atributo personalizado. Neste exemplo, [name](#) é um parâmetro posicional. Quaisquer propriedades ou campos públicos de leitura/gravação são chamados de parâmetros. Nesse caso, [version](#) é o único parâmetro nomeado. Observe o uso do atributo [AttributeUsage](#) para tornar o atributo [Author](#) válido apenas na classe e nas declarações [struct](#).

Você pode usar esse novo atributo da seguinte maneira:

C#

```
[Author("P. Ackerman", Version = 1.1)]  
class SampleClass  
{  
    // P. Ackerman's code goes here...  
}
```

`AttributeUsage` tem um parâmetro nomeado, `AllowMultiple`, com o qual você pode fazer um atributo personalizado de uso único ou multiuso. No exemplo de código a seguir, um atributo multiuso é criado.

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |  
                      System.AttributeTargets.Struct,  
                      AllowMultiple = true) // Multiuse attribute.  
]  
public class AuthorAttribute : System.Attribute  
{  
    string Name;  
    public double Version;  
  
    public AuthorAttribute(string name)  
    {  
        Name = name;  
  
        // Default value.  
        Version = 1.0;  
    }  
  
    public string GetName() => Name;  
}
```

No exemplo de código a seguir, vários atributos do mesmo tipo são aplicados a uma classe.

C#

```
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]  
public class ThirdClass  
{  
    // ...  
}
```

## Confira também

- [System.Reflection](#)
- [Escrevendo atributos personalizados](#)
- [AttributeUsage \(C#\)](#)

# Acessar atributos usando reflexão

Artigo • 20/03/2023

O fato de que você pode definir atributos personalizados e colocá-los em seu código-fonte seria de pouco valor sem alguma maneira de recuperar essas informações e tomar ação sobre elas. Por meio de reflexão, você pode recuperar as informações que foram definidas com atributos personalizados. O método principal é o `GetCustomAttributes`, que retorna uma matriz de objetos que são equivalentes, em tempo de execução, aos atributos do código-fonte. Esse método tem várias versões sobrecarregadas. Para obter mais informações, consulte [Attribute](#).

Uma especificação de atributo, como:

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass { }
```

é conceitualmente equivalente ao seguinte código:

C#

```
var anonymousAuthorObject = new Author("P. Ackerman")
{
    Version = 1.1
};
```

No entanto, o código não será executado até que `SampleClass` tenha os atributos consultados. Chamar `GetCustomAttributes` na `SampleClass` faz com que um objeto `Author` seja criado e inicializado. Se a classe tiver outros atributos, outros objetos de atributo serão construídos de forma semelhante. Então o `GetCustomAttributes` retornará o objeto `Author` e quaisquer outros objetos de atributo em uma matriz. Você poderá iterar sobre essa matriz, determinar quais atributos foram aplicados com base no tipo de cada elemento da matriz e extrair informações dos objetos de atributo.

Veja um exemplo completo. Um atributo personalizado é definido, aplicado a várias entidades e recuperado por meio da reflexão.

C#

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
    System.AttributeTargets.Struct,
```

```

                AllowMultiple = true) // Multiuse attribute.
]
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => Name;
}

// Class with the Author attribute.
[Author("P. Ackerman")]
public class FirstClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    public static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine($"Author information for {t}");

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);
    }
}

```

```
// Displaying output.
foreach (System.Attribute attr in attrs)
{
    if (attr is AuthorAttribute a)
    {
        System.Console.WriteLine($"    {a.GetName()}, version
{a.Version:f}");
    }
}
/* Output:
   Author information for FirstClass
   P. Ackerman, version 1.00
   Author information for SecondClass
   Author information for ThirdClass
   R. Koch, version 2.00
   P. Ackerman, version 1.00
*/
```

## Confira também

- [System.Reflection](#)
- [Attribute](#)
- [Recuperando informações armazenadas em atributos](#)

# Como criar uma união do C/C++ usando atributos em C#

Artigo • 03/04/2023

Usando atributos, você pode personalizar como os structs são dispostos na memória. Por exemplo, você pode criar o que é conhecido como uma união no C/C++ usando os atributos `StructLayout(LayoutKind.Explicit)` e `FieldOffset`.

Neste segmento de código, todos os campos de `TestUnion` são iniciados no mesmo local na memória.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public byte b;
}
```

O código a seguir é outro exemplo em que os campos são iniciados em locais diferentes definidos explicitamente.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public long lg;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i1;

    [System.Runtime.InteropServices.FieldOffset(4)]
    public int i2;

    [System.Runtime.InteropServices.FieldOffset(8)]
}
```

```
public double d;  
  
[System.Runtime.InteropServices.FieldOffset(12)]  
public char c;  
  
[System.Runtime.InteropServices.FieldOffset(14)]  
public byte b;  
}
```

Os dois campos inteiros, `i1` e `i2` combinados, compartilham os mesmos locais de memória que `lg`. `lg` usa os primeiros oito bytes, ou então `i1` usa os quatro primeiros bytes e `i2` usa os próximos quatro bytes. Esse tipo de controle sobre o layout do struct é útil ao usar a invocação de plataforma.

## Confira também

- [System.Reflection](#)
- [Attribute](#)
- [Atributos](#)

# Coleções (C#)

Artigo • 10/05/2023

Para muitos aplicativos, você desejará criar e gerenciar grupos de objetos relacionados. Há duas maneiras de agrupar objetos: criando matrizes de objetos e criando coleções de objetos.

As matrizes são mais úteis para criar e trabalhar com um número fixo de objetos fortemente tipados. Para obter informações sobre matrizes, consulte [Matrizes](#).

As coleções fornecem uma maneira mais flexível de trabalhar com grupos de objetos. Ao contrário das matrizes, o grupo de objetos com o qual você trabalha pode crescer e reduzir dinamicamente conforme as necessidades do aplicativo são alteradas. Para algumas coleções, você pode atribuir uma chave para qualquer objeto que coloque na coleção para que você possa recuperar rapidamente o objeto usando a chave.

Uma coleção é uma classe, portanto você deve declarar uma instância da classe antes de adicionar elementos a essa coleção.

Se a coleção contiver elementos de apenas um tipo de dados, você poderá usar uma das classes no namespace [System.Collections.Generic](#). Uma coleção genérica impõe segurança de tipos para que nenhum outro tipo de dados possa ser adicionado a ela. Ao recuperar um elemento de uma coleção genérica, você não precisa determinar seu tipo de dados ou convertê-lo.

## ⓘ Observação

Para os exemplos neste tópico, inclua diretivas `using` para os namespaces `System.Collections.Generic` e `System.Linq`.

## Neste tópico

- [Usando uma coleção simples](#)
- [Tipos de coleções](#)
  - [Classes System.Collections.Generic](#)
  - [Classes System.Collections.Concurrent](#)
  - [Classes System.Collections](#)
- [Implementando uma coleção de pares chave-valor](#)

- Usando LINQ para acessar uma coleção
- Classificando uma coleção
- Definindo uma coleção personalizada
- Iteradores

## Usando uma coleção simples

Os exemplos nesta seção usam a classe genérica `List<T>`, que habilita você a trabalhar com uma lista de objetos fortemente tipados.

O exemplo a seguir cria uma lista de cadeias de caracteres e, em seguida, itera nas cadeias de caracteres usando uma instrução `foreach`.

C#

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Se o conteúdo de uma coleção for conhecido com antecedência, você poderá usar um *inicializador de coleção* para inicializar a coleção. Para obter mais informações, consulte [Inicializadores de coleção e objeto](#).

O exemplo a seguir é igual ao exemplo anterior, exceto que um inicializador de coleção é usado para adicionar elementos à coleção.

C#

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
```

```
        Console.WriteLine(salmon + " ");
    }
// Output: chinook coho pink sockeye
```

Você pode usar uma instrução `for` em vez de uma instrução `foreach` para iterar em uma coleção. Você realiza isso acessando os elementos da coleção pela posição do índice. O índice dos elementos começa em 0 e termina na contagem de elementos, menos de 1.

O exemplo a seguir itera nos elementos de uma coleção usando `for` em vez de `foreach`.

C#

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

for (var index = 0; index < salmons.Count; index++)
{
    Console.WriteLine(salmons[index] + " ");
}
// Output: chinook coho pink sockeye
```

O exemplo a seguir remove um elemento da coleção, especificando o objeto a ser removido.

C#

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.WriteLine(salmon + " ");
}
// Output: chinook pink sockeye
```

O exemplo a seguir remove elementos de uma lista genérica. Em vez de uma instrução `foreach`, é usada uma instrução `for` que itera em ordem decrescente. Isso é feito porque o método `RemoveAt` faz com que os elementos após um elemento removido tenham um valor de índice menor.

C#

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.WriteLine(number + " "));
// Output: 0 2 4 6 8
```

Para o tipo dos elementos na `List<T>`, você também pode definir sua própria classe. No exemplo a seguir, a classe `Galaxy` que é usada pela `List<T>` é definida no código.

C#

```
private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
    {
        new Galaxy() { Name="Tadpole", MegaLightYears=400},
        new Galaxy() { Name="Pinwheel", MegaLightYears=25},
        new Galaxy() { Name="Milky Way", MegaLightYears=0},
        new Galaxy() { Name="Andromeda", MegaLightYears=3}
    };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears);
    }

    // Output:
    // Tadpole 400
    // Pinwheel 25
    // Milky Way 0
    // Andromeda 3
}

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

```
    public int MegaLightYears { get; set; }  
}
```

# Tipos de coleções

Várias coleções comuns são fornecidas pelo .NET. Cada tipo de coleção é projetado para uma finalidade específica.

Algumas das classes de coleção comuns são descritas nesta seção:

- Classes [System.Collections.Generic](#)
- Classes [System.Collections.Concurrent](#)
- Classes [System.Collections](#)

## Classes System.Collections.Generic

Você pode criar uma coleção genérica usando uma das classes no namespace [System.Collections.Generic](#). Uma coleção genérica é útil quando cada item na coleção tem o mesmo tipo de dados. Uma coleção genérica impõe tipagem forte, permitindo que apenas o tipo de dados desejado seja adicionado.

A tabela a seguir lista algumas das classes frequentemente usadas do namespace [System.Collections.Generic](#):

Classe	Descrição
<a href="#">Dictionary&lt;TKey,TValue&gt;</a>	Representa uma coleção de pares chave-valor organizados com base na chave.
<a href="#">List&lt;T&gt;</a>	Representa uma lista de objetos que podem ser acessados por índice. Fornece métodos para pesquisar, classificar e modificar listas.
<a href="#">Queue&lt;T&gt;</a>	Representa uma coleção de objetos PEPS (primeiro a entrar, primeiro a sair).
<a href="#">SortedList&lt;TKey,TValue&gt;</a>	Representa uma coleção de pares chave/valor que são classificados por chave com base na implementação de <a href="#">IComparer&lt;T&gt;</a> associada.
<a href="#">Stack&lt;T&gt;</a>	Representa uma coleção de objetos UEPS (último a entrar, primeiro a sair).

Para obter informações adicionais, consulte [Tipos de coleção comumente usados](#), [Selecionando uma classe de coleção](#) e [System.Collections.Generic](#).

## Classes System.Collections.Concurrent

No .NET Framework 4 e versões posteriores, as coleções no namespace [System.Collections.Concurrent](#) fornecem operações thread-safe eficientes para acessar itens da coleção por meio de vários threads.

As classes no namespace [System.Collections.Concurrent](#) deverão ser usadas em vez dos tipos correspondentes nos namespaces [System.Collections.Generic](#) e [System.Collections](#) sempre que vários threads estiverem acessando a coleção simultaneamente. Para obter mais informações, veja [Coleções thread-safe](#) e [System.Collections.Concurrent](#).

Algumas classes incluídas no namespace [System.Collections.Concurrent](#) são [BlockingCollection<T>](#), [ConcurrentDictionary<TKey,TValue>](#), [ConcurrentQueue<T>](#) e [ConcurrentStack<T>](#).

## Classes System.Collections

As classes no namespace [System.Collections](#) não armazenam elementos como objetos especificamente tipados, mas como objetos do tipo [Object](#).

Sempre que possível, você deve usar as coleções genéricas no namespace [System.Collections.Generic](#) ou no [System.Collections.Concurrent](#) em vez dos tipos herdados no namespace [System.Collections](#).

A tabela a seguir lista algumas das classes frequentemente usadas no namespace [System.Collections](#):

Classe	Descrição
<a href="#">ArrayList</a>	Representa uma matriz de objetos cujo tamanho é aumentado dinamicamente conforme necessário.
<a href="#">Hashtable</a>	Representa uma coleção de pares chave-valor organizados com base no código hash da chave.
<a href="#">Queue</a>	Representa uma coleção de objetos PEPS (primeiro a entrar, primeiro a sair).
<a href="#">Stack</a>	Representa uma coleção de objetos UEPS (último a entrar, primeiro a sair).

O namespace [System.Collections.Specialized](#) fornece classes de coleções especializadas e fortemente tipadas, como coleções somente de cadeias de caracteres, bem como de dicionários híbridos e de listas vinculadas.

# Implementando uma coleção de pares chave-valor

A coleção genérica `Dictionary< TKey, TValue >` permite que você acesse elementos em uma coleção usando a chave de cada elemento. Cada adição ao dicionário consiste em um valor e a respectiva chave associada. A recuperação de um valor usando sua chave é rápida, porque a classe `Dictionary` é implementada como uma tabela de hash.

O exemplo a seguir cria uma coleção `Dictionary` e itera no dicionário usando uma instrução `foreach`.

C#

```
private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
        Console.WriteLine("values: " + theElement.Symbol + " " +
                          theElement.Name + " " + theElement.AtomicNumber);
    }
}

private static Dictionary<string, Element> BuildDictionary()
{
    var elements = new Dictionary<string, Element>();

    AddToDictionary(elements, "K", "Potassium", 19);
    AddToDictionary(elements, "Ca", "Calcium", 20);
    AddToDictionary(elements, "Sc", "Scandium", 21);
    AddToDictionary(elements, "Ti", "Titanium", 22);

    return elements;
}

private static void AddToDictionary(Dictionary<string, Element> elements,
                                    string symbol, string name, int atomicNumber)
{
    Element theElement = new Element();

    theElement.Symbol = symbol;
    theElement.Name = name;
    theElement.AtomicNumber = atomicNumber;

    elements.Add(key: theElement.Symbol, value: theElement);
}
```

```
public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}
```

Para, em vez disso, usar um inicializador de coleção para criar a coleção `Dictionary`, você pode substituir os métodos `BuildDictionary` e `AddToDictionary` pelo seguinte método.

C#

```
private static Dictionary<string, Element> BuildDictionary2()
{
    return new Dictionary<string, Element>
    {
        {"K",
            new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}
```

O exemplo a seguir usa o método `ContainsKey` e a propriedade `Item[]` de `Dictionary` para localizar rapidamente um item por chave. A propriedade `Item` permite que você accesse um item na coleção `elements` usando o `elements[symbol]` no C#.

C#

```
private static void FindInDictionary(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    if (elements.ContainsKey(symbol) == false)
    {
        Console.WriteLine(symbol + " not found");
    }
    else
    {
        Element theElement = elements[symbol];
        Console.WriteLine("Found: " + theElement.Name);
    }
}
```

O exemplo a seguir usa o método [TryGetValue](#) para localizar rapidamente um item por chave.

C#

```
private static void FindInDictionary2(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    Element theElement = null;
    if (elements.TryGetValue(symbol, out theElement) == false)
        Console.WriteLine(symbol + " not found");
    else
        Console.WriteLine("found: " + theElement.Name);
}
```

## Usando LINQ para acessar uma coleção

A LINQ (consulta integrada à linguagem) pode ser usada para acessar coleções. As consultas LINQ fornecem recursos de filtragem, classificação e agrupamento. Para obter mais informações, confira [Introdução à LINQ no C#](#).

O exemplo a seguir executa uma consulta LINQ em uma `List` genérica. A consulta LINQ retorna uma coleção diferente que contém os resultados.

C#

```
private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    // Calcium 20
    // Potassium 19
    // Scandium 21
}

private static List<Element> BuildList()
```

```

{
    return new List<Element>
{
    { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
    { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
    { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
    { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
};
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}

```

## Classificando uma coleção

O exemplo a seguir ilustra um procedimento para a classificação de uma coleção. O exemplo classifica instâncias da classe `Car` que estão armazenados em uma `List<T>`. A classe `Car` implementa a interface `IComparable<T>`, que requer que o método `CompareTo` seja implementado.

Cada chamada ao método `CompareTo` faz uma comparação única que é usada para classificação. Os códigos escritos pelo usuário no método `CompareTo` retornam um valor para cada comparação do objeto atual com outro objeto. O valor retornado será menor que zero se o objeto atual for menor que o outro objeto, maior que zero se o objeto atual for maior que o outro objeto e zero, se eles forem iguais. Isso permite que você defina no código os critérios para maior que, menor que e igual.

No método `ListCars`, a instrução `cars.Sort()` classifica a lista. Essa chamada para o método `Sort` da `List<T>` faz com que o método `CompareTo` seja chamado automaticamente para os objetos `Car` na `List`.

C#

```

private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20}},
        { new Car() { Name = "car2", Color = "red", Speed = 50}},
        { new Car() { Name = "car3", Color = "green", Speed = 10}},
        { new Car() { Name = "car4", Color = "blue", Speed = 50}},
        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
        { new Car() { Name = "car6", Color = "red", Speed = 60}},
    };
}
```

```

        { new Car() { Name = "car7", Color = "green", Speed = 50}};

    }

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.Write(thisCar.Color.PadRight(5) + " ");
        Console.Write(thisCar.Speed.ToString() + " ");
        Console.WriteLine(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    // blue 50 car4
    // blue 30 car5
    // blue 20 car1
    // green 50 car7
    // green 10 car3
    // red 60 car6
    // red 50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}

```

```
    }  
}
```

## Definindo uma coleção personalizada

Você pode definir uma coleção implementando a interface `IEnumerable<T>` ou `IEnumerable`.

Embora seja possível definir uma coleção personalizada, é melhor usar as coleções que estão incluídas no .NET, descritas em [Tipos de coleções](#) anteriormente neste tópico.

O exemplo a seguir define uma classe de coleção personalizada chamada `AllColors`. Essa classe implementa a interface `IEnumerable`, que requer que o método `GetEnumerator` seja implementado.

O método `GetEnumerator` retorna uma instância da classe `ColorEnumerator`.

`ColorEnumerator` implementa a interface `IEnumerator`, que requer que a propriedade `Current`, o método `MoveNext` e o método `Reset` sejam implementados.

C#

```
private static void ListColors()  
{  
    var colors = new AllColors();  
  
    foreach (Color theColor in colors)  
    {  
        Console.Write(theColor.Name + " ");  
    }  
    Console.WriteLine();  
    // Output: red blue green  
}  
  
// Collection class.  
public class AllColors : System.Collections.IEnumerable  
{  
    Color[] _colors =  
    {  
        new Color() { Name = "red" },  
        new Color() { Name = "blue" },  
        new Color() { Name = "green" }  
    };  
  
    public System.Collections.IEnumerator GetEnumerator()  
    {  
        return new ColorEnumerator(_colors);  
  
        // Instead of creating a custom enumerator, you could  
        // use the GetEnumerator of the array.
```

```

        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
            _position = -1;
        }
    }
}

// Element class.
public class Color
{
    public string Name { get; set; }
}

```

## Iterators

Um *iterador* é usado para realizar uma iteração personalizada em uma coleção. Um iterador pode ser um método ou um acessador `get`. Um iterador usa uma instrução `yield return` para retornar um elemento da coleção por vez.

Você chama um iterador usando uma instrução `foreach`. Cada iteração do loop `foreach` chama o iterador. Quando uma instrução `yield return` é alcançada no iterador, uma

expressão é retornada e o local atual no código é retido. A execução será reiniciada desse local na próxima vez que o iterador for chamado.

Para obter mais informações, consulte [Iteradores \(C#\)](#).

O exemplo a seguir usa um método iterador. O método iterador tem uma instrução `yield return` que está dentro de um loop `for`. No método `ListEvenNumbers`, cada iteração do corpo da instrução `foreach` cria uma chamada ao método iterador, que avança para a próxima instrução `yield return`.

C#

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

## Confira também

- [Inicializadores de objeto e coleção](#)
- [Conceitos de programação \(C#\)](#)
- [Instrução Option Strict](#)
- [LINQ to Objects \(C#\)](#)
- [LINQ paralelo \(PLINQ\)](#)
- [Coleções e Estruturas de Dados](#)
- [Selecionando uma classe de coleção](#)
- [Comparações e Classificações Dentro de Coleções](#)
- [Quando Usar Coleções Genéricas](#)

- Instruções de iteração

# Covariância e contravariância (C#)

Artigo • 15/02/2023

No C#, a covariância e a contravariância habilitam a conversão de referência implícita para tipos de matriz, tipos de delegados e argumentos de tipo genérico. A covariância preserva a compatibilidade de atribuição, e a contravariância reverte.

O código a seguir demonstra a diferença entre a compatibilidade da atribuição, a covariância e a contravariância.

C#

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less
derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

A covariância para matrizes permite a conversão implícita de uma matriz de um tipo mais derivado para uma matriz de um tipo menos derivado. Mas essa operação não é fortemente tipada, conforme mostrado no exemplo de código a seguir.

C#

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

O suporte de covariância e contravariância aos grupos de método permite a correspondência de assinaturas de método com tipos de delegados. Isso permite

atribuir a delegados não apenas os métodos que têm correspondência de assinaturas, mas também métodos que retornam tipos mais derivados (covariância) ou que aceitam parâmetros que têm tipos menos derivados (contravariância) do que o especificado pelo tipo delegado. Para obter mais informações, consulte [Variância em delegados \(C#\)](#) e [Usando variância em delegados \(C#\)](#).

O exemplo de código a seguir mostra o suporte da covariância e da contravariância para grupos de método.

C#

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

No .NET Framework 4 e versões mais recentes, o C# oferece suporte à covariância e à contravariância em interfaces e delegados genéricos, e permite a conversão implícita de parâmetros de tipo genérico. Para obter mais informações, consulte [Variação em interfaces genéricas \(C#\)](#) e [Variação em delegados \(C#\)](#).

O exemplo de código a seguir mostra a conversão de referência implícita para interfaces genéricas.

C#

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

Uma interface ou delegado genérico será chamado *variante* se seus parâmetros genéricos forem declarados covariantes ou contravariantes. O C# permite que você crie suas próprias interfaces variantes e delegados. Para obter mais informações, consulte [Criando interfaces genéricas variantes \(C#\)](#) e [Variação em delegados \(C#\)](#).

# Tópicos Relacionados

Título	Descrição
<a href="#">Variância em interfaces genéricas (C#)</a>	Discute a covariância e a contravariância em interfaces genéricas e fornece uma lista de interfaces genéricas variáveis no .NET Framework.
<a href="#">Criando interfaces genéricas variáveis (C#)</a>	Mostra como criar interfaces variantes personalizadas.
<a href="#">Usando variação em interfaces para Coleções Genéricas (C#)</a>	Mostra como o suporte de covariância e contravariância nas interfaces <code>IEnumerable&lt;T&gt;</code> e <code>IComparable&lt;T&gt;</code> pode ajudar na reutilização do código.
<a href="#">Variação em delegados (C#)</a>	Discute a covariância e a contravariância em interfaces genéricas e não genéricas, e fornece uma lista de delegados genéricos variáveis no .NET.
<a href="#">Usando variação em delegados (C#)</a>	Mostra como usar o suporte de covariância e contravariância em delegados não genéricos para corresponder às assinaturas de método com tipos delegados.
<a href="#">Usando variação para delegados genéricos Func e Action (C#)</a>	Mostra como o suporte de covariância e contravariância nos delegados <code>Func</code> e <code>Action</code> pode ajudar na reutilização do código.

# Variância em interfaces genéricas (C#)

Artigo • 10/05/2023

O .NET Framework 4 introduziu o suporte à variação para diversas interfaces genéricas existentes. O suporte à variação possibilita a conversão implícita de classes que implementam essas interfaces.

Do .NET Framework 4 em diante, as seguintes interfaces são variantes:

- `IEnumerable<T>` (T é covariante)
- `IEnumerator<T>` (T é covariante)
- `IQueryable<T>` (T é covariante)
- `IGrouping< TKey, TElement >` (`TKey` e `TElement` são covariantes)
- `IComparer<T>` (T é contravariante)
- `IEqualityComparer<T>` (T é contravariante)
- `IComparable<T>` (T é contravariante)

A partir do .NET Framework 4.5, as seguintes interfaces são variantes:

- `IReadOnlyList<T>` (T é covariante)
- `IReadOnlyCollection<T>` (T é covariante)

A covariância permite que um método tenha um tipo de retorno mais derivados que aquele definidos pelo parâmetro de tipo genérico da interface. Para ilustrar o recurso de covariância, considere estas interfaces genéricas: `IEnumerable<Object>` e `IEnumerable<String>`. A interface `IEnumerable<String>` não herda a interface `IEnumerable<Object>`. No entanto, o tipo `string` herda o tipo `Object` e, em alguns casos, talvez você queira atribuir objetos dessas interfaces uns aos outros. Isso é mostrado no exemplo de código a seguir.

C#

```
IEnumerable<String> strings = new List<String>();  
IEnumerable<Object> objects = strings;
```

Em versões anteriores do .NET Framework, esse código gera um erro de compilação no C# e, se `Option Strict` estiver ativado, no Visual Basic. Mas agora você pode usar

`strings` em vez de `objects`, conforme mostrado no exemplo anterior, porque a interface `IEnumerable<T>` é covariante.

A contravariância permite que um método tenha tipos de argumentos menos derivados que aquele especificado pelo parâmetro genérico da interface. Para ilustrar a contravariância, suponha que você tenha criado uma classe `BaseComparer` para comparar instâncias da classe `BaseClass`. A classe `BaseComparer` implementa a interface `IEqualityComparer<BaseClass>`. Como a interface `IEqualityComparer<T>` agora é contravariante, você pode usar `BaseComparer` para comparar instâncias de classes que herdam a classe `BaseClass`. Isso é mostrado no exemplo de código a seguir.

C#

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

Para ver mais exemplos, consulte [Usando variação em interfaces para coleções genéricas \(C#\)](#).

A variação em interfaces genéricas tem suporte somente para tipos de referência. Tipos de valor não dão suporte à variação. Por exemplo, `IEnumerable<int>` não pode ser convertido implicitamente em `IEnumerable<object>`, porque inteiros são representados por um tipo de valor.

C#

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
// because int is a value type.
// IEnumerable<Object> objects = integers;
```

Também é importante lembrar que as classes que implementam interfaces variantes ainda são invariantes. Por exemplo, embora `List<T>` implemente a interface covariante `IEnumerable<T>`, você não pode converter implicitamente `List<String>` para `List<Object>`. Isso é ilustrado no exemplo de código a seguir.

C#

```
// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();
```

## Confira também

- Usando variação em interfaces para Coleções Genéricas (C#)
- Criando interfaces genéricas variáveis (C#)
- Interfaces genéricas
- Variação em delegados (C#)

# Criando interfaces genéricas variantes (C#)

Artigo • 10/05/2023

Você pode declarar parâmetros de tipo genérico em interfaces como covariantes ou contravariantes. A *Covariância* permite que os métodos de interface tenham tipos de retorno mais derivados que aqueles definidos pelos parâmetros de tipo genérico. A *Contravariância* permite que os métodos de interface tenham tipos de argumentos que são menos derivados que aqueles especificados pelos parâmetros genéricos. Uma interface genérica que tenha parâmetros de tipo genérico covariantes ou contravariantes é chamada de *variante*.

## ⓘ Observação

O .NET Framework 4 introduziu o suporte à variação para diversas interfaces genéricas existentes. Para obter a lista das interfaces variantes no .NET, consulte [Variância em interfaces genéricas \(C#\)](#).

## Declarando interfaces genéricas variantes

Você pode declarar interfaces genéricas variantes usando as palavras-chave `in` e `out` para parâmetros de tipo genérico.

## ⓘ Importante

Os parâmetros `ref`, `in` e `out` no C# não podem ser variantes. Os tipos de valor também não dão suporte à variância.

Você pode declarar um parâmetro de tipo genérico como covariante usando a palavra-chave `out`. O tipo de covariante deve satisfazer as condições a seguir:

- O tipo é usado apenas como um tipo de retorno dos métodos de interface e não é usado como um tipo de argumentos de método. Isso é ilustrado no exemplo a seguir, no qual o tipo `R` é declarado covariante.

C#

```
interface ICovariant<out R>
{
```

```
R GetSomething();
// The following statement generates a compiler error.
// void SetSomething(R sampleArg);

}
```

Há uma exceção a essa regra. Se você tiver um delegado genérico contravariante como um parâmetro de método, você poderá usar o tipo como um parâmetro de tipo genérico para o delegado. Isso é ilustrado pelo tipo `R` no exemplo a seguir.

Para obter mais informações, consulte [Variância em delegados \(C#\)](#) e [Usando variância para delegados genéricos Func e Action \(C#\)](#).

C#

```
interface ICovariant<out R>
{
    void DoSomething(Action<R> callback);
}
```

- O tipo não é usado como uma restrição genérica para os métodos de interface. O código a seguir ilustra isso.

C#

```
interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic constraints.
    // void DoSomething<T>() where T : R;
}
```

Você pode declarar um parâmetro de tipo genérico como contravariante usando a palavra-chave `in`. O tipo contravariante pode ser usado apenas como um tipo de argumentos de método e não como um tipo de retorno dos métodos de interface. O tipo contravariante também pode ser usado para restrições genéricas. O código a seguir mostra como declarar uma interface contravariante e usar uma restrição genérica para um de seus métodos.

C#

```
interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
```

```
// A GetSomething();  
}
```

Também é possível oferecer suporte à covariância e contravariância na mesma interface, mas para parâmetros de tipo diferentes, conforme mostrado no exemplo de código a seguir.

C#

```
interface IVariant<out R, in A>  
{  
    R GetSomething();  
    void SetSomething(A sampleArg);  
    R GetSetSomethings(A sampleArg);  
}
```

## Implementando interfaces genéricas variantes

Você pode implementar interfaces genéricas variantes em classes, através da mesma sintaxe que é usada para interfaces invariantes. O exemplo de código a seguir mostra como implementar uma interface covariante em uma classe genérica.

C#

```
interface ICovariant<out R>  
{  
    R GetSomething();  
}  
class SampleImplementation<R> : ICovariant<R>  
{  
    public R GetSomething()  
    {  
        // Some code.  
        return default(R);  
    }  
}
```

As classes que implementam interfaces variantes são invariantes. Por exemplo, considere o código a seguir.

C#

```
// The interface is covariant.  
ICovariant<Button> ibutton = new SampleImplementation<Button>();  
ICovariant<Object> iobj = ibutton;  
  
// The class is invariant.
```

```
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

## Estendendo interfaces genéricas variantes

Quando você estende uma interface genérica variante, é necessário usar as palavras-chave `in` e `out` para especificar explicitamente se a interface derivada dará suporte à variância. O compilador não deduz a variância da interface que está sendo estendida. Por exemplo, considere as seguintes interfaces.

C#

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

Na interface `IInvariant<T>`, o parâmetro de tipo genérico `T` é invariante, enquanto que no `IExtCovariant<out T>` o parâmetro de tipo é covariante, embora as duas interfaces estendam a mesma interface. A mesma regra é aplicada aos parâmetros de tipo genérico contravariantes.

Você pode criar uma interface que estende tanto a interface em que o parâmetro de tipo genérico `T` é covariante, quanto a interface em que ele é contravariante, caso na interface de extensão o parâmetro de tipo genérico `T` seja invariante. Isso é ilustrado no exemplo de código a seguir.

C#

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

No entanto, se um parâmetro de tipo genérico `T` for declarado covariante em uma interface, você não poderá declará-lo contravariante na interface de extensão ou vice-versa. Isso é ilustrado no exemplo de código a seguir.

C#

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

## Evitando ambiguidade

Ao implementar interfaces genéricas variantes, a variância, às vezes, pode levar à ambiguidade. Essa ambiguidade deve ser evitada.

Por exemplo, se você implementar explicitamente a mesma interface genérica variante com parâmetros de tipo genérico diferentes em uma classe, isso poderá criar ambiguidade. O compilador não produzirá um erro nesse caso, mas não está especificada qual implementação de interface será escolhida em runtime. Essa ambiguidade pode causar bugs sutis no seu código. Considere o exemplo de código a seguir.

C#

```
// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}

class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

Neste exemplo, não está especificado como o método `pets.GetEnumerator` escolherá entre `Cat` e `Dog`. Isso poderá causar problemas em seu código.

## Confira também

- [Variância em interfaces genéricas \(C#\)](#)
- [Usando variação para delegados genéricos Func e Action \(C#\)](#)

# Usando variação em interfaces para Coleções Genéricas (C#)

Artigo • 10/05/2023

Uma interface de covariante permite que seus métodos retornem tipos derivados daquelas especificadas na interface. Uma interface de contravariante permite que seus métodos aceitem parâmetros de tipos menos derivados do que os especificados na interface.

No .NET Framework 4, várias interfaces existentes se tornaram covariantes e contravariantes. Eles incluem `IEnumerable<T>` e `IComparable<T>`. Isso permite que você reutilize métodos que operam com coleções genéricas de tipos base para coleções de tipos derivados.

Para obter uma lista de interfaces variantes no .NET, confira [Variância em interfaces genéricas \(C#\)](#).

## Convertendo coleções genéricas

O exemplo a seguir ilustra os benefícios do suporte à covariância na interface `IEnumerable<T>`. O método `PrintFullName` aceita uma coleção do tipo `IEnumerable<Person>` como um parâmetro. No entanto, você pode reutilizá-lo para uma coleção do tipo `IEnumerable<Employee>` porque `Employee` herda `Person`.

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
                person.FirstName, person.LastName);
        }
    }
}
```

```

        }

    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.

        PrintFullName(employees);

    }
}

```

## Comparando coleções genéricas

O exemplo a seguir ilustra os benefícios do suporte à contravariância na interface `IEqualityComparer<T>`. A classe `PersonComparer` implementa a interface `IEqualityComparer<Person>`. No entanto, você pode reutilizar essa classe para comparar uma sequência de objetos do tipo `Employee` porque `Employee` herda `Person`.

C#

```

// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
        int hashFirstName = person.FirstName == null

```

```

        ? 0 : person.FirstName.GetHashCode();
    int hashLastName = person.LastName.GetHashCode();
    return hashFirstName ^ hashLastName;
}
}

class Program
{
    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
            new Employee() {FirstName = "Michael", LastName =
"Alexander"}, 
            new Employee() {FirstName = "Jeff", LastName = "Price"}
        };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());
        
        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}

```

## Confira também

- [Variância em interfaces genéricas \(C#\)](#)

# Variação em delegados (C#)

Artigo • 09/05/2023

O .NET Framework 3.5 introduziu o suporte a variação para assinaturas de método correspondentes com tipos de delegados em todos os delegados do C#. Isso significa que você pode atribuir a delegados não apenas os métodos que têm assinaturas correspondentes, mas também métodos que retornam tipos mais derivados (covariância) ou que aceitam parâmetros que têm tipos menos derivados (contravariância) do que o especificado pelo tipo de delegado. Isso inclui delegados genéricos e não genéricos.

Por exemplo, considere o código a seguir, que tem duas classes e dois delegados: genérico e não genérico.

C#

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

Quando cria delegados dos tipos `SampleDelegate` ou `SampleGenericDelegate<A, R>`, você pode atribuir qualquer um dos seguintes métodos a esses delegados.

C#

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

O exemplo de código a seguir ilustra a conversão implícita entre a assinatura do método e o tipo de delegado.

C#

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

Para obter mais informações, consulte [Usando variação em delegados \(C#\)](#) e [Usando variação para os delegados genéricos Func e Action \(C#\)](#).

## Variação em parâmetros de tipo genérico

No .NET Framework 4 ou posterior, agora você pode habilitar a conversão implícita entre delegados, de modo que delegados genéricos que têm tipos diferentes especificados por parâmetros de tipo genérico podem ser atribuídos uns aos outros, se os tipos forem herdados uns dos outros como é obrigatório de acordo com a variação.

Para habilitar a conversão implícita, você precisa declarar explicitamente os parâmetros genéricos em um delegado como covariante ou contravariante usando a palavra-chave `in` ou `out`.

O exemplo de código a seguir mostra como você pode criar um delegado que tem um parâmetro de tipo genérico covariante.

C#

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";
    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

Se você usar somente o suporte à para fazer a correspondência de assinaturas de método com tipos de delegados e não usar as palavras-chave `in` e `out`, você poderá perceber que, às vezes, é possível instanciar delegados com métodos ou expressões lambda idênticas, mas não é possível atribuir um delegado a outro.

No exemplo de código a seguir, `SampleGenericDelegate<String>` não pode ser convertido explicitamente em `SampleGenericDelegate<Object>`, embora `String` herde `Object`. Você pode corrigir esse problema marcando o parâmetro genérico `T` com a palavra-chave `out`.

C#

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

## Delegados genéricos que têm parâmetros de tipo variante no .NET

O .NET Framework 4 introduziu o suporte à variação para parâmetros de tipo genérico em diversos delegados genéricos existentes:

- `Action` delega do namespace `System`, por exemplo, `Action<T>` e `Action<T1,T2>`
- `Func` delega do namespace `System`, por exemplo, `Func<TResult>` e `Func<T,TResult>`
- O delegado `Predicate<T>`
- O delegado `Comparison<T>`

- O delegado `Converter<TInput,TOutput>`

Para obter mais informações e exemplos, consulte [Usando variação para delegados genéricos Func e Action \(C#\)](#).

## Declarando parâmetros de tipo variante em delegados genéricos

Se um delegado genérico tiver parâmetros de tipo genérico covariantes ou contravariantes, ele poderá ser considerado um *delegado genérico variante*.

Você pode declarar um parâmetro de tipo genérico covariante em um delegado genérico usando a palavra-chave `out`. O tipo covariante pode ser usado apenas como um tipo de retorno de método e não como um tipo de argumentos de método. O exemplo de código a seguir mostra como declarar um delegado genérico covariante.

C#

```
public delegate R DCovariant<out R>();
```

Você pode declarar um parâmetro de tipo genérico contravariante em um delegado genérico usando a palavra-chave `in`. O tipo contravariante pode ser usado apenas como um tipo de argumentos de método e não como um tipo de retorno de método. O exemplo de código a seguir mostra como declarar um delegado genérico contravariante.

C#

```
public delegate void DContravariant<in A>(A a);
```

### ⓘ Importante

Os parâmetros `ref`, `in` e `out` em C# não podem ser marcados como variantes.

Também é possível dar suporte à variância e à covariância no mesmo delegado, mas para parâmetros de tipo diferente. Isso é mostrado no exemplo a seguir.

C#

```
public delegate R DVariant<in A, out R>(A a);
```

## Instanciando e invocando delegados genéricos variantes

Você pode instanciar e invocar delegados variantes da mesma forma como instancia e invoca delegados invariantes. No exemplo a seguir, um delegado é instanciado por uma expressão lambda.

C#

```
DVariant<String, String> dvariant = (String str) => str + " ";  
dvariant("test");
```

## Combinando delegados genéricos variantes

Não combine delegados variantes. O método [Combine](#) não dá suporte à conversão de delegados variantes e espera que os delegados sejam exatamente do mesmo tipo. Isso pode levar a uma exceção de tempo de execução quando você combina delegados usando o método [Combine](#) ou o operador `+`, conforme mostrado no exemplo de código a seguir.

C#

```
Action<Object> actObj = x => Console.WriteLine("Object: {0}", x);  
Action<String> actStr = x => Console.WriteLine("string: {0}", x);  
// All of the following statements throw exceptions at run time.  
// Action<string> actCombine = actStr + actObj;  
// actStr += actObj;  
// Delegate.Combine(actStr, actObj);
```

## Variação em parâmetros de tipo genérico para tipos de referência e valor

A variação para parâmetros de tipo genérico tem suporte apenas para tipos de referência. Por exemplo, `DVariant<int>` não pode ser convertido implicitamente em `DVariant<Object>` ou `DVariant<long>`, pois inteiro é um tipo de valor.

O exemplo a seguir demonstra que a variação em parâmetros de tipo genérico não tem suporte para tipos de valor.

C#

```
// The type T is covariant.  
public delegate T DVariant<out T>();
```

```
// The type T is invariant.  
public delegate T DInvariant<T>();  
  
public static void Test()  
{  
    int i = 0;  
    DInvariant<int> dInt = () => i;  
    DVariant<int> dVariantInt = () => i;  
  
    // All of the following statements generate a compiler error  
    // because type variance in generic parameters is not supported  
    // for value types, even if generic type parameters are declared  
    variant.  
    // DInvariant<Object> dObject = dInt;  
    // DInvariant<long> dLong = dInt;  
    // DVariant<Object> dVariantObject = dVariantInt;  
    // DVariant<long> dVariantLong = dVariantInt;  
}
```

## Confira também

- [Genéricos](#)
- [Usando variação para delegados genéricos Func e Action \(C#\)](#)
- [Como combinar delegados \(delegados multicast\)](#)

# Usando variação em delegações (C#)

Artigo • 07/04/2023

Quando você atribui um método a um delegado, a *covariância* e a *contravariância* fornece flexibilidade para corresponder um tipo de delegado a uma assinatura de método. A covariância permite que um método tenha o tipo de retorno mais derivado do que o definido no delegado. A contravariância permite que um método que tem tipos de parâmetro menos derivados do que no tipo delegado.

## Exemplo 1: covariância

### Descrição

Este exemplo demonstra como delegados podem ser usados com métodos que têm tipos de retorno que são derivados do tipo de retorno na assinatura do delegado. O tipo de dados retornado por `DogsHandler` é do tipo `Dogs`, que deriva do tipo `Mammals` definido no delegado.

### Código

C#

```
class Mammals {}
class Dogs : Mammals {}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;

        // Covariance enables this assignment.
    }
}
```

```
    HandlerMethod handlerDogs = DogsHandler;
}
}
```

## Exemplo 2: contravariância

### Descrição

Este exemplo demonstra como representantes podem ser usados com métodos que têm parâmetros cujos tipos são tipos base do tipo de parâmetro de assinatura do representante. Com a contravariância, você pode usar um manipulador de eventos em vez de manipuladores separados. O seguinte exemplo usa dois representantes:

- Um representante [KeyEventHandler](#) que define a assinatura do evento [Button.KeyDown](#). Sua assinatura é:

```
C#
public delegate void KeyEventHandler(object sender, KeyEventArgs e)
```

- Um representante [MouseEventHandler](#) que define a assinatura do evento [Button.MouseClick](#). Sua assinatura é:

```
C#
public delegate void MouseEventHandler(object sender, MouseEventArgs e)
```

O exemplo define um manipulador de eventos com um parâmetro [EventArgs](#) e o usa para manipular os eventos [Button.KeyDown](#) e [Button.MouseClick](#). Ele pode fazer isso porque [EventArgs](#) é um tipo base de [KeyEventArgs](#) e [MouseEventArgs](#).

### Código

```
C#
// Event handler that accepts a parameter of the EventArgs type.
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
```

```
InitializeComponent();

// You can use a method that has an EventArgs parameter,
// although the event expects the KeyEventArgs parameter.
this.button1.KeyDown += this.MultiHandler;

// You can use the same method
// for an event that expects the MouseEventArgs parameter.
this.button1.MouseClick += this.MultiHandler;

}
```

## Confira também

- [Variação em delegados \(C#\)](#)
- [Usando variação para delegados genéricos Func e Action \(C#\)](#)

# Usando variância para delegados genéricos Func e Action (C#)

Artigo • 07/04/2023

Esses exemplos demonstram como usar covariância e contravariância nos delegados genéricos `Func` e `Action` para permitir a reutilização dos métodos e fornecer mais flexibilidade em seu código.

Para obter mais informações sobre covariância e contravariância, consulte [Variação em delegações \(C#\)](#).

## Usando delegados com parâmetros de tipo covariantes

O exemplo a seguir ilustra os benefícios do suporte à covariância nos delegados genéricos `Func`. O método `FindByTitle` assume um parâmetro do tipo `String` e retorna um objeto do tipo `Employee`. No entanto, você pode atribuir esse método ao delegado `Func<String, Person>` porque `Employee` herda `Person`.

C#

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
```

```
    findPerson = findEmployee;

}

}
```

## Usando delegados com parâmetros de tipo contravariantes

O exemplo a seguir ilustra os benefícios do suporte à contravariância nos delegados genéricos `Action`. O método `AddToContacts` assume um parâmetro do tipo `Person`. No entanto, você pode atribuir esse método ao delegado `Action<Employee>` porque `Employee` herda `Person`.

C#

```
public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}
```

## Confira também

- Covariância e contravariância (C#)

- Genéricos

# Expression Trees

Article • 03/09/2023

*Expression trees* represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

If you have used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you aren't familiar with LINQ, you probably want to read [the LINQ tutorial](#) and the article about [lambda expressions](#) before this one.) Expression Trees provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

You've likely already written code that uses Expression trees. Entity Framework's LINQ APIs accept Expression trees as the arguments for the LINQ Query Expression Pattern. That enables [Entity Framework](#) to translate the query you wrote in C# into SQL that executes in the database engine. Another example is [Moq](#) ↗, which is a popular mocking framework for .NET.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you examine, modify, or execute. These tools give you the power to manipulate code during run time. You write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you modify running algorithms and even translate C# expressions into another form for execution in another environment.

You compile and run code represented by expression trees. Building and running expression trees enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see [How to use expression trees to build dynamic queries](#).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see [Dynamic Language Runtime Overview](#).

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the `System.Linq.Expressions` namespace.

When a lambda expression is assigned to a variable of type `Expression<TDelegate>`, the compiler emits code to build an expression tree that represents the lambda expression.

The C# compiler generates expression trees only from expression lambdas (or single-line lambdas). It can't parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see [Lambda Expressions](#).

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

C#

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

You create expression trees in your code. You build the tree by creating each node and attaching the nodes into a tree structure. You learn how to create expressions in the article on [building expression trees](#).

Expression trees are immutable. If you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You use an expression tree visitor to traverse the existing expression tree. For more information, see the article on [translating expression trees](#).

Once you build an expression tree, you [execute the code represented by the expression tree](#).

## Limitations

There are some newer C# language elements that don't translate well into expression trees. Expression trees can't contain `await` expressions, or `async` lambda expressions. Many of the features added in C# 6 and later don't appear exactly as written in expression trees. Instead, newer features are exposed in expression trees in the equivalent, earlier syntax, where possible. Other constructs aren't available. It means that code that interprets expression trees works the same when new language features are introduced. However, even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It enables rich libraries such as Entity Framework to accomplish what they do.

Expression trees won't support new expression node types. It would be a breaking change for all libraries interpreting expression trees to introduce new node types. The following list includes most C# language elements that can't be used:

- Conditional methods that have been removed
- base access
- Method group expressions, including *address-of* (&) a method group, and anonymous method expressions
- References to local functions
- Statements, including assignment (=) and statement bodied expressions
- Partial methods with only a defining declaration
- Unsafe pointer operations
- dynamic operations
- Coalescing operators with null or default literal left side, null coalescing assignment, and the null propagating operator (?)
- Multi-dimensional array initializers, indexed properties, and dictionary initializers
- throw expressions
- Accessing static virtual or abstract interface members
- Lambda expressions that have attributes
- Interpolated strings
- UTF-8 string conversions or UTF-8 string literals
- Method invocations using variable arguments, named arguments or optional arguments
- Expressions using System.Index or System.Range, index "from end" (^) operator or range expressions (..)
- async lambda expressions or await expressions, including await foreach and await using
- Tuple literals, tuple conversions, tuple == or !=, or with expressions
- Discards (\_), deconstructing assignment, pattern matching is operator or the pattern matching switch expression
- COM call with ref omitted on the arguments
- ref, in or out parameters, ref return values, out arguments, or any values of ref struct type

# Executar árvores de expressão

Artigo • 16/03/2023

Uma *árvore de expressão* é uma estrutura de dados que representa algum código. Não se trata de código compilado nem executável. Se você quiser executar o código do .NET representado por uma árvore de expressão, precisará convertê-lo em instruções IL executáveis. Executar uma árvore de expressão pode retornar um valor ou apenas realizar uma ação, como chamar um método.

Somente árvores de expressão que representam expressões lambda podem ser executadas. Árvores de expressão que representam expressões lambda são do tipo [LambdaExpression](#) ou [Expression<TDelegate>](#). Para executar essas árvores de expressão, chame o método [Compile](#) para criar um delegado executável e, em seguida, invoque o delegado.

## ⓘ Observação

Se o tipo de delegado não for conhecido, ou seja, se a expressão lambda for do tipo [LambdaExpression](#) e não [Expression<TDelegate>](#), chame o método [DynamicInvoke](#) no delegado em vez de invocá-la diretamente.

Se uma árvore de expressão não representa uma expressão lambda, você pode criar uma expressão lambda que tenha a árvore de expressão original como corpo. Para isso, chame o método [Lambda<TDelegate>\(Expression, IEnumerable<ParameterExpression>\)](#). Em seguida, você pode executar a expressão lambda como descrito anteriormente nesta seção.

## Expressões lambda para funções

Você pode converter qualquer [LambdaExpression](#) ou qualquer tipo derivado de [LambdaExpression](#) em IL executável. Outros tipos de expressões não podem ser convertidos diretamente em código. Essa restrição tem pouco efeito na prática. As expressões lambda são os únicos tipos de expressões que você gostaria de executar convertendo em IL (linguagem intermediária) executável. (Pense no que significaria executar diretamente um [System.Linq.Expressions.ConstantExpression](#). Significaria algo útil?) Qualquer árvore de expressão, que seja um [System.Linq.Expressions.LambdaExpression](#) ou um tipo derivado de [LambdaExpression](#), pode ser convertida em IL. O tipo de expressão [System.Linq.Expressions.Expression<TDelegate>](#) é o único exemplo concreto nas

bibliotecas do .NET Core. Ele é usado para representar uma expressão que mapeia para qualquer tipo delegado. Como esse tipo mapeia para um tipo delegado, o .NET pode examinar a expressão e gerar a IL para um delegado apropriado que corresponda à assinatura da expressão lambda. O tipo delegado é baseado no tipo de expressão. Você deve conhecer o tipo de retorno e a lista de argumentos se quiser usar o objeto delegado de maneira fortemente tipada. O método `LambdaExpression.Compile()` retorna o tipo `Delegate`. Você precisará convertê-lo no tipo delegado correto para fazer com que as ferramentas de tempo de compilação verifiquem a lista de argumentos ou o tipo de retorno.

Na maioria dos casos, existe um mapeamento simples entre uma expressão e o delegado correspondente. Por exemplo, uma árvore de expressão representada por `Expression<Func<int>>` seria convertida em um delegado do tipo `Func<int>`. Para uma expressão lambda com qualquer tipo de retorno e lista de argumentos, existe um tipo delegado que é o tipo de destino para o código executável representado por essa expressão lambda.

O tipo `System.Linq.Expressions.LambdaExpression` contém membros `LambdaExpression.Compile` e `LambdaExpression.CompileToMethod` que você usaria para converter uma árvore de expressão em código executável. O método `Compile` cria um delegado. O método `CompileToMethod` atualiza um objeto `System.Reflection.Emit.MethodBuilder` com a IL que representa a saída compilada da árvore de expressão.

### ⓘ Importante

`CompileToMethod` só está disponível no .NET Framework, e não no .NET Core nem no .NET 5 e posterior.

Como opção, você também pode fornecer um `System.Runtime.CompilerServices.DebugInfoGenerator` que receberá as informações de depuração de símbolo para o objeto delegado gerado. O `DebugInfoGenerator` fornece informações completas de depuração sobre o delegado gerado.

Uma expressão seria convertida em um delegado usando o seguinte código:

C#

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

O exemplo de código a seguir demonstra os tipos concretos usados ao compilar e executar uma árvore de expressão.

C#

```
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

O exemplo de código a seguir demonstra como executar uma árvore de expressão que representa a elevação de um número a uma potência, criando uma expressão lambda e executando-a. O resultado, representado pelo número elevado à potência, é exibido.

C#

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2d),
                                         Expression.Constant(3d));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

## Execução e tempos de vida

O código é executado ao invocar o delegado que foi criado quando você chamou `LambdaExpression.Compile()`. O código anterior, `add.Compile()`, retorna um delegado. Você invoca esse delegado chamando `func()`, que executa o código.

Esse delegado representa o código na árvore de expressão. Você pode reter o identificador para esse delegado e invocá-lo mais tarde. Você não precisa compilar a árvore de expressão sempre que deseja executar o código que ela representa. (Lembre-se que as árvores de expressão são imutáveis e compilar a mesma árvore de expressão mais tarde, criará um delegado que executa o mesmo código.)

### ⊗ Cuidado

Não crie mecanismos de cache mais sofisticados para aumentar o desempenho evitando chamadas de compilação desnecessárias. Comparar duas árvores de expressão arbitrárias para determinar se elas representam o mesmo algoritmo é uma operação demorada. O tempo de computação que você economiza evitando chamadas extras a `LambdaExpression.Compile()` provavelmente é maior que o consumido pelo tempo de execução do código que determina se as duas árvores de expressão diferentes resultam no mesmo código executável.

## Advertências

A compilação de uma expressão lambda para um delegado e invocar esse delegado é uma das operações mais simples que você pode realizar com uma árvore de expressão. No entanto, mesmo com essa operação simples, há limitações que você deve estar ciente.

As expressões lambda criam fechamentos sobre todas as variáveis locais que são referenciadas na expressão. Você deve assegurar que todas as variáveis que farão parte do delegado são utilizáveis no local em que você chamar `Compile` e no momento em que você executar o delegado resultante. O compilador garante que as variáveis estejam no escopo. No entanto, se a sua expressão acessa uma variável que implementa `IDisposable`, é possível que o código descarte o objeto enquanto ele ainda é mantido pela árvore de expressão.

Por exemplo, esse código funciona bem, porque `int` não implementa `IDisposable`:

C#

```
private static Func<int, int> CreateBoundFunc()
{
```

```
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

O delegado capturou uma referência à variável local `constant`. Essa variável é acessada a qualquer momento mais tarde, quando a função retornada por `CreateBoundFunc` for executada.

No entanto, considere a seguinte classe (bastante artificial) que implementa `System.IDisposable`:

C#

```
public class Resource : IDisposable
{
    private bool _isDisposed = false;
    public int Argument
    {
        get
        {
            if (!_isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        _isDisposed = true;
    }
}
```

Se você a usar em uma expressão, conforme mostrado no seguinte código, obterá uma `System.ObjectDisposedException` ao executar o código referenciado pela propriedade `Resource.Argument`:

C#

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the
    expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument +
        b;
        var rVal = expression.Compile();
        return rVal;
}
```

```
    }  
}
```

O delegado retornado desse método fechou sobre o objeto `constant`, que foi descartado. (Foi descartado, porque foi declarado em uma instrução `using`).

Agora, ao executar o delegado retornado por esse método, uma `ObjectDisposedException` será gerada no ponto de execução.

Parece estranho ter um erro de runtime representando um constructo de tempo de compilação, mas é isso que ocorre quando trabalha com árvores de expressão.

Há várias permutações desse problema, portanto é difícil oferecer diretrizes gerais para evitá-lo. Tenha cuidado ao acessar variáveis locais quando define expressões e ao acessar o estado no objeto atual (representado por `this`) quando cria uma árvore de expressão retornada por meio de uma API pública.

O código na sua expressão pode referenciar métodos ou propriedades em outros assemblies. Esse assembly deve estar acessível quando a expressão for definida, quando ela for compilada e quando o delegado resultante for invocado. Você é recebido com um `ReferencedAssemblyNotFoundException` quando ele não está presente.

## Resumo

As árvores de expressão que representam expressões lambda podem ser compiladas para criar um delegado que pode ser executado. As árvores de expressão fornecem um mecanismo para executar o código representado por uma árvore de expressão.

A árvore de expressão representa o código que seria executado para qualquer constructo específico que você criar. Contanto que o ambiente em que você compilar e executar o código corresponda ao ambiente em que você criar a expressão, tudo funcionará conforme o esperado. Quando isso não acontece, os erros são previsíveis e capturados nos primeiros testes de qualquer código que usam as árvores de expressão.

# Mover árvores de expressão

Artigo • 16/03/2023

Neste artigo, você aprenderá a visitar cada nó em uma árvore de expressão enquanto estiver criando uma cópia modificada dessa árvore de expressão. Você converterá árvores de expressão para entender os algoritmos de modo que eles possam ser convertidos em outro ambiente. Você alterará o algoritmo criado. Você poderá adicionar registro em log, interceptar chamadas de método e monitorá-las ou realizar outras ações.

O código que você compila para mover uma árvore de expressão é uma extensão do que você já viu para visitar todos os nós em uma árvore. Quando você move uma árvore de expressão, visita todos os nós e ao visitá-los, cria a nova árvore. A nova árvore pode conter referências aos nós originais ou aos novos nós que você inseriu na árvore.

Vamos acessar uma árvore de expressão e criar uma árvore com alguns nós de substituição. Neste exemplo, substituiremos qualquer constante por uma constante dez vezes maior. Caso contrário, deixe a árvore de expressão intacta. Em vez de ler o valor da constante e substituí-la por uma nova constante, você fará essa substituição trocando o nó de constante por um novo nó que executa a multiplicação.

Aqui, quando encontrar um nó de constante, você criará um nó de multiplicação cujos filhos serão a constante original e a constante `10`:

C#

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

Crie uma árvore substituindo o nó original pelo substituto. Você verificará as alterações compilando e executando a árvore substituída.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

A criação de uma nova árvore é uma combinação da visita aos nós da árvore existentes e a criação de novos nós, inserindo-os na árvore. O anterior exemplo mostra a importância de as árvores de expressão serem imutáveis. Observe que a árvore criada no código anterior contém uma mistura de nós recém-criados e nós da árvore existente. Os nós podem ser usados em ambas as árvores porque os nós na árvore existente não podem ser modificados. Reutilizar nós resulta em eficiências significativas de memória. Os mesmos nós podem ser usados em toda a árvore ou em várias árvores de expressão. Como os nós não podem ser modificados, o mesmo nó pode ser reutilizado sempre que necessário.

## Percorrer e executar uma adição

Vamos verificar a nova árvore criando um segundo visitante que percorre a árvore de nós de adição e calcula o resultado. Faça algumas modificações no visitante que você viu até agora. Nessa nova versão, o visitante retorna a soma parcial da operação de adição até este ponto. Para uma expressão de constante, esse é simplesmente o valor da expressão de constante. Para uma expressão de adição, o resultado será a soma dos operandos esquerdos e direitos, uma vez que essas árvores forem percorridas.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three = Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so you can call it
// from itself recursively:
Func<Expression, int> aggregate = null!;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
```

```

aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
        (int)((ConstantExpression)exp).Value :
        aggregate(((BinaryExpression)exp).Left) +
        aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);

```

Tem bastante código nisso, mas os conceitos são acessíveis. Esse código visita filhos em uma pesquisa de profundidade inicial. Ao encontrar um nó constante, o visitante retorna o valor da constante. Depois de visitar ambos os filhos, o visitante terá a soma calculada para essa subárvore. Agora o nó de adição poderá computar sua soma. Uma vez que todos os nós da árvore de expressão forem visitados, a soma é calculada. Você pode executar o exemplo no depurador e rastrear a execução.

Vamos facilitar o rastreamento de como os nós são analisados e como a soma é calculada, percorrendo a árvore. Esta é uma versão atualizada do método de agregação que inclui bastante informação de rastreamento:

C#

```

private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        if (constantExp.Value is int value)
        {
            return value;
        }
        else
        {
            return 0;
        }
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
}

```

```
        else throw new NotSupportedException("Haven't written this yet");
    }
```

Executá-lo na expressão `sum` produz o seguinte resultado:

Saída

```
10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

Rastreie a saída e acompanhe no código anterior. Você será capaz de entender como o código visita cada nó e calcula a soma, à medida que percorre a árvore e localiza a soma.

Agora, vejamos uma execução diferente, com a expressão fornecida por `sum1`:

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

Aqui está a saída ao examinar essa expressão:

Saída

```
Found Addition Expression
Computing Left node
Found Constant: 1
```

```
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

Embora a resposta final seja a mesma, a forma de percorrer a árvore é diferente. Os nós são percorridos em uma ordem diferente, porque a árvore foi construída com operações diferentes que ocorrem primeiro.

## Criar uma cópia modificada

Crie um novo projeto de **Aplicativo de Console**. Adicione uma diretiva `using` ao arquivo para o namespace `System.Linq.Expressions`. Adicione a classe `AndAlsoModifier` ao seu projeto.

C#

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an
            // AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right,

```

```

        b.IsLiftedToNull, b.Method);
    }

    return base.VisitBinary(b);
}
}

```

Essa classe herda a classe [ExpressionVisitor](#) e é especializada para modificar expressões que representam operações `AND` condicionais. Ela muda essas operações de uma `AND` condicional para uma `OR` condicional. A classe substitui o método [VisitBinary](#) do tipo base, pois as expressões condicionais `AND` são representadas como expressões binárias. No método [VisitBinary](#), se a expressão que é passada a ele representa uma operação `AND` condicional, o código cria uma nova expressão que contém o operador `OR` condicional em vez do operador `AND` condicional. Se a expressão passada para o [VisitBinary](#) não representa uma operação `AND` condicional, o método adia para a implementação da classe base. Os métodos da classe base constroem nós semelhantes às árvores de expressão passadas, mas as subárvores dos nós são substituídas pelas árvores de expressão produzidas de maneira recorrente pelo visitante.

Adicione uma diretiva `using` ao arquivo para o namespace `System.Linq.Expressions`. Adicione código ao método `Main` no arquivo `Program.cs` para criar uma árvore de expressão e passá-la ao método que a modifica.

C#

```

Expression<Func<string, bool>> expr = name => name.Length > 10 &&
name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression)expr);

Console.WriteLine(modifiedExpr);

/* This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/

```

O código cria uma expressão que contém uma operação `AND` condicional. Em seguida, ele cria uma instância da classe `AndAlsoModifier` e passa a expressão ao método `Modify` dessa classe. A árvore de expressão original e a modificada são geradas para mostrar a alteração. Compile e execute o aplicativo.

## Saiba mais

Este exemplo mostra um pequeno subconjunto do código que você compilaria para percorrer e interpretar os algoritmos representados por uma árvore de expressão. Para obter informações sobre como criar uma biblioteca de uso geral que converte árvores de expressão em outra linguagem, leia [esta série](#) de Matt Warren. Ele entra em detalhes de como mover qualquer código que você pode encontrar em uma árvore de expressão.

Espero que você tenha visto o verdadeiro potencial das árvores de expressão. Você examina um conjunto de códigos, faz as alterações que deseja nesse código e executa a versão modificada. Como as árvores de expressão são imutáveis, você cria árvores usando os componentes de árvores existentes. A reutilização de nós minimiza a quantidade de memória necessária para criar árvores de expressão modificadas.

# Sintaxe do DebugView

Artigo • 16/03/2023

A propriedade **DebugView** (disponível apenas durante a depuração) fornece uma renderização de cadeia de caracteres de árvores de expressão. A maior parte da sintaxe é bastante simples de entender; os casos especiais são descritos nas seções a seguir.

Cada exemplo é seguido por um comentário de bloco, que contém **DebugView**.

## ParameterExpression

[ParameterExpression](#) os nomes de variáveis são exibidos com o símbolo `$` no início.

Se um parâmetro não tiver nome, receberá um nome gerado automaticamente, como `$var1` ou `$var2`.

```
C#  
  
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");  
/*  
 $num  
 */  
  
ParameterExpression numParam = Expression.Parameter(typeof(int));  
/*  
 $var1  
 */
```

## ConstantExpression

Para objetos [ConstantExpression](#) que representam valores inteiros, cadeias de caracteres e `null`, o valor da constante é exibido.

Para tipos numéricos que têm sufixos padrão como literais de C#, o sufixo é adicionado ao valor. A tabela a seguir mostra os sufixos associados com vários tipos numéricos.

Tipo	Palavra-chave	Sufixo
<a href="#">System.UInt32</a>	<code>uint</code>	<code>U</code>
<a href="#">System.Int64</a>	<code>longo</code>	<code>L</code>
<a href="#">System.UInt64</a>	<code>ulong</code>	<code>UL</code>

Tipos	Palavra-chave	Sufixo
System.Double	double	D
System.Single	float	F
System.Decimal	decimal	M

C#

```
int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/
```

## BlockExpression

Se o tipo de um objeto [BlockExpression](#) difere do tipo da última expressão no bloco, o tipo será exibido entre colchetes angulares (< e >). Caso contrário, o tipo do objeto [BlockExpression](#) não é exibido.

C#

```
BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block = Expression.Block(typeof(Object),
Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/
```

## LambdaExpression

Objetos [LambdaExpression](#) são exibidos junto com seus tipos delegados.

Se uma expressão lambda não tiver nome, receberá um nome gerado automaticamente, como `#Lambda1` ou `#Lambda2`.

C#

```
LambdaExpression lambda = Expression.Lambda<Func<int>>()
(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/
LambdaExpression lambda = Expression.Lambda<Func<int>>()
(Expression.Constant(1), "SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/
```

## LabelExpression

Se você especificar um valor padrão para o objeto [LabelExpression](#), esse valor será exibido antes do objeto [LabelTarget](#).

O token `.Label` indica o início do rótulo. O token `.LabelTarget` indica o local para o qual o destino deve saltar.

Se um rótulo não tiver nome, receberá um nome gerado automaticamente, como

`#Label1` ou `#Label2`.

C#

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
            -1
        .LabelTarget SampleLabel:
    }
*/
```

```
*/  
  
LabelTarget target = Expression.Label();  
BlockExpression block = Expression.Block(  
    Expression.Goto(target),  
    Expression.Label(target)  
);  
/*  
    .Block() {  
        .Goto #Label1 { };  
        .Label  
        .LabelTarget #Label1:  
    }  
*/
```

## Operadores verificados

Os operadores verificados são exibidos com o símbolo `#` na frente do operador. Por exemplo, o operador de adição verificado é exibido como `#+`.

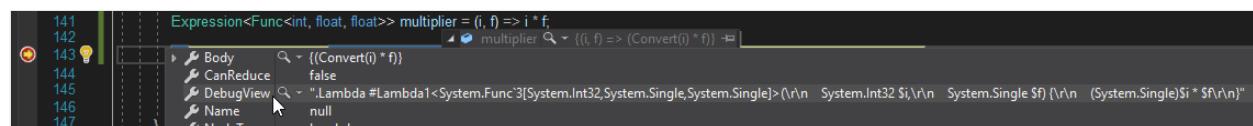
C#

```
Expression expr = Expression.AddChecked( Expression.Constant(1),  
    Expression.Constant(2));  
/*  
    1 #+ 2  
*/  
  
Expression expr = Expression.ConvertChecked( Expression.Constant(10.0),  
    typeof(int));  
/*  
    #(System.Int32)10D  
*/
```

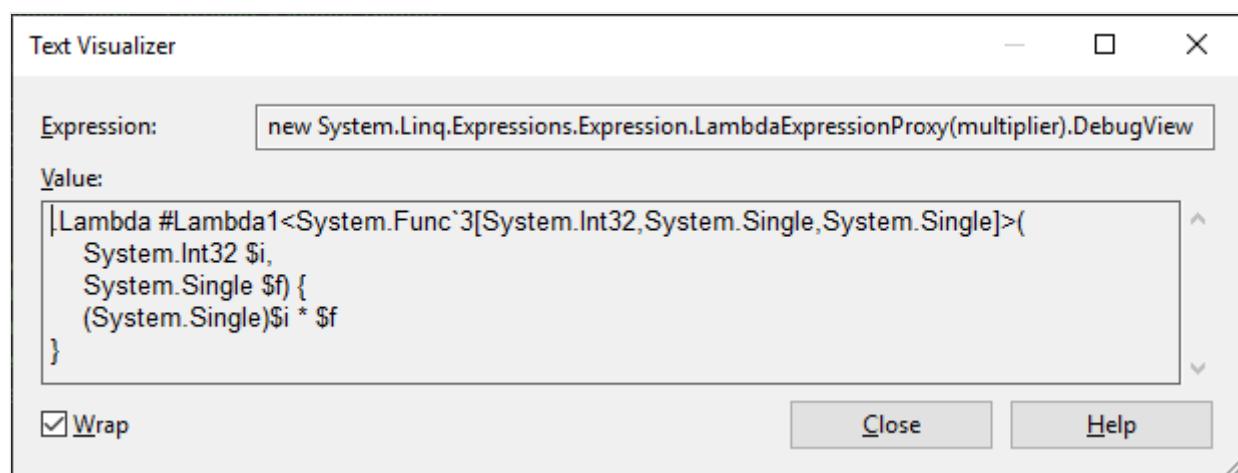
# Depurando árvores de expressão no Visual Studio

Artigo • 16/03/2023

Ao depurar seus aplicativos, você pode analisar a estrutura e o conteúdo das árvores de expressão. Para obter uma visão geral da estrutura de árvore de expressão, você pode usar a propriedade `DebugView`, que representa as árvores de expressão [usando uma sintaxe especial](#). `DebugView` está disponível apenas no modo de depuração.

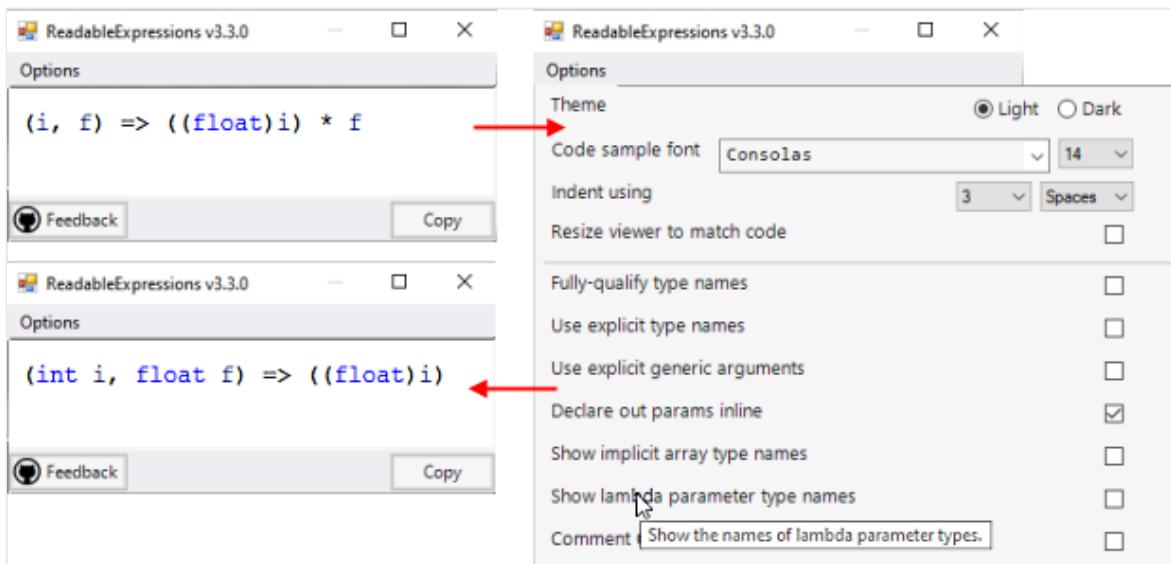


Uma vez que `DebugView` é uma cadeia de caracteres, você pode usar o [Visualizador de Texto interno](#) para exibi-lo em várias linhas, selecionando **Visualizador de Texto** do ícone de lupa ao lado do rótulo `DebugView`.

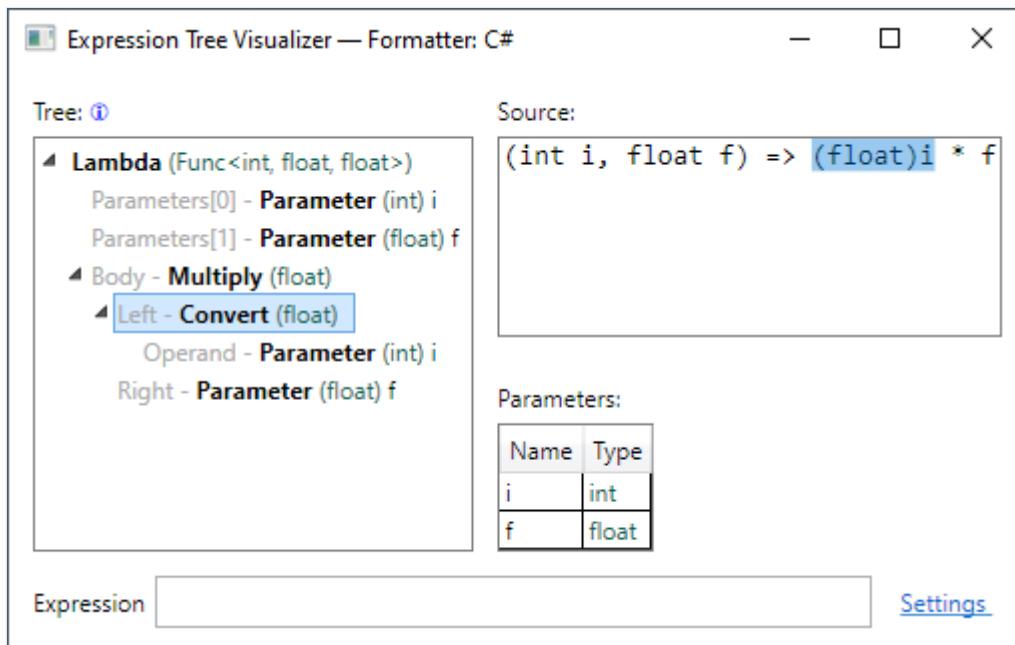


Como alternativa, você pode instalar e usar [um visualizador personalizado para árvores de expressão](#), como:

- As [Expressões Legíveis](#) ([licença do MIT](#), disponível no [Visual Studio Marketplace](#)) renderizam a árvore de expressão como código C# com tema, com várias opções de renderização:

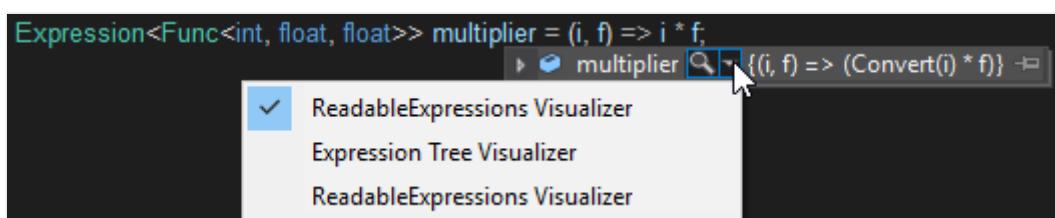


- O Visualizador de Árvore de Expressão (licença MIT) fornece um modo de exibição de árvore da árvore de expressão e dos nós individuais dela:



## Abrir um visualizador para uma árvore de expressão

Selecione o ícone de lupa que aparece ao lado da árvore de expressão em DataTips, uma janela Inspeção, a janela Autos ou a janela Locais. É exibida uma lista dos visualizadores disponíveis:



Selecione o visualizador que você deseja usar.

## Confira também

- [Depurando no Visual Studio](#)
- [Criar visualizadores personalizados](#)
- [DebugView sintaxe](#)

# Como realizar consultas com base no estado do runtime (C#)

Artigo • 16/03/2023

## ⓘ Observação

Adicione `using System.Linq.Expressions;` e `using static System.Linq.Expressions.Expression;` na parte superior do seu arquivo .cs.

Considere o código que define um `IQueryable` ou um `IQueryable<T>` em relação a uma fonte de dados:

C#

```
var companyNames = new[] {
    "Consolidated Messenger", "Alpine Ski House", "Southridge Video",
    "City Power & Light", "Coho Winery", "Wide World Importers",
    "Graphic Design Institute", "Adventure Works", "Humongous Insurance",
    "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
    "Blue Yonder Airlines", "Trey Research", "The Phone Company",
    "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee"
};

// We're using an in-memory array as the data source, but the IQueryable
// could have come
// from anywhere -- an ORM backed by a database, a web request, or any other
// LINQ provider.
IQueryable<string> companyNamesSource = companyNames.AsQueryable();
var fixedQry = companyNames.OrderBy(x => x);
```

Toda vez que você executar esse código, exatamente a mesma consulta será executada. Isso, geralmente, não é muito útil, pois você pode querer que o código execute consultas diferentes com base nas condições em tempo de execução. Este artigo descreve como você pode executar uma consulta diferente com base no estado de runtime.

## IQueryable/IQueryable<T> e árvores de expressão

Fundamentalmente, uma `IQueryable` tem dois componentes:

- **Expression** – Uma representação (agnóstica quanto à linguagem e à fonte de dados) dos componentes da consulta atual, na forma de uma árvore de expressão.
- **Provider** – Uma instância de um provedor LINQ, que sabe como materializar a consulta atual em um valor ou conjunto de valores.

No contexto de consulta dinâmica, o provedor geralmente permanece o mesmo, ao passo que a árvore de expressão da consulta varia de uma consulta para a outra.

As árvores de expressão são imutáveis; se você quiser uma árvore de expressão diferente e, portanto, uma consulta diferente, precisará traduzir a árvore de expressão existente para uma nova e, portanto, para um novo [IQueryable](#).

As seguintes seções descrevem técnicas específicas para executar consultas diferentes em resposta ao estado de runtime:

- Usar o estado de runtime de dentro da árvore de expressão
- Chamar métodos LINQ adicionais
- Variar a árvore de expressão passada para os métodos LINQ
- Construir uma árvore de expressão [Expression<TDelegate>](#) usando os métodos de fábrica em [Expression](#)
- Adicionar nós de chamada de método a uma árvore de expressão de [IQueryable](#)
- Construir cadeias de caracteres e usar a [Biblioteca dinâmica do LINQ ↗](#)

## Usar o estado de runtime de dentro da árvore de expressão

Supondo que o provedor LINQ dê suporte a isso, a maneira mais simples de executar consultas dinamicamente é referenciar o estado de runtime diretamente na consulta por meio de uma variável fechada, como `length` no seguinte exemplo de código:

C#

```
var length = 1;
var qry = companyNamesSource
    .Select(x => x.Substring(0, length))
    .Distinct();

Console.WriteLine(string.Join(", ", qry));
// prints: C, A, S, W, G, H, M, N, B, T, L, F

length = 2;
Console.WriteLine(string.Join(", ", qry));
// prints: Co, Al, So, Ci, Wi, Gr, Ad, Hu, Wo, Ma, No, Bl, Tr, Th, Lu, Fo
```

A árvore de expressão interna e, portanto, a consulta, não foi modificada; a consulta retorna valores diferentes apenas porque o valor de `length` foi alterado.

## Chamar métodos LINQ adicionais

Em geral, os [métodos LINQ internos](#) em `Queryable` executam duas etapas:

- Empacote a árvore de expressão atual em uma `MethodCallExpression` representando a chamada de método.
- Passar a árvore de expressão encapsulada de volta para o provedor, seja para retornar um valor por meio do método `IQueryProvider.Execute` do provedor, ou para retornar um objeto de consulta traduzido por meio do método `IQueryProvider.CreateQuery`.

Você pode substituir a consulta original pelo resultado do método `IQueryable<T>-returning` a fim de obter uma nova consulta. Você pode fazer isso condicionalmente, com base no estado de runtime, como no seguinte exemplo:

```
C#  
  
// bool sortByLength = /* ... */;  
  
var qry = companyNamesSource;  
if (sortByLength)  
{  
    qry = qry.OrderBy(x => x.Length);  
}
```

## Variar a árvore de expressão passada para os métodos LINQ

Você pode passar expressões diferentes para os métodos LINQ, dependendo do estado de runtime:

```
C#  
  
// string? startsWith = /* ... */;  
// string? endsWith = /* ... */;  
  
Expression<Func<string, bool>> expr = (startsWith, endsWith) switch  
{  
    ("", "" or null, "" or null) => x => true,  
    (_, "" or null) => x => x.StartsWith(startsWith),  
    ("", "" or null, _) => x => x.EndsWith(endsWith),
```

```
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);
```

Talvez você também queira compor as diversas subexpressões usando uma biblioteca de terceiros, como o [PredicateBuilder](#) da [LinqKit](#):

C#

```
// This is functionally equivalent to the previous example.

// using LinqKit;
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>>? expr = PredicateBuilder.New<string>(false);
var original = expr;
if (!string.IsNullOrEmpty(startsWith))
{
    expr = expr.Or(x => x.StartsWith(startsWith));
}
if (!string.IsNullOrEmpty(endsWith))
{
    expr = expr.Or(x => x.EndsWith(endsWith));
}
if (expr == original)
{
    expr = x => true;
}

var qry = companyNamesSource.Where(expr);
```

## Construir árvores de expressão e consultas usando métodos de fábrica

Em todos os exemplos até este ponto, conhecemos, no tempo de compilação, o tipo de elemento `string`, portanto, o tipo da consulta `IQueryable<string>`. Talvez seja necessário adicionar componentes a uma consulta de qualquer tipo de elemento ou adicionar componentes diferentes, dependendo do tipo de elemento. Você pode criar árvores de expressão do zero, usando os métodos de fábrica em `System.Linq.Expressions.Expression`, personalizando assim a expressão em tempo de execução para um tipo de elemento específico.

### Construindo um `Expression<TDelegate>`

Quando você constrói uma expressão para passar para um dos métodos LINQ, na verdade você está criando uma instância de `Expression<TDelegate>`, em que `TDelegate` é algum tipo de delegado, como `Func<string, bool>`, `Action` ou um tipo de delegado personalizado.

`Expression<TDelegate>` herda de `LambdaExpression`, que representa uma expressão lambda completa como a seguinte:

C#

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

Uma `LambdaExpression` tem dois componentes:

- Uma lista de parâmetros – `(string x)` – representada pela propriedade `Parameters`.
- Um corpo – `x.StartsWith("a")` – representado pela propriedade `Body`.

As etapas básicas para construção de um `Expression<TDelegate>` são as seguintes:

- Defina objetos `ParameterExpression` para cada um dos parâmetros (se houver) na expressão lambda, usando o método de fábrica `Parameter`.

C#

```
ParameterExpression x = Expression.Parameter(typeof(string), "x");
```

- Construa o corpo do seu `LambdaExpression`, usando os `ParameterExpression` que você definiu e os métodos de fábrica em `Expression`. Por exemplo, uma expressão que representa `x.StartsWith("a")` poderia ser construída assim:

C#

```
Expression body = Call(
    x,
    typeof(string).GetMethod("StartsWith", new[] { typeof(string) })!,
    Constant("a")
);
```

- Encapsule os parâmetros e o corpo em um `Expression<TDelegate>` do tipo tempo de compilação, usando a sobrecarga do método de fábrica `Lambda` apropriada:

C#

```
Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body, x);
```

As seções a seguir descrevem um cenário no qual talvez você queira construir um `Expression<TDelegate>` para passar para um método LINQ e fornecer um exemplo completo de como fazer isso usando os métodos de fábrica.

## Cenário

Digamos que você tenha vários tipos de entidade:

C#

```
record Person(string LastName, string FirstName, DateTime DateOfBirth);  
record Car(string Model, int Year);
```

Para qualquer um desses tipos de entidade, você deve filtrar e retornar somente as entidades que têm determinado texto dentro de um dos campos `string` delas. Para `Person`, você deve pesquisar as propriedades `FirstName` e `LastName`:

C#

```
string term = /* ... */;  
var personsQry = new List<Person>()  
    .AsQueryable()  
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

Já para `Car`, você deve pesquisar apenas a propriedade `Model`:

C#

```
string term = /* ... */;  
var carsQry = new List<Car>()  
    .AsQueryable()  
    .Where(x => x.Model.Contains(term));
```

Embora você possa gravar uma função personalizada para `IQueryable<Person>` e outra para `IQueryable<Car>`, a função a seguir adiciona essa filtragem a qualquer consulta existente, independentemente do tipo do elemento.

## Exemplo

C#

```
// using static System.Linq.Expressions.Expression;

IQueryable<T> TextFilter<T>(IQueryable<T> source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    // T is a compile-time placeholder for the element type of the query.
    Type elementType = typeof(T);

    // Get all the string properties on this specific type.
    PropertyInfo[] stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Get the right overload of String.Contains
    MethodInfo containsMethod = typeof(string).GetMethod("Contains", new[] {
        typeof(string) })!;

    // Create a parameter for the expression tree:
    // the 'x' in 'x => x.PropertyName.Contains("term")'
    // The type of this parameter is the query's element type
    ParameterExpression prm = Parameter(elementType);

    // Map each property to an expression tree node
    IEnumerable<Expression> expressions = stringProperties
        .Select(prp =>
            // For each property, we have to construct an expression tree
            // node like x.PropertyName.Contains("term")
            Call(
                Property(prm, prp),
                containsMethod,
                Constant(term)
            )
        );
}

// Combine all the resultant expression nodes using ||
Expression body = expressions
    .Aggregate(
        (prev, current) => Or(prev, current)
    );

// Wrap the expression body in a compile-time-typed lambda expression
Expression<Func<T, bool>> lambda = Lambda<Func<T, bool>>(body, prm);

// Because the lambda is compile-time-typed (albeit with a generic
// parameter), we can use it with the Where method
```

```
    return source.Where(lambda);
}
```

Como a função `TextFilter` usa e retorna um `IQueryable<T>` (e não apenas um `IQueryable`), você pode adicionar mais elementos de consulta do tipo tempo de compilação após o filtro de texto.

C#

```
var qry = TextFilter(
    new List<Person>().AsQueryable(),
    "abcd"
)
.Where(x => x.DateOfBirth < new DateTime(2001, 1, 1));

var qry1 = TextFilter(
    new List<Car>().AsQueryable(),
    "abcd"
)
.Where(x => x.Year == 2010);
```

## Adicionar nós de chamada de método à árvore de expressão de `IQueryable`

Se você tiver um `IQueryable` em vez de um `IQueryable<T>`, não poderá chamar diretamente os métodos LINQ genéricos. Uma alternativa é criar a árvore de expressão interna, conforme acima, e usar a reflexão para invocar o método LINQ apropriado e passar como parâmetro a árvore de expressão.

Você também pode duplicar a funcionalidade do método LINQ, encapsulando toda a árvore em um `MethodCallExpression` que representa uma chamada ao método LINQ:

C#

```
IQueryable TextFilter_Untyped(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
    Type elementType = source.ElementType;

    // The logic for building the ParameterExpression and the
    // LambdaExpression's body is the same as in the previous example,
    // but has been refactored into the constructBody function.
    (Expression? body, ParameterExpression? prm) =
    constructBody(elementType, term);
    if (body is null) {return source;}

    Expression filteredTree = Call(
```

```

        typeof(Queryable),
        "Where",
        new[] { elementType},
        source.Expression,
        Lambda(body, prm!)
    );

    return source.Provider.CreateQuery(filteredTree);
}

```

Nesse caso, você não tem um espaço reservado genérico `T` em tempo de compilação, portanto, você usará a sobrecarga `Lambda` que não requer informações do tipo tempo de compilação e que produz um `LambdaExpression` em vez de um `Expression<TDelegate>`.

## Biblioteca Dinâmica do LINQ

A construção de árvores de expressão usando métodos de fábrica é relativamente complexa; é mais fácil compor cadeias de caracteres. A [Biblioteca Dinâmica do LINQ](#) expõe um conjunto de métodos de extensão em `IQueryable`, correspondentes aos métodos LINQ padrão em `Queryable` e que aceitam cadeias de caracteres em uma [sintaxe especial](#) em vez de árvores de expressão. A biblioteca gera a árvore de expressão apropriada com base na cadeia de caracteres e pode retornar o `IQueryable` traduzido resultante.

Por exemplo, o exemplo anterior pode ser reescrito da seguinte maneira:

C#

```

// using System.Linq.Dynamic.Core

IQueryable TextFilter_Strings(IQueryable source, string term) {
    if (string.IsNullOrEmpty(term)) { return source; }

    var elementType = source.ElementType;

    // Get all the string property names on this specific type.
    var stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Build the string expression
    string filterExpr = string.Join(
        " || ",
        stringProperties.Select(prp => $"{prp.Name}.Contains(@0)")
    );

```

```
    return source.Where(filterExpr, term);  
}
```

## Confira também

- Executar árvores de expressão
- Especificar filtros de predicado dinamicamente em tempo de execução

# Iteradores (C#)

Artigo • 09/05/2023

Um *iterador* pode ser usado para percorrer coleções, como listas e matrizes.

Um método iterador ou um acessador `get` realiza uma iteração personalizada em uma coleção. Um método iterador usa a instrução `yield return` para retornar um elemento de cada vez. Quando uma instrução `yield return` for atingida, o local atual no código será lembrado. A execução será reiniciada desse local na próxima vez que a função iteradora for chamada.

Um iterador é consumido no código cliente, usando uma instrução `foreach` ou usando uma consulta LINQ.

No exemplo a seguir, a primeira iteração do loop `foreach` faz que a execução continue no método iterador `SomeNumbers` até que a primeira instrução `yield return` seja alcançada. Essa iteração retorna um valor de 3 e o local atual no método iterador é mantido. Na próxima iteração do loop, a execução no método iterador continuará de onde parou, parando novamente quando alcançar uma instrução `yield return`. Essa iteração retorna um valor de 5 e o local atual no método iterador é mantido novamente. O loop terminará quando o final do método iterador for alcançado.

C#

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.WriteLine(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

O tipo de retorno de um método iterador ou acessador `get` pode ser `IEnumerable`, `IEnumerable<T>`, `IEnumerator` ou `IEnumerator<T>`.

Você pode usar uma instrução `yield break` para terminar a iteração.

## (!) Observação

Todos os exemplos neste tópico, exceto o exemplo Iterador Simples, incluem diretivas `using` para os namespaces `System.Collections` e `System.Collections.Generic`.

## Iterador simples

O exemplo a seguir contém uma única instrução `yield return` que está dentro de um loop `for`. Em `Main`, cada iteração do corpo da instrução `foreach` cria uma chamada à função iteradora, que avança para a próxima instrução `yield return`.

C#

```
static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.WriteLine(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
    EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

## Criando uma classe de coleção

No exemplo a seguir, a classe `DaysOfTheWeek` implementa a interface `IEnumerable`, que requer um método `GetEnumerator`. O compilador chama implicitamente o método `GetEnumerator`, que retorna um `IEnumerator`.

O método `GetEnumerator` retorna cada cadeia de caracteres, uma de cada vez, usando a instrução `yield return`.

```
C#  
  
static void Main()  
{  
    DaysOfTheWeek days = new DaysOfTheWeek();  
  
    foreach (string day in days)  
    {  
        Console.Write(day + " ");  
    }  
    // Output: Sun Mon Tue Wed Thu Fri Sat  
    Console.ReadKey();  
}  
  
public class DaysOfTheWeek : IEnumerable  
{  
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",  
    "Sat" };  
  
    public IEnumerator GetEnumerator()  
    {  
        for (int index = 0; index < days.Length; index++)  
        {  
            // Yield each day of the week.  
            yield return days[index];  
        }  
    }  
}
```

O exemplo a seguir cria uma classe `Zoo` que contém uma coleção de animais.

A instrução `foreach`, que faz referência à instância de classe (`theZoo`), chama implicitamente o método `GetEnumerator`. As instruções `foreach`, que fazem referência às propriedades `Birds` e `Mammals`, usam o método iterador nomeado `AnimalsForType`.

```
C#  
  
static void Main()  
{  
    Zoo theZoo = new Zoo();  
  
    theZoo.AddMammal("Whale");  
    theZoo.AddMammal("Rhinoceros");  
    theZoo.AddBird("Penguin");  
    theZoo.AddBird("Warbler");  
  
    foreach (string name in theZoo)  
    {
```

```

        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler

    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros

    Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal });
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird });
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }

    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds

```

```

    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }

    // Private methods.
    private IEnumerable AnimalsForType(Animal.TypeEnum type)
    {
        foreach (Animal theAnimal in animals)
        {
            if (theAnimal.Type == type)
            {
                yield return theAnimal.Name;
            }
        }
    }

    // Private class.
    private class Animal
    {
        public enum TypeEnum { Bird, Mammal }

        public string Name { get; set; }
        public TypeEnum Type { get; set; }
    }
}

```

## Usando iteradores com uma lista genérica

No exemplo a seguir, a classe `Stack<T>` genérica implementa a interface genérica `IEnumerable<T>`. O método `Push` atribui valores a uma matriz do tipo `T`. O método `GetEnumerator` retorna os valores da matriz usando a instrução `yield return`.

Além do método `GetEnumerator` genérico, o método `GetEnumerator` não genérico também deve ser implementado. Isso ocorre porque `IEnumerable<T>` herda de `IEnumerable`. A implementação não genérica adia a implementação genérica.

O exemplo usa iteradores nomeados para dar suporte a várias maneiras de iterar na mesma coleção de dados. Esses iteradores nomeados são as propriedades `TopToBottom` e `BottomToTop` e o método `TopN`.

A propriedade `BottomToTop` usa um iterador em um acessador `get`.

C#

```

static void Main()
{
    Stack<int> theStack = new Stack<int>();

```

```

// Add items to the stack.
for (int number = 0; number <= 9; number++)
{
    theStack.Push(number);
}

// Retrieve items from the stack.
// foreach is allowed because theStack implements IEnumerable<int>.
foreach (int number in theStack)
{
    Console.Write("{0} ", number);
}
Console.WriteLine();
// Output: 9 8 7 6 5 4 3 2 1 0

// foreach is allowed, because theStack.TopToBottom returns
IEnumerable<Of Integer>.
foreach (int number in theStack.TopToBottom)
{
    Console.Write("{0} ", number);
}
Console.WriteLine();
// Output: 9 8 7 6 5 4 3 2 1 0

foreach (int number in theStack.BottomToTop)
{
    Console.Write("{0} ", number);
}
Console.WriteLine();
// Output: 0 1 2 3 4 5 6 7 8 9

foreach (int number in theStack.TopN(7))
{
    Console.Write("{0} ", number);
}
Console.WriteLine();
// Output: 9 8 7 6 5 4 3

Console.ReadKey();
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }
}

```

```

}

// This method implements the GetEnumerator method. It allows
// an instance of the class to be used in a foreach statement.
public IEnumarator<T> GetEnumerator()
{
    for (int index = top - 1; index >= 0; index--)
    {
        yield return values[index];
    }
}

IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}

public IEnumerable<T> TopToBottom
{
    get { return this; }
}

public IEnumerable<T> BottomToTop
{
    get
    {
        for (int index = 0; index <= top - 1; index++)
        {
            yield return values[index];
        }
    }
}

public IEnumerable<T> TopN(int itemsFromTop)
{
    // Return less than itemsFromTop if necessary.
    int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;

    for (int index = top - 1; index >= startIndex; index--)
    {
        yield return values[index];
    }
}
}

```

## Informações de sintaxe

Um iterador pode ocorrer como um método ou como um acessador `get`. Um iterador não pode ocorrer em um evento, um construtor de instância, um construtor estático ou um finalizador estático.

Deve existir uma conversão implícita do tipo de expressão na instrução `yield return`, para o argumento de tipo do `IEnumerable<T>` retornado pelo iterador.

Em C#, um método iterador não pode ter os parâmetros `in`, `ref` nem `out`.

No C#, `yield` não é uma palavra reservada e só terá significado especial quando for usada antes das palavras-chave `return` ou `break`.

## Implementação Técnica

Embora você escreva um iterador como um método, o compilador o traduz em uma classe aninhada que é, na verdade, uma máquina de estado. Essa classe mantém o controle da posição do iterador enquanto o loop `foreach` no código cliente continuar.

Para ver o que o compilador faz, você pode usar a ferramenta `Ildasm.exe` para exibir o código Microsoft Intermediate Language que é gerado para um método iterador.

Quando você cria um iterador para uma [classe](#) ou [struct](#), não é necessário implementar toda a interface [IEnumerator](#). Quando o compilador detecta o iterador, ele gera automaticamente os métodos `Current`, `MoveNext` e `Dispose` da interface [IEnumerator](#) ou [IEnumerator<T>](#).

A cada iteração sucessiva do loop `foreach` (ou a chamada direta ao `IEnumerator.MoveNext`), o próximo corpo de código do iterador continua, depois da instrução `yield return` anterior. Em seguida, ele continuará até a próxima instrução `yield return`, até que o final do corpo do iterador seja alcançado ou até que uma instrução `yield break` seja encontrada.

Iteradores não dão suporte ao método `IEnumerator.Reset`. Para iterar novamente desde o início, você deve obter um novo iterador. Chamar `Reset` no iterador retornado por um método iterador lança um [NotSupportedException](#).

Para obter informações adicionais, consulte a [Especificação da linguagem C#](#).

## Uso de iteradores

Os iteradores permitem que você mantenha a simplicidade de um loop `foreach` quando for necessário usar um código complexo para preencher uma sequência de lista. Isso pode ser útil quando você quiser fazer o seguinte:

- Modificar a sequência de lista após a primeira iteração de loop `foreach`.

- Evitar o carregamento completo de uma grande lista antes da primeira iteração de um loop `foreach`. Um exemplo é uma busca paginada para carregar um lote de linhas da tabela. Outro exemplo é o método [EnumerateFiles](#), que implementa os iteradores no .NET.
- Encapsular a criação da lista no iterador. No método iterador, você pode criar a lista e, em seguida, gerar cada resultado em um loop.

## Confira também

- [System.Collections.Generic](#)
- [IEnumerable<T>](#)
- [foreach, in](#)
- [Usar foreach com matrizes](#)
- [Genéricos](#)

# LINQ (Consulta Integrada à Linguagem)

Artigo • 15/02/2023

O LINQ (consulta integrada à linguagem) é o nome de um conjunto de tecnologias com base na integração de recursos de consulta diretamente na linguagem C#.

Tradicionalmente, consultas feitas em dados são expressas como cadeias de caracteres simples sem verificação de tipo no tempo de compilação ou suporte a IntelliSense. Além disso, você precisará aprender uma linguagem de consulta diferente para cada tipo de fonte de dados: bancos de dados SQL, documentos XML, vários serviços Web etc. Com o LINQ, uma consulta é um constructo de linguagem de primeira classe, como classes, métodos, eventos.

Para um desenvolvedor que escreve consultas, a parte mais visível "integrada à linguagem" do LINQ é a expressão de consulta. As expressões de consulta são uma *sintaxe declarativa de consulta*. Usando a sintaxe de consulta, você pode executar operações de filtragem, ordenação e agrupamento em fontes de dados com o mínimo de código. Você pode usar os mesmos padrões de expressão de consulta básica para consultar e transformar dados em bancos de dados SQL, conjuntos de dados do ADO.NET, documentos XML e fluxos e coleções .NET.

O exemplo a seguir mostra a operação de consulta completa. A operação completa inclui a criação de uma fonte de dados, definição da expressão de consulta e execução da consulta em uma instrução `foreach`.

```
C#  
  
// Specify the data source.  
int[] scores = { 97, 92, 81, 60 };  
  
// Define the query expression.  
IQueryable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;  
  
// Execute the query.  
foreach (int i in scoreQuery)  
{  
    Console.Write(i + " ");  
}  
  
// Output: 97 92 81
```

# Visão geral da expressão de consulta

- Expressões de consulta podem ser usadas para consultar e transformar dados de qualquer fonte de dados habilitada para LINQ. Por exemplo, uma única consulta pode recuperar dados de um Banco de Dados SQL e produzir um fluxo XML como saída.
- As expressões de consulta são fáceis de entender porque elas usam muitos constructos de linguagem C# familiares.
- As variáveis em uma expressão de consulta são fortemente tipadas, embora em muitos casos você não precise fornecer o tipo explicitamente, pois o compilador pode inferir nele. Para obter mais informações, consulte [Relacionamentos de tipo em operações de consulta LINQ](#).
- Uma consulta não é executada até que você itere sobre a variável de consulta, por exemplo, em uma instrução `foreach`. Para obter mais informações, consulte [Introdução a consultas LINQ](#).
- No tempo de compilação, as expressões de consulta são convertidas em chamadas de método do operador de consulta padrão de acordo com as regras definidas na especificação do C#. Qualquer consulta que pode ser expressa usando sintaxe de consulta também pode ser expressa usando sintaxe de método. No entanto, na maioria dos casos, a sintaxe de consulta é mais legível e concisa. Para obter mais informações, consulte [Especificação da linguagem C#](#) e [Visão geral de operadores de consulta padrão](#).
- Como uma regra ao escrever consultas LINQ, recomendamos que você use a sintaxe de consulta sempre que possível e a sintaxe de método sempre que necessário. Não há semântica ou diferença de desempenho entre as duas formas. As expressões de consulta são geralmente mais legíveis do que as expressões equivalentes escritas na sintaxe de método.
- Algumas operações de consulta, como [Count](#) ou [Max](#), não apresentam cláusulas de expressão de consulta equivalentes e, portanto, devem ser expressas como chamadas de método. A sintaxe de método pode ser combinada com a sintaxe de consulta de várias maneiras. Para obter mais informações, consulte [Sintaxe de consulta e sintaxe de método em LINQ](#).
- As expressões de consulta podem ser compiladas para árvores de expressão ou delegados, dependendo do tipo ao qual a consulta é aplicada. As consultas `IEnumerable<T>` são compiladas para representantes. As consultas `IQueryable` e

`IQueryable<T>` são compiladas para árvores de expressão. Para obter mais informações, consulte [Árvores de expressão](#).

## Próximas etapas

Para obter mais detalhes sobre o LINQ, comece se familiarizando com alguns conceitos básicos em [Noções básicas sobre expressões de consulta](#), e, em seguida, leia a documentação para a tecnologia LINQ na qual você está interessado:

- Documentos XML: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to Entities](#)
- Coleções do .NET, arquivos, cadeias de caracteres, etc.: [LINQ to Objects](#)

Para saber mais sobre o LINQ, consulte [LINQ em C#](#).

Para começar a trabalhar com o LINQ em C#, consulte o tutorial [Trabalhando com LINQ](#).

# Introdução a consultas LINQ (C#)

Artigo • 10/05/2023

Uma *consulta* é uma expressão que recupera dados de uma fonte de dados. As consultas normalmente são expressas em uma linguagem de consulta especializada. Diferentes linguagens foram desenvolvidas ao longo do tempo para os diversos tipos de fontes de dados, por exemplo, SQL para bancos de dados relacionais e o XQuery para XML. Portanto, os desenvolvedores precisaram aprender uma nova linguagem de consulta para cada tipo de fonte de dados ou formato de dados que eles tinham que oferecer suporte. O LINQ simplifica essa situação ao oferecer um modelo consistente para trabalhar com os dados em vários tipos de fontes e formatos de dados. Em uma consulta LINQ, você está sempre trabalhando com objetos. Você usa os mesmos padrões básicos de codificação para consultar e transformar dados em documentos XML, bancos de dados SQL, conjuntos de dados do ADO.NET, coleções do .NET e qualquer outro formato para o qual um provedor LINQ estiver disponível.

## Três Partes de uma Operação de Consulta

Todos as operações de consulta LINQ consistem em três ações distintas:

1. Obter a fonte de dados.
2. Criar a consulta.
3. Executar a consulta.

O exemplo a seguir mostra como as três partes de uma operação de consulta são expressas em código-fonte. O exemplo usa uma matriz de inteiros como uma fonte de dados para sua conveniência. No entanto, os mesmos conceitos também se aplicam a outras fontes de dados. Faremos referência a este exemplo todo o restante deste tópico.

C#

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
```

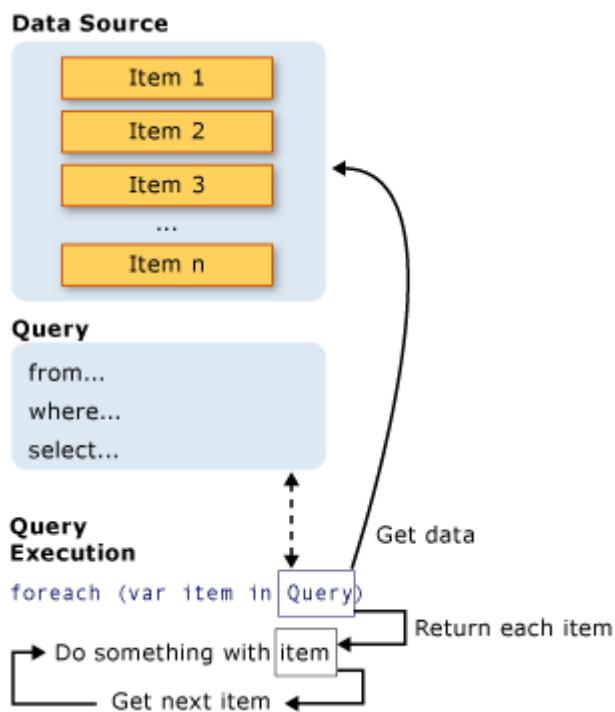
```

        from num in numbers
        where (num % 2) == 0
        select num;

    // 3. Query execution.
    foreach (int num in numQuery)
    {
        Console.WriteLine("{0,1} ", num);
    }
}

```

A ilustração a seguir mostra a operação de consulta completa. No LINQ, a execução da consulta é distinta da própria consulta. Em outras palavras, você não recupera nenhum dado apenas criando uma variável de consulta.



## A Fonte de Dados

No exemplo anterior, como a fonte de dados é uma matriz, ela dá suporte à interface genérica `IEnumerable<T>` de forma implícita. Isso significa que ela pode ser consultada com o LINQ. Uma consulta é executada em uma instrução `foreach`, e `foreach` requer `IEnumerable` ou `IEnumerable<T>`. Tipos que dão suporte a `IEnumerable<T>` ou uma interface derivada, como a genérica `IQueryable<T>`, são chamados *tipos passíveis de consulta*.

Um tipo passível de consulta não exige modificação ou tratamento especial para servir como uma fonte de dados do LINQ. Se os dados de origem ainda não estiverem na memória como um tipo passível de consulta, o provedor LINQ deverá representá-los

como tal. Por exemplo, LINQ to XML carrega um documento XML em um tipo `XElement` passível de consulta:

C#

```
// Create a data source from an XML document.  
// using System.Xml.Linq;  
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

Com o LINQ to SQL, primeiro você cria um mapeamento relacional de objeto em tempo de design, manualmente ou usando as [Ferramentas LINQ to SQL no Visual Studio](#). Você escreve suas consultas aos objetos e o LINQ to SQL manipula a comunicação com o banco de dados em tempo de execução. No exemplo a seguir, `Customers` representa uma tabela específica no banco de dados, e o tipo do resultado da consulta, `IQueryable<T>`, deriva de `IEnumerable<T>`.

C#

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");  
  
// Query for customers in London.  
IQueryable<Customer> custQuery =  
    from cust in db.Customers  
    where cust.City == "London"  
    select cust;
```

Para obter mais informações sobre como criar tipos específicos de fontes de dados, consulte a documentação para os diversos provedores LINQ. No entanto, a regra básica é muito simples: uma fonte de dados do LINQ é qualquer objeto que dá suporte à interface genérica `IEnumerable<T>` ou uma interface que herda dela.

#### ⓘ Observação

Tipos como `ArrayList`, que dão suporte à interface `IEnumerable` não genérica, também podem ser usados como uma fonte de dados LINQ. Para obter mais informações, consulte [Como consultar um ArrayList com LINQ \(C#\)](#).

## A consulta

A consulta especifica quais informações devem ser recuperadas da fonte (ou fontes) de dados. Opcionalmente, uma consulta também especifica como essas informações devem ser classificadas, agrupadas e moldadas antes de serem retornadas. Uma

consulta é armazenada em uma variável de consulta e é inicializada com uma expressão de consulta. Para tornar mais fácil escrever consultas, o C# introduziu uma nova sintaxe de consulta.

A consulta no exemplo anterior retorna todos os números pares da matriz de inteiros. A expressão de consulta contém três cláusulas: `from`, `where` e `select`. (Se você estiver familiarizado com o SQL, terá notado que a ordenação das cláusulas é o reverso da ordem no SQL.) A cláusula `from` especifica a fonte de dados, a cláusula `where` aplica o filtro e a cláusula `select` especifica o tipo dos elementos retornados. Essas e as outras cláusulas de consulta são discutidas em detalhes na seção [Consulta integrada à linguagem \(LINQ\)](#). Por enquanto, o ponto importante é que no LINQ a variável de consulta não faz nada e não retorna nenhum dado. Ele apenas armazena as informações necessárias para produzir os resultados quando a consulta for executada em um momento posterior. Para obter mais informações sobre como as consultas são construídas nos bastidores, consulte [Visão geral de operadores de consulta padrão \(C#\)](#).

#### ① Observação

As consultas também podem ser expressas usando a sintaxe de método. Para obter mais informações, consulte [Sintaxe de consulta e sintaxe de método em LINQ](#).

## Execução da consulta

### Execução Adiada

Conforme mencionado anteriormente, a variável de consulta armazena somente os comandos da consulta. A execução real da consulta é adiada até que você itere sobre a variável de consulta em uma instrução `foreach`. Esse conceito é conhecido como *execução adiada* e é demonstrado no exemplo a seguir:

C#

```
// Query execution.  
foreach (int num in numQuery)  
{  
    Console.Write("{0,1} ", num);  
}
```

A instrução `foreach` também é o local em que os resultados da consulta são recuperados. Por exemplo, na consulta anterior, a variável de iteração `num` armazena

cada valor (um de cada vez) na sequência retornada.

Como a própria variável de consulta nunca armazena os resultados da consulta, você poderá executá-la quantas vezes desejar. Por exemplo, você pode ter um banco de dados que está sendo atualizado continuamente por um aplicativo separado. Em seu aplicativo, você poderia criar uma consulta que recupera os dados mais recentes e poderia executá-la repetidamente em algum intervalo para recuperar resultados diferentes a cada vez.

## Forçando Execução Imediata

As consultas que realizam funções de agregação em um intervalo de elementos de origem devem primeiro iterar sobre esses elementos. Exemplos dessas consultas são `Count`, `Max`, `Average` e `First`. Essas consultas são executadas sem uma instrução `foreach` explícita porque a consulta em si deve usar `foreach` para retornar um resultado. Observe também que esses tipos de consultas retornam um valor único e não uma coleção `IEnumerable`. A consulta a seguir retorna uma contagem de números pares na matriz de origem:

C#

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

Para forçar a execução imediata de qualquer consulta e armazenar seus resultados em cache, você pode chamar os métodos `ToList` ou `ToArray`.

C#

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

Você também pode forçar a execução colocando o loop `foreach` imediatamente após a expressão de consulta. No entanto, ao chamar `ToList` ou `ToArray`, você também armazena em cache todos os dados em um único objeto de coleção.

## Confira também

- [Introdução a LINQ em C#](#)
- [Passo a passo: escrevendo consultas em C#](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [foreach, in](#)
- [Palavras-chave de Consulta \(LINQ\)](#)

# Relacionamentos de tipo em operações de consulta LINQ (C#)

Artigo • 20/07/2023

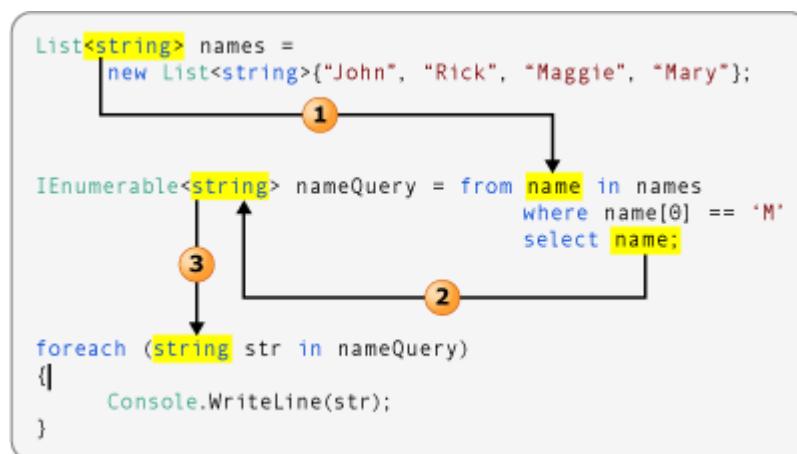
Para escrever consultas com eficiência, você precisa entender como os tipos de variáveis em uma operação de consulta completa se relacionam entre si. Se você compreender esses relacionamentos, entenderá com mais facilidade os exemplos da LINQ e as amostras de código na documentação. Além disso, você compreenderá o que ocorre nos bastidores quando variáveis são tipadas de forma implícita usando `var`.

As operações de consulta LINQ são fortemente tipadas na fonte de dados, na própria consulta e na execução da consulta. O tipo das variáveis na consulta deve ser compatível com o tipo dos elementos na fonte de dados e com o tipo da variável de iteração na instrução `foreach`. Essa tipagem forte garante que erros de tipo sejam capturados em tempo de compilação, quando podem ser corrigidos antes que os usuários os encontrem.

Para demonstrar essas relações de tipo, a maioria dos exemplos a seguir usam tipagem explícita para todas as variáveis. O último exemplo mostra como os mesmos princípios se aplicam mesmo quando você usa tipagem implícita usando `var`.

## Consultas que não transformam os dados de origem

A ilustração a seguir mostra uma operação de consulta LINQ to Objects que não executa transformações nos dados. A fonte contém uma sequência de cadeias de caracteres e a saída da consulta também é uma sequência de cadeias de caracteres.



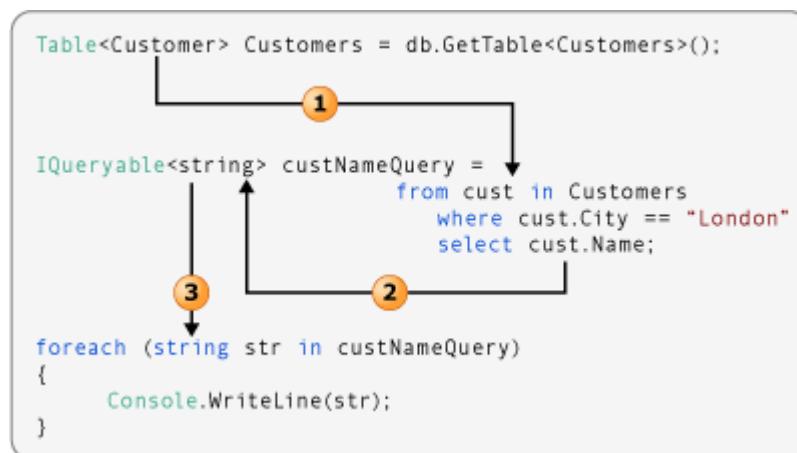
1. O argumento de tipo da fonte de dados determina o tipo da variável de intervalo.

2. O tipo do objeto selecionado determina o tipo da variável de consulta. Esta é uma cadeia de caracteres `name`. Portanto, a variável de consulta é um `IEnumerable<string>`.

3. A variável de consulta é iterada na instrução `foreach`. Como a variável de consulta é uma sequência de cadeias de caracteres, a variável de iteração também é uma cadeia de caracteres.

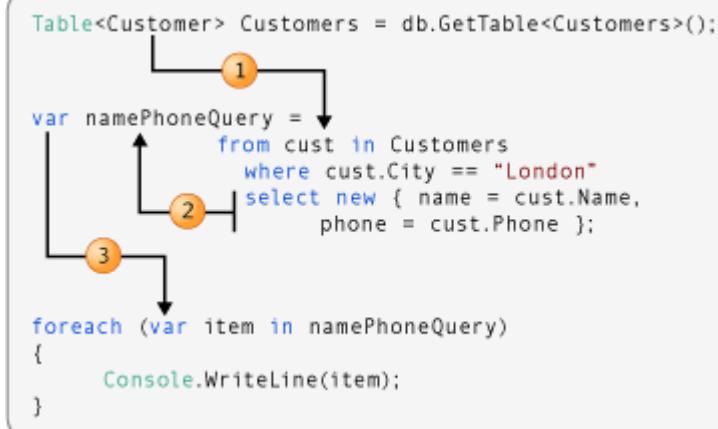
## Consultas que transformam os dados de origem

A ilustração a seguir mostra uma operação de consulta LINQ to SQL que executa uma transformação simples nos dados. A consulta usa uma sequência de objetos `Customer` como entrada e seleciona somente a propriedade `Name` no resultado. Como `Name` é uma cadeia de caracteres, a consulta produz uma sequência de cadeias de caracteres como saída.



1. O argumento de tipo da fonte de dados determina o tipo da variável de intervalo.
2. A instrução `select` retorna a propriedade `Name` em vez do objeto `Customer` completo. Como `Name` é uma cadeia de caracteres, o argumento de tipo de `custNameQuery` é `string` e não `Customer`.
3. Como `custNameQuery` é uma sequência de cadeias de caracteres, a variável de iteração do loop `foreach` também deve ser um `string`.

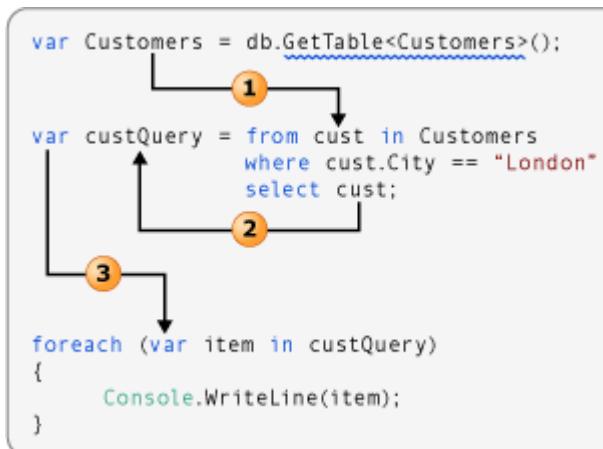
A ilustração a seguir mostra uma transformação um pouco mais complexa. A instrução `select` retorna um tipo anônimo que captura apenas dois membros do objeto `Customer` original.



1. O argumento de tipo da fonte de dados sempre é o tipo da variável de intervalo na consulta.
2. Como a instrução `select` produz um tipo anônimo, a variável de consulta deve ser tipada implicitamente usando `var`.
3. Como o tipo da variável de consulta é implícito, a variável de iteração no loop `foreach` também deve ser implícito.

## Deixando o compilador inferir informações de tipo

Embora você precise entender as relações de tipo em uma operação de consulta, você tem a opção de permitir que o compilador fazer todo o trabalho. A palavra-chave `var` pode ser usada para qualquer variável local em uma operação de consulta. A ilustração a seguir é semelhante ao exemplo número 2 que foi discutido anteriormente. No entanto, o compilador fornece o tipo forte para cada variável na operação de consulta.



Para obter mais informações sobre `var`, consulte [Variáveis de local digitadas implicitamente](#).

# LINQ e tipos genéricos (C#)

As consultas LINQ são baseadas em tipos genéricos. Não é necessário um conhecimento profundo sobre os genéricos antes de começar a escrever consultas. No entanto, convém entender dois conceitos básicos:

1. Quando você cria uma instância de uma classe de coleção genérica, como `List<T>`, substitua o "T" pelo tipo dos objetos que a lista bloqueia. Por exemplo, uma lista de cadeias de caracteres é expressa como `List<string>` e uma lista de objetos `Customer` é expressa como `List<Customer>`. Uma lista genérica é fortemente tipada e oferece muitos benefícios em coleções que armazenam seus elementos como `Object`. Se tentar adicionar um `Customer` em uma `List<string>`, você obterá um erro em tempo de compilação. É fácil usar coleções genéricas, porque você não precisa realizar a conversão de tipo em tempo de execução.
2. A `IEnumerable<T>` é a interface que permite que as classes de coleção genérica sejam enumeradas usando a instrução `foreach`. Classes de coleção genéricas dão suporte a `IEnumerable<T>` do mesmo modo que classes de coleção não genéricas, tais como `ArrayList`, dão suporte a `IEnumerable`.

Para obter mais informações sobre os genéricos, consulte [Genéricos](#).

## Variáveis `IEnumerable<T>` em consultas LINQ

As variáveis de consulta LINQ são do tipo `IEnumerable<T>` ou de um tipo derivado, como `IQueryable<T>`. Ao se deparar com uma variável de consulta que é tipada como `IEnumerable<Customer>`, significa apenas que a consulta, quando for executada, produzirá uma sequência de zero ou mais objetos `Customer`.

C#

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

Para obter mais informações, consulte [Relacionamentos de tipo em operações de consulta LINQ](#).

# Permitir que o compilador manipule as declarações de tipo genérico

Se preferir, poderá evitar a sintaxe genérica, usando a palavra-chave `var`. A palavra-chave `var` instrui o compilador a inferir o tipo de uma variável de consulta, examinando a fonte de dados especificada na cláusula `from`. O exemplo a seguir produz o mesmo código compilado que o exemplo anterior:

C#

```
var customerQuery2 =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach(var customer in customerQuery2)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
```

A palavra-chave `var` é útil quando o tipo da variável for óbvio ou quando não é tão importante especificar explicitamente os tipos genéricos aninhados, como aqueles que são produzidos por consultas de grupo. É recomendável que você note que o código poderá se tornar mais difícil de ser lido por outras pessoas, caso você use a `var`. Para obter mais informações, consulte [Variáveis locais de tipo implícito](#).

# Visão geral de operadores de consulta padrão (C#)

Artigo • 20/07/2023

Os *operadores de consulta padrão* são os métodos que formam o padrão LINQ. A maioria desses métodos opera em sequências; neste contexto, uma sequência é um objeto cujo tipo implementa a interface `IEnumerable<T>` ou a interface `IQueryable<T>`. Os operadores de consulta padrão fornecem recursos de consulta incluindo filtragem, projeção, agregação, classificação e muito mais.

Há dois conjuntos de operadores de consulta padrão do LINQ: um que opera em objetos do tipo `IEnumerable<T>`, outro que opera em objetos do tipo `IQueryable<T>`. Os métodos que compõem a cada conjunto são os membros estáticos das classes `Enumerable` e `Queryable`, respectivamente. Eles são definidos como *métodos de extensão* do tipo nos quais operam. Os métodos de extensão podem ser chamados usando a sintaxe do método estático ou a sintaxe do método de instância.

Além disso, vários métodos de operador de consulta padrão operam em tipos diferentes daqueles baseados em `IEnumerable<T>` ou `IQueryable<T>`. O tipo `Enumerable` define dois métodos tais que ambos operam em objetos do tipo `IEnumerable`. Esses métodos, `Cast<TResult>(IEnumerable)` e `OfType<TResult>(IEnumerable)`, permitem que você habilite uma coleção sem parâmetros ou não genérica, a ser consultada no padrão LINQ. Eles fazem isso criando uma coleção de objetos fortemente tipada. A classe `Queryable` define dois métodos semelhantes, `Cast<TResult>(IQueryable)` e `OfType<TResult>(IQueryable)`, que operam em objetos do tipo `IQueryable`.

Os operadores de consulta padrão são diferentes no momento de sua execução, dependendo de se eles retornam um valor singleton ou uma sequência de valores. Esses métodos que retornam um valor singleton (por exemplo, `Average` e `Sum`) são executados imediatamente. Os métodos que retornam uma sequência adiam a execução da consulta e retornam um objeto enumerável.

Para métodos que operam em coleções na memória, ou seja, os métodos que estendem `IEnumerable<T>`, o objeto `Enumerable` retornado captura os argumentos que foram passados para o método. Quando esse objeto é enumerado, a lógica do operador de consulta é empregada e os resultados da consulta são retornados.

Por outro lado, os métodos que estendem `IQueryable<T>` não implementam nenhum comportamento de consulta. Eles criam uma árvore de expressão que representa a

consulta a ser executada. O processamento de consulta é tratado pelo objeto [IQueryable<T>](#) de origem.

Chamadas para métodos de consulta podem ser encadeadas em uma consulta, o que permite que consultas se tornem arbitrariamente complexas.

O exemplo de código a seguir demonstra como os operadores de consulta padrão podem ser usados para obter informações sobre uma sequência.

C#

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

## Sintaxe de expressão de consulta

Alguns dos operadores de consulta padrão mais usados têm uma sintaxe de palavra-chave de linguagem C# e Visual Basic dedicada que possibilita que eles sejam chamados como parte de uma *expressão de consulta*. Para obter mais informações sobre operadores de consulta padrão que têm palavras-chave dedicadas e suas sintaxes correspondentes, consulte [Sintaxe de expressão de consulta para operadores de consulta padrão \(C#\)](#).

## Estendendo os operadores de consulta padrão

Você pode aumentar o conjunto de operadores de consulta padrão criando métodos específicos de domínio apropriados para o domínio ou tecnologia de destino. Você também pode substituir os operadores de consulta padrão por suas próprias implementações que fornecem serviços adicionais, como avaliação remota, conversão de consulta e otimização. Para ver um exemplo, consulte [AsEnumerable](#).

## Obter uma fonte de dados

Em uma consulta LINQ, a primeira etapa é especificar a fonte de dados. No C#, como na maioria das linguagens de programação, uma variável deve ser declarada antes que possa ser usada. Em um consulta LINQ, a cláusula `from` vem primeiro para introduzir a fonte de dados (`customers`) e a *variável de intervalo* (`cust`).

C#

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

A variável de intervalo é como a variável de iteração em um loop `foreach`, mas nenhuma iteração real ocorre em uma expressão de consulta. Quando a consulta é executada, a variável de intervalo servirá como uma referência para cada elemento sucessivo em `customers`. Uma vez que o compilador pode inferir o tipo de `cust`, você não precisa especificá-lo explicitamente. Variáveis de intervalo adicionais podem ser introduzidas por uma cláusula `let`. Para obter mais informações, consulte [Cláusula let](#).

### Observação

Para fontes de dados não genéricas, como `ArrayList`, a variável de intervalo deve ser tipada explicitamente. Para obter mais informações, consulte [Como consultar um ArrayList com LINQ \(C#\)](#) e a cláusula `from`.

# Filtragem

Provavelmente, a operação de consulta mais comum é aplicar um filtro no formulário de uma expressão booliana. O filtro faz com que a consulta retorne apenas os elementos para os quais a expressão é verdadeira. O resultado é produzido usando a cláusula `where`. O filtro em vigor especifica os elementos a serem excluídos da sequência de origem. No exemplo a seguir, somente os `customers` que têm um endereço em Londres são retornados.

C#

```
var queryLondonCustomers = from cust in customers
                            where cust.City == "London"
                            select cust;
```

Você pode usar os operadores lógicos `AND` e `OR` de C# para aplicar quantas expressões de filtro forem necessárias na cláusula `where`. Por exemplo, para retornar somente clientes de "Londres" `AND` cujo nome seja "Devon", você escreveria o seguinte código:

C#

```
where cust.City == "London" && cust.Name == "Devon"
```

Para retornar clientes de Londres ou Paris, você escreveria o seguinte código:

C#

```
where cust.City == "London" || cust.City == "Paris"
```

Para obter mais informações, consulte [Cláusula where](#).

# Ordenando

Muitas vezes é conveniente classificar os dados retornados. A cláusula `orderby` fará com que os elementos na sequência retornada sejam classificados de acordo com o comparador padrão para o tipo que está sendo classificado. Por exemplo, a consulta a seguir pode ser estendida para classificar os resultados com base na propriedade `Name`. Como `Name` é uma cadeia de caracteres, o comparador padrão executa uma classificação em ordem alfabética de A a Z.

C#

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

Para ordenar os resultados na ordem inversa, de Z para a, use a cláusula `orderby...descending`.

Para obter mais informações, consulte [Cláusula orderby](#).

## Agrupamento

A cláusula `group` permite agrupar seus resultados com base em uma chave que você especificar. Por exemplo, você pode especificar que os resultados sejam agrupados segundo o `city`, de modo que todos os clientes de Londres ou Paris fiquem em grupos individuais. Nesse caso, `cust.City` é a chave.

C#

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
var queryCustomersByCity =
    from cust in customers
    group cust by cust.City;

// customerGroup is an IGrouping<string, Customer>
foreach (var customerGroup in queryCustomersByCity)
{
    Console.WriteLine(customerGroup.Key);
    foreach (Customer customer in customerGroup)
    {
        Console.WriteLine("    {0}", customer.Name);
    }
}
```

Quando você terminar uma consulta com um cláusula `group`, seus resultados assumirão a forma de uma lista de listas. Cada elemento na lista é um objeto que tem um membro `Key` e uma lista dos elementos que estão agrupados sob essa chave. Quando itera em uma consulta que produz uma sequência de grupos, você deve usar um loop `foreach` aninhado. O loop externo itera em cada grupo e o loop interno itera nos membros de cada grupo.

Se precisar consultar os resultados de uma operação de grupo, você poderá usar a palavra-chave `into` para criar um identificador que pode ser consultado ainda mais. A

consulta a seguir retorna apenas os grupos que contêm mais de dois clientes:

C#

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

Para obter mais informações, consulte [Cláusula group](#).

## Adição

Operações de junção criam associações entre sequências que não são modeladas explicitamente nas fontes de dados. Por exemplo, você pode executar uma junção para localizar todos os clientes e distribuidores que têm o mesmo local. No LINQ, a cláusula `join` sempre funciona com coleções de objetos em vez de tabelas de banco de dados diretamente.

C#

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

No LINQ, você não precisa usar `join` com a mesma frequência que o faz no SQL, porque as chaves estrangeiras no LINQ são representados no modelo do objeto como propriedades que mantêm uma coleção de itens. Por exemplo, um objeto `Customer` que contém uma coleção de objetos `Order`. Em vez de executar uma junção, você pode acessar os pedidos usando notação de ponto:

C#

```
from order in Customer.Orders...
```

Para obter mais informações, consulte [Cláusula join](#).

## Selecionando (Projeções)

A cláusula `select` produz os resultados da consulta e especifica a "forma" ou o tipo de cada elemento retornado. Por exemplo, você pode especificar se os resultados consistirão em objetos `Customer` completos, apenas um membro, um subconjunto de membros ou algum tipo de resultado completamente diferente com base em um cálculo ou na criação de um novo objeto. Quando a cláusula `select` produz algo diferente de uma cópia do elemento de origem, a operação é chamada de *projeção*. O uso de projeções para transformar dados é um recurso poderoso de expressões de consulta LINQ. Para obter mais informações, consulte [Transformações de dados com LINQ \(C#\)](#) e [Cláusula select](#).

## Tabela de sintaxe de expressão de consulta

A tabela a seguir lista os operadores de consulta padrão que têm cláusulas de expressão de consulta equivalentes.

Método	Sintaxe de expressão de consulta em C#
<a href="#">Cast</a>	Use uma variável de intervalo de tipo explícito, por exemplo:  <code>from int i in numbers</code>  (Para obter mais informações, consulte <a href="#">Cláusula from</a> .)
<a href="#">GroupBy</a>	<code>group ... by</code>  -ou-  <code>group ... by ... into ...</code>  (Para obter mais informações, consulte <a href="#">Cláusula group</a> .)
<a href="#">GroupJoin&lt;TOuter,TInner,TKey,TResult&gt;(IEnumerable&lt;TOuter&gt;, IEnumerable&lt;TInner&gt;, Func&lt;TOuter,TKey&gt;, Func&lt;TInner,TKey&gt;, Func&lt;TOuter,IEnumerable&lt;TInner&gt;, TResult&gt;)</a>	<code>join ... in ... on ... equals ... into ...</code>  (Para obter mais informações, consulte <a href="#">Cláusula join</a> .)

Método	Sintaxe de expressão de consulta em C#
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	<pre>join ... in ... on ... equals ...</pre> <p>(Para obter mais informações, consulte <a href="#">Cláusula join</a>.)</p>
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<pre>orderby</pre> <p>(Para obter mais informações, consulte <a href="#">Cláusula orderby</a>.)</p>
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<pre>orderby ... descending</pre> <p>(Para obter mais informações, consulte <a href="#">Cláusula orderby</a>.)</p>
Select	<pre>select</pre> <p>(Para obter mais informações, consulte <a href="#">Cláusula select</a>.)</p>
SelectMany	<p>Várias cláusulas <code>from</code>.</p> <p>(Para obter mais informações, consulte <a href="#">Cláusula from</a>.)</p>
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<pre>orderby ..., ...</pre> <p>(Para obter mais informações, consulte <a href="#">Cláusula orderby</a>.)</p>
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<pre>orderby ..., ...</pre> <pre>descending</pre> <p>(Para obter mais informações, consulte <a href="#">Cláusula orderby</a>.)</p>
Where	<pre>where</pre> <p>(Para obter mais</p>

Método	Sintaxe de expressão de consulta em C#
	informações, consulte <a href="#">Cláusula where.</a> )

## Confira também

- [Enumerable](#)
- [Queryable](#)
- [Métodos de Extensão](#)
- [Palavras-chave de Consulta \(LINQ\)](#)
- [Tipos anônimos](#)

# Transformações de dados com LINQ

## (C#)

Artigo • 09/05/2023

A consulta integrada à linguagem (LINQ) não se trata apenas de recuperar dados. Também é uma ferramenta poderosa para transformação de dados. Ao usar uma consulta LINQ, você pode usar uma sequência de origem como entrada e modificá-la de várias maneiras para criar uma nova sequência de saída. Você pode modificar a própria sequência sem modificar os respectivos elementos, classificando-os e agrupando-os. Mas talvez o recurso mais poderoso das consultas LINQ é a capacidade de criar novos tipos. Isso é feito na cláusula `select`. Por exemplo, é possível executar as seguintes tarefas:

- Mesclar várias sequências de entrada em uma única sequência de saída que tenha um novo tipo.
- Criar sequências de saída cujos elementos consistem em apenas uma ou várias propriedades de cada elemento da sequência de origem.
- Criar sequências de saída cujos elementos consistem nos resultados das operações realizadas nos dados de origem.
- Criar sequências de saída em um formato diferente. Por exemplo, você pode transformar dados de linhas do SQL ou de arquivos de texto em XML.

Esses são apenas alguns exemplos. É claro que essas transformações podem ser combinadas de diversas maneiras na mesma consulta. Além disso, a sequência de saída de uma consulta pode ser usada como a sequência de entrada de uma nova consulta.

## Ingressando Várias Entradas em uma Única Sequência de Saída

Você pode usar uma consulta LINQ para criar uma sequência de saída que contenha elementos de mais de uma sequência de entrada. O exemplo a seguir mostra como combinar duas estruturas de dados na memória, mas os mesmos princípios podem ser aplicados para combinar dados de origens de XML, SQL ou DataSet. Considere os dois tipos de classe a seguir:

C#

```

class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public List<int> Scores;
}

class Teacher
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string City { get; set; }
}

```

O exemplo a seguir mostra a consulta:

C#

```

class DataTransformations
{
    static void Main()
    {
        // Create the first data source.
        List<Student> students = new List<Student>()
        {
            new Student { First="Svetlana",
                          Last="Omelchenko",
                          ID=111,
                          Street="123 Main Street",
                          City="Seattle",
                          Scores= new List<int> { 97, 92, 81, 60 } },
            new Student { First="Claire",
                          Last="O'Donnell",
                          ID=112,
                          Street="124 Main Street",
                          City="Redmond",
                          Scores= new List<int> { 75, 84, 91, 39 } },
            new Student { First="Sven",
                          Last="Mortensen",
                          ID=113,
                          Street="125 Main Street",
                          City="Lake City",
                          Scores= new List<int> { 88, 94, 65, 91 } },
        };
        // Create the second data source.
        List<Teacher> teachers = new List<Teacher>()
        {

```

```

        new Teacher { First="Ann", Last="Beebe", ID=945, City="Seattle"
    },
        new Teacher { First="Alex", Last="Robinson", ID=956,
City="Redmond" },
        new Teacher { First="Michiyo", Last="Sato", ID=972,
City="Tacoma" }
    };

    // Create the query.
var peopleInSeattle = (from student in students
    where student.City == "Seattle"
    select student.Last)
.Concat(from teacher in teachers
    where teacher.City == "Seattle"
    select teacher.Last);

Console.WriteLine("The following students and teachers live in
Seattle:");
// Execute the query.
foreach (var person in peopleInSeattle)
{
    Console.WriteLine(person);
}

Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}
/* Output:
 The following students and teachers live in Seattle:
 Omelchenko
 Beebe
 */

```

Para obter mais informações, consulte [cláusula join](#) e [cláusula select](#).

## Selecionando um Subconjunto de Cada Elemento de Origem

Há duas maneiras principais de selecionar um subconjunto de cada elemento na sequência de origem:

1. Para selecionar apenas um membro do elemento de origem, use a operação de ponto. No exemplo a seguir, suponha que um objeto `Customer` contém várias propriedades públicas, incluindo uma cadeia de caracteres denominada `City`. Quando executada, essa consulta produzirá uma sequência de saída de cadeias de caracteres.

C#

```
var query = from cust in Customers  
            select cust.City;
```

2. Para criar elementos que contenham mais de uma propriedade do elemento de origem, você pode usar um inicializador de objeto com um objeto nomeado ou um tipo anônimo. O exemplo a seguir mostra o uso de um tipo anônimo para encapsular duas propriedades de cada elemento `Customer`:

C#

```
var query = from cust in Customer  
            select new {Name = cust.Name, City = cust.City};
```

Para obter mais informações, consulte [Inicializadores de coleção e de objeto](#) e [Tipos anônimos](#).

## Transformando Objetos na Memória em XML

As consultas do LINQ facilitam a transformação de dados entre estruturas de dados na memória, bancos de dados SQL, conjuntos de dados ADO.NET e fluxos ou documentos XML. O exemplo a seguir transforma objetos de uma estrutura de dados na memória em elementos XML.

C#

```
class XMLTransform  
{  
    static void Main()  
    {  
        // Create the data source by using a collection initializer.  
        // The Student class was defined previously in this topic.  
        List<Student> students = new List<Student>()  
        {  
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores  
= new List<int>{97, 92, 81, 60}},  
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores =  
new List<int>{75, 84, 91, 39}},  
            new Student {First="Sven", Last="Mortensen", ID=113, Scores =  
new List<int>{88, 94, 65, 91}},  
        };  
  
        // Create the query.  
        var studentsToXML = new XElement("Root",  
            from student in students  
            let scores = string.Join(", ", student.Scores)
```

```

        select new XElement("student",
            new XElement("First", student.First),
            new XElement("Last", student.Last),
            new XElement("Scores", scores)
        ) // end "student"
    ); // end "Root"

    // Execute the query.
    Console.WriteLine(studentsToXML);

    // Keep the console open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}

```

O código produz a seguinte saída XML:

XML

```

<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,92,81,60</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>75,84,91,39</Scores>
  </student>
  <student>
    <First>Sven</First>
    <Last>Mortensen</Last>
    <Scores>88,94,65,91</Scores>
  </student>
</Root>

```

Para obter mais informações, consulte [Criando árvores XML em C# \(LINQ to XML\)](#).

## Realizando Operações em Elementos de Origem

Uma sequência de saída pode não conter os elementos ou propriedades de elementos da sequência de origem. Em vez disso, a saída pode ser uma sequência de valores que é calculada usando os elementos de origem como argumentos de entrada.

A consulta a seguir usará uma sequência de números que representam raios de círculos, calculará a área para cada raio e retornará uma sequência de saída contendo cadeias de caracteres formatadas com a área calculada.

Cada cadeia de caracteres para a sequência de saída será formatada usando a [interpolação de cadeia de caracteres](#). Uma cadeia de caracteres interpolada terá uma `$` na frente da aspa de abertura da cadeia de caracteres e as operações podem ser executadas dentro de chaves colocadas dentro da cadeia de caracteres interpolada. Depois que essas operações forem executadas, os resultados serão concatenados.

### ⓘ Observação

Não há suporte para chamar métodos em expressões de consulta se a consulta será movida para outro domínio. Por exemplo, você não pode chamar um método comum de C# no LINQ to SQL porque o SQL Server não tem contexto para ele. No entanto, você pode mapear procedimentos armazenados para os métodos e chamá-los. Para obter mais informações, consulte [Procedimentos armazenados](#).

C#

```
class FormatQuery
{
    static void Main()
    {
        // Data source.
        double[] radii = { 1, 2, 3 };

        // LINQ query using method syntax.
        IEnumerable<string> output =
            radii.Select(r => $"Area for a circle with a radius of '{r}' =
{r * r * Math.PI:F2}");

        /*
        // LINQ query using query syntax.
        IEnumerable<string> output =
            from rad in radii
            select $"Area for a circle with a radius of '{rad}' = {rad * rad
* Math.PI:F2}";
        */

        foreach (string s in output)
        {
            Console.WriteLine(s);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

```
    }
}

/* Output:
   Area for a circle with a radius of '1' = 3.14
   Area for a circle with a radius of '2' = 12.57
   Area for a circle with a radius of '3' = 28.27
*/
```

## Confira também

- LINQ (Consulta Integrada à Linguagem) (C#)
- LINQ to SQL
- LINQ to DataSet
- LINQ to XML (C#)
- Expressões de Consulta LINQ
- Cláusula select

# Relacionamentos de tipo em operações de consulta LINQ (C#)

Artigo • 20/07/2023

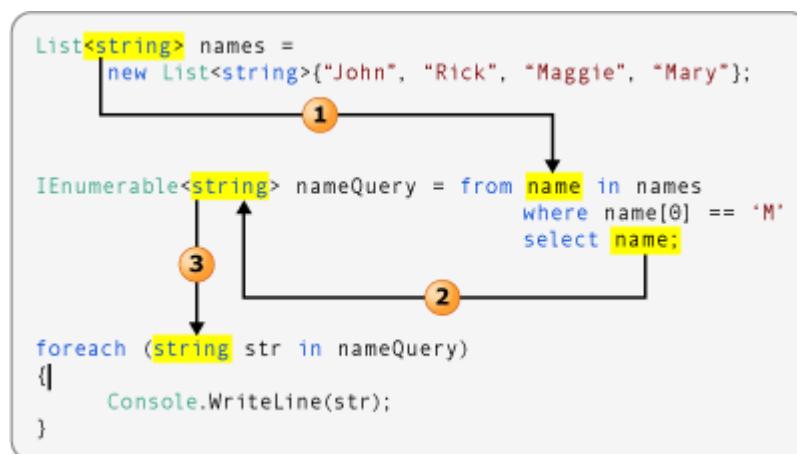
Para escrever consultas com eficiência, você precisa entender como os tipos de variáveis em uma operação de consulta completa se relacionam entre si. Se você compreender esses relacionamentos, entenderá com mais facilidade os exemplos da LINQ e as amostras de código na documentação. Além disso, você compreenderá o que ocorre nos bastidores quando variáveis são tipadas de forma implícita usando `var`.

As operações de consulta LINQ são fortemente tipadas na fonte de dados, na própria consulta e na execução da consulta. O tipo das variáveis na consulta deve ser compatível com o tipo dos elementos na fonte de dados e com o tipo da variável de iteração na instrução `foreach`. Essa tipagem forte garante que erros de tipo sejam capturados em tempo de compilação, quando podem ser corrigidos antes que os usuários os encontrem.

Para demonstrar essas relações de tipo, a maioria dos exemplos a seguir usam tipagem explícita para todas as variáveis. O último exemplo mostra como os mesmos princípios se aplicam mesmo quando você usa tipagem implícita usando `var`.

## Consultas que não transformam os dados de origem

A ilustração a seguir mostra uma operação de consulta LINQ to Objects que não executa transformações nos dados. A fonte contém uma sequência de cadeias de caracteres e a saída da consulta também é uma sequência de cadeias de caracteres.



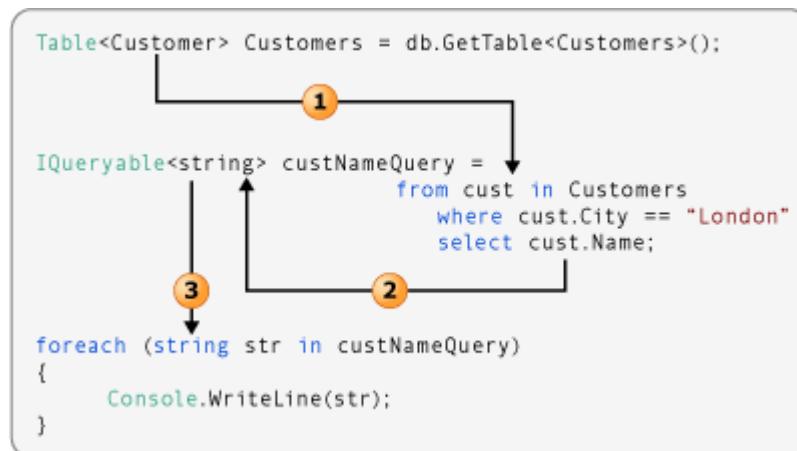
1. O argumento de tipo da fonte de dados determina o tipo da variável de intervalo.

2. O tipo do objeto selecionado determina o tipo da variável de consulta. Esta é uma cadeia de caracteres `name`. Portanto, a variável de consulta é um `IEnumerable<string>`.

3. A variável de consulta é iterada na instrução `foreach`. Como a variável de consulta é uma sequência de cadeias de caracteres, a variável de iteração também é uma cadeia de caracteres.

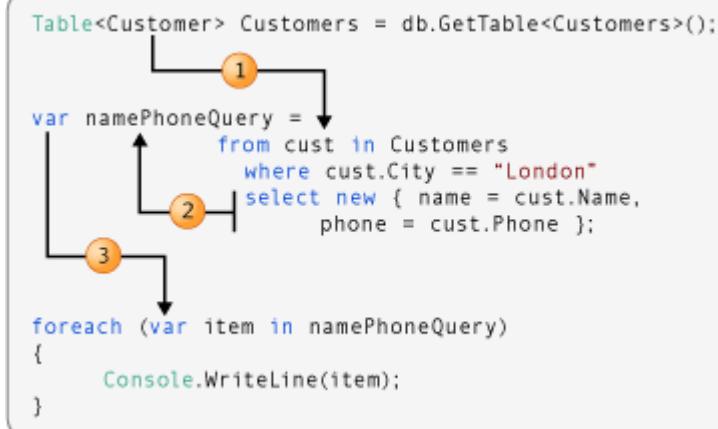
## Consultas que transformam os dados de origem

A ilustração a seguir mostra uma operação de consulta LINQ to SQL que executa uma transformação simples nos dados. A consulta usa uma sequência de objetos `Customer` como entrada e seleciona somente a propriedade `Name` no resultado. Como `Name` é uma cadeia de caracteres, a consulta produz uma sequência de cadeias de caracteres como saída.



1. O argumento de tipo da fonte de dados determina o tipo da variável de intervalo.
2. A instrução `select` retorna a propriedade `Name` em vez do objeto `Customer` completo. Como `Name` é uma cadeia de caracteres, o argumento de tipo de `custNameQuery` é `string` e não `Customer`.
3. Como `custNameQuery` é uma sequência de cadeias de caracteres, a variável de iteração do loop `foreach` também deve ser um `string`.

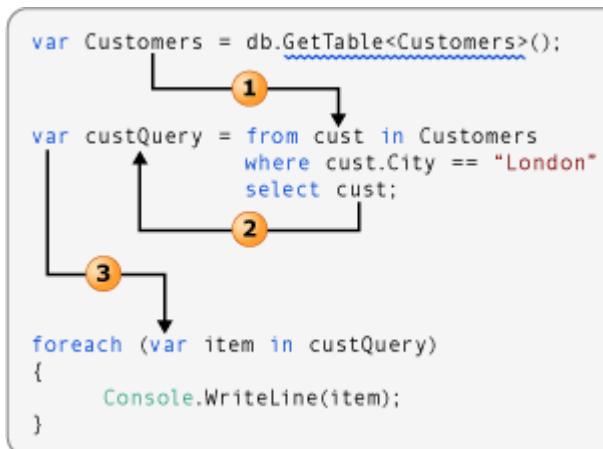
A ilustração a seguir mostra uma transformação um pouco mais complexa. A instrução `select` retorna um tipo anônimo que captura apenas dois membros do objeto `Customer` original.



1. O argumento de tipo da fonte de dados sempre é o tipo da variável de intervalo na consulta.
2. Como a instrução `select` produz um tipo anônimo, a variável de consulta deve ser tipada implicitamente usando `var`.
3. Como o tipo da variável de consulta é implícito, a variável de iteração no loop `foreach` também deve ser implícito.

## Deixando o compilador inferir informações de tipo

Embora você precise entender as relações de tipo em uma operação de consulta, você tem a opção de permitir que o compilador fazer todo o trabalho. A palavra-chave `var` pode ser usada para qualquer variável local em uma operação de consulta. A ilustração a seguir é semelhante ao exemplo número 2 que foi discutido anteriormente. No entanto, o compilador fornece o tipo forte para cada variável na operação de consulta.



Para obter mais informações sobre `var`, consulte [Variáveis de local digitadas implicitamente](#).

# LINQ e tipos genéricos (C#)

As consultas LINQ são baseadas em tipos genéricos. Não é necessário um conhecimento profundo sobre os genéricos antes de começar a escrever consultas. No entanto, convém entender dois conceitos básicos:

1. Quando você cria uma instância de uma classe de coleção genérica, como `List<T>`, substitua o "T" pelo tipo dos objetos que a lista bloqueia. Por exemplo, uma lista de cadeias de caracteres é expressa como `List<string>` e uma lista de objetos `Customer` é expressa como `List<Customer>`. Uma lista genérica é fortemente tipada e oferece muitos benefícios em coleções que armazenam seus elementos como `Object`. Se tentar adicionar um `Customer` em uma `List<string>`, você obterá um erro em tempo de compilação. É fácil usar coleções genéricas, porque você não precisa realizar a conversão de tipo em tempo de execução.
2. A `IEnumerable<T>` é a interface que permite que as classes de coleção genérica sejam enumeradas usando a instrução `foreach`. Classes de coleção genéricas dão suporte a `IEnumerable<T>` do mesmo modo que classes de coleção não genéricas, tais como `ArrayList`, dão suporte a `IEnumerable`.

Para obter mais informações sobre os genéricos, consulte [Genéricos](#).

## Variáveis `IEnumerable<T>` em consultas LINQ

As variáveis de consulta LINQ são do tipo `IEnumerable<T>` ou de um tipo derivado, como `IQueryable<T>`. Ao se deparar com uma variável de consulta que é tipada como `IEnumerable<Customer>`, significa apenas que a consulta, quando for executada, produzirá uma sequência de zero ou mais objetos `Customer`.

C#

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

Para obter mais informações, consulte [Relacionamentos de tipo em operações de consulta LINQ](#).

# Permitir que o compilador manipule as declarações de tipo genérico

Se preferir, poderá evitar a sintaxe genérica, usando a palavra-chave `var`. A palavra-chave `var` instrui o compilador a inferir o tipo de uma variável de consulta, examinando a fonte de dados especificada na cláusula `from`. O exemplo a seguir produz o mesmo código compilado que o exemplo anterior:

C#

```
var customerQuery2 =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach(var customer in customerQuery2)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
```

A palavra-chave `var` é útil quando o tipo da variável for óbvio ou quando não é tão importante especificar explicitamente os tipos genéricos aninhados, como aqueles que são produzidos por consultas de grupo. É recomendável que você note que o código poderá se tornar mais difícil de ser lido por outras pessoas, caso você use a `var`. Para obter mais informações, consulte [Variáveis locais de tipo implícito](#).

# Escrever consultas LINQ em C#

Artigo • 20/07/2023

A maioria das consultas na documentação introdutória do LINQ (Consulta Integrada à Linguagem) é escrita com o uso da sintaxe de consulta declarativa do LINQ. No entanto, a sintaxe de consulta deve ser convertida em chamadas de método para o CLR (Common Language Runtime) do .NET quando o código for compilado. Essas chamadas de método invocam os operadores de consulta padrão, que têm nomes como `Where`, `Select`, `GroupBy`, `Join`, `Max` e `Average`. Você pode chamá-los diretamente usando a sintaxe de método em vez da sintaxe de consulta.

A sintaxe de consulta e a sintaxe de método são semanticamente idênticas, mas muitas pessoas acham a sintaxe de consulta mais simples e fácil de ler. Algumas consultas devem ser expressadas como chamadas de método. Por exemplo, você deve usar uma chamada de método para expressar uma consulta que recupera o número de elementos que correspondem a uma condição especificada. Você também deve usar uma chamada de método para uma consulta que recupera o elemento que tem o valor máximo em uma sequência de origem. A documentação de referência para os operadores de consulta padrão no namespace [System.Linq](#) geralmente usa a sintaxe de método. Portanto, mesmo ao começar a escrever consultas LINQ, é útil que você esteja familiarizado com a forma de uso da sintaxe de método em consultas e nas próprias expressões de consulta.

## Métodos de extensão do operador de consulta padrão

O exemplo a seguir mostra uma *expressão de consulta* simples e a consulta semanticamente equivalente escrita como uma *consulta baseada em método*.

C#

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
```

```

    select num;

    //Method syntax:
    IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 ==
0).OrderBy(n => n);

        foreach (int i in numQuery1)
    {
        Console.Write(i + " ");
    }
    Console.WriteLine(System.Environment.NewLine);
    foreach (int i in numQuery2)
    {
        Console.Write(i + " ");
    }

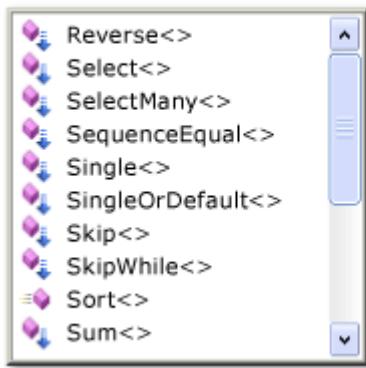
    // Keep the console open in debug mode.
    Console.WriteLine(System.Environment.NewLine);
    Console.WriteLine("Press any key to exit");
    Console.ReadKey();
}
/*
Output:
6 8 10 12
6 8 10 12
*/

```

A saída dos dois exemplos é idêntica. Você pode ver que o tipo da variável de consulta é o mesmo em ambas as formas: `IEnumerable<T>`.

Para entender a consulta baseada em método, vamos examiná-la melhor. No lado direito da expressão, observe que a cláusula `where` agora é expressa como um método de instância no objeto `numbers`, que, como você deve se lembrar, tem um tipo de `IEnumerable<int>`. Se você estiver familiarizado com a interface `IEnumerable<T>` genérica, você saberá que ela não tem um método `Where`. No entanto, se você invocar a lista de conclusão do IntelliSense no IDE do Visual Studio, verá não apenas um método `Where`, mas muitos outros métodos como `Select`, `SelectMany`, `Join` e `Orderby`. Esses são todos os operadores de consulta padrão.

```
List<string> list = new List<string>();  
list.|
```



Embora pareça como se `IEnumerable<T>` tivesse sido redefinido para incluir esses métodos adicionais, na verdade esse não é o caso. Os operadores de consulta padrão são implementados como um novo tipo de método chamado *métodos de extensão*. Métodos de extensão "estendem" um tipo existente, eles podem ser chamados como se fossem métodos de instância no tipo. Os operadores de consulta padrão estendem `IEnumerable<T>` e que é por esse motivo que você pode escrever `numbers.Where(...)`.

Para começar a usar LINQ, tudo o que você realmente precisa saber sobre os métodos de extensão é como colocá-los no escopo no seu aplicativo usando as diretivas `using` corretas. Do ponto de vista do aplicativo, um método de extensão e um método de instância normal são iguais.

Para obter mais informações sobre os métodos de extensão, consulte [Métodos de extensão](#). Para obter mais informações sobre os operadores de consulta padrão, consulte [Visão geral de operadores de consulta padrão \(C#\)](#). Alguns provedores LINQ, como LINQ to SQL e LINQ to XML, implementam seus próprios operadores de consulta padrão e métodos de extensão adicionais para outros tipos além de `IEnumerable<T>`.

## Expressões lambda

No exemplo anterior, observe que a expressão condicional (`num % 2 == 0`) é passada como um argumento embutido para o método `Where`: `Where(num => num % 2 == 0)`. Essa expressão embutida é chamada de uma expressão lambda. É uma maneira conveniente de escrever um código que de outra forma precisaria ser escrito de forma mais complicada como um método anônimo, um delegado genérico ou uma árvore de expressão. No C# `=>` é o operador lambda, que é lido como "vai para". O `num` à esquerda do operador é a variável de entrada que corresponde ao `num` na expressão de consulta. O compilador pode inferir o tipo de `num` porque ele sabe que `numbers` é um tipo `IEnumerable<T>` genérico. O corpo do lambda é exatamente igual à expressão na sintaxe de consulta ou em qualquer outra expressão ou instrução C#, ele pode incluir

chamadas de método e outra lógica complexa. O "valor retornado" é apenas o resultado da expressão.

Para começar a usar o LINQ, você não precisa usar lambdas extensivamente. No entanto, determinadas consultas só podem ser expressadas em sintaxe de método e algumas delas requerem expressões lambda. Após se familiarizar com lambdas, você verá que eles são uma ferramenta avançada e flexível na sua caixa de ferramentas do LINQ. Para obter mais informações, consulte [Expressões Lambda](#).

## Possibilidade de composição das consultas

No exemplo de código anterior, observe que o método `OrderBy` é invocado usando o operador ponto na chamada para `Where`. `Where` produz uma sequência filtrada e, em seguida, `Orderby` opera nessa sequência classificando-a. Como as consultas retornam uma `IEnumerable`, você pode escrevê-las na sintaxe de método encadeando as chamadas de método. Isso é o que o compilador faz nos bastidores quando você escreve consultas usando a sintaxe de consulta. E como uma variável de consulta não armazena os resultados da consulta, você pode modificá-la ou usá-la como base para uma nova consulta a qualquer momento, mesmo depois que ela foi executada.

Os exemplos a seguir demonstram algumas consultas LINQ simples usando cada abordagem listada anteriormente. Em geral, a regra é usar (1) sempre que possível e usar (2) e (3) sempre que necessário.

### ① Observação

Essas consultas funcionam em coleções na memória simples, no entanto, a sintaxe básica é idêntica àquela usada no LINQ to Entities e no LINQ to XML.

## Exemplo – sintaxe de consulta

A maneira recomendada de escrever a maioria das consultas é usar a *sintaxe de consulta* para criar *expressões de consulta*. O exemplo a seguir mostra três expressões de consulta. A primeira expressão de consulta demonstra como filtrar ou restringir os resultados aplicando condições com uma cláusula `where`. Ela retorna todos os elementos na sequência de origem cujos valores são maiores que 7 ou menores que 3. A segunda expressão demonstra como ordenar os resultados retornados. A terceira expressão demonstra como agrupar resultados de acordo com uma chave. Esta consulta retorna dois grupos com base na primeira letra da palavra.

C#

```
List<int> numbers = new() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

// The query variables can also be implicitly typed by using var

// Query #1.
IEnumerable<int> filteringQuery =
    from num in numbers
    where num < 3 || num > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num < 3 || num > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans",
    "barley" };
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

Observe que o tipo das consultas é `IEnumerable<T>`. Todas essas consultas poderiam ser escritas usando `var` conforme mostrado no exemplo a seguir:

```
var query = from num in numbers...
```

Em cada exemplo anterior, as consultas não são de fato executadas até você iterar na variável de consulta em uma instrução `foreach` ou outra instrução. Para obter mais informações, consulte [Introdução a Consultas LINQ](#).

## Exemplo – sintaxe de método

Algumas operações de consulta devem ser expressas como uma chamada de método. Os mais comuns desses métodos são aqueles que retornam valores numéricos singleton como `Sum`, `Max`, `Min`, `Average` e assim por diante. Esses métodos devem sempre ser chamados por último em qualquer consulta porque representam apenas um único valor e não podem atuar como a fonte para uma operação de consulta adicional. O exemplo a seguir mostra uma chamada de método em uma expressão de consulta:

C#

```
List<int> numbers1 = new() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };

// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

Se o método tiver os parâmetros Action ou Func, eles serão fornecidos na forma de uma expressão [lambda](#), como mostra o exemplo a seguir:

C#

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

Nas consultas anteriores, apenas a Query #4 é executada imediatamente. Isso ocorre porque ele retorna um único valor e não uma coleção [IEnumerable<T>](#) genérica. O próprio método tem que usar [foreach](#) para calcular seu valor.

Cada uma das consultas anteriores pode ser escrita usando a tipagem implícita com [var](#), como mostrado no exemplo a seguir:

C#

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

## Exemplo – sintaxe mista de consulta e do método

Este exemplo mostra como usar a sintaxe do método nos resultados de uma cláusula de consulta. Simplesmente coloque a expressão de consulta entre parênteses e, em seguida, aplique o operador de ponto e chame o método. No exemplo a seguir, a Query #7 retorna uma contagem dos números cujo valor está entre 3 e 7. Em geral, no entanto, é melhor usar uma segunda variável para armazenar o resultado da chamada do método. Dessa forma, é menos provável que a consulta seja confundida com os resultados da consulta.

C#

```
// Query #7.

// Using a query expression with method syntax
int numCount1 = (
    from num in numbers1
    where num < 3 || num > 7
    select num
).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

Como a Query #7 retorna um único valor e não uma coleção, a consulta é executada imediatamente.

A consulta anterior pode ser escrita usando a tipagem implícita com `var`, da seguinte maneira:

C#

```
var numCount = (from num in numbers...
```

Ela pode ser escrita na sintaxe de método da seguinte maneira:

C#

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

Ela pode ser escrita usando a tipagem explícita da seguinte maneira:

C#

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

## Confira também

- [Passo a passo: escrevendo consultas em C#](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Cláusula where](#)

# funcionalidades do C# que dão suporte a LINQ

Artigo • 09/03/2023

Esses novos recursos têm algum grau de utilização com consultas LINQ, mas não estão limitados ao LINQ e podem ser usados em qualquer contexto em que sejam úteis.

## Expressões de consulta

As expressões de consulta usam uma sintaxe declarativa semelhante ao SQL ou XQuery para consultar em coleções `IEnumerable`. No tempo de compilação, a sintaxe de consulta é convertida em chamadas de método para uma implementação LINQ dos métodos de extensão do operador de consulta padrão do provedor. Os aplicativos controlam os operadores de consulta padrão que estão no escopo, especificando o namespace apropriado com uma diretiva `using`. A expressão de consulta a seguir pega uma matriz de cadeias de caracteres, agrupa-os de acordo com o primeiro caractere da cadeia de caracteres e ordena os grupos.

C#

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

Para obter mais informações, consulte [Expressões de Consulta LINQ](#).

## Variáveis tipadas implicitamente (`var`)

Em vez de especificar explicitamente um tipo ao declarar e inicializar uma variável, você pode usar o modificador `var` para instruir o compilador a inferir e atribuir o tipo, conforme mostrado aqui:

C#

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

As variáveis declaradas como `var` são tão fortemente tipadas quanto as variáveis cujo tipo você especifica explicitamente. O uso de `var` possibilita a criação de tipos anônimos, mas ele pode ser usado para quaisquer variáveis locais. As matrizes também podem ser declaradas com tipagem implícita.

Para obter mais informações, consulte [Variáveis locais de tipo implícito](#).

## Inicializadores de objeto e coleção

Os inicializadores de objeto e de coleção possibilitam a inicialização de objetos sem chamar explicitamente um construtor para o objeto. Os inicializadores normalmente são usados em expressões de consulta quando projetam os dados de origem em um novo tipo de dados. Supondo uma classe chamada `Customer` com propriedades públicas `Name` e `Phone`, o inicializador de objeto pode ser usado como no código a seguir:

C#

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Continuando com a nossa classe `Customer`, suponha que haja uma fonte de dados chamada `IncomingOrders` e que, para cada ordem com um grande `OrderSize`, desejamos criar um novo `Customer` com base fora dessa ordem. Uma consulta LINQ pode ser executada nessa fonte de dados e usar a inicialização do objeto para preencher uma coleção:

C#

```
var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone =
o.Phone };
```

A fonte de dados pode ter mais propriedades escondidas do que a classe `Customer`, como `OrderSize`, mas com a inicialização do objeto, os dados retornados da consulta são moldados no tipo de dados desejado; escolhemos os dados que são relevantes para nossa classe. Consequentemente, agora temos um `IEnumerable` preenchido com os novos `Customer`s que queríamos. O trecho acima também pode ser escrito na sintaxe de método do LINQ:

C#

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y => new Customer { Name = y.Name, Phone = y.Phone });
```

Para obter mais informações, consulte:

- [Inicializadores de objeto e coleção](#)
- [Sintaxe de expressão da consulta para operadores de consulta padrão](#)

## Tipos anônimos

Um tipo anônimo é construído pelo compilador e o nome do tipo só fica disponível para o compilador. Os tipos anônimos fornecem uma maneira conveniente de agrupar um conjunto de propriedades temporariamente em um resultado de consulta, sem a necessidade de definir um tipo nomeado separado. Os tipos anônimos são inicializados com uma nova expressão e um inicializador de objeto, como mostrado aqui:

C#

```
select new {name = cust.Name, phone = cust.Phone};
```

Para obter mais informações, consulte [Tipos Anônimos](#).

## Métodos de Extensão

Um método de extensão é um método estático que pode ser associado a um tipo, para que ele possa ser chamado como se fosse um método de instância no tipo. Esse recurso permite que você, na verdade, "adicone" novos métodos a tipos existentes sem realmente modificá-los. Os operadores de consulta padrão são um conjunto de métodos de extensão que fornecem a funcionalidade de consulta LINQ para qualquer tipo que implementa [IEnumerable<T>](#).

Para obter mais informações, consulte [Métodos de extensão](#).

## Expressões lambda

Uma expressão lambda é uma função embutida que usa o operador `=>` para separar os parâmetros de entrada do corpo da função e podem ser convertidos em um delegado ou uma árvore de expressão, em tempo de compilação. Na programação LINQ, você

encontra as expressões lambda ao fazer chamadas de método diretas aos operadores de consulta padrão.

Para obter mais informações, consulte:

- [Expressões Lambda](#)
- [Árvores de expressão \(C#\)](#)

## Confira também

- [LINQ \(Consulta Integrada à Linguagem\) \(C#\)](#)

# Passo a passo: Escrevendo consultas em C# (LINQ)

Artigo • 07/04/2023

Essas instruções passo a passo demonstram os recursos de linguagem C# que são usados para gravar expressões de consulta LINQ.

## Criar um Projeto C#

### ⓘ Observação

As instruções a seguir são para o Visual Studio. Se você estiver usando um ambiente de desenvolvimento diferente, crie um projeto de console com uma referência a System.Core.dll e uma diretiva `using` para o namespace `System.Linq`.

### Para criar um projeto no Visual Studio

1. Inicie o Visual Studio.
2. Na barra de menus, escolha **Arquivo, Novo, Projeto**.  
A caixa de diálogo **Novo Projeto** será aberta.
3. Expanda **Instalado**, expanda **Modelos**, expanda **Visual C# e**, em seguida, escolha **Aplicativo de Console**.
4. Na caixa de texto **Nome**, insira um nome diferente ou aceite o nome padrão e escolha o botão **OK**.  
O novo projeto aparece no **Gerenciador de Soluções**.
5. Observe que o projeto tem uma referência a System.Core.dll e a uma diretiva `using` para o namespace `System.Linq`.

## Criar uma Fonte de Dados na Memória

A fonte de dados para as consultas é uma lista simples de objetos `Student`. Cada registro `Student` tem um nome, sobrenome e uma matriz de inteiros que representa

seus resultados de testes na classe. Copie este código em seu projeto. Observe as seguintes características:

- A classe `Student` consiste em propriedades autoimplementadas.
- Cada aluno na lista é inicializado com um inicializador de objeto.
- A lista em si é inicializada com um inicializador de coleção.

Essa estrutura de dados inteira será inicializada e instanciada sem chamadas explícitas para nenhum construtor ou acesso de membro explícito. Para obter mais informações sobre esses novos recursos, consulte [Propriedades autoimplementadas](#) e [Inicializadores de objeto e coleção](#).

## Para adicionar a fonte de dados

- Adicione a classe `Student` e a lista inicializada de alunos à classe `Program` em seu projeto.

C#

```
public class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public List<int> Scores;
}

// Create a data source by using a collection initializer.
static List<Student> students = new List<Student>
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
new List<int> {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new
List<int> {75, 84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new
List<int> {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new
List<int> {97, 89, 85, 82}},
    new Student {First="Debra", Last="Garcia", ID=115, Scores= new
List<int> {35, 72, 91, 70}},
    new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new
List<int> {99, 86, 90, 94}},
    new Student {First="Hanying", Last="Feng", ID=117, Scores= new
List<int> {93, 92, 80, 87}},
    new Student {First="Hugo", Last="Garcia", ID=118, Scores= new
List<int> {92, 90, 83, 78}},
    new Student {First="Lance", Last="Tucker", ID=119, Scores= new
List<int> {68, 79, 88, 92}},
```

```
    new Student {First="Terry", Last="Adams", ID=120, Scores= new  
List<int> {99, 82, 81, 79}},  
    new Student {First="Eugene", Last="Zabokritski", ID=121, Scores=  
new List<int> {96, 85, 91, 60}},  
    new Student {First="Michael", Last="Tucker", ID=122, Scores= new  
List<int> {94, 92, 91, 91}}  
};
```

## Para adicionar um novo Aluno à lista de Alunos

1. Adicione um novo `Student` à lista `Students` e use um nome e pontuações de teste de sua escolha. Tente digitar as informações do novo aluno para aprender melhor a sintaxe do inicializador de objeto.

## Criar a Consulta

### Para criar uma consulta simples

- No método `Main` do aplicativo, crie uma consulta simples que, quando for executada, produzirá uma lista de todos os alunos cuja pontuação no primeiro teste foi superior a 90. Observe que como o objeto `student` todo está selecionado, o tipo da consulta é `IEnumerable<Student>`. Embora o código também possa usar a tipagem implícita usando a palavra-chave `var`, a tipagem explícita é usada para ilustrar claramente os resultados. (Para obter mais informações sobre `var`, consulte [Variáveis locais de tipo implícito](#).)

Observe também que a variável de intervalo da consulta, `student`, também funciona como uma referência para cada `Student` na fonte, fornecendo acesso ao membro para cada objeto.

C#

```
// Create the query.  
// The first line could also be written as "var studentQuery ="  
IEnumerable<Student> studentQuery =  
    from student in students  
    where student.Scores[0] > 90  
    select student;
```

## Executar a Consulta

## Para executar a consulta.

1. Agora escreva o loop `foreach` que fará com que a consulta seja executada.

Observe o seguinte sobre o código:

- Cada elemento na sequência retornada é acessado pela variável de iteração no loop `foreach`.
- O tipo dessa variável é `Student` e o tipo da variável de consulta é compatível, `IEnumerable<Student>`.

2. Após você ter adicionado esse código, compile e execute o aplicativo para ver os resultados na janela **Console**.

C#

```
// Execute the query.  
// var could be used here also.  
foreach (Student student in studentQuery)  
{  
    Console.WriteLine("{0}, {1}", student.Last, student.First);  
  
}  
  
// Output:  
// Omelchenko, Svetlana  
// Garcia, Cesar  
// Fakhouri, Fadi  
// Feng, Hanying  
// Garcia, Hugo  
// Adams, Terry  
// Zabokritski, Eugene  
// Tucker, Michael
```

## Para adicionar outra condição de filtro

1. Você pode combinar várias condições booleanas na cláusula `where` para refinar ainda mais uma consulta. O código a seguir adiciona uma condição de forma que a consulta retorna os alunos cuja primeira pontuação foi superior a 90 e cuja última pontuação foi inferior a 80. A cláusula `where` deve parecer com o código a seguir.

C#

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

Para obter mais informações, consulte [Cláusula where](#).

# Modificar a Consulta

## Para ordenar os resultados

1. Será mais fácil verificar os resultados se eles estiverem em algum tipo de ordem.

Você pode ordenar a sequência retornada por qualquer campo acessível nos elementos de origem. Por exemplo, a cláusula `orderby` a seguir ordena os resultados em ordem alfabética de A a Z de acordo com o sobrenome de cada aluno. Adicione a cláusula `orderby` a seguir à consulta, logo após a instrução `where` e antes da instrução `select`:

```
C#  
  
orderby student.Last ascending
```

2. Agora, altere a cláusula `orderby` para que ela ordene os resultados em ordem inversa de acordo com a pontuação no primeiro teste, da pontuação mais alta para a mais baixa.

```
C#  
  
orderby student.Scores[0] descending
```

3. Altere a cadeia de caracteres de formato `WriteLine` para que você possa ver as pontuações:

```
C#  
  
Console.WriteLine("{0}, {1} {2}", student.Last, student.First,  
student.Scores[0]);
```

Para obter mais informações, consulte [Cláusula orderby](#).

## Para agrupar os resultados

1. O agrupamento é uma poderosa funcionalidade em expressões de consulta. Uma consulta com uma cláusula `group` produz uma sequência de grupos e cada grupo em si contém um `Key` e uma sequência que consiste em todos os membros desse grupo. A nova consulta a seguir agrupa os alunos usando a primeira letra do sobrenome como a chave.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery2 =  
    from student in students  
    group student by student.Last[0];
```

2. Observe que o tipo da consulta agora mudou. Ele produz uma sequência de grupos que têm um tipo `char` como uma chave e uma sequência de objetos `Student`. Como o tipo de consulta foi alterado, o código a seguir altera o loop de execução `foreach` também:

C#

```
foreach (IGrouping<char, Student> studentGroup in studentQuery2)  
{  
    Console.WriteLine(studentGroup.Key);  
    foreach (Student student in studentGroup)  
    {  
        Console.WriteLine(" {0}, {1}",  
                           student.Last, student.First);  
    }  
}  
  
// Output:  
// O  
//   Omelchenko, Svetlana  
//   O'Donnell, Claire  
// M  
//   Mortensen, Sven  
// G  
//   Garcia, Cesar  
//   Garcia, Debra  
//   Garcia, Hugo  
// F  
//   Fakhouri, Fadi  
//   Feng, Hanying  
// T  
//   Tucker, Lance  
//   Tucker, Michael  
// A  
//   Adams, Terry  
// Z  
//   Zabokritski, Eugene
```

3. Execute o aplicativo e exiba os resultados na janela **Console**.

Para obter mais informações, consulte [Cláusula group](#).

**Para deixar as variáveis tipadas implicitamente**

1. A codificação explícita `IEnumerables` de `IGroupings` pode se tornar entediante rapidamente. Você pode escrever a mesma consulta e o loop `foreach` muito mais convenientemente usando `var`. A palavra-chave `var` não altera os tipos de objetos, ela simplesmente instrui o compilador a inferir os tipos. Altere o tipo de `studentQuery` e a variável de iteração `group` para `var` e execute a consulta novamente. Observe que no loop `foreach` interno, a variável de iteração ainda tem o tipo `Student` e a consulta funciona exatamente como antes. Alterar a variável de iteração `student` para `var` e execute a consulta novamente. Você verá que obtém exatamente os mesmos resultados.

C#

```
var studentQuery3 =
    from student in students
    group student by student.Last[0];

foreach (var groupOfStudents in studentQuery3)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine(" {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

Para obter mais informações sobre `var`, consulte [Variáveis locais de tipo implícito](#).

## Para ordenar os grupos pelo valor da chave

1. Ao executar a consulta anterior, observe que os grupos não estão em ordem alfabética. Para alterar isso, você deve fornecer uma cláusula `orderby` após a cláusula `group`. Mas, para usar uma cláusula `orderby`, primeiro é necessário um identificador que serve como uma referência para os grupos criados pela cláusula `group`. Forneça o identificador usando a palavra-chave `into`, da seguinte maneira:

C#

```
var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine(" {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
//A
// Adams, Terry
//F
// Fakhouri, Fadi
// Feng, Hanying
//G
// Garcia, Cesar
// Garcia, Debra
// Garcia, Hugo
//M
// Mortensen, Sven
//O
// Omelchenko, Svetlana
// O'Donnell, Claire
//T
// Tucker, Lance
// Tucker, Michael
//Z
// Zabokritski, Eugene
```

Quando você executa essa consulta, você verá que os grupos agora estão classificados em ordem alfabética.

## Para introduzir um identificador usando let

1. Você pode usar a palavra-chave `let` para introduzir um identificador para qualquer resultado da expressão na expressão de consulta. Esse identificador pode ser uma conveniência, como no exemplo a seguir ou ele pode melhorar o desempenho armazenando os resultados de uma expressão para que ele não precise ser calculado várias vezes.

C#

```
// studentQuery5 is an IEnumerable<string>
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select student.Last + " " + student.First;

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
// Omelchenko Svetlana
// O'Donnell Claire
// Mortensen Sven
// Garcia Cesar
// Fakhouri Fadi
// Feng Hanying
// Garcia Hugo
// Adams Terry
// Zabokritski Eugene
// Tucker Michael
```

Para obter mais informações, consulte [Cláusula let](#).

## Para usar a sintaxe do método em uma expressão de consulta

1. Conforme descrito em [Sintaxe de consulta e sintaxe de método em LINQ](#), algumas operações de consulta podem ser expressadas somente usando a sintaxe de método. O código a seguir calcula a pontuação total para cada `Student` na sequência de origem e então chama o método `Average()` nos resultados da consulta para calcular a pontuação média da classe.

C#

```
var studentQuery6 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery6.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.1666666666667
```

## Para transformar ou projetar na cláusula select

1. É muito comum para uma consulta produzir uma sequência cujos elementos são diferentes dos elementos nas sequências de origem. Exclua ou comente o loop de consulta e execução anterior e substitua-o pelo código a seguir. Observe que a consulta retorna uma sequência de cadeias de caracteres (não `Students`) e esse fato é refletido no loop `foreach`.

C#

```
IEnumerable<string> studentQuery7 =
    from student in students
    where student.Last == "Garcia"
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery7)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo
```

2. O código anterior neste passo a passo indicou que a pontuação média de classe é de aproximadamente 334. Para produzir uma sequência de `Students` cuja pontuação total é maior que a média de classe, juntamente com seus `Student ID`, você pode usar um tipo anônimo na instrução `select`:

C#

```
var studentQuery8 =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in studentQuery8)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id,
item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368
```

## Próximas etapas

Depois que estiver familiarizado com os aspectos básicos de como trabalhar com consultas em C#, você estará pronto para ler a documentação e exemplos para o tipo específico de provedor LINQ que lhe interessam:

[LINQ to SQL](#)

[LINQ to DataSet](#)

[LINQ to XML \(C#\)](#)

[LINQ to Objects \(C#\)](#)

## Confira também

- [LINQ \(Consulta Integrada à Linguagem\) \(C#\)](#)
- [Expressões de Consulta LINQ](#)

# Visão geral de operadores de consulta padrão (C#)

Artigo • 20/07/2023

Os *operadores de consulta padrão* são os métodos que formam o padrão LINQ. A maioria desses métodos opera em sequências; neste contexto, uma sequência é um objeto cujo tipo implementa a interface `IEnumerable<T>` ou a interface `IQueryable<T>`. Os operadores de consulta padrão fornecem recursos de consulta incluindo filtragem, projeção, agregação, classificação e muito mais.

Há dois conjuntos de operadores de consulta padrão do LINQ: um que opera em objetos do tipo `IEnumerable<T>`, outro que opera em objetos do tipo `IQueryable<T>`. Os métodos que compõem a cada conjunto são os membros estáticos das classes `Enumerable` e `Queryable`, respectivamente. Eles são definidos como *métodos de extensão* do tipo nos quais operam. Os métodos de extensão podem ser chamados usando a sintaxe do método estático ou a sintaxe do método de instância.

Além disso, vários métodos de operador de consulta padrão operam em tipos diferentes daqueles baseados em `IEnumerable<T>` ou `IQueryable<T>`. O tipo `Enumerable` define dois métodos tais que ambos operam em objetos do tipo `IEnumerable`. Esses métodos, `Cast<TResult>(IEnumerable)` e `OfType<TResult>(IEnumerable)`, permitem que você habilite uma coleção sem parâmetros ou não genérica, a ser consultada no padrão LINQ. Eles fazem isso criando uma coleção de objetos fortemente tipada. A classe `Queryable` define dois métodos semelhantes, `Cast<TResult>(IQueryable)` e `OfType<TResult>(IQueryable)`, que operam em objetos do tipo `IQueryable`.

Os operadores de consulta padrão são diferentes no momento de sua execução, dependendo de se eles retornam um valor singleton ou uma sequência de valores. Esses métodos que retornam um valor singleton (por exemplo, `Average` e `Sum`) são executados imediatamente. Os métodos que retornam uma sequência adiam a execução da consulta e retornam um objeto enumerável.

Para métodos que operam em coleções na memória, ou seja, os métodos que estendem `IEnumerable<T>`, o objeto `Enumerable` retornado captura os argumentos que foram passados para o método. Quando esse objeto é enumerado, a lógica do operador de consulta é empregada e os resultados da consulta são retornados.

Por outro lado, os métodos que estendem `IQueryable<T>` não implementam nenhum comportamento de consulta. Eles criam uma árvore de expressão que representa a

consulta a ser executada. O processamento de consulta é tratado pelo objeto [IQueryable<T>](#) de origem.

Chamadas para métodos de consulta podem ser encadeadas em uma consulta, o que permite que consultas se tornem arbitrariamente complexas.

O exemplo de código a seguir demonstra como os operadores de consulta padrão podem ser usados para obter informações sobre uma sequência.

C#

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

## Sintaxe de expressão de consulta

Alguns dos operadores de consulta padrão mais usados têm uma sintaxe de palavra-chave de linguagem C# e Visual Basic dedicada que possibilita que eles sejam chamados como parte de uma *expressão de consulta*. Para obter mais informações sobre operadores de consulta padrão que têm palavras-chave dedicadas e suas sintaxes correspondentes, consulte [Sintaxe de expressão de consulta para operadores de consulta padrão \(C#\)](#).

## Estendendo os operadores de consulta padrão

Você pode aumentar o conjunto de operadores de consulta padrão criando métodos específicos de domínio apropriados para o domínio ou tecnologia de destino. Você também pode substituir os operadores de consulta padrão por suas próprias implementações que fornecem serviços adicionais, como avaliação remota, conversão de consulta e otimização. Para ver um exemplo, consulte [AsEnumerable](#).

## Obter uma fonte de dados

Em uma consulta LINQ, a primeira etapa é especificar a fonte de dados. No C#, como na maioria das linguagens de programação, uma variável deve ser declarada antes que possa ser usada. Em um consulta LINQ, a cláusula `from` vem primeiro para introduzir a fonte de dados (`customers`) e a *variável de intervalo* (`cust`).

C#

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

A variável de intervalo é como a variável de iteração em um loop `foreach`, mas nenhuma iteração real ocorre em uma expressão de consulta. Quando a consulta é executada, a variável de intervalo servirá como uma referência para cada elemento sucessivo em `customers`. Uma vez que o compilador pode inferir o tipo de `cust`, você não precisa especificá-lo explicitamente. Variáveis de intervalo adicionais podem ser introduzidas por uma cláusula `let`. Para obter mais informações, consulte [Cláusula let](#).

### Observação

Para fontes de dados não genéricas, como `ArrayList`, a variável de intervalo deve ser tipada explicitamente. Para obter mais informações, consulte [Como consultar um ArrayList com LINQ \(C#\)](#) e a cláusula `from`.

# Filtragem

Provavelmente, a operação de consulta mais comum é aplicar um filtro no formulário de uma expressão booliana. O filtro faz com que a consulta retorne apenas os elementos para os quais a expressão é verdadeira. O resultado é produzido usando a cláusula `where`. O filtro em vigor especifica os elementos a serem excluídos da sequência de origem. No exemplo a seguir, somente os `customers` que têm um endereço em Londres são retornados.

C#

```
var queryLondonCustomers = from cust in customers
                            where cust.City == "London"
                            select cust;
```

Você pode usar os operadores lógicos `AND` e `OR` de C# para aplicar quantas expressões de filtro forem necessárias na cláusula `where`. Por exemplo, para retornar somente clientes de "Londres" `AND` cujo nome seja "Devon", você escreveria o seguinte código:

C#

```
where cust.City == "London" && cust.Name == "Devon"
```

Para retornar clientes de Londres ou Paris, você escreveria o seguinte código:

C#

```
where cust.City == "London" || cust.City == "Paris"
```

Para obter mais informações, consulte [Cláusula where](#).

# Ordenando

Muitas vezes é conveniente classificar os dados retornados. A cláusula `orderby` fará com que os elementos na sequência retornada sejam classificados de acordo com o comparador padrão para o tipo que está sendo classificado. Por exemplo, a consulta a seguir pode ser estendida para classificar os resultados com base na propriedade `Name`. Como `Name` é uma cadeia de caracteres, o comparador padrão executa uma classificação em ordem alfabética de A a Z.

C#

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

Para ordenar os resultados na ordem inversa, de Z para a, use a cláusula `orderby...descending`.

Para obter mais informações, consulte [Cláusula orderby](#).

## Agrupamento

A cláusula `group` permite agrupar seus resultados com base em uma chave que você especificar. Por exemplo, você pode especificar que os resultados sejam agrupados segundo o `city`, de modo que todos os clientes de Londres ou Paris fiquem em grupos individuais. Nesse caso, `cust.City` é a chave.

C#

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
var queryCustomersByCity =
    from cust in customers
    group cust by cust.City;

// customerGroup is an IGrouping<string, Customer>
foreach (var customerGroup in queryCustomersByCity)
{
    Console.WriteLine(customerGroup.Key);
    foreach (Customer customer in customerGroup)
    {
        Console.WriteLine("    {0}", customer.Name);
    }
}
```

Quando você terminar uma consulta com um cláusula `group`, seus resultados assumirão a forma de uma lista de listas. Cada elemento na lista é um objeto que tem um membro `Key` e uma lista dos elementos que estão agrupados sob essa chave. Quando itera em uma consulta que produz uma sequência de grupos, você deve usar um loop `foreach` aninhado. O loop externo itera em cada grupo e o loop interno itera nos membros de cada grupo.

Se precisar consultar os resultados de uma operação de grupo, você poderá usar a palavra-chave `into` para criar um identificador que pode ser consultado ainda mais. A

consulta a seguir retorna apenas os grupos que contêm mais de dois clientes:

C#

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

Para obter mais informações, consulte [Cláusula group](#).

## Adição

Operações de junção criam associações entre sequências que não são modeladas explicitamente nas fontes de dados. Por exemplo, você pode executar uma junção para localizar todos os clientes e distribuidores que têm o mesmo local. No LINQ, a cláusula `join` sempre funciona com coleções de objetos em vez de tabelas de banco de dados diretamente.

C#

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

No LINQ, você não precisa usar `join` com a mesma frequência que o faz no SQL, porque as chaves estrangeiras no LINQ são representados no modelo do objeto como propriedades que mantêm uma coleção de itens. Por exemplo, um objeto `Customer` que contém uma coleção de objetos `Order`. Em vez de executar uma junção, você pode acessar os pedidos usando notação de ponto:

C#

```
from order in Customer.Orders...
```

Para obter mais informações, consulte [Cláusula join](#).

## Selecionando (Projeções)

A cláusula `select` produz os resultados da consulta e especifica a "forma" ou o tipo de cada elemento retornado. Por exemplo, você pode especificar se os resultados consistirão em objetos `Customer` completos, apenas um membro, um subconjunto de membros ou algum tipo de resultado completamente diferente com base em um cálculo ou na criação de um novo objeto. Quando a cláusula `select` produz algo diferente de uma cópia do elemento de origem, a operação é chamada de *projeção*. O uso de projeções para transformar dados é um recurso poderoso de expressões de consulta LINQ. Para obter mais informações, consulte [Transformações de dados com LINQ \(C#\)](#) e [Cláusula select](#).

## Tabela de sintaxe de expressão de consulta

A tabela a seguir lista os operadores de consulta padrão que têm cláusulas de expressão de consulta equivalentes.

Método	Sintaxe de expressão de consulta em C#
<a href="#">Cast</a>	Use uma variável de intervalo de tipo explícito, por exemplo:  <code>from int i in numbers</code>  (Para obter mais informações, consulte <a href="#">Cláusula from</a> .)
<a href="#">GroupBy</a>	<code>group ... by</code>  -ou-  <code>group ... by ... into ...</code>  (Para obter mais informações, consulte <a href="#">Cláusula group</a> .)
<a href="#">GroupJoin&lt;TOuter,TInner,TKey,TResult&gt;(IEnumerable&lt;TOuter&gt;, IEnumerable&lt;TInner&gt;, Func&lt;TOuter,TKey&gt;, Func&lt;TInner,TKey&gt;, Func&lt;TOuter,IEnumerable&lt;TInner&gt;, TResult&gt;)</a>	<code>join ... in ... on ... equals ... into ...</code>  (Para obter mais informações, consulte <a href="#">Cláusula join</a> .)

Método	Sintaxe de expressão de consulta em C#
<code>Join&lt;TOuter,TInner,TKey,TResult&gt;(IEnumerable&lt;TOuter&gt;, IEnumerable&lt;TInner&gt;, Func&lt;TOuter,TKey&gt;, Func&lt;TInner,TKey&gt;, Func&lt;TOuter,TInner,TResult&gt;)</code>	<code>join ... in ... on ... equals ...</code> (Para obter mais informações, consulte <a href="#">Cláusula join</a> .)
<code>OrderBy&lt;TSource,TKey&gt;(IEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</code>	<code>orderby</code> (Para obter mais informações, consulte <a href="#">Cláusula orderby</a> .)
<code>OrderByDescending&lt;TSource,TKey&gt;(IEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</code>	<code>orderby ... descending</code> (Para obter mais informações, consulte <a href="#">Cláusula orderby</a> .)
<code>Select</code>	<code>select</code> (Para obter mais informações, consulte <a href="#">Cláusula select</a> .)
<code>SelectMany</code>	Várias cláusulas <code>from</code> . (Para obter mais informações, consulte <a href="#">Cláusula from</a> .)
<code>ThenBy&lt;TSource,TKey&gt;(IOrderedEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</code>	<code>orderby ..., ...</code> (Para obter mais informações, consulte <a href="#">Cláusula orderby</a> .)
<code>ThenByDescending&lt;TSource,TKey&gt;(IOrderedEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</code>	<code>orderby ..., ... descending</code> (Para obter mais informações, consulte <a href="#">Cláusula orderby</a> .)
<code>Where</code>	<code>where</code> (Para obter mais

Método	Sintaxe de expressão de consulta em C#
	informações, consulte <a href="#">Cláusula where.</a> )

## Confira também

- [Enumerable](#)
- [Queryable](#)
- [Métodos de Extensão](#)
- [Palavras-chave de Consulta \(LINQ\)](#)
- [Tipos anônimos](#)

# Visão geral de operadores de consulta padrão (C#)

Artigo • 20/07/2023

Os *operadores de consulta padrão* são os métodos que formam o padrão LINQ. A maioria desses métodos opera em sequências; neste contexto, uma sequência é um objeto cujo tipo implementa a interface `IEnumerable<T>` ou a interface `IQueryable<T>`. Os operadores de consulta padrão fornecem recursos de consulta incluindo filtragem, projeção, agregação, classificação e muito mais.

Há dois conjuntos de operadores de consulta padrão do LINQ: um que opera em objetos do tipo `IEnumerable<T>`, outro que opera em objetos do tipo `IQueryable<T>`. Os métodos que compõem a cada conjunto são os membros estáticos das classes `Enumerable` e `Queryable`, respectivamente. Eles são definidos como *métodos de extensão* do tipo nos quais operam. Os métodos de extensão podem ser chamados usando a sintaxe do método estático ou a sintaxe do método de instância.

Além disso, vários métodos de operador de consulta padrão operam em tipos diferentes daqueles baseados em `IEnumerable<T>` ou `IQueryable<T>`. O tipo `Enumerable` define dois métodos tais que ambos operam em objetos do tipo `IEnumerable`. Esses métodos, `Cast<TResult>(IEnumerable)` e `OfType<TResult>(IEnumerable)`, permitem que você habilite uma coleção sem parâmetros ou não genérica, a ser consultada no padrão LINQ. Eles fazem isso criando uma coleção de objetos fortemente tipada. A classe `Queryable` define dois métodos semelhantes, `Cast<TResult>(IQueryable)` e `OfType<TResult>(IQueryable)`, que operam em objetos do tipo `IQueryable`.

Os operadores de consulta padrão são diferentes no momento de sua execução, dependendo de se eles retornam um valor singleton ou uma sequência de valores. Esses métodos que retornam um valor singleton (por exemplo, `Average` e `Sum`) são executados imediatamente. Os métodos que retornam uma sequência adiam a execução da consulta e retornam um objeto enumerável.

Para métodos que operam em coleções na memória, ou seja, os métodos que estendem `IEnumerable<T>`, o objeto `Enumerable` retornado captura os argumentos que foram passados para o método. Quando esse objeto é enumerado, a lógica do operador de consulta é empregada e os resultados da consulta são retornados.

Por outro lado, os métodos que estendem `IQueryable<T>` não implementam nenhum comportamento de consulta. Eles criam uma árvore de expressão que representa a

consulta a ser executada. O processamento de consulta é tratado pelo objeto [IQueryable<T>](#) de origem.

Chamadas para métodos de consulta podem ser encadeadas em uma consulta, o que permite que consultas se tornem arbitrariamente complexas.

O exemplo de código a seguir demonstra como os operadores de consulta padrão podem ser usados para obter informações sobre uma sequência.

C#

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

## Sintaxe de expressão de consulta

Alguns dos operadores de consulta padrão mais usados têm uma sintaxe de palavra-chave de linguagem C# e Visual Basic dedicada que possibilita que eles sejam chamados como parte de uma *expressão de consulta*. Para obter mais informações sobre operadores de consulta padrão que têm palavras-chave dedicadas e suas sintaxes correspondentes, consulte [Sintaxe de expressão de consulta para operadores de consulta padrão \(C#\)](#).

## Estendendo os operadores de consulta padrão

Você pode aumentar o conjunto de operadores de consulta padrão criando métodos específicos de domínio apropriados para o domínio ou tecnologia de destino. Você também pode substituir os operadores de consulta padrão por suas próprias implementações que fornecem serviços adicionais, como avaliação remota, conversão de consulta e otimização. Para ver um exemplo, consulte [AsEnumerable](#).

## Obter uma fonte de dados

Em uma consulta LINQ, a primeira etapa é especificar a fonte de dados. No C#, como na maioria das linguagens de programação, uma variável deve ser declarada antes que possa ser usada. Em um consulta LINQ, a cláusula `from` vem primeiro para introduzir a fonte de dados (`customers`) e a *variável de intervalo* (`cust`).

C#

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

A variável de intervalo é como a variável de iteração em um loop `foreach`, mas nenhuma iteração real ocorre em uma expressão de consulta. Quando a consulta é executada, a variável de intervalo servirá como uma referência para cada elemento sucessivo em `customers`. Uma vez que o compilador pode inferir o tipo de `cust`, você não precisa especificá-lo explicitamente. Variáveis de intervalo adicionais podem ser introduzidas por uma cláusula `let`. Para obter mais informações, consulte [Cláusula let](#).

### Observação

Para fontes de dados não genéricas, como `ArrayList`, a variável de intervalo deve ser tipada explicitamente. Para obter mais informações, consulte [Como consultar um ArrayList com LINQ \(C#\)](#) e a cláusula `from`.

# Filtragem

Provavelmente, a operação de consulta mais comum é aplicar um filtro no formulário de uma expressão booliana. O filtro faz com que a consulta retorne apenas os elementos para os quais a expressão é verdadeira. O resultado é produzido usando a cláusula `where`. O filtro em vigor especifica os elementos a serem excluídos da sequência de origem. No exemplo a seguir, somente os `customers` que têm um endereço em Londres são retornados.

C#

```
var queryLondonCustomers = from cust in customers
                            where cust.City == "London"
                            select cust;
```

Você pode usar os operadores lógicos `AND` e `OR` de C# para aplicar quantas expressões de filtro forem necessárias na cláusula `where`. Por exemplo, para retornar somente clientes de "Londres" `AND` cujo nome seja "Devon", você escreveria o seguinte código:

C#

```
where cust.City == "London" && cust.Name == "Devon"
```

Para retornar clientes de Londres ou Paris, você escreveria o seguinte código:

C#

```
where cust.City == "London" || cust.City == "Paris"
```

Para obter mais informações, consulte [Cláusula where](#).

# Ordenando

Muitas vezes é conveniente classificar os dados retornados. A cláusula `orderby` fará com que os elementos na sequência retornada sejam classificados de acordo com o comparador padrão para o tipo que está sendo classificado. Por exemplo, a consulta a seguir pode ser estendida para classificar os resultados com base na propriedade `Name`. Como `Name` é uma cadeia de caracteres, o comparador padrão executa uma classificação em ordem alfabética de A a Z.

C#

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

Para ordenar os resultados na ordem inversa, de Z para a, use a cláusula `orderby...descending`.

Para obter mais informações, consulte [Cláusula orderby](#).

## Agrupamento

A cláusula `group` permite agrupar seus resultados com base em uma chave que você especificar. Por exemplo, você pode especificar que os resultados sejam agrupados segundo o `city`, de modo que todos os clientes de Londres ou Paris fiquem em grupos individuais. Nesse caso, `cust.City` é a chave.

C#

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
var queryCustomersByCity =
    from cust in customers
    group cust by cust.City;

// customerGroup is an IGrouping<string, Customer>
foreach (var customerGroup in queryCustomersByCity)
{
    Console.WriteLine(customerGroup.Key);
    foreach (Customer customer in customerGroup)
    {
        Console.WriteLine("    {0}", customer.Name);
    }
}
```

Quando você terminar uma consulta com um cláusula `group`, seus resultados assumirão a forma de uma lista de listas. Cada elemento na lista é um objeto que tem um membro `Key` e uma lista dos elementos que estão agrupados sob essa chave. Quando itera em uma consulta que produz uma sequência de grupos, você deve usar um loop `foreach` aninhado. O loop externo itera em cada grupo e o loop interno itera nos membros de cada grupo.

Se precisar consultar os resultados de uma operação de grupo, você poderá usar a palavra-chave `into` para criar um identificador que pode ser consultado ainda mais. A

consulta a seguir retorna apenas os grupos que contêm mais de dois clientes:

C#

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

Para obter mais informações, consulte [Cláusula group](#).

## Adição

Operações de junção criam associações entre sequências que não são modeladas explicitamente nas fontes de dados. Por exemplo, você pode executar uma junção para localizar todos os clientes e distribuidores que têm o mesmo local. No LINQ, a cláusula `join` sempre funciona com coleções de objetos em vez de tabelas de banco de dados diretamente.

C#

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

No LINQ, você não precisa usar `join` com a mesma frequência que o faz no SQL, porque as chaves estrangeiras no LINQ são representados no modelo do objeto como propriedades que mantêm uma coleção de itens. Por exemplo, um objeto `Customer` que contém uma coleção de objetos `Order`. Em vez de executar uma junção, você pode acessar os pedidos usando notação de ponto:

C#

```
from order in Customer.Orders...
```

Para obter mais informações, consulte [Cláusula join](#).

## Selecionando (Projeções)

A cláusula `select` produz os resultados da consulta e especifica a "forma" ou o tipo de cada elemento retornado. Por exemplo, você pode especificar se os resultados consistirão em objetos `Customer` completos, apenas um membro, um subconjunto de membros ou algum tipo de resultado completamente diferente com base em um cálculo ou na criação de um novo objeto. Quando a cláusula `select` produz algo diferente de uma cópia do elemento de origem, a operação é chamada de *projeção*. O uso de projeções para transformar dados é um recurso poderoso de expressões de consulta LINQ. Para obter mais informações, consulte [Transformações de dados com LINQ \(C#\)](#) e [Cláusula select](#).

## Tabela de sintaxe de expressão de consulta

A tabela a seguir lista os operadores de consulta padrão que têm cláusulas de expressão de consulta equivalentes.

Método	Sintaxe de expressão de consulta em C#
<a href="#">Cast</a>	Use uma variável de intervalo de tipo explícito, por exemplo:  <code>from int i in numbers</code>  (Para obter mais informações, consulte <a href="#">Cláusula from</a> .)
<a href="#">GroupBy</a>	<code>group ... by</code>  -ou-  <code>group ... by ... into ...</code>  (Para obter mais informações, consulte <a href="#">Cláusula group</a> .)
<a href="#">GroupJoin&lt;TOuter,TInner,TKey,TResult&gt;(IEnumerable&lt;TOuter&gt;, IEnumerable&lt;TInner&gt;, Func&lt;TOuter,TKey&gt;, Func&lt;TInner,TKey&gt;, Func&lt;TOuter,IEnumerable&lt;TInner&gt;, TResult&gt;)</a>	<code>join ... in ... on ... equals ... into ...</code>  (Para obter mais informações, consulte <a href="#">Cláusula join</a> .)

Método	Sintaxe de expressão de consulta em C#
<code>Join&lt;TOuter,TInner,TKey,TResult&gt;(IEnumerable&lt;TOuter&gt;, IEnumerable&lt;TInner&gt;, Func&lt;TOuter,TKey&gt;, Func&lt;TInner,TKey&gt;, Func&lt;TOuter,TInner,TResult&gt;)</code>	<code>join ... in ... on ... equals ...</code> (Para obter mais informações, consulte <a href="#">Cláusula join</a> .)
<code>OrderBy&lt;TSource,TKey&gt;(IEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</code>	<code>orderby</code> (Para obter mais informações, consulte <a href="#">Cláusula orderby</a> .)
<code>OrderByDescending&lt;TSource,TKey&gt;(IEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</code>	<code>orderby ... descending</code> (Para obter mais informações, consulte <a href="#">Cláusula orderby</a> .)
<code>Select</code>	<code>select</code> (Para obter mais informações, consulte <a href="#">Cláusula select</a> .)
<code>SelectMany</code>	Várias cláusulas <code>from</code> . (Para obter mais informações, consulte <a href="#">Cláusula from</a> .)
<code>ThenBy&lt;TSource,TKey&gt;(IOrderedEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</code>	<code>orderby ..., ...</code> (Para obter mais informações, consulte <a href="#">Cláusula orderby</a> .)
<code>ThenByDescending&lt;TSource,TKey&gt;(IOrderedEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</code>	<code>orderby ..., ... descending</code> (Para obter mais informações, consulte <a href="#">Cláusula orderby</a> .)
<code>Where</code>	<code>where</code> (Para obter mais

Método	Sintaxe de expressão de consulta em C#
	informações, consulte <a href="#">Cláusula where.</a> )

## Confira também

- [Enumerable](#)
- [Queryable](#)
- [Métodos de Extensão](#)
- [Palavras-chave de Consulta \(LINQ\)](#)
- [Tipos anônimos](#)

# Consultar uma coleção de objetos

Artigo • 20/07/2023

O termo "LINQ to Objects" refere-se ao uso de consultas LINQ com qualquer coleção `IEnumerable` ou `IEnumerable<T>` diretamente, sem o uso de uma API ou provedor LINQ intermediário como o [LINQ to SQL](#) ou [LINQ to XML](#). Você pode usar o LINQ para consultar qualquer coleção enumerável como `List<T>`, `Array` ou `Dictionary< TKey, TValue >`. A coleção pode ser definida pelo usuário ou pode ser devolvida por uma API do .NET. Na abordagem da LINQ, você escreve o código declarativo que descreve o que você deseja recuperar.

Além disso, as consultas LINQ oferecem três principais vantagens sobre os loops `foreach` tradicionais:

- Elas são mais concisas e legíveis, especialmente quando você filtra várias condições.
- Elas fornecem poderosos recursos de filtragem, ordenação e agrupamento com um mínimo de código do aplicativo.
- Elas podem ser movidas para outras fontes de dados com pouca ou nenhuma modificação.

Em geral, quanto mais complexa a operação que você deseja executar sobre os dados, maior benefício você perceberá usando consultas LINQs em vez de técnicas tradicionais de iteração.

Este exemplo mostra como executar uma consulta simples em uma lista de objetos `Student`. Cada objeto `Student` contém algumas informações básicas sobre o aluno e uma lista que representa as pontuações do aluno em quatro provas.

## ⓘ Observação

Muitos outros exemplos nesta seção usam a mesma `Student` classe e coleção `students`.

C#

```
class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
```

```
public GradeLevel? Year { get; set; }
public List<int> ExamScores { get; set; }

public Student(string FirstName, string LastName, int ID, GradeLevel
Year, List<int> ExamScores)
{
    this.FirstName = FirstName;
    this.LastName = LastName;
    this.ID = ID;
    this.Year = Year;
    this.ExamScores = ExamScores;
}

public Student(string FirstName, string LastName, int StudentID,
List<int>? ExamScores = null)
{
    this.FirstName = FirstName;
    this.LastName = LastName;
    ID = StudentID;
    this.ExamScores = ExamScores ?? Enumerable.Empty<int>().ToList();
}

public static List<Student> students = new()
{
    new(
        FirstName: "Terry", LastName: "Adams", ID: 120,
        Year: GradeLevel.SecondYear,
        ExamScores: new() { 99, 82, 81, 79 }
    ),
    new(
        "Fadi", "Fakhouri", 116,
        GradeLevel.ThirdYear,
        new() { 99, 86, 90, 94 }
    ),
    new(
        "Hanying", "Feng", 117,
        GradeLevel.FirstYear,
        new() { 93, 92, 80, 87 }
    ),
    new(
        "Cesar", "Garcia", 114,
        GradeLevel.FourthYear,
        new() { 97, 89, 85, 82 }
    ),
    new(
        "Debra", "Garcia", 115,
        GradeLevel.ThirdYear,
        new() { 35, 72, 91, 70 }
    ),
    new(
        "Hugo", "Garcia", 118,
        GradeLevel.SecondYear,
        new() { 92, 90, 83, 78 }
    ),
    new(

```

```

        "Sven", "Mortensen", 113,
        GradeLevel.FirstYear,
        new() { 88, 94, 65, 91 }
    ),
    new(
        "Claire", "O'Donnell", 112,
        GradeLevel.FourthYear,
        new() { 75, 84, 91, 39 }
    ),
    new(
        "Svetlana", "Omelchenko", 111,
        GradeLevel.SecondYear,
        new() { 97, 92, 81, 60 }
    ),
    new(
        "Lance", "Tucker", 119,
        GradeLevel.ThirdYear,
        new() { 68, 79, 88, 92 }
    ),
    new(
        "Michael", "Tucker", 122,
        GradeLevel.FirstYear,
        new() { 94, 92, 91, 91 }
    ),
    new(
        "Eugene", "Zabokritski", 121,
        GradeLevel.FourthYear,
        new() { 96, 85, 91, 60 }
    )
);
}

enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

```

## Exemplo

A consulta a seguir retorna os alunos que receberam uma pontuação de 90 ou mais em sua primeira prova.

C#

```

void QueryHighScores(int exam, int score)
{
    var highScores =
        from student in students

```

```
    where student.ExamScores[exam] > score
    select new
    {
        Name = student.FirstName,
        Score = student.ExamScores[exam]
    };

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name},-{15}{item.Score}");
    }
}

QueryHighScores(1, 90);
```

Essa consulta é intencionalmente simples para que você possa testar. Por exemplo, você pode testar mais condições na cláusula `where` ou usar uma cláusula `orderby` para classificar os resultados.

## Classificação de operadores de consulta padrão por maneira de execução

As implementações de LINQ to Objects dos métodos de operador de consulta padrão em uma das duas maneiras principais: imediata ou adiada. Os operadores de consulta que usam a execução adiada podem ser divididos em mais duas categorias: streaming e não streaming. Se você souber como os operadores de consulta diferentes são executados, isso poderá ajudá-lo a entender os resultados que serão obtidos de uma determinada consulta. Isso é especialmente verdadeiro se a fonte de dados está sendo alterada ou se você estiver criando uma consulta sobre outra consulta. Este tópico classifica os operadores de consulta padrão de acordo com o modo de execução.

### Imediata

A execução imediata significa que a fonte de dados é lida e a operação é executada uma vez. Todos os operadores de consulta padrão que retornam um resultado escalar são executados imediatamente. Você pode forçar uma consulta a ser executada imediatamente usando o método `Enumerable.ToList` ou `Enumerable.ToArray`. A execução imediata fornece reutilização dos resultados da consulta, não uma declaração de consulta. Os resultados são recuperados uma vez e armazenados para uso futuro.

### Adiado

A execução adiada significa que a operação não será realizada no ponto do código em que a consulta estiver declarada. A operação será realizada somente quando a variável de consulta for enumerada, por exemplo, usando uma instrução `foreach`. Isso significa que os resultados da execução da consulta dependerão do conteúdo da fonte de dados quando a consulta for executada em vez de quando a consulta for definida. Se a variável de consulta for enumerada várias vezes, os resultados poderão ser diferentes a cada vez. Quase todos os operadores de consulta padrão cujo tipo de retorno é `IEnumerable<T>` ou `IOrderedEnumerable<TElement>` executam de maneira adiada. A execução adiada oferece a facilidade da reutilização da consulta, uma vez que ela busca os dados atualizados da fonte de dados sempre que os seus resultados são iterados.

Os operadores de consulta que usam a execução adiada podem ser, adicionalmente, classificados como streaming ou não streaming.

## Streaming

Operadores streaming não precisam ler todos os dados de origem antes de gerar elementos. No momento da execução, um operador streaming realiza sua operação em cada elemento de origem enquanto eles são lidos, gerando o elemento, se apropriado. Um operador streaming continua a ler os elementos de origem até que um elemento de resultado possa ser produzido. Isso significa que mais de um elemento de origem poderá ser lido para produzir um elemento de resultado.

## Não streaming

Os operadores não streaming devem ler todos os dados de origem antes de produzirem um elemento de resultado. Operações como classificação ou agrupamento se enquadram nesta categoria. No momento da execução, os operadores de consulta não streaming leem todos os dados de origem, colocam-nos em uma estrutura de dados, realizam a operação e geram os elementos de resultado.

## Tabela de classificação

A tabela a seguir classifica cada método de operador de consulta padrão de acordo com o respectivo método de execução.

### ⓘ Observação

Se um operador estiver marcado em duas colunas, duas sequências de entrada estarão envolvidas na operação e cada sequência será avaliada de forma diferente.

Nesses casos, a primeira sequência na lista de parâmetros é a que sempre será avaliada de maneira adiada e em modo streaming.

<b>Operador de consulta padrão</b>	<b>Tipo de retorno</b>	<b>Execução imediata</b>	<b>Execução adiada de streaming</b>	<b>Execução adiada de não streaming</b>
Aggregate	TSource	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	Valor numérico único	X		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	TSource	X		
ElementAtOrDefault	TSource?		X	
Empty	IEnumerable<T>		X	
Except	IEnumerable<T>		X	X
First	TSource	X		
FirstOrDefault	TSource?		X	
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X
Join	IEnumerable<T>		X	X
Last	TSource	X		

<b>Operador de consulta padrão</b>	<b>Tipo de retorno</b>	<b>Execução imediata</b>	<b>Execução adiada de streaming</b>	<b>Execução adiada de não streaming</b>
LastOrDefault	<code>TSource?</code>	X		
LongCount	<code>Int64</code>	X		
Max	Valor numérico único, <code>TSource</code> OU <code>TResult?</code>	X		
Min	Valor numérico único, <code>TSource</code> OU <code>TResult?</code>	X		
OfType	<code>IEnumerable&lt;T&gt;</code>	X		
OrderBy	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
OrderByDescending	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
Range	<code>IEnumerable&lt;T&gt;</code>	X		
Repeat	<code>IEnumerable&lt;T&gt;</code>	X		
Reverse	<code>IEnumerable&lt;T&gt;</code>		X	
Select	<code>IEnumerable&lt;T&gt;</code>	X		
SelectMany	<code>IEnumerable&lt;T&gt;</code>	X		
SequenceEqual	<code>Boolean</code>	X		
Single	<code>TSource</code>	X		
SingleOrDefault	<code>TSource?</code>	X		
Skip	<code>IEnumerable&lt;T&gt;</code>	X		
SkipWhile	<code>IEnumerable&lt;T&gt;</code>	X		
Sum	Valor numérico único	X		
Take	<code>IEnumerable&lt;T&gt;</code>	X		
TakeWhile	<code>IEnumerable&lt;T&gt;</code>	X		
ThenBy	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
ThenByDescending	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
ToArray	Matriz <code>TSource[]</code>	X		

<b>Operador de consulta padrão</b>	<b>Tipo de retorno</b>	<b>Execução imediata</b>	<b>Execução adiada de streaming</b>	<b>Execução adiada de não streaming</b>
ToDictionary	Dictionary<TKey,TValue>	X		
ToList	IList<T>	X		
ToLookup	ILookup<TKey,TElement>	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

## Confira também

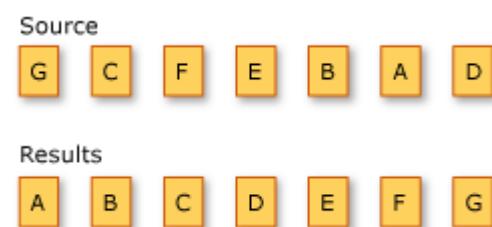
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Interpolação de cadeia de caracteres](#)

# Classificando dados (C#)

Artigo • 07/04/2023

Uma operação de classificação ordena os elementos de uma sequência com base em um ou mais atributos. O primeiro critério de classificação executa uma classificação primária dos elementos. Especificando um segundo critério de classificação, você pode classificar os elementos dentro de cada grupo de classificação primário.

A ilustração a seguir mostra os resultados de uma operação de classificação alfabética em uma sequência de caracteres:



Os métodos de operador de consulta padrão que classificam dados estão listados na seção a seguir.

## Métodos

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
OrderBy	Classifica valores em ordem crescente.	<code>orderby</code>	<a href="#">Enumerable.OrderBy</a> <a href="#">Queryable.OrderBy</a>
OrderByDescending	Classifica valores em ordem decrescente.	<code>orderby ... descending</code>	<a href="#">Enumerable.OrderByDescending</a> <a href="#">Queryable.OrderByDescending</a>
ThenBy	Executa uma classificação secundária em ordem crescente.	<code>orderby ..., ...</code>	<a href="#">Enumerable.ThenBy</a> <a href="#">Queryable.ThenBy</a>
ThenByDescending	Executa uma classificação secundária em ordem decrescente.	<code>orderby ..., ... descending</code>	<a href="#">Enumerable.ThenByDescending</a> <a href="#">Queryable.ThenByDescending</a>

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
Reverse	Inverte a ordem dos elementos em uma coleção.	Não aplicável.	<a href="#">Enumerable.Reverse</a> <a href="#">Queryable.Reverse</a>

## Exemplos de sintaxe de expressão de consulta

### Exemplos de classificação primária

#### Classificação crescente primária

O exemplo a seguir demonstra como usar a cláusula `orderby` em uma consulta de LINQ para classificar as cadeias de caracteres em uma matriz segundo o tamanho da cadeia de caracteres, em ordem crescente.

C#

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                             orderby word.Length
                             select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    brown
    jumps
*/
```

#### Classificação decrescente primária

O exemplo seguinte demonstra como usar a cláusula `orderby descending` em uma consulta de LINQ para classificar as cadeias de caracteres segundo sua primeira letra, em ordem decrescente.

C#

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                             orderby word.Substring(0, 1) descending
                             select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

the
quick
jumps
fox
brown
*/
```

## Exemplos de classificação secundária

### Classificação crescente secundária

O exemplo a seguir demonstra como usar a cláusula `orderby` em uma consulta de LINQ para executar uma classificação primária e uma classificação secundária das cadeias de caracteres em uma matriz. As cadeias de caracteres são classificadas primeiro segundo o tamanho e, depois, segundo a primeira letra da cadeia de caracteres, em ordem crescente.

C#

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                             orderby word.Length, word.Substring(0, 1)
                             select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

fox
the
brown
jumps
quick
*/
```

## Classificação decrescente secundária

O exemplo seguinte demonstra como usar a cláusula `orderby descending` em uma consulta de LINQ para executar uma classificação primária, em ordem crescente e uma classificação secundária, em ordem decrescente. As cadeias de caracteres são classificadas primeiro segundo o tamanho e, depois, segundo a primeira letra da cadeia de caracteres.

C#

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                             orderby word.Length, word.Substring(0, 1)
                             descending
                             select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    jumps
    brown
*/
```

## Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Cláusula orderby](#)
- [Ordenar os resultados de uma cláusula join](#)
- [Como classificar ou filtrar dados de texto por qualquer palavra ou campo \(LINQ\) \(C#\)](#)

# Operações de conjunto (C#)

Artigo • 07/04/2023

As operações de conjunto na LINQ referem-se a operações de consulta que geram um conjunto de resultados baseado na presença ou ausência de elementos equivalentes dentro da mesma ou de coleções (ou conjuntos) separadas.

Os métodos de operador de consulta padrão que executam operações de conjunto estão listados na seção a seguir.

## Métodos

Nomes de método	Descrição	Sintaxe de expressão de consulta em C#	Mais informações
Distinct ou DistinctBy	Remove os valores duplicados de uma coleção.	Não aplicável.	<a href="#">Enumerable.Distinct</a> <a href="#">Enumerable.DistinctBy</a> <a href="#">Queryable.Distinct</a> <a href="#">Queryable.DistinctBy</a>
Except ou ExceptBy	Retorna a diferença de conjunto, que significa os elementos de uma coleção que não aparecem em uma segunda coleção.	Não aplicável.	<a href="#">Enumerable.Except</a> <a href="#">Enumerable.ExceptBy</a> <a href="#">Queryable.Except</a> <a href="#">Queryable.ExceptBy</a>
Intersect ou IntersectBy	Retorna a interseção de conjunto, o que significa os elementos que aparecem em cada uma das duas coleções.	Não aplicável.	<a href="#">Enumerable.Intersect</a> <a href="#">Enumerable.IntersectBy</a> <a href="#">Queryable.Intersect</a> <a href="#">Queryable.IntersectBy</a>
Union ou UnionBy	Retorna a união de conjunto, o que significa os elementos únicos que aparecem em qualquer uma das duas coleções.	Não aplicável.	<a href="#">Enumerable.Union</a> <a href="#">Enumerable.UnionBy</a> <a href="#">Queryable.Union</a> <a href="#">Queryable.UnionBy</a>

## Exemplos

Alguns dos exemplos a seguir dependem de um tipo `record` que representa os planetas em nosso sistema solar.

C#

```

namespace SolarSystem;

record Planet(
    string Name,
    PlanetType Type,
    int OrderFromSun)
{
    public static readonly Planet Mercury =
        new(nameof(Mercury), PlanetType.Rock, 1);

    public static readonly Planet Venus =
        new(nameof(Venus), PlanetType.Rock, 2);

    public static readonly Planet Earth =
        new(nameof(Earth), PlanetType.Rock, 3);

    public static readonly Planet Mars =
        new(nameof(Mars), PlanetType.Rock, 4);

    public static readonly Planet Jupiter =
        new(nameof(Jupiter), PlanetType.Gas, 5);

    public static readonly Planet Saturn =
        new(nameof(Saturn), PlanetType.Gas, 6);

    public static readonly Planet Uranus =
        new(nameof(Uranus), PlanetType.Liquid, 7);

    public static readonly Planet Neptune =
        new(nameof(Neptune), PlanetType.Liquid, 8);

    // Yes, I know... not technically a planet anymore
    public static readonly Planet Pluto =
        new(nameof(Pluto), PlanetType.Ice, 9);
}

```

O `record Planet` é um registro de posição, e requer argumentos `Name`, `Type` e `OrderFromSun` para instanciá-lo. Há várias instâncias de planeta `static readonly` no tipo `Planet`. Estas são definições baseadas em conveniência para planetas conhecidos. O membro `Type` identifica o tipo de planeta.

C#

```

namespace SolarSystem;

enum PlanetType
{
    Rock,
    Ice,
    Gas,

```

```
    Liquid  
};
```

## Distinct e DistinctBy

O exemplo a seguir descreve o comportamento do método `Enumerable.Distinct` em uma sequência de cadeias de caracteres. A sequência retornada contém os elementos exclusivos da sequência de entrada.

```
a, b, b, c, d, c → a, b, c, d
```

C#

```
string[] planets = { "Mercury", "Venus", "Venus", "Earth", "Mars", "Earth" };  
  
IEnumerable<string> query = from planet in planets.Distinct()  
                               select planet;  
  
foreach (var str in query)  
{  
    Console.WriteLine(str);  
}  
  
/* This code produces the following output:  
 *  
 * Mercury  
 * Venus  
 * Earth  
 * Mars  
 */
```

`DistinctBy` é uma abordagem alternativa a `Distinct` que usa `keySelector`. `keySelector` é usado como o discriminador comparativo do tipo de origem. Considere a seguinte matriz de planeta:

C#

```
Planet[] planets =  
{  
    Planet.Mercury,  
    Planet.Venus,  
    Planet.Earth,  
    Planet.Mars,  
    Planet.Jupiter,  
    Planet.Saturn,  
    Planet.Uranus,
```

```
    Planet.Neptune,  
    Planet.Pluto  
};
```

No código a seguir, os planetas são discriminados com base no seu `PlanetType`, e o primeiro planeta de cada tipo é exibido:

C#

```
foreach (Planet planet in planets.DistinctBy(p => p.Type))  
{  
    Console.WriteLine(planet);  
}  
  
// This code produces the following output:  
//     Planet { Name = Mercury, Type = Rock, OrderFromSun = 1 }  
//     Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }  
//     Planet { Name = Uranus, Type = Liquid, OrderFromSun = 7 }  
//     Planet { Name = Pluto, Type = Ice, OrderFromSun = 9 }
```

No código anterior do C#:

- A matriz `Planet` é filtrada distintamente para a primeira ocorrência de cada tipo de planeta exclusivo.
- As instâncias resultantes `planet` são gravadas no console.

## Except e ExceptBy

O exemplo a seguir descreve o comportamento de `Enumerable.Except`. A sequência retornada contém apenas os elementos da primeira sequência de entrada que não estão na segunda sequência de entrada.



C#

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };  
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };  
  
IEnumerable<string> query = from planet in planets1.Except(planets2)  
                                select planet;  
  
foreach (var str in query)  
{
```

```
        Console.WriteLine(str);
    }

/* This code produces the following output:
 *
 * Venus
 */
```

O método `ExceptBy` é uma abordagem alternativa a `Except` que usa duas sequências de tipos possivelmente heterogêneos e um `keySelector`. O `keySelector` é o mesmo tipo que o tipo da segunda coleção, e é usado como o discriminador comparativo do tipo de origem. Considere a seguinte matriz de planetas:

```
C#

Planet[] planets =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Jupiter
};

Planet[] morePlanets =
{
    Planet.Mercury,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};
```

Para encontrar planetas na primeira coleção que não estão na segunda coleção, você pode projetar os nomes de planeta como a coleção `second` e fornecer o mesmo `keySelector`:

```
C#

// A shared "keySelector"
static string PlanetNameSelector(Planet planet) => planet.Name;

foreach (Planet planet in
    planets.ExceptBy(
        morePlanets.Select(PlanetNameSelector), PlanetNameSelector))
{
    Console.WriteLine(planet);

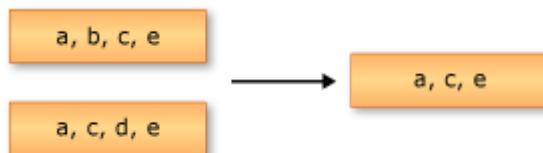
// This code produces the following output:
//      Planet { Name = Venus, Type = Rock, OrderFromSun = 2 }
```

No código anterior do C#:

- A função `keySelector` é definida como uma função local `static` que discrimina um nome de planeta.
- A primeira matriz de planetas é filtrada para planetas não encontrados na segunda matriz de planetas, com base nos nomes.
- As instâncias resultantes `planet` são gravadas no console.

## Intersect e IntersectBy

O exemplo a seguir descreve o comportamento de `Enumerable.Intersect`. A sequência retornada contém os elementos que são comuns a ambas as sequências de entrada.



C#

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Intersect(planets2)
                                select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Earth
 * Jupiter
 */
```

O método `IntersectBy` é uma abordagem alternativa a `Intersect` que usa duas sequências de tipos possivelmente heterogêneos e um `keySelector`. O `keySelector` é usado como o discriminatório comparativo do tipo da segunda coleção. Considere a seguinte matriz de planetas:

C#

```

Planet[] firstFivePlanetsFromTheSun =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};

Planet[] lastFivePlanetsFromTheSun =
{
    Planet.Mars,
    Planet.Jupiter,
    Planet.Saturn,
    Planet.Uranus,
    Planet.Neptune
};

```

Há duas matrizes de planetas; a primeira representa os cinco primeiros planetas a partir do Sol, e a segunda representa os últimos cinco planetas a partir do Sol. Como o tipo `Planet` é um tipo posicional `record`, você pode usar sua semântica de comparação de valor na forma do `keySelector`:

C#

```

foreach (Planet planet in
    firstFivePlanetsFromTheSun.IntersectBy(
        lastFivePlanetsFromTheSun, planet => planet))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
//      Planet { Name = Mars, Type = Rock, OrderFromSun = 4 }
//      Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }

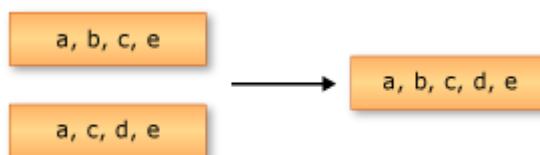
```

No código anterior do C#:

- As duas matrizes `Planet` são interseccionadas por sua semântica de comparação de valor.
- Somente os planetas encontrados em ambas as matrizes estão presentes na sequência resultante.
- As instâncias resultantes `planet` são gravadas no console.

## Union e UnionBy

O exemplo a seguir descreve uma operação de união em duas sequências de cadeias de caracteres. A sequência retornada contém os elementos exclusivos das duas sequências de entrada.



C#

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Union(planets2)
                               select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Jupiter
 * Mars
 */
```

O método `UnionBy` é uma abordagem alternativa a `Union` que usa duas sequências do mesmo tipo e uma `keySelector`. `keySelector` é usado como o discriminador comparativo do tipo de origem. Considere a seguinte matriz de planetas:

C#

```
Planet[] firstFivePlanetsFromTheSun =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};

Planet[] lastFivePlanetsFromTheSun =
{
    Planet.Mars,
    Planet.Jupiter,
```

```
    Planet.Saturn,  
    Planet.Uranus,  
    Planet.Neptune  
};
```

Para unir essas duas coleções em uma única sequência, você fornece `keySelector`:

C#

```
foreach (Planet planet in  
    firstFivePlanetsFromTheSun.UnionBy(  
        lastFivePlanetsFromTheSun, planet => planet))  
{  
    Console.WriteLine(planet);  
}  
  
// This code produces the following output:  
//     Planet { Name = Mercury, Type = Rock, OrderFromSun = 1 }  
//     Planet { Name = Venus, Type = Rock, OrderFromSun = 2 }  
//     Planet { Name = Earth, Type = Rock, OrderFromSun = 3 }  
//     Planet { Name = Mars, Type = Rock, OrderFromSun = 4 }  
//     Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }  
//     Planet { Name = Saturn, Type = Gas, OrderFromSun = 6 }  
//     Planet { Name = Uranus, Type = Liquid, OrderFromSun = 7 }  
//     Planet { Name = Neptune, Type = Liquid, OrderFromSun = 8 }
```

No código anterior do C#:

- As duas matrizes `Planet` são tecidas juntas usando a semântica `record` de comparação de valores.
- As instâncias resultantes `planet` são gravadas no console.

## Confira também

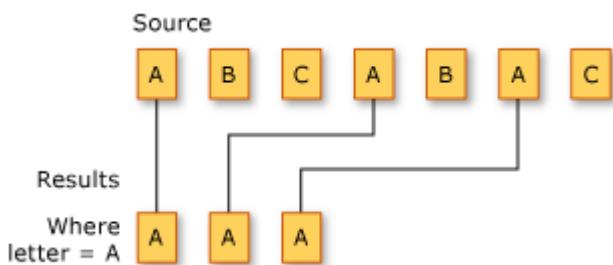
- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Como combinar e comparar coleções de cadeias de caracteres \(LINQ\) \(C#\)](#)
- [Como localizar a diferença de conjunto entre duas listas \(LINQ\) \(C#\)](#)

# Filtrando dados (C#)

Artigo • 16/03/2023

A filtragem é a operação de restringir o conjunto de resultados de forma que ele contenha apenas os elementos correspondentes a uma condição especificada. Ela também é conhecida como seleção.

A ilustração a seguir mostra os resultados da filtragem de uma sequência de caracteres. O predicado para a operação de filtragem especifica que o caractere deve ser "A".



Os métodos de operador de consulta padrão que realizam a seleção estão listados na seção a seguir.

## Métodos

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
OfType	Seleciona valores, dependendo da capacidade de serem convertidos em um tipo especificado.	Não aplicável.	<a href="#">Enumerable.OfType</a> <a href="#">Queryable.OfType</a>
Where	Seleciona valores com base em uma função de predicado.	<code>where</code>	<a href="#">Enumerable.Where</a> <a href="#">Queryable.Where</a>

## Exemplo de sintaxe de expressão de consulta

O exemplo a seguir usa a cláusula `where` para filtrar em uma matriz as cadeias de caracteres com um tamanho específico.

C#

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };
```

```
IEnumerable<string> query = from word in words
                             where word.Length == 3
                             select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
*/
```

## Confira também

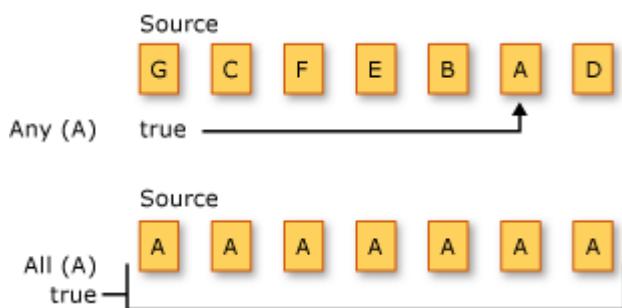
- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Cláusula where](#)
- [Especificar filtros de predicado dinamicamente em tempo de execução](#)
- [Como consultar metadados de um assembly com Reflexão \(LINQ\) \(C#\)](#)
- [Como consultar arquivos com um atributo ou nome especificado \(C#\)](#)
- [Como classificar ou filtrar dados de texto por qualquer palavra ou campo \(LINQ\) \(C#\)](#)

# Operações do quantificador no LINQ (C#)

Artigo • 07/04/2023

As operações de quantificador retornam um valor [Boolean](#) que indica se alguns ou todos os elementos em uma sequência satisfazem uma condição.

A ilustração a seguir mostra duas operações de quantificador diferentes em duas sequências de origem diferentes. A primeira operação pergunta se algum dos elementos é o caractere 'A'. A segunda operação pergunta se todos os elementos são o caractere 'A'. Ambos os métodos retornam `true` neste exemplo.



Os métodos de operador de consulta padrão que realizam operações de quantificador estão listados na seção a seguir.

## Métodos

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
Todo	Determina se todos os elementos em uma sequência satisfazem uma condição.	Não aplicável.	<a href="#">Enumerable.All</a> <a href="#">Queryable.All</a>
Qualquer	Determina se todos os elementos em uma sequência satisfazem uma condição.	Não aplicável.	<a href="#">Enumerable.Any</a> <a href="#">Queryable.Any</a>
Contém	Determina se uma sequência contém um elemento especificado.	Não aplicável.	<a href="#">Enumerable.Contains</a> <a href="#">Queryable.Contains</a>

## Exemplos de sintaxe de expressão de consulta

## Tudo

O exemplo a seguir usa `All` para verificar se todas as cadeias de caracteres têm comprimento específico.

C#

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi",
"cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon",
"mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi",
"apple", "orange" } },
    };

    // Determine which market have all fruit names length equal to 5
    IEnumerable<string> names = from market in markets
                                  where market.Items.All(item => item.Length
== 5)
                                  select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
}
```

## Qualquer

O exemplo a seguir usa `Any` para verificar se todas as cadeias de caracteres são iniciadas com 'o'.

C#

```
class Market
{
```

```

        public string Name { get; set; }
        public string[] Items { get; set; }
    }

    public static void Example()
    {
        List<Market> markets = new List<Market>
        {
            new Market { Name = "Emily's", Items = new string[] { "kiwi",
"cheery", "banana" } },
            new Market { Name = "Kim's", Items = new string[] { "melon",
"mango", "olive" } },
            new Market { Name = "Adam's", Items = new string[] { "kiwi",
"apple", "orange" } },
        };

        // Determine which market have any fruit names start with 'o'
        IEnumerable<string> names = from market in markets
                                      where market.Items.Any(item =>
item.StartsWith("o"))
                                      select market.Name;

        foreach (string name in names)
        {
            Console.WriteLine($"{name} market");
        }

        // This code produces the following output:
        //
        // Kim's market
        // Adam's market
    }
}

```

## Contém

O exemplo a seguir usa `Contains` para verificar se uma matriz tem um elemento específico.

C#

```

class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi",

```

```

    "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon",
    "mango", "olive" } },
            new Market { Name = "Adam's", Items = new string[] { "kiwi",
    "apple", "orange" } },
        };
}

// Determine which market contains fruit names equal 'kiwi'
IEnumerable<string> names = from market in markets
                                where market.Items.Contains("kiwi")
                                select market.Name;

foreach (string name in names)
{
    Console.WriteLine($"{name} market");
}

// This code produces the following output:
//
// Emily's market
// Adam's market
}

```

## Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Especificar filtros de predicado dinamicamente em tempo de execução](#)
- [Como consultar sentenças que contenham um conjunto especificado de palavras \(LINQ\) \(C#\)](#)

# Operações de projeção (C#)

Artigo • 07/04/2023

Projeção refere-se à operação de transformar um objeto em um novo formulário que geralmente consiste apenas nas propriedades que serão usadas posteriormente. Usando a projeção, você pode construir um novo tipo que é criado de cada objeto. É possível projetar uma propriedade e executar uma função matemática nela. Também é possível projetar o objeto original sem alterá-lo.

Os métodos de operador de consulta padrão que realizam a projeção estão listados na seção a seguir.

## Métodos

Nomes de método	Descrição	Sintaxe de expressão de consulta em C#	Mais informações
Selecionar	Projeta valores com base em uma função de transformação.	<code>select</code>	<a href="#">Enumerable.Select</a> <a href="#">Queryable.Select</a>
SelectMany	Projeta sequências de valores baseados em uma função de transformação e os mescla em uma sequência.	Use várias cláusulas <code>from</code>	<a href="#">Enumerable.SelectMany</a> <a href="#">Queryable.SelectMany</a>
Zip	Produz uma sequência de tuplas com elementos das 2 a 3 sequências especificadas.	Não aplicável.	<a href="#">Enumerable.Zip</a> <a href="#">Queryable.Zip</a>

### Select

O exemplo a seguir usa a cláusula `select` para projetar a primeira letra de cada cadeia de caracteres em uma lista de cadeias de caracteres.

C#

```
List<string> words = new() { "an", "apple", "a", "day" };

var query = from word in words
            select word.Substring(0, 1);

foreach (string s in query)
    Console.WriteLine(s);
```

```
/* This code produces the following output:
```

```
 a  
 a  
 a  
 d  
 */
```

## SelectMany

O exemplo a seguir usa várias cláusulas `from` para projetar cada palavra de cada cadeia de caracteres em uma lista de cadeias de caracteres.

```
C#
```

```
List<string> phrases = new() { "an apple a day", "the quick brown fox" };

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
 */
```

## Zip

Há várias sobrecargas para o operador de projeção `Zip`. Todos os métodos `Zip` funcionam em sequências de dois ou mais tipos possivelmente heterogêneos. As duas primeiras sobrecargas retornam tuplas com o tipo posicional correspondente das sequências fornecidas.

Considere as seguintes coleções:

```
C#
```

```
// An int array with 7 elements.
IEnumerable<int> numbers = new[]
{
    1, 2, 3, 4, 5, 6, 7
};
// A char array with 6 elements.
IEnumerable<char> letters = new[]
{
    'A', 'B', 'C', 'D', 'E', 'F'
};
```

Para projetar essas sequências em conjunto, use o operador `Enumerable.Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>)`:

C#

```
foreach ((int number, char letter) in numbers.Zip(letters))
{
    Console.WriteLine($"Number: {number} zipped with letter: '{letter}'");
}
// This code produces the following output:
//      Number: 1 zipped with letter: 'A'
//      Number: 2 zipped with letter: 'B'
//      Number: 3 zipped with letter: 'C'
//      Number: 4 zipped with letter: 'D'
//      Number: 5 zipped with letter: 'E'
//      Number: 6 zipped with letter: 'F'
```

### ⓘ Importante

A sequência resultante de uma operação zip nunca tem mais comprimento do que a sequência mais curta. As coleções `numbers` e `letters` diferem em comprimento e a sequência resultante omite o último elemento da coleção `numbers`, pois ela não possui nada com o que zipar.

A segunda sobrecarga aceita uma sequência `third`. Vamos criar outra coleção, ou seja, `emoji`:

C#

```
// A string array with 8 elements.
IEnumerable<string> emoji = new[]
{
    "👀", "🔥", "🎉", "👀", "⭐", "❤️", "✓", "💯"
};
```

Para projetar essas sequências em conjunto, use o operador `Enumerable.Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)`:

C#

```
foreach ((int number, char letter, string emoji) in numbers.Zip(letters,
emoji))
{
    Console.WriteLine(
        $"Number: {number} is zipped with letter: '{letter}' and emoji:
{emoji}");
}

// This code produces the following output:
// Number: 1 is zipped with letter: 'A' and emoji: 😊
// Number: 2 is zipped with letter: 'B' and emoji: 🔥
// Number: 3 is zipped with letter: 'C' and emoji: 🎉
// Number: 4 is zipped with letter: 'D' and emoji: 🕸️
// Number: 5 is zipped with letter: 'E' and emoji: ⭐
// Number: 6 is zipped with letter: 'F' and emoji: 💜
```

Assim como a sobrecarga anterior, o método `Zip` projeta uma tupla, mas desta vez com três elementos.

A terceira sobrecarga aceita um argumento `Func<TFirst, TSecond, TResult>` que atua como um seletor de resultados. Considerando os dois tipos das sequências que estão sendo compactadas, você pode projetar uma nova sequência resultante.

C#

```
foreach (string result in
    numbers.Zip(letters, (number, letter) => $"{number} = {letter}
({(int)letter})"))
{
    Console.WriteLine(result);
}

// This code produces the following output:
// 1 = A (65)
// 2 = B (66)
// 3 = C (67)
// 4 = D (68)
// 5 = E (69)
// 6 = F (70)
```

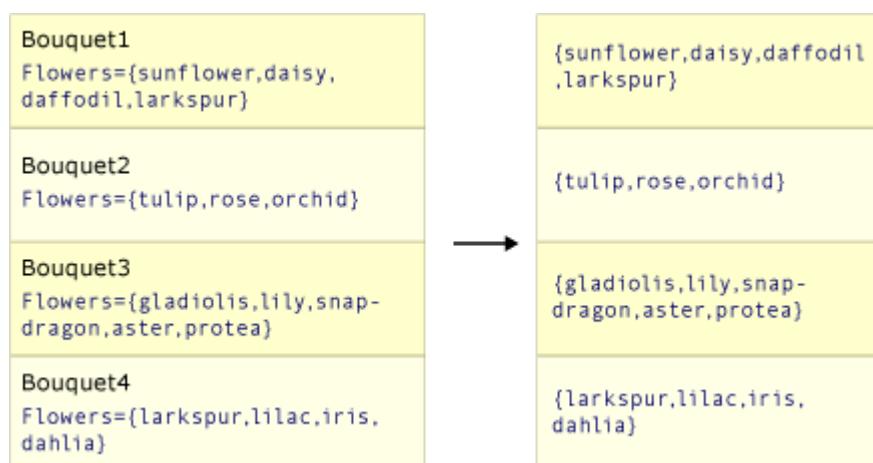
Com a sobrecarga `Zip` anterior, a função especificada é aplicada aos elementos correspondentes `numbers` e `letter`, produzindo uma sequência dos resultados `string`.

## Select versus SelectMany

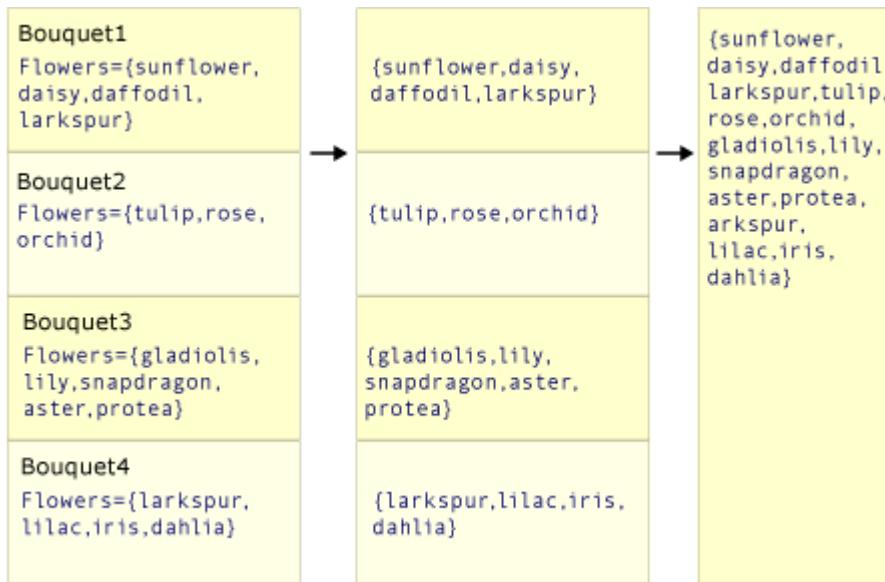
O trabalho de `Select` e `SelectMany` é produzir um valor (ou valores) de resultado dos valores de origem. `Select` produz um valor de resultado para cada valor de origem. O resultado geral, portanto, é uma coleção que tem o mesmo número de elementos que a coleção de origem. Por outro lado, `SelectMany` produz um único resultado geral que contém subcoleções concatenadas de cada valor de origem. A função de transformação passada como um argumento para `SelectMany` deve retornar uma sequência enumerável de valores para cada valor de origem. Essas sequências enumeráveis, então, são concatenadas por `SelectMany` para criar uma sequência grande.

As duas ilustrações a seguir mostram a diferença conceitual entre as ações desses dois métodos. Em cada caso, presuma que a função de seletor (transformação) seleciona a matriz de flores de cada valor de origem.

A ilustração mostra como `Select` retorna uma coleção que tem o mesmo número de elementos que a coleção de origem.



Esta ilustração mostra como `SelectMany` concatena a sequência intermediária de matrizes em um valor de resultado final que contém cada valor de cada matriz intermediária.



## Exemplo de código

O exemplo a seguir compara o comportamento de `Select` e de `SelectMany`. O código cria um "buquê" de flores tirando os itens de cada lista de nomes de flores na coleção de origem. Neste exemplo, o "valor único" que a função de transformação `Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)` usa é uma coleção de valores. Isso requer o loop `foreach` extra para enumerar cada cadeia de caracteres em cada subsequência.

C#

```

class Bouquet
{
    public List<string> Flowers { get; set; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets = new()
    {
        new Bouquet { Flowers = new List<string> { "sunflower", "daisy",
"larkspur" } },
        new Bouquet { Flowers = new List<string> { "tulip", "rose", "orchid" } },
        new Bouquet { Flowers = new List<string> { "gladiolus", "lily",
"snapdragon", "aster", "protea" } },
        new Bouquet { Flowers = new List<string> { "larkspur", "lilac",
"iris", "dahlia" } }
    };

    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);
}

```

```

Console.WriteLine("Results by using Select():");
// Note the extra foreach loop here.
foreach (IEnumerable<String> collection in query1)
    foreach (string item in collection)
        Console.WriteLine(item);

Console.WriteLine("\nResults by using SelectMany():");
foreach (string item in query2)
    Console.WriteLine(item);

/* This code produces the following output:

Results by using Select():
sunflower
daisy
daffodil
larkspur
tulip
rose
orchid
gladiolus
lily
snapdragon
aster
protea
larkspur
lilac
iris
dahlia

Results by using SelectMany():
sunflower
daisy
daffodil
larkspur
tulip
rose
orchid
gladiolus
lily
snapdragon
aster
protea
larkspur
lilac
iris
dahlia
*/
}

```

## Confira também

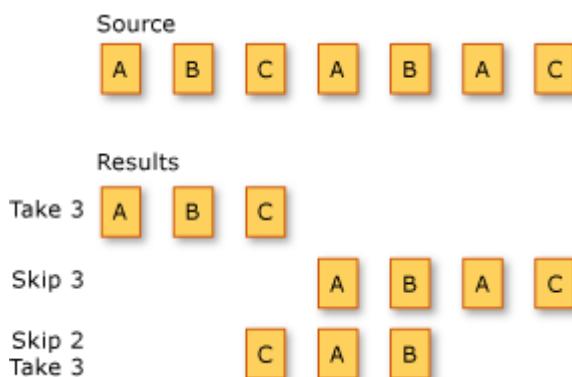
- System.Linq
- Visão geral de operadores de consulta padrão (C#)
- Cláusula select
- Como preencher coleções de objetos de várias fontes (LINQ) (C#)
- Como dividir um arquivo em vários arquivos usando grupos (LINQ) (C#)

# Como particionar dados (C#)

Artigo • 07/04/2023

Particionamento em LINQ refere-se à operação de dividir uma sequência de entrada em duas seções sem reorganizar os elementos e, depois, retornar uma das seções.

A ilustração a seguir mostra os resultados de três operações de particionamento diferentes em uma sequência de caracteres. A primeira operação retorna os três primeiros elementos na sequência. A segunda operação ignora os três primeiros elementos e retorna os elementos restantes. A terceira operação ignora os dois primeiros elementos na sequência e retorna os três elementos seguintes.



Os métodos de operador de consulta padrão que particionam sequências estão listados na seção a seguir.

## Operadores

Nomes de método	Descrição	Sintaxe de expressão de consulta em C#	Mais informações
Ignorar	Ignora elementos até uma posição especificada na sequência.	Não aplicável.	<a href="#">Enumerable.Skip</a> <a href="#">Queryable.Skip</a>
SkipWhile	Ignora elementos com base em uma função de predicado até que um elemento não satisfaça a condição.	Não aplicável.	<a href="#">Enumerable.SkipWhile</a> <a href="#">Queryable.SkipWhile</a>
Take	Aceita elementos até uma posição especificada na sequência.	Não aplicável.	<a href="#">Enumerable.Take</a> <a href="#">Queryable.Take</a>
TakeWhile	Aceita elementos com base em uma função de predicado até que um elemento não satisfaça a condição.	Não aplicável.	<a href="#">Enumerable.TakeWhile</a> <a href="#">Queryable.TakeWhile</a>

Nomes de método	Descrição	Sintaxe de expressão de consulta em C#	Mais informações
Chunk	Divide os elementos de uma sequência em partes com tamanho máximo especificado.	Não aplicável.	<a href="#">Enumerable.Chunk</a> <a href="#">Queryable.Chunk</a>

## Exemplo

O operador `Chunk` é usado para dividir elementos de uma sequência com base em determinado `size`.

```
C#
int chunkNumber = 1;
foreach (int[] chunk in Enumerable.Range(0, 8).Chunk(3))
{
    Console.WriteLine($"Chunk {chunkNumber++}:");
    foreach (int item in chunk)
    {
        Console.WriteLine($"      {item}");
    }

    Console.WriteLine();
}
// This code produces the following output:
// Chunk 1:
//      0
//      1
//      2
//
//Chunk 2:
//      3
//      4
//      5
//
//Chunk 3:
//      6
//      7
```

O código anterior do C#:

- Depende de `Enumerable.Range(Int32, Int32)` para gerar uma sequência de números.
- Aplica o operador `Chunk`, dividindo a sequência em partes com tamanho máximo igual a três.

## Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)

# Operações Join (C#)

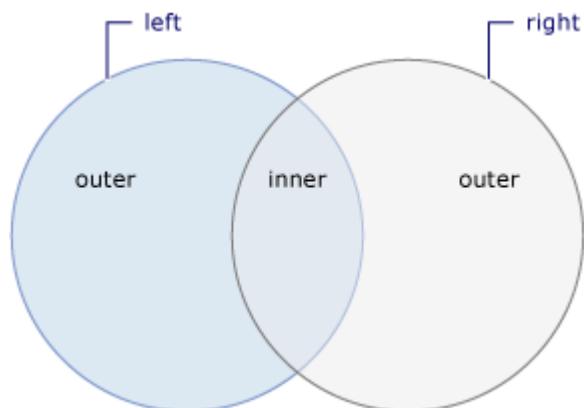
Artigo • 28/04/2023

Uma *junção* de duas fontes de dados é a associação de objetos em uma fonte de dados, com objetos que compartilham um atributo comum em outra fonte de dados.

Join é uma operação importante em consultas que têm como destino fontes de dados cujas relações entre si não podem ser seguidas diretamente. Na programação orientada a objeto, isso pode significar uma correlação entre objetos que não são modelados, como a direção retroativa de uma relação unidirecional. Um exemplo de uma relação unidirecional é uma classe Cliente que tem uma propriedade do tipo Cidade, mas a classe Cidade ainda não tem uma propriedade que é uma coleção de objetos Cliente. Se você tem uma lista de objetos Cidade e você quer encontrar todos os clientes em cada cidade, você pode usar uma operação de junção para encontrá-los.

Os métodos de junção fornecidos na estrutura do LINQ são [Join](#) e [GroupJoin](#). Esses métodos executam junção por igualdade ou junções que correspondem duas fontes de dados com base na igualdade de suas chaves. (Para comparação, o Transact-SQL dá suporte a operadores de junção diferentes de 'equals', por exemplo, o operador 'less than'). Em termos de banco de dados relacional, [Join](#) implementa uma junção interna, um tipo de junção no qual apenas os objetos que têm uma correspondência no outro conjunto de dados são retornados. O método [GroupJoin](#) não tem equivalente direto em termos de banco de dados relacional, mas ele implementa um superconjunto de junções internas e junções externas esquerdas. Uma junção externa esquerda é uma junção que retorna cada elemento da primeira (esquerda) fonte de dados, mesmo que ele não tenha elementos correlacionados na outra fonte de dados.

A ilustração a seguir mostra uma visão conceitual de dois conjuntos e os elementos dentro desses conjuntos que estão incluídos em uma junção interna ou externa à esquerda.



# Métodos

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
Join	Faz Join de duas sequências com base nas funções de seletor de chave e extrai pares de valores.	join ... in ... on ... equals ...	Enumerable.Join Queryable.Join
GroupJoin	Faz Join de duas sequências baseadas em funções de seletor de chave e agrupa as correspondências resultantes para cada elemento.	join ... in ... on ... equals ... into ...	Enumerable.GroupJoin Queryable.GroupJoin

## Exemplos de sintaxe de expressão de consulta

### Join

O exemplo a seguir usa a cláusula `join ... in ... on ... equals ...` para unir duas sequências com base em um valor específico:

```
C#  
  
class Product  
{  
    public string? Name { get; set; }  
    public int CategoryId { get; set; }  
}  
  
class Category  
{  
    public int Id { get; set; }  
    public string? CategoryName { get; set; }  
}  
  
public static void Example()  
{  
    List<Product> products = new List<Product>  
    {  
        new Product { Name = "Cola", CategoryId = 0 },  
        new Product { Name = "Tea", CategoryId = 0 },  
        new Product { Name = "Apple", CategoryId = 1 },  
        new Product { Name = "Kiwi", CategoryId = 1 },  
        new Product { Name = "Carrot", CategoryId = 2 },  
    };  
}
```

```

List<Category> categories = new List<Category>
{
    new Category { Id = 0, CategoryName = "Beverage" },
    new Category { Id = 1, CategoryName = "Fruit" },
    new Category { Id = 2, CategoryName = "Vegetable" }
};

// Join products and categories based on CategoryId
var query = from product in products
            join category in categories on product.CategoryId equals
category.Id
            select new { product.Name, category.CategoryName };

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.CategoryName}");
}

// This code produces the following output:
//
// Cola - Beverage
// Tea - Beverage
// Apple - Fruit
// Kiwi - Fruit
// Carrot - Vegetable
}

```

## GroupJoin

O exemplo a seguir usa a cláusula `join ... in ... on ... equals ... into ...` para unir duas sequências com base em um valor específico e agrupa as correspondências resultantes para cada elemento:

C#

```

class Product
{
    public string? Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string? CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {

```

```

        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join categories and product based on CategoryId and grouping result
    var productGroups = from category in categories
                        join product in products on category.Id equals
    product.CategoryId into productGroup
                        select productGroup;

    foreach (IEnumerable<Product> productGroup in productGroups)
    {
        Console.WriteLine("Group");
        foreach (Product product in productGroup)
        {
            Console.WriteLine($"{product.Name,8}");
        }
    }

    // This code produces the following output:
    //
    // Group
    //     Cola
    //     Tea
    // Group
    //     Apple
    //     Kiwi
    // Group
    //     Carrot
}

```

## Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Tipos anônimos](#)
- [Formular Join e consultas entre produtos](#)
- [Cláusula join](#)
- [Fazer Join usando chaves compostas](#)
- [Como unir conteúdo de arquivos diferentes \(LINQ\) \(C#\)](#)

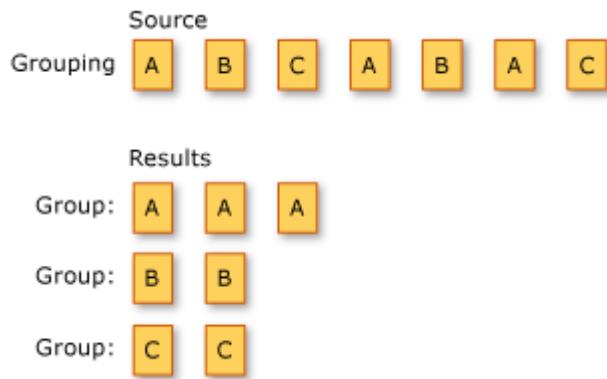
- Ordenar os resultados de uma cláusula join
- Executar operações de junção personalizadas
- Executar junções agrupadas
- Executar junções internas
- Executar junções externas esquerdas
- Como preencher coleções de objetos de várias fontes (LINQ) (C#)

# Agrupando dados (C#)

Artigo • 08/06/2023

O agrupamento refere-se à operação de colocação de dados em grupos, de modo que os elementos em cada grupo compartilhem um atributo comum.

A ilustração a seguir mostra os resultados do agrupamento de uma sequência de caracteres. A chave para cada grupo é o caractere.



Os métodos do operador de consulta padrão que agrupam elementos de dados estão listados na seção a seguir.

## Métodos

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
GroupBy	Agrupa elementos que compartilham um atributo comum. Cada grupo é representado por um objeto <a href="#">IGrouping&lt;TKey, TElement&gt;</a> .	<code>group ... by</code> -ou- <code>group ... by ... into ...</code>	<a href="#">Enumerable.GroupBy</a> <a href="#">Queryable.GroupBy</a>
ToLookup	Insere os elementos em um <a href="#">Lookup&lt;TKey, TElement&gt;</a> (um dicionário one-to-many) com base em uma função de seletor de chave.	Não aplicável.	<a href="#">Enumerable.ToLookup</a>

## Exemplo de sintaxe de expressão de consulta

O seguinte exemplo de código usa a cláusula `group by` para agrupar inteiros em uma lista de acordo com se eles são pares ou ímpares.

C#

```
List<int> numbers = new List<int>() { 35, 44, 200, 84, 3987, 4, 199, 329, 446, 208 };

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                             group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd
numbers:");
    foreach (int i in group)
        Console.WriteLine(i);
}

/* This code produces the following output:

Odd numbers:
35
3987
199
329

Even numbers:
44
200
84
4
446
208
*/
```

## Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Cláusula group](#)
- [Criar um grupo aninhado](#)
- [Como agrupar arquivos por extensão \(LINQ\) \(C#\)](#)
- [Agrupar resultados de consultas](#)
- [Executar uma subconsulta em uma operação de agrupamento](#)
- [Como dividir um arquivo em vários arquivos usando grupos \(LINQ\) \(C#\)](#)

# Operações de geração (C#)

Artigo • 07/04/2023

Geração refere-se à criação de uma nova sequência de valores.

Os métodos de operador de consulta padrão que realizam a geração estão listados na seção a seguir.

## Métodos

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
DefaultIfEmpty	Substitui uma coleção vazia por uma coleção de singletons com valor padrão.	Não aplicável.	<a href="#">Enumerable.DefaultIfEmpty</a> <a href="#">Queryable.DefaultIfEmpty</a>
Vazio	Retorna uma coleção vazia.	Não aplicável.	<a href="#">Enumerable.Empty</a>
Intervalo	Gera uma coleção que contém uma sequência de números.	Não aplicável.	<a href="#">Enumerable.Range</a>
Repetir	Gera uma coleção que contém um valor repetido.	Não aplicável.	<a href="#">Enumerable.Repeat</a>

## Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)

# Operações de Igualdade (C#)

Artigo • 07/04/2023

Duas sequências cujos elementos correspondentes são iguais e que têm o mesmo número de elementos são consideradas iguais.

## Métodos

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
SequenceEqual	Determina se duas sequências são iguais comparando seus elementos por pares.	Não aplicável.	<a href="#">Enumerable.SequenceEqual</a> <a href="#">Queryable.SequenceEqual</a>

## Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Como comparar o conteúdo de duas pastas \(LINQ\) \(C#\)](#)

# Operações de elemento (C#)

Artigo • 07/04/2023

Operações de elemento retornam um único elemento específico de uma sequência.

Os métodos de operador de consulta padrão que executam operações de elemento estão listados na seção a seguir.

## Métodos

Nome do método	Descrição	Sintaxe de expressão de consulta em C#	Mais informações
ElementAt	Retorna o elemento em um índice especificado em uma coleção.	Não aplicável.	<a href="#">Enumerable.ElementAt</a> <a href="#">Queryable.ElementAt</a>
ElementAtOrDefault	Retorna o elemento em um índice especificado em uma coleção ou um valor padrão se o índice estiver fora do intervalo.	Não aplicável.	<a href="#">Enumerable.ElementAtOrDefault</a> <a href="#">Queryable.ElementAtOrDefault</a>
Primeiro	Retorna o primeiro elemento de uma coleção ou o primeiro elemento que satisfaz uma condição.	Não aplicável.	<a href="#">Enumerable.First</a> <a href="#">Queryable.First</a>
FirstOrDefault	Retorna o primeiro elemento de uma coleção ou o primeiro elemento que satisfaz uma condição. Retorna um valor padrão se esse elemento não existir.	Não aplicável.	<a href="#">Enumerable.FirstOrDefault</a> <a href="#">Queryable.FirstOrDefault</a> <a href="#">Queryable.FirstOrDefault&lt;TSource&gt;</a> <a href="#">IQueryable&lt;TSource&gt;</a>
Último	Retorna o último elemento de uma coleção ou o último elemento que satisfaz uma condição.	Não aplicável.	<a href="#">Enumerable.Last</a> <a href="#">Queryable.Last</a>

<b>Nome do método</b>	<b>Descrição</b>	<b>Sintaxe de expressão de consulta em C#</b>	<b>Mais informações</b>
LastOrDefault	<p>Retorna o último elemento de uma coleção ou o último elemento que satisfaz uma condição.</p> <p>Retorna um valor padrão se esse elemento não existir.</p>	Não aplicável.	<a href="#">Enumerable.LastOrDefault</a> <a href="#">Queryable.LastOrDefault</a>
Single	<p>Retorna o único elemento de uma coleção ou o único elemento que satisfaz uma condição.</p> <p>Gera um <a href="#">InvalidOperationException</a> se não houver elemento ou mais de um elemento a ser retornado.</p>	Não aplicável.	<a href="#">Enumerable.Single</a> <a href="#">Queryable.Single</a>
SingleOrDefault	<p>Retorna o único elemento de uma coleção ou o único elemento que satisfaz uma condição.</p> <p>Retorna um valor padrão se não houver elemento a ser retornado. Gera um <a href="#">InvalidOperationException</a> se houver mais de um elemento a ser retornado.</p>	Não aplicável.	<a href="#">Enumerable.SingleOrDefault</a> <a href="#">Queryable.SingleOrDefault</a>

## Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Como consultar os maiores arquivos em uma árvore de diretório \(LINQ\) \(C#\)](#)

# Convertendo Tipos de Dados (C#)

Artigo • 08/06/2023

Os métodos de conversão alteram o tipo dos objetos de entrada.

As operações de conversão em consultas LINQ são úteis em diversas aplicações. A seguir estão alguns exemplos:

- O método [Enumerable.AsEnumerable](#) pode ser usado para ocultar a implementação personalizada de um tipo de um operador de consulta padrão.
- O método [Enumerable.OfType](#) pode ser usado para habilitar coleções sem parâmetros para consulta LINQ.
- Os métodos [Enumerable.ToArray](#), [Enumerable.ToDictionary](#), [Enumerable.ToList](#) e [Enumerable.ToLookup](#) podem ser usados para forçar a execução de consulta imediata em vez de adiá-la até que a consulta seja enumerada.

## Métodos

A tabela a seguir lista os métodos de operador de consulta padrão que realizam conversões de tipo de dados.

Os métodos de conversão nesta tabela cujos nomes começam com "As" alteram o tipo estático da coleção de origem, mas não a enumeram. Os métodos cujos nomes começam com "To" enumeram a coleção de origem e colocam os itens na coleção de tipo correspondente.

Nome do método	Descrição	Sintaxe de expressão de consulta	Mais informações
AsEnumerable	Retorna a entrada digitada como <a href="#">IEnumerable&lt;T&gt;</a> .	Não aplicável.	<a href="#">Enumerable.AsEnumerable</a>
AsQueryable	Converte um <a href="#">IEnumerable</a> (genérico) em um <a href="#">IQueryable</a> (genérico).	Não aplicável.	<a href="#">Queryable.AsQueryable</a>

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
Conversão	Converte os elementos de uma coleção em um tipo especificado.	Use uma variável de intervalo de tipo explícito. Por exemplo:	<a href="#">Enumerable.Cast</a> <a href="#">Queryable.Cast</a>
OfType	Filtre valores, dependendo da capacidade de serem convertidos em um tipo especificado.	Não aplicável.	<a href="#">Enumerable.OfType</a> <a href="#">Queryable.OfType</a>
ToArray	Converte uma coleção em uma matriz. Esse método força a execução de consulta.	Não aplicável.	<a href="#">Enumerable.ToArray</a>
ToDictionary	Coloca os elementos em um <a href="#">Dictionary&lt;TKey,TValue&gt;</a> com base em uma função de seletor de chave. Esse método força a execução de consulta.	Não aplicável.	<a href="#">Enumerable.ToDictionary</a>
ToList	Converte uma coleção em um <a href="#">List&lt;T&gt;</a> . Esse método força a execução de consulta.	Não aplicável.	<a href="#">Enumerable.ToList</a>
ToLookup	Coloca os elementos em um <a href="#">Lookup&lt;TKey,TElement&gt;</a> (um dicionário one-to-many) com base em uma função de seletor de chave. Esse método força a execução de consulta.	Não aplicável.	<a href="#">Enumerable.ToLookup</a>

## Exemplo de sintaxe de expressão de consulta

O exemplo de código a seguir usa uma variável de intervalo de tipo explícito para converter um tipo em um subtipo antes de acessar um membro que está disponível somente no subtipo.

C#

```

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

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}

static void Cast()
{
    Plant[] plants = new Plant[] {
        new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap Trap" },
        new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall Trap" },
        new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap" },
        new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap Trap" }
    };

    var query = from CarnivorousPlant cPlant in plants
               where cPlant.TrapType == "Snap Trap"
               select cPlant;

    foreach (Plant plant in query)
        Console.WriteLine(plant.Name);
}

/* This code produces the following output:

    Venus Fly Trap
    Waterwheel Plant
*/
}

```

## Confira também

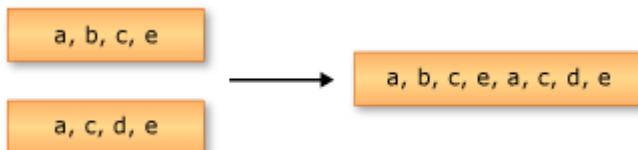
- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Cláusula From](#)
- [Expressões de Consulta LINQ](#)
- [Como consultar um ArrayList com LINQ \(C#\)](#)

# Operações de concatenação (C#)

Artigo • 07/04/2023

A concatenação refere-se a operação de acrescentar uma sequência à outra.

A ilustração a seguir mostra uma operação de concatenação em duas sequências de caracteres.



Os métodos de operador de consulta padrão que realizam a concatenação estão listados na seção a seguir.

## Métodos

Nome do método	Descrição	Sintaxe de expressão de consulta C#	Mais informações
Concat	Concatena duas sequências para formar uma sequência.	Não aplicável.	<a href="#">Enumerable.Concat</a> <a href="#">Queryable.Concat</a>

## Confira também

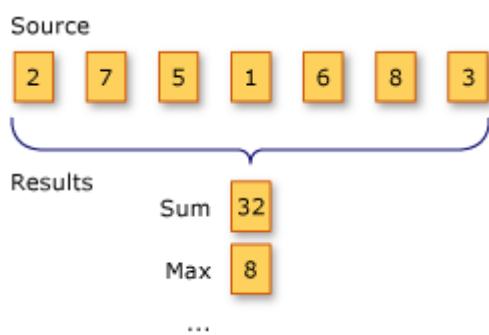
- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Como combinar e comparar coleções de cadeias de caracteres \(LINQ\) \(C#\)](#)

# Operações de agregação (C#)

Artigo • 07/04/2023

Uma operação de agregação computa um único valor de uma coleção de valores. Um exemplo de uma operação de agregação é o cálculo da temperatura média diária dos valores válidos de temperatura diária de um mês.

A ilustração a seguir mostra os resultados de duas operações de agregação diferentes em uma sequência de números. A primeira operação soma os números. A segunda operação retorna o valor máximo na sequência.



Os métodos de operador de consulta padrão que realizam operações de agregação são listados na seção a seguir.

## Métodos

Nome do método	Descrição	Sintaxe de expressão de consulta em C#	Mais informações
Agregado	Executa uma operação de agregação personalizada nos valores de uma coleção.	Não aplicável.	<a href="#">Enumerable.Aggregate</a> <a href="#">Queryable.Aggregate</a>
Média	Calcula o valor médio de uma coleção de valores.	Não aplicável.	<a href="#">Enumerable.Average</a> <a href="#">Queryable.Average</a>
Contagem	Conta os elementos em uma coleção e, opcionalmente, apenas os elementos que satisfazem a uma função de predicado.	Não aplicável.	<a href="#">Enumerable.Count</a> <a href="#">Queryable.Count</a>

<b>Nome do método</b>	<b>Descrição</b>	<b>Sintaxe de expressão de consulta em C#</b>	<b>Mais informações</b>
LongCount	Conta os elementos em uma coleção grande e, opcionalmente, apenas os elementos que satisfazem a uma função de predicado.	Não aplicável.	<a href="#">Enumerable.LongCount</a> <a href="#">Queryable.LongCount</a>
Max ou MaxBy	Determina o valor máximo em uma coleção.	Não aplicável.	<a href="#">Enumerable.Max</a> <a href="#">Enumerable.MaxBy</a> <a href="#">Queryable.Max</a> <a href="#">Queryable.MaxBy</a>
Min ou MinBy	Retorna o valor mínimo em uma coleção.	Não aplicável.	<a href="#">Enumerable.Min</a> <a href="#">Enumerable.MinBy</a> <a href="#">Queryable.Min</a> <a href="#">Queryable.MinBy</a>
Somar	Calcula a soma dos valores em uma coleção.	Não aplicável.	<a href="#">Enumerable.Sum</a> <a href="#">Queryable.Sum</a>

## Confira também

- [System.Linq](#)
- [Visão geral de operadores de consulta padrão \(C#\)](#)
- [Como computar valores de coluna em um arquivo de texto CSV \(LINQ\) \(C#\)](#)
- [Como consultar o maior arquivo ou arquivos em uma árvore de diretório \(LINQ\) \(C#\)](#)
- [Como consultar o número total de bytes em um conjunto de pastas \(LINQ\) \(C#\)](#)

# Consultar uma coleção de objetos

Artigo • 20/07/2023

O termo "LINQ to Objects" refere-se ao uso de consultas LINQ com qualquer coleção `IEnumerable` ou `IEnumerable<T>` diretamente, sem o uso de uma API ou provedor LINQ intermediário como o [LINQ to SQL](#) ou [LINQ to XML](#). Você pode usar o LINQ para consultar qualquer coleção enumerável como `List<T>`, `Array` ou `Dictionary< TKey, TValue >`. A coleção pode ser definida pelo usuário ou pode ser devolvida por uma API do .NET. Na abordagem da LINQ, você escreve o código declarativo que descreve o que você deseja recuperar.

Além disso, as consultas LINQ oferecem três principais vantagens sobre os loops `foreach` tradicionais:

- Elas são mais concisas e legíveis, especialmente quando você filtra várias condições.
- Elas fornecem poderosos recursos de filtragem, ordenação e agrupamento com um mínimo de código do aplicativo.
- Elas podem ser movidas para outras fontes de dados com pouca ou nenhuma modificação.

Em geral, quanto mais complexa a operação que você deseja executar sobre os dados, maior benefício você perceberá usando consultas LINQs em vez de técnicas tradicionais de iteração.

Este exemplo mostra como executar uma consulta simples em uma lista de objetos `Student`. Cada objeto `Student` contém algumas informações básicas sobre o aluno e uma lista que representa as pontuações do aluno em quatro provas.

## ⓘ Observação

Muitos outros exemplos nesta seção usam a mesma `Student` classe e coleção `students`.

C#

```
class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
```

```
public GradeLevel? Year { get; set; }
public List<int> ExamScores { get; set; }

public Student(string FirstName, string LastName, int ID, GradeLevel
Year, List<int> ExamScores)
{
    this.FirstName = FirstName;
    this.LastName = LastName;
    this.ID = ID;
    this.Year = Year;
    this.ExamScores = ExamScores;
}

public Student(string FirstName, string LastName, int StudentID,
List<int>? ExamScores = null)
{
    this.FirstName = FirstName;
    this.LastName = LastName;
    ID = StudentID;
    this.ExamScores = ExamScores ?? Enumerable.Empty<int>().ToList();
}

public static List<Student> students = new()
{
    new(
        FirstName: "Terry", LastName: "Adams", ID: 120,
        Year: GradeLevel.SecondYear,
        ExamScores: new() { 99, 82, 81, 79 }
    ),
    new(
        "Fadi", "Fakhouri", 116,
        GradeLevel.ThirdYear,
        new() { 99, 86, 90, 94 }
    ),
    new(
        "Hanying", "Feng", 117,
        GradeLevel.FirstYear,
        new() { 93, 92, 80, 87 }
    ),
    new(
        "Cesar", "Garcia", 114,
        GradeLevel.FourthYear,
        new() { 97, 89, 85, 82 }
    ),
    new(
        "Debra", "Garcia", 115,
        GradeLevel.ThirdYear,
        new() { 35, 72, 91, 70 }
    ),
    new(
        "Hugo", "Garcia", 118,
        GradeLevel.SecondYear,
        new() { 92, 90, 83, 78 }
    ),
    new(

```

```

        "Sven", "Mortensen", 113,
        GradeLevel.FirstYear,
        new() { 88, 94, 65, 91 }
    ),
    new(
        "Claire", "O'Donnell", 112,
        GradeLevel.FourthYear,
        new() { 75, 84, 91, 39 }
    ),
    new(
        "Svetlana", "Omelchenko", 111,
        GradeLevel.SecondYear,
        new() { 97, 92, 81, 60 }
    ),
    new(
        "Lance", "Tucker", 119,
        GradeLevel.ThirdYear,
        new() { 68, 79, 88, 92 }
    ),
    new(
        "Michael", "Tucker", 122,
        GradeLevel.FirstYear,
        new() { 94, 92, 91, 91 }
    ),
    new(
        "Eugene", "Zabokritski", 121,
        GradeLevel.FourthYear,
        new() { 96, 85, 91, 60 }
    )
);
}

enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

```

## Exemplo

A consulta a seguir retorna os alunos que receberam uma pontuação de 90 ou mais em sua primeira prova.

C#

```

void QueryHighScores(int exam, int score)
{
    var highScores =
        from student in students

```

```
    where student.ExamScores[exam] > score
    select new
    {
        Name = student.FirstName,
        Score = student.ExamScores[exam]
    };

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name},-{15}{item.Score}");
    }
}

QueryHighScores(1, 90);
```

Essa consulta é intencionalmente simples para que você possa testar. Por exemplo, você pode testar mais condições na cláusula `where` ou usar uma cláusula `orderby` para classificar os resultados.

## Classificação de operadores de consulta padrão por maneira de execução

As implementações de LINQ to Objects dos métodos de operador de consulta padrão em uma das duas maneiras principais: imediata ou adiada. Os operadores de consulta que usam a execução adiada podem ser divididos em mais duas categorias: streaming e não streaming. Se você souber como os operadores de consulta diferentes são executados, isso poderá ajudá-lo a entender os resultados que serão obtidos de uma determinada consulta. Isso é especialmente verdadeiro se a fonte de dados está sendo alterada ou se você estiver criando uma consulta sobre outra consulta. Este tópico classifica os operadores de consulta padrão de acordo com o modo de execução.

### Imediata

A execução imediata significa que a fonte de dados é lida e a operação é executada uma vez. Todos os operadores de consulta padrão que retornam um resultado escalar são executados imediatamente. Você pode forçar uma consulta a ser executada imediatamente usando o método `Enumerable.ToList` ou `Enumerable.ToArray`. A execução imediata fornece reutilização dos resultados da consulta, não uma declaração de consulta. Os resultados são recuperados uma vez e armazenados para uso futuro.

### Adiado

A execução adiada significa que a operação não será realizada no ponto do código em que a consulta estiver declarada. A operação será realizada somente quando a variável de consulta for enumerada, por exemplo, usando uma instrução `foreach`. Isso significa que os resultados da execução da consulta dependerão do conteúdo da fonte de dados quando a consulta for executada em vez de quando a consulta for definida. Se a variável de consulta for enumerada várias vezes, os resultados poderão ser diferentes a cada vez. Quase todos os operadores de consulta padrão cujo tipo de retorno é `IEnumerable<T>` ou `IOrderedEnumerable<TElement>` executam de maneira adiada. A execução adiada oferece a facilidade da reutilização da consulta, uma vez que ela busca os dados atualizados da fonte de dados sempre que os seus resultados são iterados.

Os operadores de consulta que usam a execução adiada podem ser, adicionalmente, classificados como streaming ou não streaming.

## Streaming

Operadores streaming não precisam ler todos os dados de origem antes de gerar elementos. No momento da execução, um operador streaming realiza sua operação em cada elemento de origem enquanto eles são lidos, gerando o elemento, se apropriado. Um operador streaming continua a ler os elementos de origem até que um elemento de resultado possa ser produzido. Isso significa que mais de um elemento de origem poderá ser lido para produzir um elemento de resultado.

## Não streaming

Os operadores não streaming devem ler todos os dados de origem antes de produzirem um elemento de resultado. Operações como classificação ou agrupamento se enquadram nesta categoria. No momento da execução, os operadores de consulta não streaming leem todos os dados de origem, colocam-nos em uma estrutura de dados, realizam a operação e geram os elementos de resultado.

## Tabela de classificação

A tabela a seguir classifica cada método de operador de consulta padrão de acordo com o respectivo método de execução.

### ⓘ Observação

Se um operador estiver marcado em duas colunas, duas sequências de entrada estarão envolvidas na operação e cada sequência será avaliada de forma diferente.

Nesses casos, a primeira sequência na lista de parâmetros é a que sempre será avaliada de maneira adiada e em modo streaming.

<b>Operador de consulta padrão</b>	<b>Tipo de retorno</b>	<b>Execução imediata</b>	<b>Execução adiada de streaming</b>	<b>Execução adiada de não streaming</b>
Aggregate	TSource	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	Valor numérico único	X		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	TSource	X		
ElementAtOrDefault	TSource?		X	
Empty	IEnumerable<T>		X	
Except	IEnumerable<T>		X	X
First	TSource	X		
FirstOrDefault	TSource?		X	
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X
Join	IEnumerable<T>		X	X
Last	TSource	X		

<b>Operador de consulta padrão</b>	<b>Tipo de retorno</b>	<b>Execução imediata</b>	<b>Execução adiada de streaming</b>	<b>Execução adiada de não streaming</b>
LastOrDefault	<code>TSource?</code>	X		
LongCount	<code>Int64</code>	X		
Max	Valor numérico único, <code>TSource</code> OU <code>TResult?</code>	X		
Min	Valor numérico único, <code>TSource</code> OU <code>TResult?</code>	X		
OfType	<code>IEnumerable&lt;T&gt;</code>	X		
OrderBy	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
OrderByDescending	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
Range	<code>IEnumerable&lt;T&gt;</code>	X		
Repeat	<code>IEnumerable&lt;T&gt;</code>	X		
Reverse	<code>IEnumerable&lt;T&gt;</code>		X	
Select	<code>IEnumerable&lt;T&gt;</code>	X		
SelectMany	<code>IEnumerable&lt;T&gt;</code>	X		
SequenceEqual	<code>Boolean</code>	X		
Single	<code>TSource</code>	X		
SingleOrDefault	<code>TSource?</code>	X		
Skip	<code>IEnumerable&lt;T&gt;</code>	X		
SkipWhile	<code>IEnumerable&lt;T&gt;</code>	X		
Sum	Valor numérico único	X		
Take	<code>IEnumerable&lt;T&gt;</code>	X		
TakeWhile	<code>IEnumerable&lt;T&gt;</code>	X		
ThenBy	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
ThenByDescending	<code>IOrderedEnumerable&lt;TElement&gt;</code>		X	
ToArray	Matriz <code>TSource[]</code>	X		

<b>Operador de consulta padrão</b>	<b>Tipo de retorno</b>	<b>Execução imediata</b>	<b>Execução adiada de streaming</b>	<b>Execução adiada de não streaming</b>
ToDictionary	Dictionary<TKey,TValue>	X		
ToList	IList<T>	X		
ToLookup	ILookup<TKey,TElement>	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

## Confira também

- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Interpolação de cadeia de caracteres](#)

# LINQ (Consulta Integrada à Linguagem)

Artigo • 15/02/2023

O LINQ (consulta integrada à linguagem) é o nome de um conjunto de tecnologias com base na integração de recursos de consulta diretamente na linguagem C#.

Tradicionalmente, consultas feitas em dados são expressas como cadeias de caracteres simples sem verificação de tipo no tempo de compilação ou suporte a IntelliSense. Além disso, você precisará aprender uma linguagem de consulta diferente para cada tipo de fonte de dados: bancos de dados SQL, documentos XML, vários serviços Web etc. Com o LINQ, uma consulta é um constructo de linguagem de primeira classe, como classes, métodos, eventos.

Para um desenvolvedor que escreve consultas, a parte mais visível "integrada à linguagem" do LINQ é a expressão de consulta. As expressões de consulta são uma *sintaxe declarativa de consulta*. Usando a sintaxe de consulta, você pode executar operações de filtragem, ordenação e agrupamento em fontes de dados com o mínimo de código. Você pode usar os mesmos padrões de expressão de consulta básica para consultar e transformar dados em bancos de dados SQL, conjuntos de dados do ADO.NET, documentos XML e fluxos e coleções .NET.

O exemplo a seguir mostra a operação de consulta completa. A operação completa inclui a criação de uma fonte de dados, definição da expressão de consulta e execução da consulta em uma instrução `foreach`.

```
C#  
  
// Specify the data source.  
int[] scores = { 97, 92, 81, 60 };  
  
// Define the query expression.  
IQueryable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;  
  
// Execute the query.  
foreach (int i in scoreQuery)  
{  
    Console.Write(i + " ");  
}  
  
// Output: 97 92 81
```

# Visão geral da expressão de consulta

- Expressões de consulta podem ser usadas para consultar e transformar dados de qualquer fonte de dados habilitada para LINQ. Por exemplo, uma única consulta pode recuperar dados de um Banco de Dados SQL e produzir um fluxo XML como saída.
- As expressões de consulta são fáceis de entender porque elas usam muitos constructos de linguagem C# familiares.
- As variáveis em uma expressão de consulta são fortemente tipadas, embora em muitos casos você não precise fornecer o tipo explicitamente, pois o compilador pode inferir nele. Para obter mais informações, consulte [Relacionamentos de tipo em operações de consulta LINQ](#).
- Uma consulta não é executada até que você itere sobre a variável de consulta, por exemplo, em uma instrução `foreach`. Para obter mais informações, consulte [Introdução a consultas LINQ](#).
- No tempo de compilação, as expressões de consulta são convertidas em chamadas de método do operador de consulta padrão de acordo com as regras definidas na especificação do C#. Qualquer consulta que pode ser expressa usando sintaxe de consulta também pode ser expressa usando sintaxe de método. No entanto, na maioria dos casos, a sintaxe de consulta é mais legível e concisa. Para obter mais informações, consulte [Especificação da linguagem C#](#) e [Visão geral de operadores de consulta padrão](#).
- Como uma regra ao escrever consultas LINQ, recomendamos que você use a sintaxe de consulta sempre que possível e a sintaxe de método sempre que necessário. Não há semântica ou diferença de desempenho entre as duas formas. As expressões de consulta são geralmente mais legíveis do que as expressões equivalentes escritas na sintaxe de método.
- Algumas operações de consulta, como [Count](#) ou [Max](#), não apresentam cláusulas de expressão de consulta equivalentes e, portanto, devem ser expressas como chamadas de método. A sintaxe de método pode ser combinada com a sintaxe de consulta de várias maneiras. Para obter mais informações, consulte [Sintaxe de consulta e sintaxe de método em LINQ](#).
- As expressões de consulta podem ser compiladas para árvores de expressão ou delegados, dependendo do tipo ao qual a consulta é aplicada. As consultas `IEnumerable<T>` são compiladas para representantes. As consultas `IQueryable` e

`IQueryable<T>` são compiladas para árvores de expressão. Para obter mais informações, consulte [Árvores de expressão](#).

## Próximas etapas

Para obter mais detalhes sobre o LINQ, comece se familiarizando com alguns conceitos básicos em [Noções básicas sobre expressões de consulta](#), e, em seguida, leia a documentação para a tecnologia LINQ na qual você está interessado:

- Documentos XML: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to Entities](#)
- Coleções do .NET, arquivos, cadeias de caracteres, etc.: [LINQ to Objects](#)

Para saber mais sobre o LINQ, consulte [LINQ em C#](#).

Para começar a trabalhar com o LINQ em C#, consulte o tutorial [Trabalhando com LINQ](#).

# Como contar ocorrências de uma palavra em uma cadeia de caracteres (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo mostra como usar uma consulta LINQ para contar as ocorrências de uma palavra especificada em uma cadeia de caracteres. Observe que para executar a contagem, primeiro o método `Split` é chamado para criar uma matriz de palavras. Há um custo de desempenho para o método `Split`. Se for a única operação na cadeia de caracteres para contar as palavras, você deverá considerar o uso dos métodos `Matches` ou `IndexOf` em vez dele. No entanto, se o desempenho não for um problema crítico ou se você já tiver dividido a sentença para executar outros tipos de consulta nela, faz sentido usar LINQ para contar as palavras ou frases também.

## Exemplo

C#

```
class CountWords
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of
objects" +
                      @" have not been well integrated. Programmers work in C# or Visual
Basic" +
                      @" and also in SQL or XQuery. On the one side are concepts such as
classes," +
                      @" objects, fields, inheritance, and .NET APIs. On the other side"
+
                      @" are tables, columns, rows, nodes, and separate languages for
dealing with" +
                      @" them. Data types often require translation between the two
worlds; there are" +
                      @" different standard functions. Because the object world has no
notion of query, a" +
                      @" query can only be represented as a string without compile-time
type checking or" +
                      @" IntelliSense support in the IDE. Transferring data from SQL
tables or XML trees to" +
                      @" objects in memory is often tedious and error-prone.";

        string searchTerm = "data";

        //Convert the string into an array of words
```

```

        string[] source = text.Split(new char[] { '.', '?', '!', ' ', ';' ,
        ':', ',' }, StringSplitOptions.RemoveEmptyEntries);

        // Create the query. Use the InvariantCultureIgnoreCase comparision
        // to match "data" and "Data"
        var matchQuery = from word in source
                         where word.Equals(searchTerm,
StringComparison.InvariantCultureIgnoreCase)
                         select word;

        // Count the matches, which executes the query.
        int wordCount = matchQuery.Count();
        Console.WriteLine("{0} occurrence(s) of the search term \"{1}\"
were found.", wordCount, searchTerm);

        // Keep console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

/* Output:
   3 occurrences(s) of the search term "data" were found.
*/

```

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como consultar sentenças que contêm um conjunto especificado de palavras (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo mostra como localizar frases em um arquivo de texto que contêm correspondências para cada conjunto de palavras especificado. Embora a matriz de termos de pesquisa seja embutida em código neste exemplo, ela também pode ser preenchida dinamicamente em tempo de execução. Neste exemplo, a consulta retorna as frases que contêm as palavras "Historically", "data" e "integrated".

## Exemplo

C#

```
class FindSentences
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of
objects " +
            @"have not been well integrated. Programmers work in C# or Visual
Basic " +
            @"and also in SQL or XQuery. On the one side are concepts such as
classes, " +
            @"objects, fields, inheritance, and .NET APIs. On the other side " +
            @"are tables, columns, rows, nodes, and separate languages for
dealing with " +
            @"them. Data types often require translation between the two worlds;
there are " +
            @"different standard functions. Because the object world has no
notion of query, a " +
            @"query can only be represented as a string without compile-time
type checking or " +
            @"IntelliSense support in the IDE. Transferring data from SQL tables
or XML trees to " +
            @"objects in memory is often tedious and error-prone.';

        // Split the text block into an array of sentences.
        string[] sentences = text.Split(new char[] { '.', '?', '!' });

        // Define the search terms. This list could also be dynamically
        // populated at run time.
        string[] wordsToMatch = { "Historically", "data", "integrated" };

        // Find sentences that contain all the terms in the wordsToMatch
```

```

array.
    // Note that the number of terms to match is not specified at
    // compile time.
    var sentenceQuery = from sentence in sentences
                        let w = sentence.Split(new char[] { '.', '?',
                            '!', ' ', ';' , ':' , ',' },
StringSplitOptions.RemoveEmptyEntries)
                        where
w.Distinct().Intersect(wordsToMatch).Count() == wordsToMatch.Count()
                        select sentence;

    // Execute the query. Note that you can explicitly type
    // the iteration variable here even though sentenceQuery
    // was implicitly typed.
    foreach (string str in sentenceQuery)
    {
        Console.WriteLine(str);
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit");
    Console.ReadKey();
}
}
/* Output:
Historically, the world of data and the world of objects have not been well
integrated
*/

```

A consulta funciona primeiro dividindo o texto em frases e, em seguida, dividindo as sentenças em uma matriz de cadeias de caracteres que contêm cada palavra. Para cada uma dessas matrizes, o método `Distinct` remove todas as palavras duplicadas e, em seguida, a consulta executa uma operação `Intersect` na matriz de palavras e na matriz `wordsToMatch`. Se a contagem da interseção for igual à contagem da matriz `wordsToMatch`, todas as palavras foram encontradas nas palavras e a frase original será retornada.

Na chamada para `Split`, as marcas de pontuação são usadas como separadores para removê-las da cadeia de caracteres. Se não fizer isso, por exemplo, você poderia ter uma cadeia de caracteres "Historically" que não corresponderia a "Historically" na matriz `wordsToMatch`. Talvez você precise usar separadores adicionais, dependendo dos tipos de pontuação encontrados no texto de origem.

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como consultar caracteres em uma cadeia de caracteres (LINQ) (C#)

Artigo • 20/07/2023

Já que a classe `String` implementa a interface `IEnumerable<T>` genérica, qualquer cadeia de caracteres pode ser consultada como uma sequência de caracteres. No entanto, esse não é um uso comum da LINQ. Para operações de correspondência de padrões complexas, use a classe `Regex`.

## Exemplo

O exemplo a seguir consulta uma cadeia de caracteres para determinar quantos dígitos numéricos ela contém. Observe que a consulta é "reutilizada" depois que é executada pela primeira vez. Isso é possível porque a consulta em si não armazena nenhum resultado real.

C#

```
class QueryAString
{
    static void Main()
    {
        string aString = "ABCDE99F-J74-12-89A";

        // Select only those characters that are numbers
        IEnumerable<char> stringQuery =
            from ch in aString
            where Char.IsDigit(ch)
            select ch;

        // Execute the query
        foreach (char c in stringQuery)
            Console.Write(c + " ");

        // Call the Count method on the existing query.
        int count = stringQuery.Count();
        Console.WriteLine("Count = {0}", count);

        // Select all characters before the first '-'
        IEnumerable<char> stringQuery2 = aString.TakeWhile(c => c != '-');

        // Execute the second query
        foreach (char c in stringQuery2)
            Console.Write(c);

        Console.WriteLine(System.Environment.NewLine + "Press any key to")
```

```
    exit");
        Console.ReadKey();
    }
}
/* Output:
Output: 9 9 7 4 1 2 8 9
Count = 8
ABCDE99F
*/

```

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

## Confira também

- [Como combinar consultas da LINQ com expressões regulares \(C#\)](#)

# Como combinar consultas LINQ com expressões regulares (C#)

Artigo • 20/07/2023

Este exemplo mostra como usar a classe [Regex](#) para criar uma expressão regular para correspondências mais complexas em cadeias de texto. A consulta LINQ torna fácil a aplicação de filtro exatamente nos arquivos que você deseja pesquisar com a expressão regular e formatar os resultados.

## Exemplo

C#

```
class QueryWithRegEx
{
    public static void Main()
    {
        // Modify this path as necessary so that it accesses your version of
        Visual Studio.
        string startFolder = @"C:\Program Files (x86)\Microsoft Visual
        Studio 14.0\";
        // One of the following paths may be more appropriate on your
        computer.
        //string startFolder = @"C:\Program Files (x86)\Microsoft Visual
        Studio\2017\";

        // Take a snapshot of the file system.
        IEnumerable<System.IO.FileInfo> fileList = GetFiles(startFolder);

        // Create the regular expression to find all things "Visual".
        System.Text.RegularExpressions.Regex searchTerm =
            new System.Text.RegularExpressions.Regex(@"Visual
        (Basic|C#|C\+\+|Studio)"));

        // Search the contents of each .htm file.
        // Remove the where clause to find even more matchedValues!
        // This query produces a list of files where a match
        // was found, and a list of the matchedValues in that file.
        // Note: Explicit typing of "Match" in select clause.
        // This is required because MatchCollection is not a
        // generic IEnumerable collection.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = System.IO.File.ReadAllText(file.FullName)
            let matches = searchTerm.Matches(fileText)
            where matches.Count > 0
            select new
```

```

    {
        name = file.FullName,
        matchedValues = from System.Text.RegularExpressions.Match
    match in matches
        select match.Value
    };

    // Execute the query.
    Console.WriteLine("The term \"{0}\" was found in:",
searchTerm.ToString());

    foreach (var v in queryMatchingFiles)
    {
        // Trim the path a bit, then write
        // the file name in which a match was found.
        string s = v.name.Substring(startFolder.Length - 1);
        Console.WriteLine(s);

        // For this file, write out all the matching strings
        foreach (var v2 in v.matchedValues)
        {
            Console.WriteLine(" " + v2);
        }
    }

    // Keep the console window open in debug mode
    Console.WriteLine("Press any key to exit");
    Console.ReadKey();
}

// This method assumes that the application has discovery
// permissions for all folders under the specified path.
static IEnumerable<System.IO.FileInfo> GetFiles(string path)
{
    if (!System.IO.Directory.Exists(path))
        throw new System.IO.DirectoryNotFoundException();

    string[] fileNames = null;
    List<System.IO.FileInfo> files = new List<System.IO.FileInfo>();

    fileNames = System.IO.Directory.GetFiles(path, "*.*",
System.IO.SearchOption.AllDirectories);
    foreach (string name in fileNames)
    {
        files.Add(new System.IO.FileInfo(name));
    }
    return files;
}
}

```

Observe que também é possível consultar o objeto **MatchCollection** retornado por uma pesquisa **RegEx**. Neste exemplo, apenas o valor de cada correspondência é produzido nos resultados. No entanto, também é possível usar a LINQ para executar todos os tipos

de filtragem, classificação e agrupamento nessa coleção. Como [MatchCollection](#) é uma coleção [IEnumerable](#) não genérica, é necessário declarar explicitamente o tipo da variável de intervalo na consulta.

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como localizar a diferença de conjunto entre duas listas (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo mostra como usar o LINQ para comparar duas listas de cadeias de caracteres e retornar as linhas que estão no names1.txt, mas não no names2.txt.

## Para criar os arquivos de dados

1. Copiar names1.txt e names2.txt para a pasta da sua solução, conforme mostrado em [Como combinar e comparar coleções de cadeias de caracteres \(LINQ\) \(C#\)](#).

## Exemplo

C#

```
class CompareLists
{
    static void Main()
    {
        // Create the IEnumerable data sources.
        string[] names1 =
System.IO.File.ReadAllLines(@"../../names1.txt");
        string[] names2 =
System.IO.File.ReadAllLines(@"../../names2.txt");

        // Create the query. Note that method syntax must be used here.
        IEnumerable<string> differenceQuery =
            names1.Except(names2);

        // Execute the query.
        Console.WriteLine("The following lines are in names1.txt but not
names2.txt");
        foreach (string s in differenceQuery)
            Console.WriteLine(s);

        // Keep the console window open until the user presses a key.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
   The following lines are in names1.txt but not names2.txt
   Potra, Cristina
   Noriega, Fabricio
   Aw, Kam Foo
```

Toyoshima, Tim  
Guy, Wey Yuan  
Garcia, Debra  
\*/

Alguns tipos de operações de consulta em C#, tais como [Except](#), [Distinct](#), [Union](#) e [Concat](#), só podem ser expressas em sintaxe baseada em método.

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como classificar ou filtrar dados de texto por qualquer palavra ou campo (LINQ) (C#)

Artigo • 20/07/2023

O exemplo a seguir mostra como classificar linhas de texto estruturado, como valores separados por vírgulas, por qualquer campo na linha. O campo pode ser especificado dinamicamente no tempo de execução. Suponha que os campos em scores.csv representam o número de ID do aluno, seguido por uma série de quatro resultados de teste.

## Para criar um arquivo que contém dados

1. Copie os dados de scores.csv do tópico [Como unir conteúdo de arquivos diferentes \(LINQ\) \(C#\)](#) e salve-os na pasta da sua solução.

## Exemplo

C#

```
public class SortLines
{
    static void Main()
    {
        // Create an IEnumerable data source
        string[] scores =
System.IO.File.ReadAllLines(@"../../scores.csv");

        // Change this to any value from 0 to 4.
        int sortField = 1;

        Console.WriteLine("Sorted highest to lowest by field [{0}]:",
sortField);

        // Demonstrates how to return query from a method.
        // The query is executed here.
        foreach (string str in RunQuery(scores, sortField))
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

```

    }

    // Returns the query variable, not query results!
    static IEnumerable<string> RunQuery(IEnumerable<string> source, int num)
    {
        // Split the string and sort on field[num]
        var scoreQuery = from line in source
                          let fields = line.Split(',')
                          orderby fields[num] descending
                          select line;

        return scoreQuery;
    }
}

/* Output (if sortField == 1):
Sorted highest to lowest by field [1]:
116, 99, 86, 90, 94
120, 99, 82, 81, 79
111, 97, 92, 81, 60
114, 97, 89, 85, 82
121, 96, 85, 91, 60
122, 94, 92, 91, 91
117, 93, 92, 80, 87
118, 92, 90, 83, 78
113, 88, 94, 65, 91
112, 75, 84, 91, 39
119, 68, 79, 88, 92
115, 35, 72, 91, 70
*/

```

Este exemplo também demonstra como retornar uma variável de consulta de um método.

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como reordenar os campos de um arquivo delimitado (LINQ) (C#)

Artigo • 20/07/2023

Um CSV (arquivo de valores separados por vírgula) é um arquivo de texto que é frequentemente usado para armazenar dados de planilha ou outros dados de tabela que são representados por linhas e colunas. Ao usar o método [Split](#) para separar os campos, é muito fácil consultar e manipular arquivos CSV usando LINQ. Na verdade, a mesma técnica pode ser usada para reordenar as partes de qualquer linha estruturada de texto. Ela não é limitada a arquivos CSV.

No exemplo a seguir, suponha que as três colunas representem "sobrenome", "nome" e "ID" dos alunos. Os campos estão em ordem alfabética com base nos sobrenomes dos alunos. A consulta gera uma nova sequência, na qual a coluna ID é exibida em primeiro, seguida por uma segunda coluna que combina o nome e o sobrenome do aluno. As linhas são reordenadas acordo com o campo ID. Os resultados são salvos em um novo arquivo e os dados originais não são modificados.

## Para criar o arquivo de dados

1. Copie as seguintes linhas em um arquivo de texto sem formatação denominado `spreadsheet1.csv`. Salve o arquivo na pasta do seu projeto.

```
csv

Adams,Terry,120
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Cesar,114
Garcia,Debra,115
Garcia,Hugo,118
Mortensen,Sven,113
O'Donnell,Claire,112
Omelchenko,Svetlana,111
Tucker,Lance,119
Tucker,Michael,122
Zabokritski,Eugene,121
```

## Exemplo

C#

```

class CSVFiles
{
    static void Main(string[] args)
    {
        // Create the IEnumerable data source
        string[] lines =
System.IO.File.ReadAllLines(@"../../spreadsheet1.csv");

        // Create the query. Put field 2 first, then
        // reverse and combine fields 0 and 1 from the old field
        IEnumerable<string> query =
            from line in lines
            let x = line.Split(',')
            orderby x[2]
            select x[2] + ", " + (x[1] + " " + x[0]);

        // Execute the query and write out the new file. Note that
        WriteAllLines
            // takes a string[], so ToArray is called on the query.
            System.IO.File.WriteAllLines(@"../../spreadsheet2.csv",
query.ToArray());

        Console.WriteLine("Spreadsheet2.csv written to disk. Press any key
to exit");
        Console.ReadKey();
    }
}
/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
*/

```

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como combinar e comparar coleções de cadeias de caracteres (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo mostra como mesclar arquivos que contêm linhas de texto e, em seguida, classificar os resultados. Especificamente, mostra como executar uma concatenação, uma união e uma interseção simples nos dois conjuntos de linhas de texto.

## Para configurar o projeto e os arquivos de texto

1. Copie esses nomes em um arquivo de texto chamado names1.txt e salve-o na sua pasta do projeto:

```
text  
Bankov, Peter  
Holm, Michael  
Garcia, Hugo  
Potra, Cristina  
Noriega, Fabricio  
Aw, Kam Foo  
Beebe, Ann  
Toyoshima, Tim  
Guy, Wey Yuan  
Garcia, Debra
```

2. Copie esses nomes em um arquivo de texto chamado names2.txt e salve-o na sua pasta do projeto. Observe que os dois arquivos tem alguns nomes em comum.

```
text  
Liu, Jinghao  
Bankov, Peter  
Holm, Michael  
Garcia, Hugo  
Beebe, Ann  
Gilchrist, Beth  
Myrcha, Jacek  
Giakoumakis, Leo  
McLin, Nkenge  
El Yassir, Mehdi
```

## Exemplo

C#

```
class MergeStrings
{
    static void Main(string[] args)
    {
        //Put text files in your solution folder
        string[] fileA =
System.IO.File.ReadAllLines(@"../../names1.txt");
        string[] fileB =
System.IO.File.ReadAllLines(@"../../names2.txt");

        //Simple concatenation and sort. Duplicates are preserved.
        IEnumerable<string> concatQuery =
            fileA.Concat(fileB).OrderBy(s => s);

        // Pass the query variable to another function for execution.
        OutputQueryResults(concatQuery, "Simple concatenate and sort.
Duplicates are preserved:");

        // Concatenate and remove duplicate names based on
        // default string comparer.
        IEnumerable<string> uniqueNamesQuery =
            fileA.Union(fileB).OrderBy(s => s);
        OutputQueryResults(uniqueNamesQuery, "Union removes duplicate
names:");

        // Find the names that occur in both files (based on
        // default string comparer).
        IEnumerable<string> commonNamesQuery =
            fileA.Intersect(fileB);
        OutputQueryResults(commonNamesQuery, "Merge based on
intersect:");

        // Find the matching fields in each list. Merge the two
        // results by using Concat, and then
        // sort using the default string comparer.
        string nameMatch = "Garcia";

        IEnumerable<String> tempQuery1 =
            from name in fileA
            let n = name.Split(',')
            where n[0] == nameMatch
            select name;

        IEnumerable<string> tempQuery2 =
            from name2 in fileB
            let n2 = name2.Split(',')
            where n2[0] == nameMatch
            select name2;

        IEnumerable<string> nameMatchQuery =
            tempQuery1.Concat(tempQuery2).OrderBy(s => s);
        OutputQueryResults(nameMatchQuery, $"Concat based on partial
```

```
name match \"{nameMatch}\":");
    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit");
    Console.ReadKey();
}

static void OutputQueryResults(IEnumerable<string> query, string
message)
{
    Console.WriteLine(System.Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("{0} total names in list", query.Count());
}
/* Output:
   Simple concatenate and sort. Duplicates are preserved:
   Aw, Kam Foo
   Bankov, Peter
   Bankov, Peter
   Beebe, Ann
   Beebe, Ann
   El Yassir, Mehdi
   Garcia, Debra
   Garcia, Hugo
   Garcia, Hugo
   Giakoumakis, Leo
   Gilchrist, Beth
   Guy, Wey Yuan
   Holm, Michael
   Holm, Michael
   Liu, Jinghao
   McLin, Nkenge
   Myrcha, Jacek
   Noriega, Fabricio
   Potra, Cristina
   Toyoshima, Tim
   20 total names in list

   Union removes duplicate names:
   Aw, Kam Foo
   Bankov, Peter
   Beebe, Ann
   El Yassir, Mehdi
   Garcia, Debra
   Garcia, Hugo
   Giakoumakis, Leo
   Gilchrist, Beth
   Guy, Wey Yuan
   Holm, Michael
   Liu, Jinghao
   McLin, Nkenge
```

```
Myrcha, Jacek
Noriega, Fabricio
Potra, Cristina
Toyoshima, Tim
16 total names in list

Merge based on intersect:
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
4 total names in list

Concat based on partial name match "Garcia":
Garcia, Debra
Garcia, Hugo
Garcia, Hugo
3 total names in list
*/
```

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como preencher coleções de objetos de várias fontes (LINQ) (C#)

Artigo • 07/04/2023

Este exemplo mostra como mesclar dados de diferentes fontes em uma sequência de novos tipos.

## ⓘ Observação

Não tente unir dados na memória ou dados no sistema de arquivos com os dados que ainda estão em um banco de dados. Essas junções entre domínios podem gerar resultados indefinidos, devido às diferentes formas em que as operações de junção podem ser definidas para consultas de banco de dados e outros tipos de fontes. Além disso, há um risco de que essa operação possa causar uma exceção de falta de memória, se a quantidade de dados no banco de dados for grande o suficiente. Para unir dados de um banco de dados com os dados na memória, primeiro chame `ToList` ou `ToArray` na consulta de banco de dados e, em seguida, realize a junção na coleção retornada.

## Para criar o arquivo de dados

Copie os arquivos `names.csv` e `scores.csv` para a pasta do projeto, conforme descrito em [Como unir conteúdo de arquivos diferentes \(LINQ\) \(C#\)](#).

## Exemplo

O exemplo a seguir mostra como usar um tipo nomeado `Student` para armazenar dados mesclados, de duas coleções na memória de cadeias de caracteres, que simulam dados de planilha no formato `.csv`. A primeira coleção de cadeias de caracteres representa os nomes e as IDs dos alunos e a segunda coleção, representa a ID do aluno (na primeira coluna) e quatro pontuações de exames. A ID é usada como a chave estrangeira.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;

class Student
```

```
{  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
    public int ID { get; set; }  
    public List<int> ExamScores { get; set; }  
}  
  
class PopulateCollection  
{  
    static void Main()  
{  
        // These data files are defined in How to join content from  
        // dissimilar files (LINQ).  
  
        // Each line of names.csv consists of a last name, a first name, and  
        // an  
        // ID number, separated by commas. For example,  
        Omelchenko,Svetlana,111  
        string[] names = System.IO.File.ReadAllLines(@"../../names.csv");  
  
        // Each line of scores.csv consists of an ID number and four test  
        // scores, separated by commas. For example, 111, 97, 92, 81, 60  
        string[] scores =  
System.IO.File.ReadAllLines(@"../../scores.csv");  
  
        // Merge the data sources using a named type.  
        // var could be used instead of an explicit type. Note the dynamic  
        // creation of a list of ints for the ExamScores member. The first  
item  
        // is skipped in the split string because it is the student ID,  
        // not an exam score.  
        IEnumerable<Student> queryNamesScores =  
            from nameLine in names  
            let splitName = nameLine.Split(',')  
            from scoreLine in scores  
            let splitScoreLine = scoreLine.Split(',')  
            where Convert.ToInt32(splitName[2]) ==  
Convert.ToInt32(splitScoreLine[0])  
            select new Student()  
            {  
                FirstName = splitName[0],  
                LastName = splitName[1],  
                ID = Convert.ToInt32(splitName[2]),  
                ExamScores = (from scoreAsText in splitScoreLine.Skip(1)  
                            select Convert.ToInt32(scoreAsText)).  
                            ToList()  
            };  
  
        // Optional. Store the newly created student objects in memory  
        // for faster access in future queries. This could be useful with  
        // very large data files.  
        List<Student> students = queryNamesScores.ToList();  
  
        // Display each student's name and exam score average.  
        foreach (var student in students)
```

```

    {
        Console.WriteLine("The average score of {0} {1} is {2}.",
            student.FirstName, student.LastName,
            student.ExamScores.Average());
    }

    //Keep console window open in debug mode
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
/* Output:
   The average score of Omelchenko Svetlana is 82.5.
   The average score of O'Donnell Claire is 72.25.
   The average score of Mortensen Sven is 84.5.
   The average score of Garcia Cesar is 88.25.
   The average score of Garcia Debra is 67.
   The average score of Fakhouri Fadi is 92.25.
   The average score of Feng Hanying is 88.
   The average score of Garcia Hugo is 85.75.
   The average score of Tucker Lance is 81.75.
   The average score of Adams Terry is 85.25.
   The average score of Zabokritski Eugene is 83.
   The average score of Tucker Michael is 92.
*/

```

Na cláusula `select`, um inicializador de objeto é usado para instanciar cada novo objeto `Student`, usando os dados das duas fontes.

Se você não tiver que armazenar os resultados de uma consulta, os tipos anônimos poderão ser mais convenientes que os tipos nomeados. Os tipos nomeados são necessários se você passa os resultados da consulta para fora do método em que a consulta é executada. O exemplo a seguir realiza a mesma tarefa do exemplo anterior, mas usa tipos anônimos em vez de tipos nomeados:

C#

```

// Merge the data sources by using an anonymous type.
// Note the dynamic creation of a list of ints for the
// ExamScores member. We skip 1 because the first string
// in the array is the student ID, not an exam score.
var queryNamesScores2 =
    from nameLine in names
    let splitName = nameLine.Split(',')
    from scoreLine in scores
    let splitScoreLine = scoreLine.Split(',')
    where Convert.ToInt32(splitName[2]) ==
        Convert.ToInt32(splitScoreLine[0])
    select new
    {
        First = splitName[0],

```

```
    Last = splitName[1],  
    ExamScores = (from scoreAsText in splitScoreLine.Skip(1)  
                  select Convert.ToInt32(scoreAsText))  
                  .ToList()  
};  
  
// Display each student's name and exam score average.  
foreach (var student in queryNamesScores2)  
{  
    Console.WriteLine("The average score of {0} {1} is {2}.",  
                      student.First, student.Last, student.ExamScores.Average());  
}
```

## Confira também

- [LINQ e cadeias de caracteres \(C#\)](#)
- [Inicializadores de objeto e coleção](#)
- [Tipos anônimos](#)

# Como dividir um arquivo em vários usando grupos (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo mostra uma maneira de mesclar o conteúdo de dois arquivos e, em seguida, criar um conjunto de novos arquivos que organizam os dados em uma nova forma.

## Para criar os arquivos de dados

1. Copie esses nomes em um arquivo de texto chamado names1.txt e salve-o na sua pasta do projeto:

```
text  
Bankov, Peter  
Holm, Michael  
Garcia, Hugo  
Potra, Cristina  
Noriega, Fabricio  
Aw, Kam Foo  
Beebe, Ann  
Toyoshima, Tim  
Guy, Wey Yuan  
Garcia, Debra
```

2. Copie esses nomes em um arquivo de texto chamado names2.txt e salve-o na sua pasta do projeto: observe que os dois arquivos têm alguns nomes em comum.

```
text  
Liu, Jinghao  
Bankov, Peter  
Holm, Michael  
Garcia, Hugo  
Beebe, Ann  
Gilchrist, Beth  
Myrcha, Jacek  
Giakoumakis, Leo  
McLin, Nkenge  
El Yassir, Mehdi
```

## Exemplo

C#

```
class SplitWithGroups
{
    static void Main()
    {
        string[] fileA =
System.IO.File.ReadAllLines(@"../../names1.txt");
        string[] fileB =
System.IO.File.ReadAllLines(@"../../names2.txt");

        // Concatenate and remove duplicate names based on
        // default string comparer
        var mergeQuery = fileA.Union(fileB);

        // Group the names by the first letter in the last name.
        var groupQuery = from name in mergeQuery
                          let n = name.Split(',')
                          group name by n[0][0] into g
                          orderby g.Key
                          select g;

        // Create a new file for each group that was created
        // Note that nested foreach loops are required to access
        // individual items with each group.
        foreach (var g in groupQuery)
        {
            // Create the new file name.
            string fileName = @"../../testFile_" + g.Key + ".txt";

            // Output to display.
            Console.WriteLine(g.Key);

            // Write file.
            using (System.IO.StreamWriter sw = new
System.IO.StreamWriter(fileName))
            {
                foreach (var item in g)
                {
                    sw.WriteLine(item);
                    // Output to console for example purposes.
                    Console.WriteLine("  {0}", item);
                }
            }
        }
        // Keep console window open in debug mode.
        Console.WriteLine("Files have been written. Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
 A
 Aw, Kam Foo
 B
```

```
Bankov, Peter
Beebe, Ann
E
El Yassir, Mehdi
G
Garcia, Hugo
Guy, Wey Yuan
Garcia, Debra
Gilchrist, Beth
Giakoumakis, Leo
H
Holm, Michael
L
Liu, Jinghao
M
Myrcha, Jacek
McLin, Nkenge
N
Noriega, Fabricio
P
Potra, Cristina
T
Toyoshima, Tim
*/
```

O programa grava um arquivo separado para cada grupo na mesma pasta que os arquivos de dados.

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como unir conteúdo de arquivos diferentes (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo mostra como unir dados de dois arquivos delimitados por vírgulas que compartilham um valor comum que é usado como uma chave correspondente. Essa técnica pode ser útil se você precisa combinar dados de duas planilhas ou de uma planilha e um arquivo com outro formato, em um novo arquivo. Você pode modificar o exemplo para funcionar com qualquer tipo de texto estruturado.

## Para criar os arquivos de dados

1. Copie as seguintes linhas para um arquivo chamado *scores.csv* e salve-o na sua pasta do projeto. O arquivo representa dados da planilha. A coluna 1 é a ID do aluno e as colunas 2 a 5 são resultados de testes.

CSV

```
111, 97, 92, 81, 60  
112, 75, 84, 91, 39  
113, 88, 94, 65, 91  
114, 97, 89, 85, 82  
115, 35, 72, 91, 70  
116, 99, 86, 90, 94  
117, 93, 92, 80, 87  
118, 92, 90, 83, 78  
119, 68, 79, 88, 92  
120, 99, 82, 81, 79  
121, 96, 85, 91, 60  
122, 94, 92, 91, 91
```

2. Copie as seguintes linhas para um arquivo chamado *names.csv* e salve-o na sua pasta do projeto. O arquivo representa uma planilha que contém o sobrenome, o nome e a ID do aluno.

CSV

```
Omelchenko,Svetlana,111  
O'Donnell,Claire,112  
Mortensen,Sven,113  
Garcia,Cesar,114  
Garcia,Debra,115  
Fakhouri,Fadi,116  
Feng,Hanying,117
```

```
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

## Exemplo

C#

```
using System;
using System.Collections.Generic;
using System.Linq;

class JoinStrings
{
    static void Main()
    {
        // Join content from dissimilar files that contain
        // related information. File names.csv contains the student
        // name plus an ID number. File scores.csv contains the ID
        // and a set of four test scores. The following query joins
        // the scores to the student names by using ID as a
        // matching key.

        string[] names = System.IO.File.ReadAllLines(@"../../names.csv");
        string[] scores =
System.IO.File.ReadAllLines(@"../../scores.csv");

        // Name:    Last[0],      First[1],   ID[2]
        //          Omelchenko,   Svetlana,   11
        // Score:   StudentID[0], Exam1[1]   Exam2[2],   Exam3[3],   Exam4[4]
        //          111,           97,         92,         81,         60

        // This query joins two dissimilar spreadsheets based on common ID
value.
        // Multiple from clauses are used instead of a join clause
        // in order to store results of id.Split.
        IEnumerable<string> scoreQuery1 =
            from name in names
            let nameFields = name.Split(',')
            from id in scores
            let scoreFields = id.Split(',')
            where Convert.ToInt32(nameFields[2]) ==
Convert.ToInt32(scoreFields[0])
            select nameFields[0] + "," + scoreFields[1] + "," +
scoreFields[2]
                + "," + scoreFields[3] + "," + scoreFields[4];

        // Pass a query variable to a method and execute it
        // in the method. The query itself is unchanged.
        OutputQueryResults(scoreQuery1, "Merge two spreadsheets:");
    }
}
```

```
// Keep console window open in debug mode.
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

static void OutputQueryResults(IEnumerable<string> query, string
message)
{
    Console.WriteLine(System.Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("{0} total names in list", query.Count());
}
/*
* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
*/
```

# Como calcular valores de coluna em um arquivo de texto CSV (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo mostra como executar cálculos de agregação, como soma, média, mín. e máx. nas colunas de um arquivo .csv. Os princípios de exemplo mostrados aqui podem ser aplicados a outros tipos de texto estruturado.

## Para criar o arquivo de origem

1. Copie as seguintes linhas para um arquivo chamado scores.csv e salve-o na sua pasta do projeto. Suponha que a primeira coluna representa uma ID do aluno e as colunas subsequentes representam as notas de quatro provas.

```
CSV  
111, 97, 92, 81, 60  
112, 75, 84, 91, 39  
113, 88, 94, 65, 91  
114, 97, 89, 85, 82  
115, 35, 72, 91, 70  
116, 99, 86, 90, 94  
117, 93, 92, 80, 87  
118, 92, 90, 83, 78  
119, 68, 79, 88, 92  
120, 99, 82, 81, 79  
121, 96, 85, 91, 60  
122, 94, 92, 91, 91
```

## Exemplo

```
C#  
  
class SumColumns  
{  
    static void Main(string[] args)  
    {  
        string[] lines =  
System.IO.File.ReadAllLines(@"../../scores.csv");  
  
        // Specifies the column to compute.  
        int exam = 3;  
  
        // Spreadsheet format:  
    }  
}
```

```

        // Student ID      Exam#1   Exam#2   Exam#3   Exam#4
        // 111,           97,       92,       81,       60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
{
    Console.WriteLine("Single Column Query:");

    // Parameter examNum specifies the column to
    // run the calculations on. This value could be
    // passed in dynamically at run time.

    // Variable columnQuery is an IEnumerable<int>.
    // The following query performs two steps:
    // 1) use Split to break each row (a string) into an array
    //     of strings,
    // 2) convert the element at position examNum to an int
    //     and select it.
    var columnQuery =
        from line in strs
        let elements = line.Split(',')
        select Convert.ToInt32(elements[examNum]);

    // Execute the query and cache the results to improve
    // performance. This is helpful only with very large files.
    var results = columnQuery.ToList();

    // Perform aggregate calculations Average, Max, and
    // Min on the column specified by examNum.
    double average = results.Average();
    int max = results.Max();
    int min = results.Min();

    Console.WriteLine("Exam #{0}: Average:{1:##.##} High Score:{2} Low
Score:{3}",
                      examNum, average, max, min);
}

static void MultiColumns(IEnumerable<string> strs)
{
    Console.WriteLine("Multi Column Query:");

    // Create a query, multiColQuery. Explicit typing is used
    // to make clear that, when executed, multiColQuery produces
    // nested sequences. However, you get the same results by
    // using 'var'.
}

```

```

// The multiColQuery query performs the following steps:
// 1) use Split to break each row (a string) into an array
//     of strings,
// 2) use Skip to skip the "Student ID" column, and store the
//     rest of the row in scores.
// 3) convert each score in the current row from a string to
//     an int, and select that entire sequence as one row
//     in the results.
IEnumerable<IEnumerable<int>> multiColQuery =
    from line in strs
    let elements = line.Split(',')
    let scores = elements.Skip(1)
    select (from str in scores
            select Convert.ToInt32(str));

// Execute the query and cache the results to improve
// performance.
// ToArray could be used instead of ToList.
var results = multiColQuery.ToList();

// Find out how many columns you have in results.
int columnCount = results[0].Count();

// Perform aggregate calculations Average, Max, and
// Min on each column.
// Perform one iteration of the loop for each column
// of scores.
// You can use a for loop instead of a foreach loop
// because you already executed the multiColQuery
// query by calling ToList.
for (int column = 0; column < columnCount; column++)
{
    var results2 = from row in results
                  select row.ElementAt(column);
    double average = results2.Average();
    int max = results2.Max();
    int min = results2.Min();

    // Add one to column because the first exam is Exam #1,
    // not Exam #0.
    Console.WriteLine("Exam #{0} Average: {1:##.##} High Score: {2}
Low Score: {3}",
                      column + 1, average, max, min);
}
}

/* Output:
Single Column Query:
Exam #4: Average:76.92 High Score:94 Low Score:39

Multi Column Query:
Exam #1 Average: 86.08 High Score: 99 Low Score: 35
Exam #2 Average: 86.42 High Score: 94 Low Score: 72
Exam #3 Average: 84.75 High Score: 91 Low Score: 65

```

A consulta funciona usando o método `Split` para converter cada linha de texto em uma matriz. Cada elemento da matriz representa uma coluna. Por fim, o texto em cada coluna é convertido em sua representação numérica. Se o arquivo for um arquivo separado por tabulações, é só atualizar o argumento no método `Split` para `\t`.

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como consultar metadados de um assembly com reflexão (LINQ)

Artigo • 20/03/2023

Você usa as APIs de reflexão do .NET para examinar os metadados no assembly do .NET e criar coleções de tipos, membros de tipo e parâmetros que estão nesse assembly. Como essas coleções dão suporte à interface `IEnumerable<T>` genéricas, elas podem ser consultadas usando LINQ.

O exemplo a seguir mostra como o LINQ pode ser usado com a reflexão para recuperar metadados específicos sobre os métodos que correspondem a um critério de pesquisa especificado. Nesse caso, a consulta localiza os nomes de todos os métodos no assembly que retornam tipos enumeráveis como matrizes.

C#

```
Assembly assembly = Assembly.Load("System.Private.CoreLib, Version=7.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e");
var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                    from method in type.GetMethods()
                    where method.ReturnType.IsArray == true
                          || (method.ReturnType.GetInterface(
typeof(System.Collections.Generic.IEnumerable<>)).FullName != null
                            && method.ReturnType.FullName != "System.String")
                    group method.ToString() by type.ToString();

foreach (var groupOfMethods in pubTypesQuery)
{
    Console.WriteLine("Type: {0}", groupOfMethods.Key);
    foreach (var method in groupOfMethods)
    {
        Console.WriteLine("  {0}", method);
    }
}
```

O exemplo usa o método `Assembly.GetTypes` para retornar uma matriz de tipos no assembly especificado. O filtro `where` é aplicado para que apenas tipos públicos sejam retornados. Para cada tipo de público, uma subconsulta é gerada usando a matriz `MethodInfo` que é retornada da chamada `Type.GetMethods`. Esses resultados são filtrados para retornar apenas os métodos cujo tipo de retorno é uma matriz ou um tipo que implementa `IEnumerable<T>`. Por fim, esses resultados são agrupados usando o nome do tipo como uma chave.

# Como consultar metadados de um assembly com reflexão (LINQ)

Artigo • 20/03/2023

Você usa as APIs de reflexão do .NET para examinar os metadados no assembly do .NET e criar coleções de tipos, membros de tipo e parâmetros que estão nesse assembly. Como essas coleções dão suporte à interface `IEnumerable<T>` genéricas, elas podem ser consultadas usando LINQ.

O exemplo a seguir mostra como o LINQ pode ser usado com a reflexão para recuperar metadados específicos sobre os métodos que correspondem a um critério de pesquisa especificado. Nesse caso, a consulta localiza os nomes de todos os métodos no assembly que retornam tipos enumeráveis como matrizes.

C#

```
Assembly assembly = Assembly.Load("System.Private.CoreLib, Version=7.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e");
var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                    from method in type.GetMethods()
                    where method.ReturnType.IsArray == true
                          || (method.ReturnType.GetInterface(
typeof(System.Collections.Generic.IEnumerable<>)).FullName != null
                            && method.ReturnType.FullName != "System.String")
                    group method.ToString() by type.ToString();

foreach (var groupOfMethods in pubTypesQuery)
{
    Console.WriteLine("Type: {0}", groupOfMethods.Key);
    foreach (var method in groupOfMethods)
    {
        Console.WriteLine("  {0}", method);
    }
}
```

O exemplo usa o método `Assembly.GetTypes` para retornar uma matriz de tipos no assembly especificado. O filtro `where` é aplicado para que apenas tipos públicos sejam retornados. Para cada tipo de público, uma subconsulta é gerada usando a matriz `MethodInfo` que é retornada da chamada `Type.GetMethods`. Esses resultados são filtrados para retornar apenas os métodos cujo tipo de retorno é uma matriz ou um tipo que implementa `IEnumerable<T>`. Por fim, esses resultados são agrupados usando o nome do tipo como uma chave.

# LINQ e diretórios de arquivos (C#)

Artigo • 07/04/2023

Muitas operações do sistema de arquivos são essencialmente consultas e, portanto, são ideais para a abordagem do LINQ.

As consultas nesta seção não são destrutivas. Elas não são usadas para alterar o conteúdo dos arquivos ou pastas originais. Isso segue a regra de que consultas não devem causar efeitos colaterais. Em geral, qualquer código (incluindo consultas que executam operadores criar / atualizar / excluir) que modifique dados de origem deve ser mantido separado do código que apenas consulta os dados.

Esta seção contém os seguintes tópicos:

## [Como consultar arquivos com um atributo ou nome especificado \(C#\)](#)

Mostra como pesquisar arquivos, examinando uma ou mais propriedades de seu objeto [FileInfo](#).

## [Como agrupar arquivos por extensão \(LINQ\) \(C#\)](#)

Mostra como retornar grupos de objetos [FileInfo](#) com base em sua extensão de nome de arquivo.

## [Como consultar o número total de bytes em um conjunto de pastas \(LINQ\) \(C#\)](#)

Mostra como retornar o número total de bytes em todos os arquivos de uma árvore de diretórios especificada.

## [Como comparar o conteúdo de duas pastas \(LINQ\) \(C#\)](#)

Mostra como retornar todos os arquivos que estão presentes em duas pastas especificadas e também todos os arquivos que estão presentes em uma pasta, mas não na outra.

## [Como consultar o maior arquivo ou arquivos em uma árvore de diretório \(LINQ\) \(C#\)](#)

Mostra como retornar o maior ou o menor arquivo ou um número especificado de arquivos, em uma árvore de diretório.

## [Como consultar arquivos duplicados em uma árvore de diretório \(LINQ\) \(C#\)](#)

Mostra como agrupar todos os nomes de arquivo que ocorrem em mais de um local em uma árvore de diretórios especificada. Também mostra como realizar comparações mais complexas com base em um comparador personalizado.

## [Como consultar o conteúdo de arquivos em uma pasta \(LINQ\) \(C#\)](#)

Mostra como iterar pelas pastas em uma árvore, abrir cada arquivo e consultar o conteúdo do arquivo.

# Comentários

Há certa complexidade envolvida na criação de uma fonte de dados que representa o conteúdo do sistema de arquivos com precisão e trata exceções de maneira elegante. Os exemplos nesta seção criam uma coleção de instantâneos de objetos [FileInfo](#) que representa todos os arquivos em uma pasta raiz especificada e todas as suas subpastas. O estado real de cada [FileInfo](#) pode ser alterado no tempo entre o momento em que você começa e termina a execução de uma consulta. Por exemplo, você pode criar uma lista de objetos [FileInfo](#) para usar como uma fonte de dados. Se você tentar acessar a propriedade `Length` em uma consulta, o objeto [FileInfo](#) tentará acessar o sistema de arquivos para atualizar o valor de `Length`. Se o arquivo não existir, você obterá uma [FileNotFoundException](#) em sua consulta, embora não esteja consultando diretamente o sistema de arquivos. Algumas consultas nesta seção usam um método separado que consome essas exceções específicas em determinados casos. Outra opção é manter a fonte de dados atualizada dinamicamente usando o [FileSystemWatcher](#).

## Confira também

- [LINQ to Objects \(C#\)](#)

# Como consultar arquivos com um atributo ou nome especificado (C#)

Artigo • 20/07/2023

Este exemplo mostra como localizar todos os arquivos que têm uma extensão de nome de arquivo especificada (por exemplo ".txt") em uma árvore de diretório especificada. Ele também mostra como retornar tanto os arquivos mais recentes como os mais antigo na árvore com base na hora de criação.

## Exemplo

C#

```
class FindFileByExtension
{
    // This query will produce the full path for all .txt files
    // under the specified folder including subfolders.
    // It orders the list according to the file name.
    static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery
permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.",
System.IO.SearchOption.AllDirectories);

        //Create the query
        IEnumerable<System.IO.FileInfo> fileQuery =
            from file in fileList
            where file.Extension == ".txt"
            orderby file.Name
            select file;

        //Execute the query. This might write out a lot of files!
        foreach (System.IO.FileInfo fi in fileQuery)
        {
            Console.WriteLine(fi.FullName);
        }

        // Create and execute a new query by using the previous
        // query as a starting point. fileQuery is not
```

```
// executed again until the call to Last()
var newestFile =
    (from file in fileQuery
     orderby file.CreationTime
     select new { file.FullName, file.CreationTime })
    .Last();

Console.WriteLine("\r\nThe newest .txt file is {0}. Creation time:
{1}",
    newestFile.FullName, newestFile.CreationTime);

// Keep the console window open in debug mode.
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}
```

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como agrupar arquivos por extensão (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo mostra como o LINQ pode ser usado para realizar operações avançadas de classificação e agrupamento em listas de arquivos ou pastas. Ele também mostra como paginar a saída na janela do console usando os métodos [Skip](#) e [Take](#).

## Exemplo

A consulta a seguir mostra como agrupar o conteúdo de uma árvore de diretórios especificada, pela extensão de nome de arquivo.

C#

```
class GroupByExtension
{
    // This query will sort all the files under the specified folder
    // and subfolder into groups keyed by the file extension.
    private static void Main()
    {
        // Take a snapshot of the file system.
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\Common7";

        // Used in WriteLine to trim output lines.
        int trimLength = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery
permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        // Create the query.
        var queryGroupByExt =
            from file in fileList
            group file by file.Extension.ToLower() into fileGroup
            orderby fileGroup.Key
            select fileGroup;

        // Display one group at a time. If the number of
        // entries is greater than the number of lines
```

```

        // in the console window, then page the output.
        PageOutput(trimLength, queryGroupByExt);
    }

    // This method specifically handles group queries of FileInfo objects
    // with string keys.
    // It can be modified to work for any long listings of data. Note that
    // explicit typing
    // must be used in method signatures. The groupbyExtList parameter is a
    // query that produces
    // groups of FileInfo objects with string keys.
    private static void PageOutput(int rootLength,
        IEnumerable<System.Linq.IGrouping<string, System.IO.FileInfo>>
        groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input
        // cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
            int currentLine = 0;

            // Output only as many lines of the current group as will fit in
            // the window.
            do
            {
                Console.Clear();
                Console.WriteLine(filegroup.Key == String.Empty ? "[none]" :
filegroup.Key);

                // Get 'numLines' number of items starting at number
                // 'currentLine'.
                var resultPage = filegroup.Skip(currentLine).Take(numLines);

                //Execute the resultPage query
                foreach (var f in resultPage)
                {
                    Console.WriteLine("\t{0}",
f.FullName.Substring(rootLength));
                }

                // Increment the line counter.
                currentLine += numLines;

                // Give the user a chance to escape.
                Console.WriteLine("Press any key to continue or the 'End'
key to break...");

                ConsoleKey key = Console.ReadKey().Key;
            }
        }
    }
}

```

```
        if (key == ConsoleKey.End)
    {
        goAgain = false;
        break;
    }
} while (currentLine < filegroup.Count());

if (goAgain == false)
    break;
}
}
```

A saída desse programa pode ser longa, dependendo dos detalhes do sistema de arquivos local e o que está definido em `startFolder`. Para habilitar a exibição de todos os resultados, este exemplo mostra como paginá-los. As mesmas técnicas podem ser aplicadas a aplicativos do Windows e aplicativos Web. Observe que, como o código dispõe os itens em um grupo, é necessário um loop `foreach` aninhado. Há também alguma lógica adicional para calcular a posição atual na lista e para permitir que o usuário interrompa a paginação e saia do programa. Nesse caso específico, a consulta de paginação é executada nos resultados da consulta original armazenados em cache. Em outros contextos, como LINQ to SQL, esse cache não é necessário.

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como consultar o número total de bytes em um conjunto de pastas (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo mostra como recuperar o número total de bytes usado por todos os arquivos em uma pasta especificada e todas as suas subpastas.

## Exemplo

O método `Sum` adiciona os valores de todos os itens selecionados na cláusula `select`. Você pode modificar essa consulta para recuperar o maior ou o menor arquivo na árvore de diretório especificada chamando o método `Min` ou `Max` em vez de `Sum`.

C#

```
class QuerySize
{
    public static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\VC#";

        // Take a snapshot of the file system.
        // This method assumes that the application has discovery
permissions
        // for all folders under the specified path.
        IEnumerable<string> fileList =
System.IO.Directory.GetFiles(startFolder, "*.*",
System.IO.SearchOption.AllDirectories);

        var fileQuery = from file in fileList
                        select GetFileLength(file);

        // Cache the results to avoid multiple trips to the file system.
        long[] fileLengths = fileQuery.ToArray();

        // Return the size of the largest file
        long largestFile = fileLengths.Max();

        // Return the total number of bytes in all the files under the
specified folder.
        long totalBytes = fileLengths.Sum();

        Console.WriteLine("There are {0} bytes in {1} files under {2}",
totalBytes, fileList.Count(), startFolder);
        Console.WriteLine("The largest files is {0} bytes.", largestFile);
```

```

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the System.IO.FileInfo.Length
    property.
    static long GetFileLength(string filename)
    {
        long retval;
        try
        {
            System.IO.FileInfo fi = new System.IO.FileInfo(filename);
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }
}

```

Se precisar apenas contar o número de bytes em uma árvore de diretório especificada, você pode fazer isso com mais eficiência sem criar uma consulta LINQ, que gera a sobrecarga de criação da coleção de lista como uma fonte de dados. A utilidade da abordagem da LINQ aumenta conforme a consulta se torna mais complexa ou quando você precisa executar várias consultas na mesma fonte de dados.

A consulta chama um método separado para obter o tamanho do arquivo. Ela faz isso para consumir a possível exceção que será gerada se o arquivo tiver sido excluído em outro thread após o objeto `FileInfo` ter sido criado na chamada para `GetFiles`. Embora o objeto `FileInfo` já tenha sido criado, a exceção poderá ocorrer porque um objeto `FileInfo` tentará atualizar sua propriedade `Length` com o tamanho mais atual na primeira vez que a propriedade foi acessada. Ao colocar essa operação em um bloco `try-catch` fora da consulta, o código segue a regra de evitar operações em consultas que podem causar efeitos colaterais. Em geral, deve-se ter muito cuidado ao consumir exceções para garantir que um aplicativo não seja deixado em um estado desconhecido.

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como comparar o conteúdo de duas pastas (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo demonstra três modos de se comparar duas listagens de arquivo:

- Consultando um valor booleano que especifica se as duas listas de arquivos são idênticas.
- Consultando a interseção para recuperar os arquivos que estão em ambas as pastas.
- Consultando a diferença de conjunto para recuperar os arquivos que estão em uma pasta, mas não na outra.

## Observação

As técnicas mostradas aqui podem ser adaptadas para comparar sequências de objetos de qualquer tipo.

A classe `FileComparer` mostrada aqui demonstra como usar uma classe de comparação personalizada junto com operadores de consulta padrão. A classe não se destina ao uso em cenários do mundo real. Ela apenas utiliza o nome e o comprimento em bytes de cada arquivo para determinar se o conteúdo de cada pasta é idêntico ou não. Em um cenário do mundo real, você deve modificar esse comparador para executar uma verificação mais rigorosa de igualdade.

## Exemplo

C#

```
namespace QueryCompareTwoDirs
{
    class CompareDirs
    {

        static void Main(string[] args)
        {

            // Create two identical or different temporary folders
            // on a local drive and change these file paths.
            string pathA = @"C:\TestDir";
```

```

        string pathB = @"C:\TestDir2";

        System.IO.DirectoryInfo dir1 = new
System.IO.DirectoryInfo(pathA);
        System.IO.DirectoryInfo dir2 = new
System.IO.DirectoryInfo(pathB);

        // Take a snapshot of the file system.
        IEnumerable<System.IO.FileInfo> list1 = dir1.GetFiles(".*",
System.IO.SearchOption.AllDirectories);
        IEnumerable<System.IO.FileInfo> list2 = dir2.GetFiles(".*",
System.IO.SearchOption.AllDirectories);

        //A custom file comparer defined below
        FileCompare myFileCompare = new FileCompare();

        // This query determines whether the two folders contain
        // identical file lists, based on the custom file comparer
        // that is defined in the FileCompare class.
        // The query executes immediately because it returns a bool.
        bool areIdentical = list1.SequenceEqual(list2, myFileCompare);

        if (areIdentical == true)
        {
            Console.WriteLine("the two folders are the same");
        }
        else
        {
            Console.WriteLine("The two folders are not the same");
        }

        // Find the common files. It produces a sequence and doesn't
        // execute until the foreach statement.
        var queryCommonFiles = list1.Intersect(list2, myFileCompare);

        if (queryCommonFiles.Any())
        {
            Console.WriteLine("The following files are in both
folders:");
            foreach (var v in queryCommonFiles)
            {
                Console.WriteLine(v.FullName); //shows which items end
up in result list
            }
        }
        else
        {
            Console.WriteLine("There are no common files in the two
folders.");
        }

        // Find the set difference between the two folders.
        // For this example we only check one way.
        var queryList1Only = (from file in list1
                            select file).Except(list2, myFileCompare);

```

```

        Console.WriteLine("The following files are in list1 but not
list2:");
        foreach (var v in queryList1Only)
        {
            Console.WriteLine(v.FullName);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

// This implementation defines a very simple comparison
// between two FileInfo objects. It only compares the name
// of the files being compared and their length in bytes.
class FileCompare :
System.Collections.Generic.IEqualityComparer<System.IO.FileInfo>
{
    public FileCompare() { }

    public bool Equals(System.IO.FileInfo f1, System.IO.FileInfo f2)
    {
        return (f1.Name == f2.Name &&
                f1.Length == f2.Length);
    }

    // Return a hash that reflects the comparison criteria. According to
    // the
    // rules for IEqualityComparer<T>, if Equals is true, then the hash
    // codes must
    // also be equal. Because equality as defined here is a simple value
    // equality, not
    // reference identity, it is possible that two or more objects will
    // produce the same
    // hash code.
    public int GetHashCode(System.IO.FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}
}

```

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como consultar o maior arquivo ou arquivos em uma árvore de diretório (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo mostra cinco consultas relacionadas ao tamanho do arquivo em bytes:

- Como recuperar o tamanho em bytes do maior arquivo.
- Como recuperar o tamanho em bytes do menor arquivo.
- Como recuperar o maior ou menor arquivo do objeto [FileInfo](#) de uma ou mais pastas em uma pasta raiz especificada.
- Como recuperar uma sequência, como os 10 maiores arquivos.
- Como ordenar os arquivos em grupos com base no tamanho do arquivo em bytes, ignorando arquivos menores do que um tamanho especificado.

## Exemplo

O exemplo a seguir contém cinco consultas separadas que mostram como consultar e agrupar arquivos, dependendo do tamanho do arquivo em bytes. Você pode modificar facilmente esses exemplos para basear a consulta em outra propriedade do objeto [FileInfo](#).

C#

```
class QueryBySize
{
    static void Main(string[] args)
    {
        QueryFilesBySize();
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    private static void QueryFilesBySize()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
```

```

System.IO.DirectoryInfo(startFolder);

    // This method assumes that the application has discovery
permissions
    // for all folders under the specified path.
    IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

    //Return the size of the largest file
    long maxSize =
        (from file in fileList
         let len = GetFileLength(file)
         select len)
        .Max();

    Console.WriteLine("The length of the largest file under {0} is {1}",
                      startFolder, maxSize);

    // Return the FileInfo object for the largest file
    // by sorting and selecting from beginning of list
    System.IO.FileInfo longestFile =
        (from file in fileList
         let len = GetFileLength(file)
         where len > 0
         orderby len descending
         select file)
        .First();

    Console.WriteLine("The largest file under {0} is {1} with a length
of {2} bytes",
                      startFolder, longestFile.FullName,
                      longestFile.Length);

    //Return the FileInfo of the smallest file
    System.IO.FileInfo smallestFile =
        (from file in fileList
         let len = GetFileLength(file)
         where len > 0
         orderby len ascending
         select file).First();

    Console.WriteLine("The smallest file under {0} is {1} with a length
of {2} bytes",
                      startFolder, smallestFile.FullName,
                      smallestFile.Length);

    //Return the FileInfos for the 10 largest files
    // queryTenLargest is an IEnumerable<System.IO.FileInfo>
    var queryTenLargest =
        (from file in fileList
         let len = GetFileLength(file)
         orderby len descending
         select file).Take(10);

    Console.WriteLine("The 10 largest files under {0} are:",

```

```

startFolder);

    foreach (var v in queryTenLargest)
    {
        Console.WriteLine("{0}: {1} bytes", v.FullName, v.Length);
    }

    // Group the files according to their size, leaving out
    // files that are less than 200000 bytes.
    var querySizeGroups =
        from file in fileList
        let len = GetFileLength(file)
        where len > 0
        group file by (len / 100000) into fileGroup
        where fileGroup.Key >= 2
        orderby fileGroup.Key descending
        select fileGroup;

    foreach (var filegroup in querySizeGroups)
    {
        Console.WriteLine(filegroup.Key.ToString() + "0000");
        foreach (var item in filegroup)
        {
            Console.WriteLine("\t{0}: {1}", item.Name, item.Length);
        }
    }
}

// This method is used to swallow the possible exception
// that can be raised when accessing the FileInfo.Length property.
// In this particular case, it is safe to swallow the exception.
static long GetFileLength(System.IO.FileInfo fi)
{
    long retval;
    try
    {
        retval = fi.Length;
    }
    catch (System.IO.FileNotFoundException)
    {
        // If a file is no longer present,
        // just add zero bytes to the total.
        retval = 0;
    }
    return retval;
}

}

```

Para retornar um ou mais objetos [FileInfo](#) completos, a consulta deve primeiro examinar cada um dos objetos na fonte de dados e, em seguida, classificá-los segundo o valor de sua propriedade Length. Em seguida, ela pode retornar um único elemento ou a sequência com os maiores tamanhos. Use [First](#) para retornar o primeiro elemento em

uma lista. Use `Take` para retornar o primeiro número  $n$  de elementos. Especifique uma ordem de classificação decrescente para colocar os menores elementos no início da lista.

A consulta chama um método separado para obter o tamanho do arquivo em bytes para consumir a exceção possível que ocorrerá caso um arquivo tenha sido excluído em outro thread no período desde que o objeto `FileInfo` foi criado na chamada para `GetFiles`. Embora o objeto `FileInfo` já tenha sido criado, a exceção poderá ocorrer porque um objeto `FileInfo` tentará atualizar sua propriedade `Length` usando o tamanho mais atual em bytes na primeira vez que a propriedade foi acessada. Ao colocar essa operação em um bloco try-catch fora da consulta, nós seguimos a regra de evitar operações em consultas que podem causar efeitos colaterais. Em geral, deve-se ter muito cuidado ao consumir exceções para garantir que um aplicativo não seja deixado em um estado desconhecido.

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como consultar arquivos duplicados em uma árvore de diretório (LINQ) (C#)

Artigo • 20/07/2023

Às vezes, arquivos que têm o mesmo nome podem ser localizados em mais de uma pasta. Por exemplo, sob a pasta de instalação do Visual Studio, várias pastas têm um arquivo readme.htm. Este exemplo mostra como consultar esses nomes de arquivos duplicados sob uma pasta raiz especificada. O segundo exemplo mostra como consultar arquivos cujo tamanho e os tempos LastWrite também são semelhantes.

## Exemplo

C#

```
class QueryDuplicateFileNames
{
    static void Main(string[] args)
    {
        // Uncomment QueryDuplicates2 to run that query.
        QueryDuplicates();
        // QueryDuplicates2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryDuplicates()
    {
        // Change the root drive or folder if necessary
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery
permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        // used in WriteLine to keep the lines shorter
        int charsToSkip = startFolder.Length;

        // var can be used for convenience with groups.
    }
}
```

```

        var queryDupNames =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by file.Name into
fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        // Pass the query to a method that will
        // output one page at a time.
        PageOutput<string, string>(queryDupNames);
    }

    // A Group key that can be passed to a separate method.
    // Override Equals and GetHashCode to define equality for the key.
    // Override ToString to provide a friendly name for Key.ToString()
    class PortableKey
    {
        public string Name { get; set; }
        public DateTime LastWriteTime { get; set; }
        public long Length { get; set; }

        public override bool Equals(object obj)
        {
            PortableKey other = (PortableKey)obj;
            return other.LastWriteTime == this.LastWriteTime &&
                other.Length == this.Length &&
                other.Name == this.Name;
        }

        public override int GetHashCode()
        {
            string str = $"{this.LastWriteTime}{this.Length}{this.Name}";
            return str.GetHashCode();
        }

        public override string ToString()
        {
            return $"{this.Name} {this.Length} {this.LastWriteTime}";
        }
    }
    static void QueryDuplicates2()
    {
        // Change the root drive or folder if necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\Common7";

        // Make the lines shorter for the console display
        int charsToSkip = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
System.IO.DirectoryInfo(startFolder);
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        // Note the use of a compound key. Files that match

```

```

        // all three properties belong to the same group.
        // A named type is used to enable the query to be
        // passed to another method. Anonymous types can also be used
        // for composite keys but cannot be passed across method boundaries
        //
        var queryDupFiles =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by
                new PortableKey { Name = file.Name, LastWriteTime =
file.LastWriteTime, Length = file.Length } into fileGroup
                where fileGroup.Count() > 1
                select fileGroup;

        var list = queryDupFiles.ToList();

        int i = queryDupFiles.Count();

        PageOutput<PortableKey, string>(queryDupFiles);
    }

    // A generic method to page the output of the QueryDuplications methods
    // Here the type of the group must be specified explicitly. "var" cannot
    // be used in method signatures. This method does not display more than
    one
    // group per page.
    private static void PageOutput<K, V>
(IEnumerable<System.Linq.IGrouping<K, V>> groupByExtList)
{
    // Flag to break out of paging loop.
    bool goAgain = true;

    // "3" = 1 line for extension + 1 for "Press any key" + 1 for input
cursor.
    int numLines = Console.WindowHeight - 3;

    // Iterate through the outer collection of groups.
    foreach (var filegroup in groupByExtList)
    {
        // Start a new extension at the top of a page.
        int currentLine = 0;

        // Output only as many lines of the current group as will fit in
the window.
        do
        {
            Console.Clear();
            Console.WriteLine("Filename = {0}", filegroup.Key.ToString()
== String.Empty ? "[none]" : filegroup.Key.ToString());

            // Get 'numLines' number of items starting at number
            'currentLine'.
            var resultPage = filegroup.Skip(currentLine).Take(numLines);

            //Execute the resultPage query
            foreach (var fileName in resultPage)

```

```
        {
            Console.WriteLine("\t{0}", fileName);
        }

        // Increment the line counter.
        currentLine += numLines;

        // Give the user a chance to escape.
        Console.WriteLine("Press any key to continue or the 'End' key to break...");
```

ConsoleKey key = Console.ReadKey().Key;

```
        if (key == ConsoleKey.End)
        {
            goAgain = false;
            break;
        }
    } while (currentLine < filegroup.Count());
```

```
    if (goAgain == false)
        break;
}
```

A primeira consulta usa uma chave simples para determinar uma correspondência. Ela localiza arquivos que têm o mesmo nome, mas cujo conteúdo pode ser diferente. A segunda consulta usa uma chave composta para comparar em relação a três propriedades do objeto [FileInfo](#). É muito mais provável que essa consulta localize arquivos que têm o mesmo nome e conteúdo semelhante ou idêntico.

# Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como consultar o conteúdo de arquivos de texto em uma pasta (LINQ) (C#)

Artigo • 20/07/2023

Este exemplo mostra como consultar todos os arquivos em uma árvore de diretório especificada, abrir cada arquivo e inspecionar seu conteúdo. Este tipo de técnica pode ser usado para criar índices ou inverter os índices do conteúdo de uma árvore de diretório. Uma pesquisa de cadeia de caracteres simples é executada neste exemplo. No entanto, os tipos de correspondência de padrões mais complexos podem ser executados com uma expressão regular. Para obter mais informações, consulte [Como combinar consultas LINQ com expressões regulares \(C#\)](#).

## Exemplo

C#

```
class QueryContents
{
    public static void Main()
    {
        // Modify this path as necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery
permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        string searchTerm = @"Visual Studio";

        // Search the contents of each file.
        // A regular expression created with the RegEx class
        // could be used instead of the Contains method.
        // queryMatchingFiles is an IEnumerable<string>.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = GetFileText(file.FullName)
            where fileText.Contains(searchTerm)
            select file.FullName;
```

```
// Execute the query.  
Console.WriteLine("The term \"{0}\" was found in:", searchTerm);  
foreach (string filename in queryMatchingFiles)  
{  
    Console.WriteLine(filename);  
}  
  
// Keep the console window open in debug mode.  
Console.WriteLine("Press any key to exit");  
Console.ReadKey();  
}  
  
// Read the contents of the file.  
static string GetFileText(string name)  
{  
    string fileContents = String.Empty;  
  
    // If the file has been deleted since we took  
    // the snapshot, ignore it and return the empty string.  
    if (System.IO.File.Exists(name))  
    {  
        fileContents = System.IO.File.ReadAllText(name);  
    }  
    return fileContents;  
}  
}
```

## Compilando o código

Criar um projeto de aplicativo de console em C# com diretivas `using` para os namespaces `System.Linq` e `System.IO`.

# Como consultar uma ArrayList com LINQ (C#)

Artigo • 20/07/2023

Ao usar a LINQ para consultar coleções `IEnumerable` não genéricas como `ArrayList`, você deve declarar explicitamente o tipo da variável de intervalo para refletir o tipo específico dos objetos na coleção. Por exemplo, se você tiver um `ArrayList` de objetos `Student`, sua cláusula `from` deverá ter uma aparência semelhante a esta:

C#

```
var query = from Student s in arrList  
//...
```

Especificando o tipo da variável de intervalo, você está convertendo cada item na `ArrayList` em um `Student`.

O uso de uma variável de intervalo de tipo explícito em uma expressão de consulta é equivalente a chamar o método `Cast`. `Cast` lança uma exceção se a conversão especificada não puder ser realizada. `Cast` e `OfType` são os dois métodos de operador de consulta padrão que operam em tipos `IEnumerable` não genéricos. Para obter mais informações, consulte [Relacionamentos de tipo em operações de consulta LINQ](#).

## Exemplo

O exemplo a seguir mostra uma consulta simples sobre um `ArrayList`. Observe que este exemplo usa os inicializadores de objeto quando o código chama o método `Add`, mas isso não é um requisito.

C#

```
using System;  
using System.Collections;  
using System.Linq;  
  
namespace NonGenericLINQ  
{  
    public class Student  
    {  
        public string FirstName { get; set; }  
        public string LastName { get; set; }  
        public int[] Scores { get; set; }  
    }  
}
```

```

class Program
{
    static void Main(string[] args)
    {
        ArrayList arrList = new ArrayList();
        arrList.Add(
            new Student
            {
                FirstName = "Svetlana", LastName = "Omelchenko",
                Scores = new int[] { 98, 92, 81, 60 }
            });
        arrList.Add(
            new Student
            {
                FirstName = "Claire", LastName = "O'Donnell", Scores =
                new int[] { 75, 84, 91, 39 }
            });
        arrList.Add(
            new Student
            {
                FirstName = "Sven", LastName = "Mortensen", Scores =
                new int[] { 88, 94, 65, 91 }
            });
        arrList.Add(
            new Student
            {
                FirstName = "Cesar", LastName = "Garcia", Scores =
                new int[] { 97, 89, 85, 82 }
            });

        var query = from Student student in arrList
                   where student.Scores[0] > 95
                   select student;

        foreach (Student s in query)
            Console.WriteLine(s.LastName + ": " + s.Scores[0]);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Omelchenko: 98
   Garcia: 97
*/

```

# Como adicionar métodos personalizados para consultas LINQ (C#)

Artigo • 07/04/2023

Você pode estender o conjunto de métodos que usa para consultas LINQ adicionando métodos de extensão à interface `IEnumerable<T>`. Por exemplo, além do padrão médio ou máximo de operações, é possível criar um método de agregação personalizado para calcular um único valor de uma sequência de valores. Também é possível criar um método que funciona como filtro personalizado ou transformação de dados específicos para uma sequência de valores e retorna uma nova sequência. Exemplos desses métodos são [Distinct](#), [Skip](#) e [Reverse](#).

Ao estender a interface `IEnumerable<T>`, você pode aplicar seus métodos personalizados para qualquer coleção enumerável. Para obter mais informações, consulte [Métodos de extensão](#).

## Adicionar um método de agregação

Um método de agregação calcula um valor único de um conjunto de valores. O LINQ fornece vários métodos de agregação, incluindo [Average](#), [Min](#) e [Max](#). Você pode criar seu próprio método de agregação, adicionando um método de extensão à interface `IEnumerable<T>`.

O exemplo de código a seguir mostra como criar um método de extensão chamado `Median` para calcular uma mediana de uma sequência de números do tipo `double`.

C#

```
public static class EnumerableExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a
null or empty set.");
        }

        var sortedList =
            source.OrderBy(number => number).ToList();

        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
```

```
    {
        // Even number of items.
        return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
    }
    else
    {
        // Odd number of items.
        return sortedList[itemIndex];
    }
}
}
```

Você chama esse método de extensão para qualquer coleção enumerável da mesma maneira que chama outros métodos de agregação da interface `IEnumerable<T>`.

O exemplo de código a seguir mostra como usar o método `Median` para uma matriz do tipo `double`.

C#

```
double[] numbers = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };
var query = numbers.Median();

Console.WriteLine($"double: Median = {query}");
// This code produces the following output:
//      double: Median = 4.85
```

## Sobrecarregar um método de agregação para aceitar vários tipos

Você pode sobrecarregar o método de agregação para que ele aceite sequências de vários tipos. A abordagem padrão é criar uma sobrecarga para cada tipo. Outra abordagem é criar uma sobrecarga que vai pegar um tipo genérico e convertê-lo em um tipo específico, usando um delegado. Você também pode combinar as duas abordagens.

### Criar uma sobrecarga para cada tipo

Você pode criar uma sobrecarga específica para cada tipo que deseja oferecer suporte. O exemplo de código a seguir mostra uma sobrecarga do método `Median` para o tipo `int`.

C#

```
// int overload
public static double Median(this IEnumerable<int> source) =>
    (from number in source select (double)number).Median();
```

Agora você pode chamar as sobrecargas `Median` para os tipos `integer` e `double`, conforme mostrado no código a seguir:

C#

```
double[] numbers1 = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };
var query1 = numbers1.Median();

Console.WriteLine($"double: Median = {query1}");

int[] numbers2 = { 1, 2, 3, 4, 5 };
var query2 = numbers2.Median();

Console.WriteLine($"int: Median = {query2}");
// This code produces the following output:
//      double: Median = 4.85
//      int: Median = 3
```

## Criar uma sobrecarga genérica

Você também pode criar uma sobrecarga que aceita uma sequência de objetos genéricos. Essa sobrecarga recebe um delegado como parâmetro e usa-o para converter uma sequência de objetos de um tipo genérico em um tipo específico.

O código a seguir mostra uma sobrecarga do método `Median` que recebe o delegado `Func<T,TResult>` como um parâmetro. Esse delegado recebe um objeto de tipo genérico `T` e retorna um objeto do tipo `double`.

C#

```
// generic overload
public static double Median<T>(
    this IEnumerable<T> numbers, Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();
```

Agora você pode chamar o método `Median` para uma sequência de objetos de qualquer tipo. Se o tipo não tiver sua própria sobrecarga de método, será necessário passar um parâmetro delegado. No C# você pode usar uma expressão lambda para essa finalidade. Além disso, no Visual Basic, se você usar a cláusula `Aggregate` ou `Group By` em vez da

chamada de método, você pode passar qualquer valor ou expressão que estiver no escopo dessa cláusula.

O exemplo de código a seguir mostra como chamar o método `Median` para uma matriz de inteiros e para uma matriz de cadeias de caracteres. Será calculada a mediana dos comprimentos das cadeias de caracteres na matriz. O exemplo também mostra como passar o parâmetro delegado `Func<T,TResult>` ao método `Median` para cada caso.

C#

```
int[] numbers3 = { 1, 2, 3, 4, 5 };

/*
    You can use the num => num lambda expression as a parameter for the
    Median method
        so that the compiler will implicitly convert its value to double.
        If there is no implicit conversion, the compiler will display an error
        message.
*/
var query3 = numbers3.Median(num => num);

Console.WriteLine($"int: Median = {query3}");

string[] numbers4 = { "one", "two", "three", "four", "five" };

// With the generic overload, you can also use numeric properties of
// objects.
var query4 = numbers4.Median(str => str.Length);

Console.WriteLine($"string: Median = {query4}");
// This code produces the following output:
//      int: Median = 3
//      string: Median = 4
```

## Adicionar um método que retorna uma sequência

Você pode estender a interface `IEnumerable<T>` com um método de consulta personalizada que retorna uma sequência de valores. Nesse caso, o método deve retornar uma coleção do tipo `IEnumerable<T>`. Esses métodos podem ser usados para aplicar transformações de dados ou filtros a uma sequência de valores.

O exemplo a seguir mostra como criar um método de extensão chamado `AlternateElements` que retorna todos os outros elementos em uma coleção, começando pelo primeiro elemento.

C#

```
// Extension method for the IEnumerable<T> interface.  
// The method returns every other element of a sequence.  
public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T>  
source)  
{  
    int index = 0;  
    foreach (T element in source)  
    {  
        if (index % 2 == 0)  
        {  
            yield return element;  
        }  
  
        index++;  
    }  
}
```

Você pode chamar esse método de extensão para qualquer coleção enumerável exatamente como chamaria outros métodos da interface `IEnumerable<T>`, conforme mostrado no código a seguir:

C#

```
string[] strings = { "a", "b", "c", "d", "e" };  
  
var query5 = stringsAlternateElements();  
  
foreach (var element in query5)  
{  
    Console.WriteLine(element);  
}  
// This code produces the following output:  
//      a  
//      c  
//      e
```

## Confira também

- [IEnumerable<T>](#)
- [Métodos de Extensão](#)

# LINQ (Consulta Integrada à Linguagem)

Artigo • 15/02/2023

O LINQ (consulta integrada à linguagem) é o nome de um conjunto de tecnologias com base na integração de recursos de consulta diretamente na linguagem C#.

Tradicionalmente, consultas feitas em dados são expressas como cadeias de caracteres simples sem verificação de tipo no tempo de compilação ou suporte a IntelliSense. Além disso, você precisará aprender uma linguagem de consulta diferente para cada tipo de fonte de dados: bancos de dados SQL, documentos XML, vários serviços Web etc. Com o LINQ, uma consulta é um constructo de linguagem de primeira classe, como classes, métodos, eventos.

Para um desenvolvedor que escreve consultas, a parte mais visível "integrada à linguagem" do LINQ é a expressão de consulta. As expressões de consulta são uma *sintaxe declarativa de consulta*. Usando a sintaxe de consulta, você pode executar operações de filtragem, ordenação e agrupamento em fontes de dados com o mínimo de código. Você pode usar os mesmos padrões de expressão de consulta básica para consultar e transformar dados em bancos de dados SQL, conjuntos de dados do ADO.NET, documentos XML e fluxos e coleções .NET.

O exemplo a seguir mostra a operação de consulta completa. A operação completa inclui a criação de uma fonte de dados, definição da expressão de consulta e execução da consulta em uma instrução `foreach`.

```
C#  
  
// Specify the data source.  
int[] scores = { 97, 92, 81, 60 };  
  
// Define the query expression.  
IQueryable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;  
  
// Execute the query.  
foreach (int i in scoreQuery)  
{  
    Console.Write(i + " ");  
}  
  
// Output: 97 92 81
```

# Visão geral da expressão de consulta

- Expressões de consulta podem ser usadas para consultar e transformar dados de qualquer fonte de dados habilitada para LINQ. Por exemplo, uma única consulta pode recuperar dados de um Banco de Dados SQL e produzir um fluxo XML como saída.
- As expressões de consulta são fáceis de entender porque elas usam muitos constructos de linguagem C# familiares.
- As variáveis em uma expressão de consulta são fortemente tipadas, embora em muitos casos você não precise fornecer o tipo explicitamente, pois o compilador pode inferir nele. Para obter mais informações, consulte [Relacionamentos de tipo em operações de consulta LINQ](#).
- Uma consulta não é executada até que você itere sobre a variável de consulta, por exemplo, em uma instrução `foreach`. Para obter mais informações, consulte [Introdução a consultas LINQ](#).
- No tempo de compilação, as expressões de consulta são convertidas em chamadas de método do operador de consulta padrão de acordo com as regras definidas na especificação do C#. Qualquer consulta que pode ser expressa usando sintaxe de consulta também pode ser expressa usando sintaxe de método. No entanto, na maioria dos casos, a sintaxe de consulta é mais legível e concisa. Para obter mais informações, consulte [Especificação da linguagem C#](#) e [Visão geral de operadores de consulta padrão](#).
- Como uma regra ao escrever consultas LINQ, recomendamos que você use a sintaxe de consulta sempre que possível e a sintaxe de método sempre que necessário. Não há semântica ou diferença de desempenho entre as duas formas. As expressões de consulta são geralmente mais legíveis do que as expressões equivalentes escritas na sintaxe de método.
- Algumas operações de consulta, como [Count](#) ou [Max](#), não apresentam cláusulas de expressão de consulta equivalentes e, portanto, devem ser expressas como chamadas de método. A sintaxe de método pode ser combinada com a sintaxe de consulta de várias maneiras. Para obter mais informações, consulte [Sintaxe de consulta e sintaxe de método em LINQ](#).
- As expressões de consulta podem ser compiladas para árvores de expressão ou delegados, dependendo do tipo ao qual a consulta é aplicada. As consultas `IEnumerable<T>` são compiladas para representantes. As consultas `IQueryable` e

`IQueryable<T>` são compiladas para árvores de expressão. Para obter mais informações, consulte [Árvores de expressão](#).

## Próximas etapas

Para obter mais detalhes sobre o LINQ, comece se familiarizando com alguns conceitos básicos em [Noções básicas sobre expressões de consulta](#), e, em seguida, leia a documentação para a tecnologia LINQ na qual você está interessado:

- Documentos XML: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to Entities](#)
- Coleções do .NET, arquivos, cadeias de caracteres, etc.: [LINQ to Objects](#)

Para saber mais sobre o LINQ, consulte [LINQ em C#](#).

Para começar a trabalhar com o LINQ em C#, consulte o tutorial [Trabalhando com LINQ](#).

# LINQ (Consulta Integrada à Linguagem)

Artigo • 15/02/2023

O LINQ (consulta integrada à linguagem) é o nome de um conjunto de tecnologias com base na integração de recursos de consulta diretamente na linguagem C#.

Tradicionalmente, consultas feitas em dados são expressas como cadeias de caracteres simples sem verificação de tipo no tempo de compilação ou suporte a IntelliSense. Além disso, você precisará aprender uma linguagem de consulta diferente para cada tipo de fonte de dados: bancos de dados SQL, documentos XML, vários serviços Web etc. Com o LINQ, uma consulta é um constructo de linguagem de primeira classe, como classes, métodos, eventos.

Para um desenvolvedor que escreve consultas, a parte mais visível "integrada à linguagem" do LINQ é a expressão de consulta. As expressões de consulta são uma *sintaxe declarativa de consulta*. Usando a sintaxe de consulta, você pode executar operações de filtragem, ordenação e agrupamento em fontes de dados com o mínimo de código. Você pode usar os mesmos padrões de expressão de consulta básica para consultar e transformar dados em bancos de dados SQL, conjuntos de dados do ADO.NET, documentos XML e fluxos e coleções .NET.

O exemplo a seguir mostra a operação de consulta completa. A operação completa inclui a criação de uma fonte de dados, definição da expressão de consulta e execução da consulta em uma instrução `foreach`.

```
C#  
  
// Specify the data source.  
int[] scores = { 97, 92, 81, 60 };  
  
// Define the query expression.  
IQueryable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;  
  
// Execute the query.  
foreach (int i in scoreQuery)  
{  
    Console.Write(i + " ");  
}  
  
// Output: 97 92 81
```

# Visão geral da expressão de consulta

- Expressões de consulta podem ser usadas para consultar e transformar dados de qualquer fonte de dados habilitada para LINQ. Por exemplo, uma única consulta pode recuperar dados de um Banco de Dados SQL e produzir um fluxo XML como saída.
- As expressões de consulta são fáceis de entender porque elas usam muitos constructos de linguagem C# familiares.
- As variáveis em uma expressão de consulta são fortemente tipadas, embora em muitos casos você não precise fornecer o tipo explicitamente, pois o compilador pode inferir nele. Para obter mais informações, consulte [Relacionamentos de tipo em operações de consulta LINQ](#).
- Uma consulta não é executada até que você itere sobre a variável de consulta, por exemplo, em uma instrução `foreach`. Para obter mais informações, consulte [Introdução a consultas LINQ](#).
- No tempo de compilação, as expressões de consulta são convertidas em chamadas de método do operador de consulta padrão de acordo com as regras definidas na especificação do C#. Qualquer consulta que pode ser expressa usando sintaxe de consulta também pode ser expressa usando sintaxe de método. No entanto, na maioria dos casos, a sintaxe de consulta é mais legível e concisa. Para obter mais informações, consulte [Especificação da linguagem C#](#) e [Visão geral de operadores de consulta padrão](#).
- Como uma regra ao escrever consultas LINQ, recomendamos que você use a sintaxe de consulta sempre que possível e a sintaxe de método sempre que necessário. Não há semântica ou diferença de desempenho entre as duas formas. As expressões de consulta são geralmente mais legíveis do que as expressões equivalentes escritas na sintaxe de método.
- Algumas operações de consulta, como [Count](#) ou [Max](#), não apresentam cláusulas de expressão de consulta equivalentes e, portanto, devem ser expressas como chamadas de método. A sintaxe de método pode ser combinada com a sintaxe de consulta de várias maneiras. Para obter mais informações, consulte [Sintaxe de consulta e sintaxe de método em LINQ](#).
- As expressões de consulta podem ser compiladas para árvores de expressão ou delegados, dependendo do tipo ao qual a consulta é aplicada. As consultas `IEnumerable<T>` são compiladas para representantes. As consultas `IQueryable` e

`IQueryable<T>` são compiladas para árvores de expressão. Para obter mais informações, consulte [Árvores de expressão](#).

## Próximas etapas

Para obter mais detalhes sobre o LINQ, comece se familiarizando com alguns conceitos básicos em [Noções básicas sobre expressões de consulta](#), e, em seguida, leia a documentação para a tecnologia LINQ na qual você está interessado:

- Documentos XML: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to Entities](#)
- Coleções do .NET, arquivos, cadeias de caracteres, etc.: [LINQ to Objects](#)

Para saber mais sobre o LINQ, consulte [LINQ em C#](#).

Para começar a trabalhar com o LINQ em C#, consulte o tutorial [Trabalhando com LINQ](#).

# LINQ (Consulta Integrada à Linguagem)

Artigo • 15/02/2023

O LINQ (consulta integrada à linguagem) é o nome de um conjunto de tecnologias com base na integração de recursos de consulta diretamente na linguagem C#.

Tradicionalmente, consultas feitas em dados são expressas como cadeias de caracteres simples sem verificação de tipo no tempo de compilação ou suporte a IntelliSense. Além disso, você precisará aprender uma linguagem de consulta diferente para cada tipo de fonte de dados: bancos de dados SQL, documentos XML, vários serviços Web etc. Com o LINQ, uma consulta é um constructo de linguagem de primeira classe, como classes, métodos, eventos.

Para um desenvolvedor que escreve consultas, a parte mais visível "integrada à linguagem" do LINQ é a expressão de consulta. As expressões de consulta são uma *sintaxe declarativa de consulta*. Usando a sintaxe de consulta, você pode executar operações de filtragem, ordenação e agrupamento em fontes de dados com o mínimo de código. Você pode usar os mesmos padrões de expressão de consulta básica para consultar e transformar dados em bancos de dados SQL, conjuntos de dados do ADO.NET, documentos XML e fluxos e coleções .NET.

O exemplo a seguir mostra a operação de consulta completa. A operação completa inclui a criação de uma fonte de dados, definição da expressão de consulta e execução da consulta em uma instrução `foreach`.

```
C#  
  
// Specify the data source.  
int[] scores = { 97, 92, 81, 60 };  
  
// Define the query expression.  
IQueryable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;  
  
// Execute the query.  
foreach (int i in scoreQuery)  
{  
    Console.Write(i + " ");  
}  
  
// Output: 97 92 81
```

# Visão geral da expressão de consulta

- Expressões de consulta podem ser usadas para consultar e transformar dados de qualquer fonte de dados habilitada para LINQ. Por exemplo, uma única consulta pode recuperar dados de um Banco de Dados SQL e produzir um fluxo XML como saída.
- As expressões de consulta são fáceis de entender porque elas usam muitos constructos de linguagem C# familiares.
- As variáveis em uma expressão de consulta são fortemente tipadas, embora em muitos casos você não precise fornecer o tipo explicitamente, pois o compilador pode inferir nele. Para obter mais informações, consulte [Relacionamentos de tipo em operações de consulta LINQ](#).
- Uma consulta não é executada até que você itere sobre a variável de consulta, por exemplo, em uma instrução `foreach`. Para obter mais informações, consulte [Introdução a consultas LINQ](#).
- No tempo de compilação, as expressões de consulta são convertidas em chamadas de método do operador de consulta padrão de acordo com as regras definidas na especificação do C#. Qualquer consulta que pode ser expressa usando sintaxe de consulta também pode ser expressa usando sintaxe de método. No entanto, na maioria dos casos, a sintaxe de consulta é mais legível e concisa. Para obter mais informações, consulte [Especificação da linguagem C#](#) e [Visão geral de operadores de consulta padrão](#).
- Como uma regra ao escrever consultas LINQ, recomendamos que você use a sintaxe de consulta sempre que possível e a sintaxe de método sempre que necessário. Não há semântica ou diferença de desempenho entre as duas formas. As expressões de consulta são geralmente mais legíveis do que as expressões equivalentes escritas na sintaxe de método.
- Algumas operações de consulta, como [Count](#) ou [Max](#), não apresentam cláusulas de expressão de consulta equivalentes e, portanto, devem ser expressas como chamadas de método. A sintaxe de método pode ser combinada com a sintaxe de consulta de várias maneiras. Para obter mais informações, consulte [Sintaxe de consulta e sintaxe de método em LINQ](#).
- As expressões de consulta podem ser compiladas para árvores de expressão ou delegados, dependendo do tipo ao qual a consulta é aplicada. As consultas `IEnumerable<T>` são compiladas para representantes. As consultas `IQueryable` e

`IQueryable<T>` são compiladas para árvores de expressão. Para obter mais informações, consulte [Árvores de expressão](#).

## Próximas etapas

Para obter mais detalhes sobre o LINQ, comece se familiarizando com alguns conceitos básicos em [Noções básicas sobre expressões de consulta](#), e, em seguida, leia a documentação para a tecnologia LINQ na qual você está interessado:

- Documentos XML: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to Entities](#)
- Coleções do .NET, arquivos, cadeias de caracteres, etc.: [LINQ to Objects](#)

Para saber mais sobre o LINQ, consulte [LINQ em C#](#).

Para começar a trabalhar com o LINQ em C#, consulte o tutorial [Trabalhando com LINQ](#).

# Attributes

Article • 03/15/2023

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*.

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required.
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection.

Reflection provides objects (of type [Type](#)) that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you're using attributes in your code, reflection enables you to access them. For more information, see [Attributes](#).

Here's a simple example of reflection using the [GetType\(\)](#) method - inherited by all types from the `Object` base class - to obtain the type of a variable:

## ① Note

Make sure you add `using System;` and `using System.Reflection;` at the top of your .cs file.

C#

```
// Using GetType to obtain type information:  
int i = 42;  
Type type = i.GetType();  
Console.WriteLine(type);
```

The output is: `System.Int32`.

The following example uses reflection to obtain the full name of the loaded assembly.

C#

```
// Using Reflection to get information of an Assembly:  
Assembly info = typeof(int).Assembly;  
Console.WriteLine(info);
```

The output is something like: `System.Private.CoreLib, Version=7.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`.

### ⓘ Note

The C# keywords `protected` and `internal` have no meaning in Intermediate Language (IL) and are not used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. To identify an `internal` method using reflection, use the `IsAssembly` property. To identify a `protected internal` method, use the `IsFamilyOrAssembly`.

## Using attributes

Attributes can be placed on almost any declaration, though a specific attribute might restrict the types of declarations on which it's valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets (`[]`) above the declaration of the entity to which it applies.

In this example, the `SerializableAttribute` attribute is used to apply a specific characteristic to a class:

C#

```
[Serializable]  
public class SampleClass  
{  
    // Objects of this type can be serialized.  
}
```

A method with the attribute `DllImportAttribute` is declared like the following example:

C#

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

More than one attribute can be placed on a declaration as the following example shows:

C#

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is [ConditionalAttribute](#):

C#

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

### ① Note

By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Class Library.

## Attribute parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and can't be omitted. Named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

C#

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Positional parameters correspond to the parameters of the attribute constructor. Named or optional parameters correspond to either properties or fields of the attribute. Refer to the individual attribute's documentation for information on default parameter values.

For more information on allowed parameter types, see the [Attributes](#) section of the [C# language specification](#)

## Attribute targets

The *target* of an attribute is the entity that the attribute applies to. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that follows it. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
C#  
[target : attribute-list]
```

The list of possible `target` values is shown in the following table.

Target value	Applies to
<code>assembly</code>	Entire assembly
<code>module</code>	Current assembly module
<code>field</code>	Field in a class or a struct
<code>event</code>	Event
<code>method</code>	Method or <code>get</code> and <code>set</code> property accessors
<code>param</code>	Method parameters or <code>set</code> property accessor parameters
<code>property</code>	Property
<code>return</code>	Return value of a method, property indexer, or <code>get</code> property accessor
<code>type</code>	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an [auto-implemented property](#).

The following example shows how to apply attributes to assemblies and modules. For more information, see [Common Attributes \(C#\)](#).

C#

```
using System;
using System.Reflection;
[assembly: AssemblyTitle("Production assembly 4")]
[module: CLSCompliant(true)]
```

The following example shows how to apply attributes to methods, method parameters, and method return values in C#.

C#

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

### ⓘ Note

Regardless of the targets on which `ValidatedContract` is defined to be valid, the `return` target has to be specified, even if `ValidatedContract` were defined to apply only to return values. In other words, the compiler will not use `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see [AttributeUsage](#).

## Common uses for attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see [WebMethodAttribute](#).
- Describing how to marshal method parameters when interoperating with native code. For more information, see [MarshalAsAttribute](#).
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the `DllImportAttribute` class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

## Reflection overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see [Retrieving Information Stored in Attributes](#).
- For examining and instantiating types in an assembly.
- For building new types at run time. Use classes in [System.Reflection.Emit](#).
- For performing late binding, accessing methods on types created at run time. See the article [Dynamically Loading and Using Types](#).

## Related sections

For more information:

- [Common Attributes \(C#\)](#)
- [Caller Information \(C#\)](#)
- [Attributes](#)
- [Reflection](#)
- [Viewing Type Information](#)
- [Reflection and Generic Types](#)
- [System.Reflection.Emit](#)
- [Retrieving Information Stored in Attributes](#)

# Serialização no .NET

Artigo • 15/02/2023

A serialização é o processo de conversão do estado de um objeto em um formulário que possa ser persistido ou transportado. O complemento de serialização é desserialização, que converte um fluxo em um objeto. Juntos, esses processos permitem que os dados sejam armazenados e transferidos com facilidade.

O .NET apresenta as seguintes tecnologias de serialização:

- A [serialização binária](#) preserva a fidelidade do tipo, o que é útil para preservar o estado de um objeto entre diferentes chamadas de um aplicativo. Por exemplo, você pode compartilhar um objeto entre diferentes aplicativos serializando-o para a área de transferência. Você pode serializar um objeto para um fluxo, um disco, a memória, pela rede, e assim por diante. O acesso remoto usa a serialização para passar objetos “por valor” de um computador ou domínio de aplicativo para outro.
- A [serialização XML e SOAP](#) só serializa as propriedades públicas e os campos e não preserva a fidelidade de tipo. Isso é útil quando você deseja fornecer ou consumir dados sem restringir o aplicativo que usa os dados. Como o XML é um padrão aberto, é uma opção atrativa para compartilhar dados pela Web. SOAP é, da mesma forma, um padrão aberto, uma opção atrativa.
- A [serialização JSON](#) serializa somente as propriedades públicas e não preserva a fidelidade de tipo. O JSON é um padrão aberto que é uma opção interessante para compartilhar dados na Web.

## Referência

### [System.Runtime.Serialization](#)

Contém classes que podem ser usadas para serialização e desserialização de objetos.

### [System.Xml.Serialization](#)

Contém classes que podem ser usadas para serializar objetos em documentos ou fluxos de formato XML.

### [System.Text.Json](#)

Contém classes que podem ser usadas para serializar objetos em documentos ou fluxos de formato JSON.

# Como serializar um objeto

Artigo • 07/04/2023

Para serializar um objeto, primeiro crie o objeto a ser serializado e defina seus campos e propriedades públicos. Para fazer isso, você deve determinar o formato de transporte em que o fluxo XML deve ser armazenado: como um fluxo ou como um arquivo. Por exemplo, se o fluxo XML precisar ser salvo de uma forma permanente, crie um objeto [FileStream](#).

## ⓘ Observação

Para obter mais exemplos de serialização XML, consulte [Exemplos de Serialização XML](#).

## Para serializar um objeto

1. Crie o objeto e defina seus campos e propriedades públicos.
2. Construa um [XmlSerializer](#) usando o tipo do objeto. Para obter mais informações, consulte os construtores da classe [XmlSerializer](#).
3. Chame o método [Serialize](#) para gerar um fluxo XML ou uma representação em arquivo de propriedades e campos públicos do objeto. O exemplo a seguir cria um arquivo.

C#

```
MySerializableClass myObject = new MySerializableClass();
// Insert code to set properties and fields of the object.
XmlSerializer mySerializer = new
XmlSerializer(typeof(MySerializableClass));
// To write to a file, create a StreamWriter object.
StreamWriter myWriter = new StreamWriter("myFileName.xml");
mySerializer.Serialize(myWriter, myObject);
myWriter.Close();
```

## Confira também

- [Apresentando a serialização XML](#)
- [Como desserializar um objeto](#)

# Como serializar e desserializar (realizar marshaling e cancelar a realização de marshalling) JSON no .NET

Artigo • 01/06/2023

Este artigo mostra como usar o namespace [System.Text.Json](#) para serializar e desserializar da JSON (JavaScript Object Notation). Se você estiver portando o código existente de [Newtonsoft.Json](#), confira [Como migrar para System.Text.Json](#).

## Exemplos de código

Os exemplos de código neste artigo:

- Use a biblioteca diretamente, não por meio de uma estrutura como [ASP.NET Core](#).
- Use a classe [JsonSerializer](#) com tipos personalizados para serializar e desserializar.

Para informações sobre como ler e gravar dados JSON sem usar [JsonSerializer](#), confira [Como usar o JSON DOM](#), [Como usar o Utf8JsonReader](#) e [Como usar o Utf8JsonWriter](#).

- Use a opção [WriteIndented](#) para formatar o JSON para legibilidade humana quando isso for útil.

Para uso em produção, você normalmente aceitaria o valor padrão dessa configuração `false`, uma vez que adicionar espaço em branco desnecessário pode causar um impacto negativo no desempenho e no uso da largura de banda.

- Confira a seguinte classe e variantes dela:

C#

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

## Namespaces

O namespace [System.Text.Json](#) contém todos os pontos de entrada e os tipos principais. O namespace [System.Text.Json.Serialization](#) contém atributos e APIs para cenários avançados e personalização específicos para serialização e desserialização. Os exemplos de código mostrados neste artigo exigem diretivas `using` para um ou ambos os namespaces:

```
C#
```

```
using System.Text.Json;
using System.Text.Json.Serialization;
```

### ⓘ Importante

- Atributos do namespace [System.Runtime.Serialization](#) não têm suporte de [System.Text.Json](#).
- [System.SerializableAttribute](#) e a interface [ISerializable](#) não têm suporte de [System.Text.Json](#). Esses tipos são usados apenas para [Serialização binária e XML](#).

## Como escrever objetos .NET como JSON (serializar)

Para gravar JSON em uma cadeia de caracteres ou em um arquivo, chame o método [JsonSerializer.Serialize](#).

O seguinte exemplo cria JSON como uma cadeia de caracteres:

```
C#
```

```
using System.Text.Json;

namespace SerializeBasic
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
```

```

    {
        var weatherForecast = new WeatherForecast
        {
            Date = DateTime.Parse("2019-08-01"),
            TemperatureCelsius = 25,
            Summary = "Hot"
        };

        string jsonString = JsonSerializer.Serialize(weatherForecast);

        Console.WriteLine(jsonString);
    }
}

// output:
//{"Date":"2019-08-01T00:00:00-
07:00","TemperatureCelsius":25,"Summary":"Hot"}

```

A saída JSON é minimizada (caracteres de espaço em branco, recuo e nova linha são removidos) por padrão.

O seguinte exemplo usa código síncrono para criar um arquivo JSON:

```

C#

using System.Text.Json;

namespace SerializeToFile
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string fileName = "WeatherForecast.json";
            string jsonString = JsonSerializer.Serialize(weatherForecast);
            File.WriteAllText(fileName, jsonString);

            Console.WriteLine(File.ReadAllText(fileName));
        }
    }
}

```

```
        }
    }
// output:
//>{"Date":"2019-08-01T00:00:00-
07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

O seguinte exemplo usa código assíncrono para criar um arquivo JSON:

C#

```
using System.Text.Json;

namespace SerializeToFileAsync
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static async Task Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTimeOffset.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string fileName = "WeatherForecast.json";
            using FileStream createStream = File.Create(fileName);
            await JsonSerializer.SerializeAsync(createStream,
weatherForecast);
            await createStream.DisposeAsync();

            Console.WriteLine(File.ReadAllText(fileName));
        }
    }
}
// output:
//>{"Date":"2019-08-01T00:00:00-
07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

Os exemplos anteriores usam inferência de tipos para o tipo que está sendo serializado. Uma sobrecarga de `Serialize()` usa um parâmetro de tipo genérico:

C#

```

using System.Text.Json;

namespace SerializeWithGenericParameter
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string jsonString = JsonSerializer.Serialize<WeatherForecast>
(weatherForecast);

            Console.WriteLine(jsonString);
        }
    }
}

// output:
//{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot"}

```

## Exemplo de serialização

Veja um exemplo mostrando como uma classe que contém propriedades de coleção e um tipo definido pelo usuário é serializada:

C#

```

using System.Text.Json;

namespace SerializeExtra
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
        public string? SummaryField;
        public IList<DateTimeOffset>? DatesAvailable { get; set; }
    }
}

```

```

        public Dictionary<string, HighLowTemps>? TemperatureRanges { get;
set; }
        public string[]? SummaryWords { get; set; }

    }

    public class HighLowTemps
    {
        public int High { get; set; }
        public int Low { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot",
                SummaryField = "Hot",
                DatesAvailable = new List<DateTimeOffset>()
                    { DateTime.Parse("2019-08-01"), DateTime.Parse("2019-08-
02") },
                TemperatureRanges = new Dictionary<string, HighLowTemps>
                {
                    ["Cold"] = new HighLowTemps { High = 20, Low = -10
},
                    ["Hot"] = new HighLowTemps { High = 60 , Low = 20 }
                },
                SummaryWords = new[] { "Cool", "Windy", "Humid" }
            };

            var options = new JsonSerializerOptions { WriteIndented = true
};
            string jsonString = JsonSerializer.Serialize(weatherForecast,
options);

            Console.WriteLine(jsonString);
        }
    }
}

// output:
//{
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 25,
//  "Summary": "Hot",
//  "DatesAvailable": [
//    "2019-08-01T00:00:00-07:00",
//    "2019-08-02T00:00:00-07:00"
//  ],
//  "TemperatureRanges": {
//    "Cold": {
//      "High": 20,
//      "Low": -10
//    }
//  }
//}
```

```
//    },
//    "Hot": {
//        "High": 60,
//        "Low": 20
//    }
// },
// "SummaryWords": [
//     "Cool",
//     "Windy",
//     "Humid"
// ]
//}
```

## Serialize para UTF-8

Serialize para uma matriz de bytes UTF-8 é cerca de 5 a 10% mais rápido do que usar os métodos baseados em cadeia de caracteres. A diferença ocorre porque os bytes (como UTF-8) não precisam ser convertidos em cadeias de caracteres (UTF-16).

Para serialize para uma matriz de bytes UTF-8, chame o método [JsonSerializer.SerializeToUtf8Bytes](#):

C#

```
byte[] jsonUtf8Bytes = JsonSerializer.SerializeToUtf8Bytes(weatherForecast);
```

Uma sobrecarga [Serialize](#) que usa um [Utf8JsonWriter](#) também está disponível.

## Comportamento de serialização

- Por padrão, todas as propriedades públicas são serializadas. Você pode [especificar propriedades a serem ignoradas](#).
- O [codificador padrão](#) escapa caracteres não ASCII, caracteres sensíveis a HTML dentro do intervalo ASCII e caracteres que devem ser escapados de acordo com a [especificação JSON RFC 8259](#).
- Por padrão, o JSON é minimizado. Você pode [definir os estilos de formatação do JSON](#).
- Por padrão, o uso de maiúsculas e minúsculas de nomes JSON corresponde aos nomes .NET. Você pode [personalizar o uso de maiúsculas e minúsculas de nome JSON](#).
- Por padrão, são detectadas referências circulares e exceções geradas. Você pode [preservar referências e manipular referências circulares](#).
- Por padrão, os [campos](#) são ignorados. Você pode [incluir campos](#).

Quando você usa `System.Text.Json` indiretamente em um aplicativo ASP.NET Core, alguns comportamentos padrão são diferentes. Para obter mais informações, confira [Padrões da Web para `JsonSerializerOptions`](#).

Os tipos compatíveis incluem:

- Primitivos do .NET que são mapeados para primitivos JavaScript, como tipos numéricos, cadeias de caracteres e boolianos.
- [POCOs \(objetos CLR básicos\)](#) definidos pelo usuário.
- Matrizes unidimensionais e irregulares (`T[][]`).
- Coleções e dicionários dos namespaces a seguir.
  - [System.Collections](#)
  - [System.Collections.Generic](#)
  - [System.Collections.Immutable](#)
  - [System.Collections.Concurrent](#)
  - [System.Collections.Specialized](#)
  - [System.Collections.ObjectModel](#)

Para mais informações, confira [Tipos de coleção com suporte em `System.Text.Json`](#).

Você pode [implementar conversores personalizados](#) para lidar com tipos adicionais ou fornecer funcionalidades que não são compatíveis com os conversores internos.

## Como ler JSON como objetos .NET (desserializar)

Um modo comum de desserializar o JSON é primeiro criar uma classe com propriedades e campos que representam uma ou mais das propriedades JSON. Em seguida, para desserializar de uma cadeia de caracteres ou um arquivo, chame o método `JsonSerializer.Deserialize`. Para as sobrecargas genéricas, você passa o tipo da classe que criou como o parâmetro de tipo genérico. Para as sobrecargas não genéricas, você passa o tipo da classe que criou como um parâmetro de método. Você pode desserializar de maneira síncrona ou assíncrona.

Todas as propriedades JSON que não são representadas na sua classe são ignoradas [por padrão](#). Além disso, se alguma propriedade no tipo for [necessária](#), mas não estiver presente no conteúdo JSON, a desserialização falhará.

O seguinte exemplo mostra como desserializar uma cadeia de caracteres JSON:

```
C#
```

```
using System.Text.Json;

namespace DeserializeExtra
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
        public string? SummaryField;
        public IList<DateTimeOffset>? DatesAvailable { get; set; }
        public Dictionary<string, HighLowTemps>? TemperatureRanges { get;
set; }
        public string[]? SummaryWords { get; set; }
    }

    public class HighLowTemps
    {
        public int High { get; set; }
        public int Low { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            string jsonString =
@"
{
    ""Date"": ""2019-08-01T00:00:00-07:00"",
    ""TemperatureCelsius"": 25,
    ""Summary"": ""Hot"",
    ""DatesAvailable"": [
        ""2019-08-01T00:00:00-07:00"",
        ""2019-08-02T00:00:00-07:00""
    ],
    ""TemperatureRanges"": {
        ""Cold"": {
            ""High"": 20,
            ""Low"": -10
        },
        ""Hot"": {
            ""High"": 60,
            ""Low"": 20
        }
    },
    ""SummaryWords"": [
        ""Cool"",
        ""Windy"",
        ""Humid""
    ]
}
";
```

```

        WeatherForecast? weatherForecast =
            JsonSerializer.Deserialize<WeatherForecast>(jsonString);

            Console.WriteLine($"Date: {weatherForecast?.Date}");
            Console.WriteLine($"TemperatureCelsius:
{weatherForecast?.TemperatureCelsius}");
            Console.WriteLine($"Summary: {weatherForecast?.Summary}");
        }
    }
}

// output:
//Date: 8/1/2019 12:00:00 AM -07:00
//TemperatureCelsius: 25
//Summary: Hot

```

Para desserializar de um arquivo usando código síncrono, leia o arquivo em uma cadeia de caracteres, conforme mostrado no seguinte exemplo:

C#

```

using System.Text.Json;

namespace DeserializeFromFile
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            string fileName = "WeatherForecast.json";
            string jsonString = File.ReadAllText(fileName);
            WeatherForecast weatherForecast =
JsonSerializer.Deserialize<WeatherForecast>(jsonString)!

            Console.WriteLine($"Date: {weatherForecast.Date}");
            Console.WriteLine($"TemperatureCelsius:
{weatherForecast.TemperatureCelsius}");
            Console.WriteLine($"Summary: {weatherForecast.Summary}");
        }
    }
}

// output:
//Date: 8/1/2019 12:00:00 AM -07:00
//TemperatureCelsius: 25
//Summary: Hot

```

Para desserializar de um arquivo usando código assíncrono, chame o método [DeserializeAsync](#):

C#

```
using System.Text.Json;

namespace DeserializeFromFileAsync
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static async Task Main()
        {
            string fileName = "WeatherForecast.json";
            using FileStream openStream = File.OpenRead(fileName);
            WeatherForecast? weatherForecast =
                await JsonSerializer.DeserializeAsync<WeatherForecast>
(openStream);
            Console.WriteLine($"Date: {weatherForecast?.Date}");
            Console.WriteLine($"TemperatureCelsius:
{weatherForecast?.TemperatureCelsius}");
            Console.WriteLine($"Summary: {weatherForecast?.Summary}");
        }
    }
}

// output:
//Date: 8/1/2019 12:00:00 AM -07:00
//TemperatureCelsius: 25
//Summary: Hot
```

### Dica

Se você tiver o JSON no qual deseja desserializar e não tiver a classe para desserializá-lo, terá opções diferentes de criar manualmente a classe de que precisa:

- Dessaialize em um **DOM JSON (modelo de objeto do documento)** e extraia o que você precisa do DOM.

O DOM permite que você navegue até uma subseção de um conteúdo JSON e desserialize um só valor, um tipo personalizado ou uma matriz. Para

informações sobre o **JsonNode** DOM, confira [Desserializar subseções de um conteúdo JSON](#). Para informações sobre o DOM **JsonDocument**, confira [Como pesquisar um JsonDocument e JsonElement em busca de subconjuntos](#).

- Use o **Utf8JsonReader** diretamente.
- Use o Visual Studio 2022 para gerar automaticamente a classe necessária:
  - Copie o JSON de que você precisa para desserializar.
  - Crie um arquivo de classe e exclua o código do modelo.
  - Escolha **Editar>Colar especial>Colar JSON como Classes**. O resultado é uma classe que você pode usar para seu destino de desserialização.

## Desserializar de UTF-8

Para desserializar de UTF-8, chame uma sobrecarga **JsonSerializer.Deserialize** que usa um `ReadOnlySpan<byte>` ou um `Utf8JsonReader`, conforme mostrado nos exemplos a seguir. Os exemplos pressupõem que o JSON está em uma matriz de bytes chamada `jsonUtf8Bytes`.

C#

```
var readOnlySpan = new ReadOnlySpan<byte>(jsonUtf8Bytes);
WeatherForecast deserializedWeatherForecast =
    JsonSerializer.Deserialize<WeatherForecast>(readOnlySpan)!
```

C#

```
var utf8Reader = new Utf8JsonReader(jsonUtf8Bytes);
WeatherForecast deserializedWeatherForecast =
    JsonSerializer.Deserialize<WeatherForecast>(ref utf8Reader)!
```

## Comportamento de desserialização

Os seguintes comportamentos se aplicam ao desserializar o JSON:

- Por padrão, a correspondência de nome de propriedade diferencia maiúsculas de minúsculas. Você pode [especificar não diferenciar maiúsculas de minúsculas](#).
- Se o JSON contiver um valor para uma propriedade somente leitura, o valor será ignorado por padrão. Você pode definir a opção [PreferredObjectCreationHandling](#)

como `JsonObjectCreationHandling.Populate` a fim de habilitar a desserialização para propriedades somente leitura.

- Construtores não públicos são ignorados pelo serializador.
- Há suporte para desserialização para objetos imutáveis ou propriedades que não têm acessadores públicos `set`. Confira [Tipos imutáveis e registros](#).
- Por padrão, há suporte para enumerações como números. Você pode [serializar nomes de enumeração como cadeias de caracteres](#).
- Por padrão, os campos são ignorados. Você pode [incluir campos](#).
- Por padrão, comentários ou vírgulas à direita no JSON geram exceções. Você pode [permitir comentários e vírgulas à direita](#).
- A [profundidade máxima padrão](#) é 64.

Quando você usa `System.Text.Json` indiretamente em um aplicativo ASP.NET Core, alguns comportamentos padrão são diferentes. Para obter mais informações, confira [Padrões da Web para JsonSerializerOptions](#).

Você pode [implementar conversores personalizados](#) para fornecer funcionalidades que não são compatíveis com os conversores internos.

## Serializar para JSON formatado

Para estruturar a saída JSON, defina `JsonSerializerOptions.WriteLineIndented` como `true`:

C#

```
using System.Text.Json;

namespace SerializeWriteIndented
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };
        }
    }
}
```

```
        var options = new JsonSerializerOptions { WriteIndented = true
    };
    string jsonString = JsonSerializer.Serialize(weatherForecast,
options);

    Console.WriteLine(jsonString);
}
}

// output:
//{
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 25,
//  "Summary": "Hot"
//}
```

Se você usar `JsonSerializerOptions` repetidamente com as mesmas opções, não crie uma instância `JsonSerializerOptions` sempre que a usar. Reutilize a mesma instância para cada chamada. Para mais informações, confira [Reutilizar instâncias de JsonSerializerOptions](#).

## Incluir campos

Use a configuração global `JsonSerializerOptions.IncludeFields` ou o atributo `[JsonInclude]` para incluir campos ao serializar ou desserializar, conforme mostrado no seguinte exemplo:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace Fields
{
    public class Forecast
    {
        public DateTime Date;
        public int TemperatureC;
        public string? Summary;
    }

    public class Forecast2
    {
        [JsonInclude]
        public DateTime Date;
        [JsonInclude]
        public int TemperatureC;
        [JsonInclude]
        public string? Summary;
    }
}
```

```
}

public class Program
{
    public static void Main()
    {
        var json =
            @"{""Date"":""2020-09-
06T11:31:01.923395"",""TemperatureC"":-1,""Summary"":""Cold""} ";
        Console.WriteLine($"Input JSON: {json}");

        var options = new JsonSerializerOptions
        {
            IncludeFields = true,
        };
        var forecast = JsonSerializer.Deserialize<Forecast>(json,
options)!;

        Console.WriteLine($"forecast.Date: {forecast.Date}");
        Console.WriteLine($"forecast.TemperatureC:
{forecast.TemperatureC}");
        Console.WriteLine($"forecast.Summary: {forecast.Summary}");

        var roundTrippedJson =
            JsonSerializer.Serialize<Forecast>(forecast, options);

        Console.WriteLine($"Output JSON: {roundTrippedJson}");

        var forecast2 = JsonSerializer.Deserialize<Forecast2>(json)!

        Console.WriteLine($"forecast2.Date: {forecast2.Date}");
        Console.WriteLine($"forecast2.TemperatureC:
{forecast2.TemperatureC}");
        Console.WriteLine($"forecast2.Summary: {forecast2.Summary}");

        roundTrippedJson = JsonSerializer.Serialize<Forecast2>
(forecast2);

        Console.WriteLine($"Output JSON: {roundTrippedJson}");
    }
}

// Produces output like the following example:
//
//Input JSON: { "Date":"2020-09-
06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}
//forecast.Date: 9/6/2020 11:31:01 AM
//forecast.TemperatureC: -1
//forecast.Summary: Cold
//Output JSON: { "Date":"2020-09-
06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}
//forecast2.Date: 9/6/2020 11:31:01 AM
//forecast2.TemperatureC: -1
//forecast2.Summary: Cold
```

```
//Output JSON: { "Date":"2020-09-  
06T11:31:01.923395", "TemperatureC":-1, "Summary":"Cold"}
```

Para ignorar campos somente leitura, use a configuração global [JsonSerializerOptions.IgnoreReadOnlyFields](#).

## Métodos de extensão HttpClient e HttpContent

Serializar e desserializar cargas JSON da rede são operações comuns. Os métodos de extensão em [HttpClient](#) e [HttpContent](#) permitem que você faça essas operações em uma só linha de código. Esses métodos de extensão usam [padrões da Web para JsonSerializerOptions](#).

O seguinte exemplo ilustra o uso de [HttpClientJsonExtensions.GetFromJsonAsync](#) e [HttpClientJsonExtensions.PostAsJsonAsync](#):

C#

```
using System.Net.Http.Json;  
  
namespace HttpClientExtensionMethods  
{  
    public class User  
    {  
        public int Id { get; set; }  
        public string? Name { get; set; }  
        public string? Username { get; set; }  
        public string? Email { get; set; }  
    }  
  
    public class Program  
    {  
        public static async Task Main()  
        {  
            using HttpClient client = new()  
            {  
                BaseAddress = new  
Uri("https://jsonplaceholder.typicode.com")  
            };  
  
            // Get the user information.  
            User? user = await client.GetFromJsonAsync<User>("users/1");  
            Console.WriteLine($"Id: {user?.Id}");  
            Console.WriteLine($"Name: {user?.Name}");  
            Console.WriteLine($"Username: {user?.Username}");  
            Console.WriteLine($"Email: {user?.Email}");  
  
            // Post a new user.  
        }  
    }  
}
```

```
        HttpResponseMessage response = await
client.PostAsJsonAsync("users", user);
Console.WriteLine(
    $"{{(response.IsSuccessStatusCode ? "Success" : "Error")}} - 
{response.StatusCode}");
}
}

// Produces output like the following example but with different names:
//Id: 1
//Name: Tyler King
//Username: Tyler
//Email: Tyler @contoso.com
//Success - Created
```

Também há métodos de extensão para System.Text.Json em [HttpContent](#).

## Confira também

- [System.Text.Json overview](#)

# Instruções (Guia de Programação em C#)

Artigo • 07/04/2023

As ações que usa um programa executa são expressas em instruções. Ações comuns incluem declarar variáveis, atribuir valores, chamar métodos, fazer loops pelas coleções e ramificar para um ou para outro bloco de código, dependendo de uma determinada condição. A ordem na qual as instruções são executadas em um programa é chamada de fluxo de controle ou fluxo de execução. O fluxo de controle pode variar sempre que um programa é executado, dependendo de como o programa reage às entradas que recebe em tempo de execução.

Uma instrução pode consistir em uma única linha de código que termina em um ponto e vírgula ou uma série de instruções de uma linha em um bloco. Um bloco de instrução é colocado entre colchetes {} e pode conter blocos aninhados. O código a seguir mostra dois exemplos de instruções de linha única, bem como um bloco de instrução de várias linhas:

```
C#  
  
static void Main()  
{  
    // Declaration statement.  
    int counter;  
  
    // Assignment statement.  
    counter = 1;  
  
    // Error! This is an expression, not an expression statement.  
    // counter + 1;  
  
    // Declaration statements with initializers are functionally  
    // equivalent to declaration statement followed by assignment  
    // statement:  
    int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an  
    // array.  
    const double pi = 3.14159; // Declare and initialize constant.  
  
    // foreach statement block that contains multiple statements.  
    foreach (int radius in radii)  
    {  
        // Declaration statement with initializer.  
        double circumference = pi * (2 * radius);  
  
        // Expression statement (method invocation). A single-line  
        // statement can span multiple text lines because line breaks  
        // are treated as white space, which is ignored by the compiler.  
    }  
}
```

```

        System.Console.WriteLine("Radius of circle #{0} is {1}.
Circumference = {2:N2}",
                                counter, radius, circumference);

        // Expression statement (postfix increment).
        counter++;
    } // End of foreach statement block
} // End of Main method body.
} // End of SimpleStatements class.
/*
Output:
Radius of circle #1 = 15. Circumference = 94.25
Radius of circle #2 = 32. Circumference = 201.06
Radius of circle #3 = 108. Circumference = 678.58
Radius of circle #4 = 74. Circumference = 464.96
Radius of circle #5 = 9. Circumference = 56.55
*/

```

## Tipos de instruções

A tabela a seguir lista os diferentes tipos de instruções em C# e as palavras-chave associadas a elas, com links para tópicos que contêm mais informações:

Categoría	Palavras-chave do C#/observações
Instruções de declaração	Uma declaração de instrução introduz uma nova variável ou constante. Uma declaração variável pode, opcionalmente, atribuir um valor à variável. Uma declaração constante, a atribuição é obrigatória.
Instruções de expressão	Instruções de expressão que calculam um valor devem armazenar o valor em uma variável.
Instruções de seleção	Instruções de seleção permitem que você ramifique para diferentes seções de código, dependendo de uma ou mais condições especificadas. Para obter mais informações, consulte estes tópicos: <ul style="list-style-type: none"> <li>• <a href="#">if</a></li> <li>• <a href="#">switch</a></li> </ul>
Instruções de iteração	Instruções de iteração permitem que você percorra coleções como matrizes ou execute o mesmo conjunto de instruções repetidamente até que uma determinada condição seja atendida. Para obter mais informações, consulte estes tópicos: <ul style="list-style-type: none"> <li>• <a href="#">do</a></li> <li>• <a href="#">for</a></li> <li>• <a href="#">foreach</a></li> <li>• <a href="#">while</a></li> </ul>

Categoria	Palavras-chave do C#/observações
Instruções de atalho	<p>Instruções de hiperlink transferem o controle para outra seção de código. Para obter mais informações, consulte estes tópicos:</p> <ul style="list-style-type: none"> <li>• <a href="#">break</a></li> <li>• <a href="#">continue</a></li> <li>• <a href="#">goto</a></li> <li>• <a href="#">return</a></li> <li>• <a href="#">yield</a></li> </ul>
Instruções para tratamento de exceções	<p>Instruções para tratamento de exceções permitem que você se recupere normalmente de condições excepcionais que ocorrem em tempo de execução. Para obter mais informações, consulte estes tópicos:</p> <ul style="list-style-type: none"> <li>• <a href="#">throw</a></li> <li>• <a href="#">try-catch</a></li> <li>• <a href="#">try-finally</a></li> <li>• <a href="#">try-catch-finally</a></li> </ul>
<a href="#">checked</a> e <a href="#">unchecked</a>	<p>As instruções <code>checked</code> e <code>unchecked</code> permitem especificar se as operações numéricas de tipo integral têm permissão para causar um estouro quando o resultado é armazenado em uma variável muito pequena para manter o valor resultante.</p>
A instrução <a href="#">await</a>	<p>Se marcar um método com o modificador <a href="#">async</a>, você poderá usar o operador <a href="#">await</a> no método. Quando o controle atinge uma expressão <code>await</code> no método assíncrono, ele retorna para o chamador e o progresso no método é suspenso até a tarefa aguardada ser concluída. Quando a tarefa for concluída, a execução poderá ser retomada no método.</p> <p>Para obter um exemplo simples, consulte a seção "Métodos assíncronos" em <a href="#">Métodos</a>. Para obter mais informações, consulte <a href="#">Programação assíncrona com async e await</a>.</p>
A instrução <a href="#">yield return</a>	<p>Um iterador realiza uma iteração personalizada em uma coleção, como uma lista ou uma matriz. Um iterador usa a instrução <a href="#">yield return</a> para retornar um elemento de cada vez. Quando uma instrução <code>yield return</code> for atingida, o local atual no código será lembrado. A execução será reiniciada desse local quando o iterador for chamado na próxima vez.</p> <p>Para obter mais informações, consulte <a href="#">Iteradores</a>.</p>
A instrução <a href="#">fixed</a>	<p>A instrução <code>fixed</code> impede que o coletor de lixo faça a realocação de uma variável móvel. Para obter mais informações, consulte <a href="#">fixed</a>.</p>
A instrução <a href="#">lock</a>	<p>A instrução <code>lock</code> permite limitar o acesso a blocos de código a apenas um thread por vez. Para obter mais informações, consulte <a href="#">lock</a>.</p>

Categoria	Palavras-chave do C#/observações
Instruções rotuladas	Você pode atribuir um rótulo a uma instrução e, em seguida, usar a palavra-chave <code>goto</code> para ir diretamente para a instrução rotulada. (Veja o exemplo na linha a seguir.)
A instrução vazia	A instrução vazia consiste em um único ponto e vírgula. Ela não faz nada e pode ser usada em locais em que uma instrução é necessária, mas nenhuma ação precisa ser executada.

## Instruções de declaração

O código a seguir mostra exemplos de declarações de variável com e sem uma atribuição inicial e uma declaração de constante com a inicialização necessária.

```
C#
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

## Instruções de expressão

O código a seguir mostra exemplos de instruções de expressão, incluindo a atribuição, a criação de objeto com a atribuição e a invocação de método.

```
C#
// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();
```

# A instrução vazia

Os exemplos a seguir mostram dois usos de uma instrução vazia:

```
C#  
  
void ProcessMessages()  
{  
    while (ProcessMessage())  
        ; // Statement needed here.  
}  
  
void F()  
{  
    //...  
    if (done) goto exit;  
//...  
exit:  
    ; // Statement needed here.  
}
```

## Instruções inseridas

Algumas instruções, por exemplo, [instruções de iteração](#), sempre têm uma instrução inserida que as segue. Essa instrução inserida pode ser uma instrução única ou várias instruções colocadas entre colchetes {} em um bloco de instrução. Até mesmo instruções inseridas de uma única linha podem ser colocadas entre colchetes {}, conforme mostrado no seguinte exemplo:

```
C#  
  
// Recommended style. Embedded statement in block.  
foreach (string s in System.IO.Directory.GetDirectories(  
            System.Environment.CurrentDirectory))  
{  
    System.Console.WriteLine(s);  
}  
  
// Not recommended.  
foreach (string s in System.IO.Directory.GetDirectories(  
            System.Environment.CurrentDirectory))  
    System.Console.WriteLine(s);
```

Uma instrução inserida que não está entre colchetes {} não pode ser uma instrução de declaração ou uma instrução rotulada. Isso é mostrado no exemplo a seguir:

```
C#
```

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Coloque a instrução inserida em um bloco para corrigir o erro:

C#

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToString("yyyy-MM-dd"));
}
```

## Blocos de instrução aninhados

Blocos de instrução podem ser aninhados, conforme mostrado no código a seguir:

C#

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }
}
return "Not found.;"
```

## Instruções inacessíveis

Se o compilador determinar que o fluxo de controle nunca pode atingir uma determinada instrução em nenhuma circunstância, ele produzirá o aviso CS0162, conforme mostrado no exemplo a seguir:

C#

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
```

```
{  
    System.Console.WriteLine("I'll never write anything."); //CS0162  
}
```

## Especificação da linguagem C#

Para saber mais, confira a seção [Instruções](#) da [Especificação da linguagem C#](#).

## Confira também

- [Guia de Programação em C#](#)
- [Palavras-chave de instrução](#)
- [Operadores e expressões C#](#)

# Membros aptos para expressão (Guia de Programação em C#)

Artigo • 10/05/2023

As definições de corpo da expressão permitem que você forneça uma implementação de um membro em uma forma bastante concisa e legível. Você pode usar uma definição de corpo da expressão sempre que a lógica para qualquer membro com suporte, como um método ou propriedade, consiste em uma única expressão. Uma definição de corpo da expressão tem a seguinte sintaxe geral:

```
C#
```

```
member => expression;
```

em que *expression* é uma expressão válida.

As definições do corpo da expressão podem ser usadas com os seguintes membros de tipo:

- [Método](#)
- [Propriedade somente leitura](#)
- [Propriedade](#)
- [Construtor](#)
- [Finalizer](#)
- [Indexador](#)

## Métodos

Um método apto para expressão consiste em uma única expressão que retorna um valor cujo tipo corresponde ao tipo de retorno do método, ou, para métodos que retornam `void`, que executam uma operação. Por exemplo, os tipos que substituem o método `ToString` normalmente incluem uma única expressão que retorna a representação da cadeia de caracteres do objeto atual.

O exemplo a seguir define uma classe `Person` que substitui o método `ToString` por uma definição de corpo da expressão. Ele também define um método `DisplayName` que exibe um nome para o console. Observe que a palavra-chave `return` não é usada na definição de corpo da expressão `ToString`.

```
C#
```

```
using System;

public class Person
{
    public Person(string firstName, string lastName)
    {
        fname = firstName;
        lname = lastName;
    }

    private string fname;
    private string lname;

    public override string ToString() => $"{fname} {lname}".Trim();
    public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
    static void Main()
    {
        Person p = new Person("Mandy", "Dejesus");
        Console.WriteLine(p);
        p.DisplayName();
    }
}
```

Para obter mais informações, consulte [Métodos \(Guia de Programação em C#\)](#).

## Propriedades somente leitura

Você pode usar a definição de corpo da expressão para implementar uma propriedade somente leitura. Para isso, use a seguinte sintaxe:

C#

```
.PropertyType PropertyName => expression;
```

O exemplo a seguir define uma classe `Location` cuja propriedade somente leitura `Name` é implementada como uma definição de corpo da expressão que retorna o valor do campo `locationName` particular:

C#

```
public class Location
{
    private string locationName;
```

```
public Location(string name)
{
    locationName = name;
}

public string Name => locationName;
}
```

Para obter mais informações sobre as propriedades, confira [Propriedades \(Guia de Programação em C#\)](#).

## Propriedades

Você pode usar as definições de corpo da expressão para implementar a propriedade `get` e os acessadores `set`. O exemplo a seguir demonstra como fazer isso:

C#

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

Para obter mais informações sobre as propriedades, confira [Propriedades \(Guia de Programação em C#\)](#).

## Construtores

Uma definição de corpo da expressão para um construtor normalmente consiste em uma expressão de atribuição simples ou uma chamada de método que manipula os argumentos do construtor ou inicializa o estado da instância.

O exemplo a seguir define uma classe `Location` cujo construtor tem um único parâmetro de cadeia de caracteres chamado *nome*. A definição de corpo da expressão atribui o argumento à propriedade `Name`.

C#

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

Para obter mais informações, consulte [Construtores \(Guia de Programação em C#\)](#).

## Finalizadores

Uma definição de corpo da expressão para um finalizador normalmente contém instruções de limpeza, como instruções que liberam recursos não gerenciados.

O exemplo a seguir define um finalizador que usa uma definição de corpo da expressão para indicar que o finalizador foi chamado.

C#

```
public class Destroyer
{
    public override string ToString() => GetType().Name;

    ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is
executing.");
}
```

Para obter mais informações, consulte [Finalizadores \(Guia de Programação em C#\)](#).

## Indexadores

Como as propriedades, os acessadores `get` e `set` do indexador consistirão em definições de corpo da expressão se o acessador `get` consistir em uma única expressão que retorna um valor ou o acessador `set` executar uma atribuição simples.

O exemplo a seguir define uma classe chamada `Sports` que inclui uma matriz `String` interna que contém os nomes de vários esportes. Os acessadores `get` e `set` do indexador são implementados como definições de corpo da expressão.

C#

```
using System;
using System.Collections.Generic;

public class Sports
{
    private string[] types = { "Baseball", "Basketball", "Football",
                               "Hockey", "Soccer", "Tennis",
                               "Volleyball" };

    public string this[int i]
    {
        get => types[i];
        set => types[i] = value;
    }
}
```

Para obter mais informações, consulte [Indexadores \(Guia de Programação em C#\)](#).

## Confira também

- Regras de estilo de código do .NET para membros do corpo da expressão

# Comparações de igualdade (Guia de Programação em C#)

Artigo • 10/05/2023

Às vezes, é necessário comparar dois valores em relação à igualdade. Em alguns casos, testa-se a *igualdade de valor*, também conhecida como *equivalência*, o que significa que os valores contidos pelas duas variáveis são iguais. Em outros casos, é necessário determinar se duas variáveis se referem ao mesmo objeto subjacente na memória. Esse tipo de igualdade é chamado *igualdade de referência* ou *identidade*. Este tópico descreve esses dois tipos de igualdade e fornece links para outros tópicos que fornecem mais informações.

## Igualdade de referência

Igualdade de referência significa que as duas referências de objeto se referem ao mesmo objeto subjacente. Isso pode ocorrer por meio de uma atribuição simples, conforme mostrado no exemplo a seguir.

C#

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

```
    }  
}
```

Nesse código, dois objetos são criados, mas após a instrução de atribuição, ambas as referências se referem ao mesmo objeto. Portanto, eles têm igualdade de referência. Use o método [ReferenceEquals](#) para determinar se duas referências referenciam o mesmo objeto.

O conceito de igualdade de referência se aplica apenas a tipos de referência. Objetos de tipo de valor não podem ter igualdade de referência, pois quando uma instância de um tipo de valor é atribuída a uma variável, uma cópia do valor é gerada. Portanto, não é possível ter dois structs desconvertidos que referenciam o mesmo local na memória. Além disso, se [ReferenceEquals](#) for usado para comparar dois tipos de valor, o resultado sempre será `false`, mesmo se os valores contidos nos objetos forem idênticos. Isso ocorre porque cada variável é convertido em uma instância de objeto separada. Para obter mais informações, consulte [Como testar a igualdade de referência \(Identidade\)](#).

## Igualdade de valor

Igualdade de valor significa que dois objetos contêm o mesmo valor ou valores. Para tipos de valor primitivos, como `int` ou `bool`, os testes de igualdade de valor são simples. É possível usar o operador `==`, conforme mostrado no exemplo a seguir.

```
C#  
  
int a = GetOriginalValue();  
int b = GetCurrentValue();  
  
// Test for value equality.  
if (b == a)  
{  
    // The two integers are equal.  
}
```

Para a maioria dos outros tipos, o teste de igualdade de valor é mais complexo, pois é necessário entender como o tipo o define. Para classes e structs que têm vários campos ou propriedades, a igualdade de valor geralmente é definida para determinar que todos os campos ou propriedades tenham o mesmo valor. Por exemplo, dois objetos `Point` podem ser definidos para serem equivalentes se `pointA.X` for igual a `pointB.X` e `pointA.Y` for igual a `pointB.Y`. Para registros, a igualdade de valor significa que duas variáveis de um tipo de registro são iguais se os tipos corresponderem e todos os valores de propriedade e de campo corresponderem.

No entanto, não há nenhuma exigência de que a equivalência seja baseada em todos os campos em um tipo. Ela pode ser baseada em um subconjunto. Ao comparar tipos que não são de sua propriedade, certifique-se de que a forma como a equivalência é definida especificamente para esse tipo foi entendida. Para obter mais informações sobre como definir a igualdade de valor em suas próprias classes e structs, consulte [Como definir a igualdade de valor para um tipo](#).

## Igualdade de valor para valores de ponto flutuante

As comparações de igualdade de valores de ponto flutuante ([double](#) e [float](#)) são problemáticas devido à imprecisão da aritmética de ponto flutuante em computadores binários. Para obter mais informações, consulte os comentários no tópico [System.Double](#).

## Tópicos relacionados

Título	Descrição
<a href="#">Como testar a igualdade de referência (Identidade)</a>	Descreve como determinar se duas variáveis têm igualdade de referência.
<a href="#">Como definir a igualdade de valor para um tipo</a>	Descreve como fornecer uma definição personalizada de igualdade de valor a um tipo.
<a href="#">Guia de Programação em C#</a>	Fornece links para informações detalhadas sobre recursos importantes da linguagem C# e recursos disponíveis para C# por meio do .NET.
<a href="#">Types</a>	Fornece informações sobre o sistema de tipos do C# e links para mais informações.
<a href="#">Registros</a>	Fornece informações sobre tipos de registro que testam a igualdade de valor por padrão.

## Confira também

- [Guia de Programação em C#](#)

# Como definir a igualdade de valor para uma classe ou struct (Guia de programação em C#)

Artigo • 07/04/2023

Os [registros](#) implementam automaticamente a igualdade de valor. Considere definir um `record` em vez de um `class` quando seu tipo modelar dados e tiver que implementar a igualdade de valor.

Quando você define uma classe ou struct, decide se faz sentido criar uma definição personalizada de igualdade de valor (ou equivalência) para o tipo. Normalmente, você implementa igualdade de valor quando espera adicionar objetos do tipo a uma coleção, ou quando seu objetivo principal for armazenar um conjunto de campos ou propriedades. Você pode basear sua definição de igualdade de valor em uma comparação de todos os campos e propriedades no tipo ou pode basear a definição em um subconjunto.

Em ambos os casos e em classes e structs, sua implementação deve seguir as cinco garantias de equivalência (para as seguintes regras, suponha que `x`, `y` e `z` não sejam nulos):

1. Propriedade reflexiva: `x.Equals(x)` retorna `true`.
2. Propriedade simétrica: `x.Equals(y)` retorna o mesmo valor que `y.Equals(x)`.
3. Propriedade transitiva: se `(x.Equals(y) && y.Equals(z))` retorna `true`, então `x.Equals(z)` retorna `true`.
4. Invocações sucessivas de `x.Equals(y)` retornam o mesmo valor, contanto que os objetos referenciados por `x` e `y` não sejam modificados.
5. Qualquer valor não nulo não é igual a nulo. No entanto, `x.Equals(y)` gera uma exceção quando `x` é nulo. Isso quebra as regras 1 ou 2, a depender do argumento para `Equals`.

Qualquer struct que você define já tem uma implementação padrão de igualdade de valor que ele herda da substituição [System.ValueType](#) do método [Object.Equals\(Object\)](#). Essa implementação usa a reflexão para examinar todos os campos e propriedades no tipo. Embora essa implementação produza resultados corretos, ela é relativamente lenta

em comparação com uma implementação personalizada escrita especificamente para o tipo.

Os detalhes de implementação para a igualdade de valor são diferentes para classes e struct. No entanto, as classes e structs exigem as mesmas etapas básicas para implementar a igualdade:

1. Substitua o método `virtualObject.Equals(Object)`. Na maioria dos casos, sua implementação de `bool Equals( object obj )` deve apenas chamar o método `Equals` específico do tipo que é a implementação da interface `System.IEquatable<T>`. (Consulte a etapa 2.)
2. Implemente a interface `System.IEquatable<T>` fornecendo um método `Equals` específico do tipo. Isso é o local em que a comparação de equivalência de fato é realizada. Por exemplo, você pode decidir definir a igualdade comparando apenas um ou dois campos em seu tipo. Não lance exceções a partir de `Equals`. Para classes relacionadas por herança:
  - esse método deve examinar somente os campos que são declarados na classe. Ele deve chamar `base.Equals` para examinar os campos que estão na classe base. (Não chame `base.Equals` se o tipo herdar diretamente de `Object`, pois a implementação `Object` de `Object.Equals(Object)` executa uma verificação de igualdade de referência.)
  - Duas variáveis devem ser consideradas iguais somente se os tipos de tempo de execução das variáveis que estão sendo comparadas forem os mesmos. Além disso, verifique se a implementação `IEquatable` do método `Equals` para o tipo de tempo de execução será usada se os tipos de tempo de execução e tempo de compilação de uma variável forem diferentes. Uma estratégia para garantir que os tipos de tempo de execução sejam sempre comparados corretamente é implementar `IEquatable` somente em classes `sealed`. Para obter mais informações, consulte o [exemplo de classe](#) mais adiante neste artigo.
3. Opcional, mas recomendado: sobrecarregue os operadores `==` e `!=`.
4. Substitua `Object.GetHashCode` para que os dois objetos que têm a igualdade de valor produzam o mesmo código hash.
5. Opcional: para dar suporte às definições para “maior que” ou “menor que”, implemente a interface `IComparable<T>` para seu tipo e também sobrecarregue os operadores `<=` e `>=`.

## ① Observação

A partir do C# 9.0, você pode usar registros para obter semântica de igualdade de valor sem qualquer código clichê desnecessário.

## Exemplo de classe

O exemplo a seguir mostra como implementar a igualdade de valor em uma classe (tipo de referência).

C#

```
namespace ValueEqualityClass;

class TwoDPoint : IEquatable<TwoDPoint>
{
    public int X { get; private set; }
    public int Y { get; private set; }

    public TwoDPoint(int x, int y)
    {
        if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.X = x;
        this.Y = y;
    }

    public override bool Equals(object obj) => this.Equals(obj as
TwoDPoint);

    public bool Equals(TwoDPoint p)
    {
        if (p is null)
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // If run-time types are not exactly the same, return false.
        if (this.GetType() != p.GetType())
        {
            return false;
        }
    }
}
```

```

        // Return true if the fields match.
        // Note that the base class is not invoked because it is
        // System.Object, which defines Equals as reference equality.
        return (X == p.X) && (Y == p.Y);
    }

    public override int GetHashCode() => (X, Y).GetHashCode();

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs ==
rhs);
}

// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
{
    public int Z { get; private set; }

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        if ((z < 1) || (z > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj) => this.Equals(obj as
ThreeDPoint);

    public bool Equals(ThreeDPoint p)
    {
        if (p is null)
        {
            return false;
        }

        // Optimization for a common success case.
    }
}

```

```

        if (Object.ReferenceEquals(this, p))
    {
        return true;
    }

    // Check properties that this class declares.
    if (Z == p.Z)
    {
        // Let base class check its own fields
        // and do the run-time type comparison.
        return base.Equals((TwoDPoint)p);
    }
    else
    {
        return false;
    }
}

public override int GetHashCode() => (X, Y, Z).GetHashCode();

public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
{
    if (lhs is null)
    {
        if (rhs is null)
        {
            // null == null = true.
            return true;
        }

        // Only the left side is null.
        return false;
    }
    // Equals handles the case of null on right side.
    return lhs.Equals(rhs);
}

public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs) => !(lhs == rhs);
}

class Program
{
    static void Main(string[] args)
    {
        ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointC = null;
        int i = 5;

        Console.WriteLine("pointA.Equals(pointB) = {0}",
pointA.Equals(pointB));
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
        Console.WriteLine("Compare to some other type = {0}",

```

```

        pointA.Equals(i));

        TwoDPoint pointD = null;
        TwoDPoint pointE = null;

        Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD ==
pointE);

        pointE = new TwoDPoint(3, 4);
        Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
        Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
        Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

        System.Collections.ArrayList list = new
System.Collections.ArrayList();
        list.Add(new ThreeDPoint(3, 4, 5));
        Console.WriteLine("pointE.Equals(list[0]): {0}",
pointE.Equals(list[0]));

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   pointA.Equals(pointB) = True
   pointA == pointB = True
   null comparison = False
   Compare to some other type = False
   Two null TwoDPoints are equal: True
   (pointE == pointA) = False
   (pointA == pointE) = False
   (pointA != pointE) = True
   pointE.Equals(list[0]): False
*/

```

Em classes (tipo de referência), a implementação padrão de ambos os métodos `Object.Equals(Object)` executa uma comparação de igualdade de referência, não uma verificação de igualdade de valor. Quando um implementador substitui o método virtual, o objetivo é fornecer semântica de igualdade de valor.

Os operadores `==` e `!=` podem ser usados com classes, mesmo se a classe não sobrecarregá-los. No entanto, o comportamento padrão é executar uma verificação de igualdade de referência. Em uma classe, se você sobrecarregar o método `Equals`, você deverá sobrecarregar os operadores `==` e `!=`, mas isso não é necessário.

### ⓘ Importante

O código de exemplo anterior pode não lidar com todos os cenários de herança da maneira esperada. Considere o seguinte código:

C#

```
TwoDPoint p1 = new ThreeDPoint(1, 2, 3);
TwoDPoint p2 = new ThreeDPoint(1, 2, 4);
Console.WriteLine(p1.Equals(p2)); // output: True
```

Esse código relata que `p1` é igual a `p2` apesar da diferença de valores `z`. A diferença é ignorada porque o compilador escolhe a implementação `TwoDPoint` de `IEquatable` com base no tipo de tempo de compilação.

A igualdade de valores interna de tipos `record` lida com cenários como este corretamente. Se `TwoDPoint` e `ThreeDPoint` fossem tipos `record`, o resultado de `p1.Equals(p2)` seria `False`. Para obter mais informações, consulte [Igualdade nas hierarquias de herança de tipo record](#).

## Exemplo de struct

O exemplo a seguir mostra como implementar a igualdade de valor em um struct (tipo de valor):

C#

```
namespace ValueEqualityStruct
{
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 - 2000");
            }
            X = x;
            Y = y;
        }

        public override bool Equals(object? obj) => obj is TwoDPoint other
&& this.Equals(other);
```

```

    public bool Equals(TwoDPoint p) => X == p.X && Y == p.Y;

    public override int GetHashCode() => (X, Y).GetHashCode();

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs) =>
lhs.Equals(rhs);

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !
(lhs == rhs);
}

class Program
{
    static void Main(string[] args)
    {
        TwoDPoint pointA = new TwoDPoint(3, 4);
        TwoDPoint pointB = new TwoDPoint(3, 4);
        int i = 5;

        // True:
        Console.WriteLine("pointA.Equals(pointB) = {0}",
pointA.Equals(pointB));
        // True:
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        // True:
        Console.WriteLine("object.Equals(pointA, pointB) = {0}",
object.Equals(pointA, pointB));
        // False:
        Console.WriteLine("pointA.Equals(null) = {0}",
pointA.Equals(null));
        // False:
        Console.WriteLine("(pointA == null) = {0}", pointA == null);
        // True:
        Console.WriteLine("(pointA != null) = {0}", pointA != null);
        // False:
        Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
        // CS0019:
        // Console.WriteLine("pointA == i = {0}", pointA == i);

        // Compare unboxed to boxed.
        System.Collections.ArrayList list = new
System.Collections.ArrayList();
        list.Add(new TwoDPoint(3, 4));
        // True:
        Console.WriteLine("pointA.Equals(list[0]): {0}",
pointA.Equals(list[0]));

        // Compare nullable to nullable and to non-nullable.
        TwoDPoint? pointC = null;
        TwoDPoint? pointD = null;
        // False:
        Console.WriteLine("pointA == (pointC = null) = {0}", pointA ==
pointC);
        // True:

```

```

        Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

        TwoDPoint temp = new TwoDPoint(3, 4);
        pointC = temp;
        // True:
        Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA ==
pointC);

        pointD = temp;
        // True:
        Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD ==
pointC);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   pointA.Equals(pointB) = True
   pointA == pointB = True
   Object.Equals(pointA, pointB) = True
   pointA.Equals(null) = False
   (pointA == null) = False
   (pointA != null) = True
   pointA.Equals(i) = False
   pointE.Equals(list[0]): True
   pointA == (pointC = null) = False
   pointC == pointD = True
   pointA == (pointC = 3,4) = True
   pointD == (pointC = 3,4) = True
*/
}

```

Para estruturas, a implementação padrão de `Object.Equals(Object)` (que é a versão substituída em `System.ValueType`) executa uma verificação de igualdade de valor por meio de reflexão para comparar os valores de cada campo no tipo. Quando um implementador substitui o método virtual `Equals` em um struct, a finalidade é fornecer uma maneira mais eficiente de executar a verificação de igualdade de valor e, opcionalmente, basear a comparação em algum subconjunto dos campos ou propriedades do struct.

Os operadores `==` e `!=` não podem operar em um struct a menos que o struct os sobrecarregue explicitamente.

## Confira também

- [Comparações de igualdade](#)
- [Guia de programação em C#](#)



# Como testar a igualdade de referência (identidade) (Guia de Programação em C#)

Artigo • 07/04/2023

Não é necessário implementar qualquer lógica personalizada para dar suporte a comparações de igualdade de referência em seus tipos. Essa funcionalidade é fornecida para todos os tipos pelo método estático [Object.ReferenceEquals](#).

O exemplo a seguir mostra como determinar se duas variáveis têm *igualdade de referência*, que significa que elas se referem ao mesmo objeto na memória.

O exemplo também mostra por que [Object.ReferenceEquals](#) sempre retorna `false` para tipos de valor e por que você não deve usar [ReferenceEquals](#) para determinar igualdade de cadeia de caracteres.

## Exemplo

```
C#  
  
using System.Text;  
  
namespace TestReferenceEquality  
{  
    struct TestStruct  
    {  
        public int Num { get; private set; }  
        public string Name { get; private set; }  
  
        public TestStruct(int i, string s) : this()  
        {  
            Num = i;  
            Name = s;  
        }  
    }  
  
    class TestClass  
    {  
        public int Num { get; set; }  
        public string? Name { get; set; }  
    }  
  
    class Program  
    {  
        static void Main()
```

```

{
    // Demonstrate reference equality with reference types.

    #region ReferenceTypes

        // Create two reference type instances that have identical
        values.
        TestClass tcA = new TestClass() { Num = 1, Name = "New
TestClass" };
        TestClass tcB = new TestClass() { Num = 1, Name = "New
TestClass" };

        Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                           Object.ReferenceEquals(tcA, tcB)); // false

        // After assignment, tcB and tcA refer to the same object.
        // They now have reference equality.
        tcB = tcA;
        Console.WriteLine("After assignment: ReferenceEquals(tcA, tcB) =
{0}",
                           Object.ReferenceEquals(tcA, tcB)); // true

        // Changes made to tcA are reflected in tcB. Therefore, objects
        // that have reference equality also have value equality.
        tcA.Num = 42;
        tcA.Name = "TestClass 42";
        Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name,
tcB.Num);
    #endregion

    // Demonstrate that two value type instances never have
    reference equality.

    #region ValueTypes

        TestStruct tsC = new TestStruct( 1, "TestStruct 1");

        // Value types are copied on assignment. tsD and tsC have
        // the same values but are not the same object.
        TestStruct tsD = tsC;
        Console.WriteLine("After assignment: ReferenceEquals(tsC, tsD) =
{0}",
                           Object.ReferenceEquals(tsC, tsD)); // false
    #endregion

    #region stringRefEquality
        // Constant strings within the same assembly are always interned
        by the runtime.
        // This means they are stored in the same location in memory.
Therefore,
        // the two strings have reference equality although no
assignment takes place.
        string strA = "Hello world!";
        string strB = "Hello world!";
        Console.WriteLine("ReferenceEquals(strA, strB) = {0}",
                           Object.ReferenceEquals(strA, strB)); // true
    
```

```

        // After a new string is assigned to strA, strA and strB
        // are no longer interned and no longer have reference equality.
        strA = "Goodbye world!";
        Console.WriteLine("strA = \"{0}\" strB = \"{1}\\"", strA, strB);

        Console.WriteLine("After strA changes, ReferenceEquals(strA,
strB) = {0}",
                           Object.ReferenceEquals(strA, strB)); // false

        // A string that is created at runtime cannot be interned.
        StringBuilder sb = new StringBuilder("Hello world!");
        string stringC = sb.ToString();
        // False:
        Console.WriteLine("ReferenceEquals(stringC, strB) = {0}",
                           Object.ReferenceEquals(stringC, strB));

        // The string class overloads the == operator to perform an
equality comparison.
        Console.WriteLine("stringC == strB = {0}", stringC == strB); // true

    #endregion

    // Keep the console open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

}

/*
/* Output:
    ReferenceEquals(tcA, tcB) = False
    After assignment: ReferenceEquals(tcA, tcB) = True
    tcB.Name = TestClass 42 tcB.Num: 42
    After assignment: ReferenceEquals(tsC, tsD) = False
    ReferenceEquals(strA, strB) = True
    strA = "Goodbye world!" strB = "Hello world!"
    After strA changes, ReferenceEquals(strA, strB) = False
    ReferenceEquals(stringC, strB) = False
    stringC == strB = True
*/

```

A implementação de `Equals` na classe base universal `System.Object` também realiza uma verificação de igualdade de referência, mas é melhor não usar isso, porque, se uma classe substituir o método, os resultados poderão não ser o que você espera. O mesmo é verdadeiro para os operadores `==` e `!=`. Quando eles estiverem operando em tipos de referência, o comportamento padrão de `==` e `!=` é realizar uma verificação de igualdade de referência. No entanto, as classes derivadas podem sobrecarregar o operador para executar uma verificação de igualdade de valor. Para minimizar o potencial de erro, será melhor usar sempre `ReferenceEquals` quando for necessário determinar se os dois objetos têm igualdade de referência.

Cadeias de caracteres constantes dentro do mesmo assembly sempre são internalizadas pelo runtime. Ou seja, apenas uma instância de cada cadeia de caracteres literal única é mantida. No entanto, o runtime não garante que cadeias de caracteres criadas em runtime sejam internalizadas nem garante que duas de cadeias de caracteres constantes iguais em diferentes assemblies sejam internalizadas.

## Confira também

- [Comparações de igualdade](#)

# Coerções e conversões de tipo (Guia de Programação em C#)

Artigo • 07/04/2023

Como o C# é tipado estaticamente no tempo de compilação, depois que uma variável é declarada, ela não pode ser declarada novamente ou atribuída a um valor de outro tipo, a menos que esse tipo possa ser convertido implicitamente no tipo da variável. Por exemplo, a `string` não pode ser convertida implicitamente em `int`. Portanto, depois de declarar `i` como um `int`, não é possível atribuir a cadeia de caracteres "Hello" a ele, como mostra o código a seguir:

C#

```
int i;

// error CS0029: Cannot implicitly convert type 'string' to 'int'
i = "Hello";
```

No entanto, às vezes é necessário copiar um valor para uma variável ou um parâmetro de método de outro tipo. Por exemplo, você pode ter que passar uma variável de inteiro para um método cujo parâmetro é digitado como `double`. Ou talvez precise atribuir uma variável de classe a uma variável de um tipo de interface. Esses tipos de operações são chamados de *conversões de tipo*. No C#, você pode realizar os seguintes tipos de conversões:

- **Conversões implícitas:** nenhuma sintaxe especial é necessária porque a conversão sempre é bem sucedida e nenhum dado será perdido. Exemplos incluem conversões de tipos inteiros menores para maiores e conversões de classes derivadas para classes base.
- **Conversões explícitas (casts):** as conversões explícitas exigem uma [expressão cast](#). A conversão é necessária quando as informações podem ser perdidas na conversão ou quando a conversão pode não funcionar por outros motivos. Exemplos típicos incluem a conversão numérica para um tipo que tem menos precisão ou um intervalo menor e a conversão de uma instância de classe base para uma classe derivada.
- **Conversões definidas pelo usuário:** as conversões definidas pelo usuário são realizadas por métodos especiais que podem ser definidos para habilitar conversões explícitas e implícitas entre tipos personalizados que não têm uma

relação de classe base/classe derivada. Para saber mais, confira [Operadores de conversão definidos pelo usuário](#).

- **Conversões com classes auxiliares:** para converter entre tipos não compatíveis, assim como inteiros e objetos `System.DateTime`, ou cadeias de caracteres hexadecimais e matrizes de bytes, você pode usar a classe `System.BitConverter`, a classe `System.Convert` e os métodos `Parse` dos tipos numéricos internos, tais como `Int32.Parse`. Para obter mais informações, consulte [Como converter uma matriz de bytes em um int](#), [Como converter uma cadeia de caracteres em um número](#) e [Como converter entre cadeias de caracteres hexadecimais e tipos numéricos](#).

## Conversões implícitas

Para tipos numéricos internos, uma conversão implícita poderá ser feita quando o valor a ser armazenado puder se ajustar à variável sem ser truncado ou arredondado. Para tipos inteiros, isso significa que o intervalo do tipo de origem é um subconjunto apropriado do intervalo para o tipo de destino. Por exemplo, uma variável do tipo `long` (inteiro de 64 bits) pode armazenar qualquer valor que um `int` (inteiro de 32 bits) pode armazenar. No exemplo a seguir, o compilador converte implicitamente o valor de `num` à direita em um tipo `long` antes de atribuí-lo a `bigNum`.

C#

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

Para obter uma lista completa de todas as conversões numéricas implícitas, consulte a seção [Conversões numéricas implícitas](#) do artigo [Conversões numéricas internas](#).

Para tipos de referência, uma conversão implícita sempre existe de uma classe para qualquer uma das suas interfaces ou classes base diretas ou indiretas. Nenhuma sintaxe especial é necessária porque uma classe derivada sempre contém todos os membros de uma classe base.

C#

```
Derived d = new Derived();
// Always OK.
Base b = d;
```

# Conversões explícitas

No entanto, se uma conversão não puder ser realizada sem o risco de perda de informações, o compilador exigirá que você execute uma conversão explícita, que é chamada de *cast*. Uma conversão é uma maneira de informar explicitamente ao compilador que você pretende fazer a conversão e que você está ciente de que poderá ocorrer perda de dados ou a conversão poderá falhar em tempo de execução. Para executar uma conversão, especifique entre parênteses o tipo para o qual você está convertendo, na frente do valor ou da variável a ser convertida. O programa a seguir converte um `double` em um `int`. O programa não será compilado sem a conversão.

C#

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

Para obter uma lista completa de conversões numéricas explícitas com suporte, consulte a seção [Conversões numéricas explícitas](#) do artigo [Conversões numéricas internas](#).

Para tipos de referência, uma conversão explícita será necessária se você precisar converter de um tipo base para um tipo derivado:

C#

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe)a;
```

Uma operação de conversão entre tipos de referência não altera o tipo de tempo de execução do objeto subjacente. Ela apenas altera o tipo do valor que está sendo usado como uma referência a esse objeto. Para obter mais informações, consulte [Polimorfismo](#).

## Exceções de conversão de tipo em tempo de execução

Em algumas conversões de tipo de referência, o compilador não poderá determinar se uma conversão será válida. É possível que uma operação de conversão que é compilada corretamente falhe em tempo de execução. Conforme mostrado no exemplo a seguir, um tipo de conversão que falha em tempo de execução fará com que uma [InvalidCastException](#) seja lançada.

C#

```
class Animal
{
    public void Eat() => System.Console.WriteLine("Eating.");
    public override string ToString() => "I am an animal.";
}

class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // System.InvalidCastException at run time
        // Unable to cast object of type 'Mammal' to type 'Reptile'
        Reptile r = (Reptile)a;
    }
}
```

O método `Test` tem um parâmetro `Animal`, assim, converter explicitamente o argumento `a` em um `Reptile` é fazer uma suposição perigosa. É mais seguro não fazer suposições, mas sim verificar o tipo. O C# fornece o operador `is` para habilitar o teste de

compatibilidade antes de realmente executar uma conversão. Para saber mais, confira [Como converter com segurança usando a correspondência de padrões e os operadores "is" e "as"](#).

## Especificação da linguagem C#

Para saber mais, confira a seção [Conversões da Especificação da linguagem C#](#).

## Confira também

- [Guia de Programação em C#](#)
- [Types](#)
- [Expressão de conversão](#)
- [Operadores de conversões definidas pelo usuário](#)
- [Conversão de tipos generalizada](#)
- [Como converter uma cadeia de caracteres em um número](#)

# Conversões boxing e unboxing (Guia de Programação em C#)

Artigo • 07/04/2023

Conversão boxing é o processo de conversão de um [tipo de valor](#) para o tipo `object` ou para qualquer tipo de interface implementada por esse tipo de valor. Quando o CLR (Common Language Runtime) realiza a conversão de um tipo de valor, ele encapsula o valor dentro de uma instância `System.Object` e a armazena no heap gerenciado. A conversão unboxing extrai o tipo de valor do objeto. A conversão boxing é implícita, a conversão unboxing é explícita. O conceito de conversões boxing e unboxing serve como base para a exibição unificada de C# do sistema de tipos em que um valor de qualquer tipo pode ser tratado como um objeto.

No exemplo a seguir, a variável de inteiro `i` é submetida à *conversão boxing* e atribuída ao objeto `o`.

C#

```
int i = 123;
// The following line boxes i.
object o = i;
```

O objeto `o` pode ser submetido à conversão unboxing e atribuído à variável de inteiro `i`:

C#

```
o = 123;
i = (int)o; // unboxing
```

Os exemplos a seguir ilustram como a conversão boxing é usada em C#.

C#

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
```

```
// of elements.
List<object> mixedList = new List<object>();

// Add a string element to the list.
mixedList.Add("First Group:");

// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j]);

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
```

```
// 2
// 3
// 4
// Second Group:
// 5
// 6
// 7
// 8
// 9
// Sum: 30
```

## Desempenho

Em relação às atribuições simples, as conversões boxing e unboxing são processos computacionalmente dispendiosos. Quando um tipo de valor é submetido à conversão boxing, um novo objeto deve ser alocado e construído. A um grau menor, a conversão necessária para a conversão unboxing também é computacionalmente dispendiosa. Para obter mais informações, consulte [Desempenho](#).

## Conversão boxing

A conversão boxing é usada para armazenar tipos de valor no heap coletado como lixo. A conversão boxing é uma conversão implícita de um [tipo de valor](#) para o tipo [object](#) ou para qualquer tipo de interface implementada por esse tipo de valor. A conversão boxing de um tipo de valor aloca uma instância de objeto no heap e copia o valor no novo objeto.

Considere a seguinte declaração de uma variável de tipo de valor:

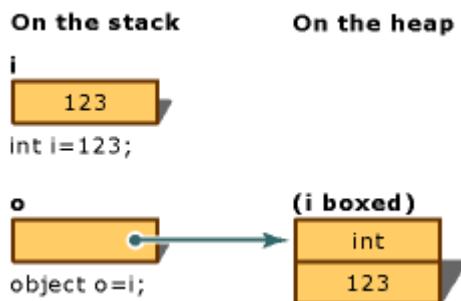
```
C#
int i = 123;
```

A instrução a seguir aplica implicitamente a operação de conversão boxing na variável `i`:

```
C#
// Boxing copies the value of i into object o.
object o = i;
```

O resultado dessa instrução é a criação de uma referência de objeto `o`, na pilha, que faz referência a um valor do tipo `int`, no heap. Esse valor é uma cópia do valor do tipo de

valor atribuído à variável `i`. A diferença entre as duas variáveis, `i` e `o`, é ilustrada na figura de conversão boxing a seguir:



Também é possível executar a conversão boxing explicitamente como no exemplo a seguir, mas a conversão boxing explícita nunca é necessária:

C#

```
int i = 123;
object o = (object)i; // explicit boxing
```

## Exemplo

Este exemplo converte uma variável de inteiro `i` em um objeto `o` usando a conversão boxing. Em seguida, o valor armazenado na variável `i` é alterado de `123` para `456`. O exemplo mostra que o tipo do valor original e o objeto submetido à conversão boxing usa locais de memória separados e, portanto, pode armazenar valores diferentes.

C#

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i doesn't affect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
   The value-type value = 456
   The object-type value = 123
```

```
The object-type value = 123  
*/
```

## Conversão unboxing

A conversão unboxing é uma conversão explícita do tipo `object` para um [tipo de valor](#) ou de um tipo de interface para um tipo de valor que implementa a interface. Uma operação de conversão unboxing consiste em:

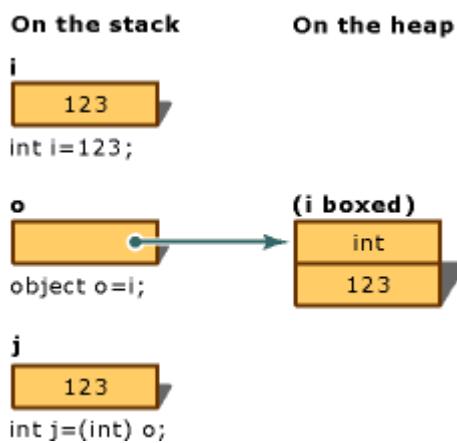
- Verificar a instância do objeto para garantir que ele é um valor da conversão boxing de um determinado tipo de valor.
- Copiar o valor da instância para a variável de tipo de valor.

As instruções a seguir demonstram operações conversão boxing e unboxing:

C#

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

A figura a seguir demonstra o resultado das instruções anteriores:



Para a conversão unboxing de tipos de valor ter êxito em tempo de execução, o item sendo submetido à conversão unboxing deve ser uma referência para um objeto que foi criado anteriormente ao realizar a conversão boxing de uma instância desse tipo de valor. Tentar realizar a conversão unboxing de `null` causa uma [NullReferenceException](#). Tentar realizar a conversão unboxing de uma referência para um tipo de valor incompatível causa uma [InvalidCastException](#).

## Exemplo

O exemplo a seguir demonstra um caso de conversão unboxing inválida e o `InvalidOperationException` resultante. Usando `try` e `catch`, uma mensagem de erro é exibida quando o erro ocorre.

C#

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i; // implicit boxing

        try
        {
            int j = (short)o; // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
        }
    }
}
```

Este programa produz:

```
Specified cast is not valid. Error: Incorrect unboxing.
```

Se você alterar a instrução:

C#

```
int j = (short)o;
```

para:

C#

```
int j = (int)o;
```

a conversão será executada e você receberá a saída:

```
Unboxing OK.
```

# Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Guia de programação em C#](#)
- [Tipos de referência](#)
- [Tipos de valor](#)

# Como converter uma matriz de bytes em um int (Guia de Programação em C#)

Artigo • 07/04/2023

Este exemplo mostra como usar a classe [BitConverter](#) para converter uma matriz de bytes em um `int` e de volta em uma matriz de bytes. Talvez você precise converter bytes em um tipo de dados interno depois de ler bytes da rede, por exemplo. Além do método [ToInt32\(Byte\[\], Int32\)](#) no exemplo, a tabela a seguir lista métodos na classe [BitConverter](#) que convertem bytes (de uma matriz de bytes) em outros tipos internos.

Tipo retornado	Método
<code>bool</code>	<a href="#">ToBoolean(Byte[], Int32)</a>
<code>char</code>	<a href="#">ToChar(Byte[], Int32)</a>
<code>double</code>	<a href="#">.ToDouble(Byte[], Int32)</a>
<code>short</code>	<a href="#">ToInt16(Byte[], Int32)</a>
<code>int</code>	<a href="#">ToInt32(Byte[], Int32)</a>
<code>long</code>	<a href="#">ToInt64(Byte[], Int32)</a>
<code>float</code>	<a href="#">ToSingle(Byte[], Int32)</a>
<code>ushort</code>	<a href="#">ToUInt16(Byte[], Int32)</a>
<code>uint</code>	<a href="#">ToUInt32(Byte[], Int32)</a>
<code>ulong</code>	<a href="#">ToUInt64(Byte[], Int32)</a>

## Exemplos

Este exemplo inicializa uma matriz de bytes, reverte a matriz se a arquitetura do computador for little-endian (ou seja, se o byte menos significativo for armazenado primeiro) e, em seguida, chama o método [ToInt32\(Byte\[\], Int32\)](#) para converter quatro bytes da matriz em um `int`. O segundo argumento para [ToInt32\(Byte\[\], Int32\)](#) especifica o índice de início da matriz de bytes.

### Observação

A saída pode ser diferente dependendo da extremidade (ordenação dos bytes) da arquitetura do computador.

C#

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

Neste exemplo, o método [GetBytes\(Int32\)](#) da classe [BitConverter](#) é chamado para converter um `int` em uma matriz de bytes.

### ⓘ Observação

A saída pode ser diferente dependendo da extremidade (ordenação dos bytes) da arquitetura do computador.

C#

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

## Confira também

- [BitConverter](#)
- [IsLittleEndian](#)
- [Types](#)

# Como converter uma cadeia de caracteres em um número (Guia de Programação em C#)

Artigo • 07/04/2023

Converta um `string` em um número chamando o método `Parse` ou `TryParse` encontrado em tipos numéricos (`int`, `long`, `double` e assim por diante) ou usando métodos na classe `System.Convert`.

Será um pouco mais eficiente e simples chamar um método `TryParse` (por exemplo, `int.TryParse("11", out number)`) ou o método `Parse` (por exemplo, `var number = int.Parse("11")`). Usar um método `Convert` é mais útil para objetos gerais que implementam `IConvertible`.

É possível usar métodos `Parse` ou `TryParse` no tipo numérico que se espera que a cadeia de caracteres contenha, como o tipo `System.Int32`. O método `Convert.ToInt32` usa `Parse` internamente. O método `Parse` retorna o número convertido; o método `TryParse` retorna um valor booleano que indica se a conversão foi bem-sucedida e retorna o número convertido em um parâmetro `out`. Se a cadeia de caracteres não estiver em um formato válido, `Parse` lançará uma exceção, mas `TryParse` retornará `false`. Ao chamar um método `Parse`, você sempre deve usar o tratamento de exceções para capturar um `FormatException` quando a operação de análise falhar.

## Chamar métodos Parse ou TryParse

Os métodos `Parse` e `TryParse` ignoram o espaço em branco no início e no final da cadeia de caracteres; porém, todos os outros caracteres devem formar o tipo numérico correto (`int`, `long`, `ulong`, `float`, `decimal` e assim por diante). Qualquer espaço em branco na cadeia de caracteres que forma o número causa um erro. Por exemplo, é possível usar `decimal.TryParse` para analisar "10", "10,3" ou " 10 ", mas não é possível usar esse método para analisar 10 de "10X", "1 0" (observe o espaço inserido), "10 .3" (observe o espaço inserido), "10e1" (`float.TryParse` funcionará neste caso) e assim por diante. Uma cadeia de caracteres cujo valor seja `null` ou `String.Empty` falhará ao ser analisada com êxito. Você pode verificar uma cadeia de caracteres nula ou vazia antes de tentar analisá-la chamando o método `String.IsNullOrEmpty`.

O exemplo a seguir demonstra chamadas com e sem êxito para `Parse` e `TryParse`.

C#

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        string input = String.Empty;
        try
        {
            int result = Int32.Parse(input);
            Console.WriteLine(result);
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{input}'");
        }
        // Output: Unable to parse ''

        try
        {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: -105

        if (Int32.TryParse("-105", out int j))
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
        }
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        const string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
        {
```

```

        Console.WriteLine(numValue);
    }
    else
    {
        Console.WriteLine($"Int32.TryParse could not parse
'{inputString}' to an int.");
    }
    // Output: Int32.TryParse could not parse 'abc' to an int.
}
}

```

O exemplo a seguir ilustra uma abordagem para analisar uma cadeia de caracteres que deve incluir caracteres numéricos à esquerda (incluindo caracteres hexadecimais) e caracteres não numéricos à direita. Ele atribui caracteres válidos do início de uma cadeia de caracteres até uma nova cadeia de caracteres antes de chamar o método [TryParse](#). Como as cadeias de caracteres a ser analisadas contêm poucos caracteres, o exemplo chama o método [String.Concat](#) para atribuir os caracteres válidos a uma nova cadeia de caracteres. Para cadeias de caracteres maiores, pode ser usada a classe [StringBuilder](#).

C#

```

using System;

public static class StringConversion
{
    public static void Main()
    {
        var str = " 10FFxxx";
        string numericString = string.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or
            // trailing spaces.
            if ((c >= '0' && c <= '9') || (char.ToUpperInvariant(c) >= 'A'
            && char.ToUpperInvariant(c) <= 'F') || c == ' ')
            {
                numericString = string.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString,
System.Globalization.NumberStyles.HexNumber, null, out int i))
        {
            Console.WriteLine($"{str} --> '{numericString}' --> {i}");
        }
        // Output: ' 10FFxxx' --> ' 10FF' --> 4351
    }
}

```

```

        str = " -10FFXXX";
        numericString = "";
        foreach (char c in str)
        {
            // Check for numeric characters (0-9), a negative sign, or
            // leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == '-' || c == ' ')
            {
                numericString = string.Concat(numericString, c);
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, out int j))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
        }
        // Output: ' -10FFXXX' --> ' -10' --> -10
    }
}

```

## Chamar métodos Convert

A tabela a seguir lista alguns dos métodos da classe [Convert](#) que podem ser usados para converter uma cadeia de caracteres em um número.

Tipo numérico	Método
<code>decimal</code>	<a href="#">ToDecimal(String)</a>
<code>float</code>	<a href="#">ToSingle(String)</a>
<code>double</code>	<a href="#">.ToDouble(String)</a>
<code>short</code>	<a href="#">ToInt16(String)</a>
<code>int</code>	<a href="#">ToInt32(String)</a>
<code>long</code>	<a href="#">ToInt64(String)</a>
<code>ushort</code>	<a href="#">ToUInt16(String)</a>
<code>uint</code>	<a href="#">ToUInt32(String)</a>
<code>ulong</code>	<a href="#">ToUInt64(String)</a>

O exemplo a seguir chama o método `Convert.ToInt32(String)` para converter uma cadeia de caracteres de entrada em um `int`. O exemplo captura as duas exceções mais comuns que podem ser geradas por esse método `FormatException` e `OverflowException`. Se o número resultante puder ser incrementado sem exceder `Int32.MaxValue`, o exemplo adicionará 1 ao resultado e exibirá a saída.

C#

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between -2,147,483,648 and
+2,147,483,647 (inclusive): ");

            string? input = Console.ReadLine();

            //ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond
its current value");
                }
            }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of
digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string? go = Console.ReadLine();
            if (go?.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}
```

```
        }
    }
}

// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
473
//   The new value is 474
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
2147483647
//   numVal cannot be incremented beyond its current value
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
-1000
//   The new value is -999
//   Go again? Y/N: n
```

# Como converter entre cadeias de caracteres hexadecimais e tipos numéricos (Guia de Programação em C#)

Artigo • 10/05/2023

Estes exemplos mostram como realizar as seguintes tarefas:

- Obter o valor hexadecimal de cada caractere em uma cadeia de caracteres.
- Obter o `char` que corresponde a cada valor em uma cadeia de caracteres hexadecimal.
- Converter um `string` hexadecimal em um `int`.
- Converter um `string` hexadecimal em um `float`.
- Como converter uma matriz de `bytes` em um `string` hexadecimal.

## Exemplos

Este exemplo gera o valor hexadecimal de cada caractere em um `string`. Primeiro, ele analisa o `string` como uma matriz de caracteres. Em seguida, ele chama `ToInt32(Char)` em cada caractere para obter seu valor numérico. Por fim, ele formata o número como sua representação hexadecimal em um `string`.

```
C#  
  
string input = "Hello World!";  
char[] values = input.ToCharArray();  
foreach (char letter in values)  
{  
    // Get the integral value of the character.  
    int value = Convert.ToInt32(letter);  
    // Convert the integer value to a hexadecimal value in string form.  
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");  
}  
/* Output:  
   Hexadecimal value of H is 48  
   Hexadecimal value of e is 65  
   Hexadecimal value of l is 6C  
   Hexadecimal value of l is 6C  
   Hexadecimal value of o is 6F
```

```

    Hexadecimal value of   is 20
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
    Hexadecimal value of ! is 21
*/

```

Este exemplo analisa um `string` de valores hexadecimais e gera o caractere correspondente a cada valor hexadecimal. Primeiro, ele chama o método `Split(Char[])` para obter cada valor hexadecimal como um `string` individual em uma matriz. Em seguida, ele chama `ToInt32(String, Int32)` para converter o valor hexadecimal em um valor decimal representado como um `int`. Ele mostra duas maneiras diferentes de obter o caractere correspondente a esse código de caractere. A primeira técnica usa `ConvertFromUtf32(Int32)`, que retorna o caractere correspondente ao argumento de inteiro como um `string`. A segunda técnica converte explicitamente o `int` em um `char`.

C#

```

string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value
= {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
   hexadecimal value = 48, int value = 72, char value = H or h
   hexadecimal value = 65, int value = 101, char value = e or E
   hexadecimal value = 6C, int value = 108, char value = l or L
   hexadecimal value = 6C, int value = 108, char value = l or L
   hexadecimal value = 6F, int value = 111, char value = o or O
   hexadecimal value = 20, int value = 32, char value = or
   hexadecimal value = 57, int value = 87, char value = W or w
   hexadecimal value = 6F, int value = 111, char value = o or O
   hexadecimal value = 72, int value = 114, char value = r or R
   hexadecimal value = 6C, int value = 108, char value = l or L
   hexadecimal value = 64, int value = 100, char value = d or D
   hexadecimal value = 21, int value = 33, char value = ! or !
*/

```

Este exemplo mostra outra maneira de converter um hexadecimal `string` em um inteiro, chamando o método `Parse(String, NumberStyles)`.

C#

```
string hexString = "8E2";
int num = Int32.Parse(hexString,
System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274
```

O exemplo a seguir mostra como converter um hexadecimal `string` para um `float` usando a classe `System.BitConverter` e o método `UInt32.Parse`.

C#

```
string hexString = "43480170";
uint num = uint.Parse(hexString,
System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056
```

O exemplo a seguir mostra como converter uma matriz de `bytes` em uma cadeia de caracteres hexadecimal usando a classe `System.BitConverter`.

C#

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
01-AA-B1-DC-10-DD
01AAB1DC10DD
*/
```

O exemplo a seguir mostra como converter uma matriz de `byte` em uma cadeia de caracteres hexadecimal chamando o método `Convert.ToString`, introduzido no .NET 5.0.

C#

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToString(array);
Console.WriteLine(hexValue);

/*Output:
646F74636574
*/
```

## Confira também

- [Cadeias de Caracteres de Formato Numérico Padrão](#)
- [Types](#)
- [Como determinar se uma cadeia de caracteres representa um valor numérico](#)

# Usar o tipo dinâmico

Artigo • 16/06/2023

O tipo `dynamic` é estático, mas um objeto do tipo `dynamic` ignora a verificação de tipo estático. Na maioria dos casos, ele funciona como se tivesse o tipo `object`. O compilador pressupõe que um elemento `dynamic` dá suporte a qualquer operação. Portanto, você não precisa determinar se o objeto obtém seu valor de uma API COM, de uma linguagem dinâmica como o IronPython, do HTML DOM (Modelo de Objeto do Documento), a reflexão ou de algum outro lugar no programa. No entanto, se o código não for válido, os erros aparecerão em tempo de execução.

Por exemplo, se método de instância `exampleMethod1` no código a seguir tiver apenas um parâmetro, o compilador reconhecerá que a primeira chamada para o método, `ec.exampleMethod1(10, 4)`, não será válido porque ele contém dois argumentos. Essa chamada causa um erro do compilador. O compilador não verifica a segunda chamada para o método, `dynamic_ec.exampleMethod1(10, 4)`, porque o tipo `dynamic_ec` é `dynamic`. Portanto, nenhum erro de compilador é relatado. No entanto, o erro não passa despercebido indefinidamente. Ele aparece em tempo de execução e causa uma exceção em tempo de execução.

C#

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

C#

```
class ExampleClass
{
```

```
public ExampleClass() { }
public ExampleClass(int v) { }

public void exampleMethod1(int i) { }

public void exampleMethod2(string str) { }
}
```

A função do compilador nesses exemplos é reunir informações sobre o que cada instrução está propondo fazer para o objeto ou expressão que tem o tipo `dynamic`. O runtime examina as informações armazenadas e qualquer instrução que não seja válida causa uma exceção em tempo de execução.

O resultado de operações mais dinâmicas é `dynamic`. Por exemplo, se você passar o ponteiro do mouse sobre o uso de `testSum` no exemplo a seguir, o IntelliSense exibirá o tipo (variável local) **testSum dinâmico**.

C#

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

As operações em que o resultado não é `dynamic` incluem:

- Conversões de `dynamic` em outro tipo.
- Chamadas de construtor que incluem argumentos do tipo `dynamic`.

Por exemplo, o tipo de `testInstance` na seguinte declaração é `ExampleClass` e não `dynamic`:

C#

```
var testInstance = new ExampleClass(d);
```

## Conversões

As conversões entre objetos dinâmicos e outros tipos são fáceis. As conversões permitem que o desenvolvedor alterne entre o comportamento dinâmico e o não dinâmico.

Você pode converter qualquer um para `dynamic` implicitamente, conforme mostrado nos exemplos a seguir.

C#

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

Por outro lado, você pode aplicar dinamicamente qualquer conversão implícita a qualquer expressão do tipo `dynamic`.

C#

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

## Resolução de sobrecarga com argumentos de tipo `dynamic`

A resolução de sobrecarga ocorre em tempo de execução em vez de em tempo de compilação se um ou mais dos argumentos em uma chamada de método tem o tipo `dynamic` ou se o receptor da chamada do método é do tipo `dynamic`. No exemplo a seguir, se o único método `exampleMethod2` acessível for definido para obter um argumento de cadeia de caracteres, enviar `d1` como o argumento não causará um erro de compilador, mas causará uma exceção de tempo de execução. A resolução de sobrecarga falha em tempo de execução porque o tipo de tempo de execução de `d1` é `int` e `exampleMethod2` requer uma cadeia de caracteres.

C#

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec
is not
// dynamic. A run-time exception is raised because the run-time type of d1
is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

# runtime de linguagem dinâmico

O DLR (tempo de execução de linguagem dinâmica) fornece a infraestrutura que dá suporte ao tipo `dynamic` em C# e também à implementação das linguagens de programação dinâmicas como IronPython e IronRuby. Para obter mais informações sobre o DLR, consulte [Visão geral do Dynamic Language Runtime](#).

## interoperabilidade COM

Muitos métodos COM permitem variação nos tipos de argumento e tipo de retorno, especificando os tipos como `object`. A interoperabilidade COM exigiu a conversão explícita dos valores para coordenar com variáveis fortemente tipadas no C#. Se você compilar usando a opção [EmbedInteropTypes](#) (opções do compilador C#), a introdução do tipo `dynamic` permitirá tratar as ocorrências de `object` em assinaturas COM, como se fossem do tipo `dynamic` e, portanto, evitar grande parte da conversão. Para obter mais informações sobre como usar o tipo `dynamic` com objetos COM, confira o artigo sobre [Como acessar objetos de interoperabilidade do Office usando recursos C#](#).

## Artigos relacionados

Título	Descrição
<a href="#">dinâmico</a>	Descreve o uso da palavra-chave <code>dynamic</code> .
<a href="#">Visão geral do Dynamic Language Runtime</a>	Fornece uma visão geral do DLR, que é um ambiente de tempo de execução que adiciona um conjunto de serviços para as linguagens dinâmicas para o CLR (Common Language Runtime).
<a href="#">Passo a passo: Criando e usando objetos dinâmicos</a>	Fornece instruções passo a passo para criar um objeto dinâmico personalizado e para criar um projeto que acessa uma biblioteca <a href="#">IronPython</a> .

# Passo a passo: criar e usar objetos dinâmicos em C#

Artigo • 09/05/2023

Os objetos dinâmicos expõem membros como propriedades e métodos em tempo de execução, em vez de em tempo de compilação. Os objetos dinâmicos permitem que você crie objetos para trabalhar com estruturas que não correspondem a um formato ou tipo estático. Por exemplo, você pode usar um objeto dinâmico para fazer referência ao DOM (Modelo de Objeto do Documento) HTML, que pode conter qualquer combinação de atributos e elementos de marcação HTML válidos. Como cada documento HTML é único, os membros de um determinado documento HTML são determinados em tempo de execução. Um método comum para fazer referência a um atributo de um elemento HTML é passar o nome do atributo para o método `GetProperty` do elemento. Para fazer referência ao atributo `id` do elemento HTML `<div id="Div1">`, primeiro você obtém uma referência ao elemento `<div>` e, depois, usa `divElement.GetProperty("id")`. Se usar um objeto dinâmico, você poderá fazer referência ao atributo `id` como `divElement.id`.

Objetos dinâmicos também fornecem acesso conveniente a linguagens dinâmicas, como IronPython e IronRuby. É possível usar um objeto dinâmico para fazer referência a um script dinâmico interpretado em tempo de execução.

Você faz referência a um objeto dinâmico usando a associação tardia. Especifique o tipo de um objeto com associação tardia como `dynamic`. Para obter mais informações, confira [dynamic](#).

Você pode criar objetos dinâmicos personalizados usando classes no namespace [System.Dynamic](#). Por exemplo, é possível criar um [ExpandoObject](#) e especificar os membros desse objeto em tempo de execução. Você também pode criar seu próprio tipo que herda da classe [DynamicObject](#). Em seguida, você pode substituir os membros da classe [DynamicObject](#) para fornecer funcionalidade dinâmica de tempo de execução.

Este artigo contém dois passo a passo independentes:

- Criar um objeto personalizado que expõe dinamicamente o conteúdo de um arquivo de texto como propriedades de um objeto.
- Criar um projeto que usa uma biblioteca [IronPython](#).

## Pré-requisitos

- Visual Studio 2022 versão 17.3 ou uma versão posterior com a carga de trabalho de desenvolvimento para desktop com o .NET instalada. O SDK do .NET 7 é incluído quando você seleciona essa carga de trabalho.

### ⓘ Observação

Seu computador pode mostrar diferentes nomes ou locais para alguns dos elementos de interface do usuário do Visual Studio nas instruções a seguir. A edição do Visual Studio que você possui e as configurações que você usa determinam esses elementos. Para obter mais informações, consulte [Personalizando o IDE](#).

- Para o segundo passo a passo, instale o [IronPython](#) para .NET. Navegue até a respectiva [página de download](#) para obter a versão mais recente.

## Criar um objeto dinâmico personalizado

O primeiro passo a passo define um objeto dinâmico personalizado que pesquisa o conteúdo de um arquivo de texto. Uma propriedade dinâmica especifica o texto a ser pesquisado. Por exemplo, se o código de chamada especificar `dynamicFile.Sample`, a classe dinâmica retornará uma lista genérica de cadeias de caracteres que contém todas as linhas do arquivo que começam com "Sample". A pesquisa diferencia maiúsculas de minúsculas. A classe dinâmica também dá suporte a dois argumentos opcionais. O primeiro argumento é um valor de enum de opção de pesquisa que especifica que a classe dinâmica deve pesquisar por correspondências no início da linha, no final da linha ou em qualquer lugar da linha. O segundo argumento especifica que a classe dinâmica deve cortar espaços iniciais e finais de cada linha antes de pesquisar. Por exemplo, se o código de chamada especificar `dynamicFile.Sample(StringSearchOption.Contains)`, a classe dinâmica pesquisará por "Sample" em qualquer lugar de uma linha. Se o código de chamada especificar `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, a classe dinâmica pesquisará por "Sample" no início de cada linha e não removerá espaços à direita e à esquerda. O comportamento padrão da classe dinâmica é pesquisar por uma correspondência no início de cada linha e remover espaços à direita e à esquerda.

## Criar uma classe dinâmica personalizada

Inicie o Visual Studio. Selecione **Criar um novo projeto**. Na caixa de diálogo **Criar um novo projeto**, selecione C#, selecione **Aplicativo de Console** e, em seguida, **Avançar**. Na caixa de diálogo **Configurar novo projeto**, digite `DynamicSample` como **Nome do**

projeto e selecione em **Avançar**. Na caixa de diálogo **Informações adicionais**, selecione **.NET 7.0 (atual)** como **Estrutura de destino** e, em seguida, selecione **Criar**. No **Gerenciador de Soluções**, clique com o botão direito do mouse no projeto **DynamicSample** e selecione **Adicionar>Classe**. Na caixa **Nome**, digite **ReadOnlyFile** e selecione **Adicionar**. Na parte superior do arquivo *ReadOnlyFile.cs* ou *ReadOnlyFile.vb*, adicione o código a seguir para importar os namespaces **System.IO** e **System.Dynamic**.

C#

```
using System.IO;
using System.Dynamic;
```

O objeto dinâmico personalizado usa um enum para determinar os critérios de pesquisa. Antes da instrução de classe, adicione a seguinte definição de enum.

C#

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

Atualize a instrução de classe para herdar a classe **DynamicObject**, conforme mostrado no exemplo de código a seguir.

C#

```
class ReadOnlyFile : DynamicObject
```

Adicione o código a seguir para a classe **ReadOnlyFile** para definir um campo particular para o caminho do arquivo e um construtor para a classe **ReadOnlyFile**.

C#

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }
}
```

```
    }  
  
    p_filePath = filePath;  
}
```

1. Adicione o seguinte método `GetPropertyValues` à classe `ReadOnlyFile`. O método `GetPropertyValues` usa, como entrada, critérios de pesquisa e retorna as linhas de um arquivo de texto que correspondem a esse critério de pesquisa. Os métodos dinâmicos fornecidos pela classe `ReadOnlyFile` chamam o método `GetPropertyValues` para recuperar seus respectivos resultados.

C#

```

        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return
        null;
        results = null;
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }

    return results;
}

```

Após o método `GetPropertyValues`, adicione o seguinte código para substituir o método `TryGetMember` da classe `DynamicObject`. O método `TryGetMember` é chamado quando um membro de uma classe dinâmica é solicitado e nenhum argumento é especificado. O argumento `binder` contém informações sobre o membro referenciado e o argumento `result` faz referência ao resultado retornado para o membro especificado. O método `TryGetMember` retorna um valor booleano que retorna `true` se o membro solicitado existe; caso contrário, ele retorna `false`.

C#

```

// Implement the TryGetMember method of the DynamicObject class for dynamic
member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                   out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}

```

Após o método `TryGetMember`, adicione o seguinte código para substituir o método `TryInvokeMember` da classe `DynamicObject`. O método `TryInvokeMember` é chamado quando um membro de uma classe dinâmica é solicitado com argumentos. O argumento `binder` contém informações sobre o membro referenciado e o argumento `result` faz referência ao resultado retornado para o membro especificado. O argumento `args` contém uma matriz de argumentos que são passados ao membro. O método `TryInvokeMember` retorna um valor booleano que retorna `true` se o membro solicitado existe; caso contrário, ele retorna `false`.

A versão personalizada do método `TryInvokeMember` espera que o primeiro argumento seja um valor do enum `StringSearchOption` que você definiu na etapa anterior. O

método `TryInvokeMember` espera que o segundo argumento seja um valor booleano. Se um ou os dois argumentos forem valores válidos, eles serão passados para o método `GetPropertyValues` para recuperar os resultados.

C#

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                      object[] args,
                                      out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption =
(StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a
StringSearchOption enum value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean
value.");
    }

    result = GetPropertyValues(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

Salve e feche o arquivo.

## Criar um arquivo de texto de exemplo

No Gerenciador de Soluções, clique com o botão direito do mouse no projeto `DynamicSample` e selecione **Adicionar>Novo item**. No painel **Modelos Instalados**, selecione **Geral** e, então, selecione o modelo **Arquivo de Texto**. Deixe o nome padrão

de *TextFile1.txt* na caixa **Nome** e, em seguida, selecione **Adicionar**. Copie o seguinte texto para o arquivo *TextFile1.txt*.

```
text

List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

Salve e feche o arquivo.

## Criar um aplicativo de exemplo que usa o objeto dinâmico personalizado

Em **Gerenciador de Soluções**, clique duas vezes no arquivo *Program.cs*. Adicione o código a seguir ao procedimento `Main` para criar uma instância da classe `ReadOnlyFile` para o arquivo *TextFile1.txt*. O código usa associação tardia para chamar membros dinâmicos e recuperar linhas de texto que contêm a cadeia de caracteres "Customer".

```
C#

dynamic rFile = new ReadOnlyFile(@"..\..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("-----");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

Salve o arquivo e pressione `Ctrl+F5` para compilar e executar o aplicativo.

## Chamar uma biblioteca de linguagem dinâmica

O próximo passo a passo cria um projeto que acessa uma biblioteca escrita na linguagem dinâmica IronPython.

## Para criar uma classe dinâmica personalizada

No Visual Studio, selecione Arquivo > Novo > Projeto. Na caixa de diálogo Criar um novo projeto, selecione C#, selecione Aplicativo de Console e, em seguida, Avançar. Na caixa de diálogo Configurar novo projeto, digite `DynamicIronPythonSample` como Nome do projeto e selecione em Avançar. Na caixa de diálogo Informações adicionais, selecione .NET 7.0 (atual) como Estrutura de destino e, em seguida, selecione Criar. Instale o pacote NuGet do [IronPython](#). Edite o arquivo `Program.cs`. Na parte superior do arquivo, adicione o código a seguir para importar os namespaces `Microsoft.Scripting.Hosting` e `IronPython.Hosting` das bibliotecas do IronPython e o namespace `System.Linq`.

C#

```
using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

No método Main, adicione o código a seguir para criar um novo objeto `Microsoft.Scripting.Hosting.ScriptRuntime` para hospedar as bibliotecas do IronPython. O objeto `ScriptRuntime` carrega o módulo `random.py` da biblioteca do IronPython.

C#

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
    Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
    @"\IronPython 2.7\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

Após o código carregar o módulo `random.py`, adicione o seguinte código para criar uma matriz de inteiros. A matriz é passada para o método `shuffle` do módulo `random.py`, que classifica aleatoriamente os valores na matriz.

C#

```
// Initialize an enumerable set of integers.  
int[] items = Enumerable.Range(1, 7).ToArray();  
  
// Randomly shuffle the array of integers by using IronPython.  
for (int i = 0; i < 5; i++)  
{  
    random.shuffle(items);  
    foreach (int item in items)  
    {  
        Console.WriteLine(item);  
    }  
    Console.WriteLine("-----");  
}
```

Salve o arquivo e pressione **Ctrl+F5** para compilar e executar o aplicativo.

## Confira também

- [System.Dynamic](#)
- [System.Dynamic.DynamicObject](#)
- [Usando o tipo dynamic](#)
- [dinâmico](#)
- [Implementando interfaces dinâmicas \(PDF para download do Microsoft TechNet\)](#) ↗

# Controle de versão com as palavras-chave override e new (Guia de Programação em C#)

Artigo • 07/04/2023

A linguagem C# foi projetada para que o controle de versão entre classes derivadas e `base` em diferentes bibliotecas possa evoluir e manter a compatibilidade com versões anteriores. Isso significa, por exemplo, que a introdução de um novo membro em uma `classe` base com o mesmo nome que um membro em uma classe derivada tem suporte completo pelo C# e não leva a comportamento inesperado. Isso também significa que uma classe deve declarar explicitamente se um método destina-se a substituir um método herdado ou se um método é um novo método que oculta um método herdado de nome semelhante.

No C#, as classes derivadas podem conter métodos com o mesmo nome que os métodos da classe base.

- Se o método na classe derivada não for precedido pelas palavras-chave `new` ou `override`, o compilador emitirá um aviso e o método se comportará como se a palavra-chave `new` estivesse presente.
- Se o método na classe derivada for precedido pela palavra-chave `new`, o método será definido como sendo independente do método na classe base.
- Se o método na classe derivada for precedido pela palavra-chave `override`, os objetos da classe derivada chamarão esse método em vez do método da classe base.
- Para aplicar a palavra-chave `override` ao método na classe derivada, o método da classe base deve ser definido como `virtual`.
- O método da classe base pode ser chamado de dentro da classe derivada usando a palavra-chave `base`.
- As palavras-chave `override`, `virtual` e `new` também podem ser aplicadas a propriedades, indexadores e eventos.

Por padrão, os métodos C# não são virtuais. Se um método for declarado como `virtual`, qualquer classe que herdar o método pode implementar sua própria versão. Para tornar um método virtual, o modificador `virtual` é usado na declaração de método da classe base. A classe derivada pode, em seguida, substituir o método virtual base usando a

palavra-chave `override` ou ocultar o método virtual na classe base usando a palavra-chave `new`. Se nem a palavra-chave `override` nem a `new` for especificada, o compilador emitirá um aviso e o método na classe derivada ocultará o método na classe base.

Para demonstrar isso na prática, suponha por um momento que a Empresa A tenha criado uma classe chamada `GraphicsClass`, que seu programa usa. A seguir está

`GraphicsClass`:

```
C#  
  
class GraphicsClass  
{  
    public virtual void DrawLine() { }  
    public virtual void DrawPoint() { }  
}
```

Sua empresa usa essa classe e você a usa para derivar sua própria classe, adicionando um novo método:

```
C#  
  
class YourDerivedGraphicsClass : GraphicsClass  
{  
    public void DrawRectangle() { }  
}
```

Seu aplicativo é usado sem problemas, até a Empresa A lançar uma nova versão de `GraphicsClass`, que se parece com o seguinte código:

```
C#  
  
class GraphicsClass  
{  
    public virtual void DrawLine() { }  
    public virtual void DrawPoint() { }  
    public virtual void DrawRectangle() { }  
}
```

A nova versão de `GraphicsClass` agora contém um método chamado `DrawRectangle`. Inicialmente, nada ocorre. A nova versão ainda é compatível em relação ao binário com a versão antiga. Qualquer software que você implantou continuará a funcionar, mesmo se a nova classe for instalada nesses sistemas de computador. Todas as chamadas existentes para o método `DrawRectangle` continuarão a fazer referência à sua versão, em sua classe derivada.

No entanto, assim que você recompilar seu aplicativo usando a nova versão do `GraphicsClass`, você receberá um aviso do compilador, CS0108. Este aviso informa que você deve considerar como deseja que seu método `DrawRectangle` se comporte em seu aplicativo.

Se desejar que seu método substitua o novo método de classe base, use a palavra-chave `override`:

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
```

A palavra-chave `override` garante que todos os objetos derivados de `YourDerivedGraphicsClass` usarão a versão da classe derivada de `DrawRectangle`. Os objetos derivados de `YourDerivedGraphicsClass` ainda poderão acessar a versão da classe base de `DrawRectangle` usando a palavra-chave `base`:

C#

```
base.DrawRectangle();
```

Se você não quiser que seu método substitua o novo método de classe base, as seguintes considerações se aplicam. Para evitar confusão entre os dois métodos, você pode renomear seu método. Isso pode ser demorado e propenso a erros e simplesmente não ser prático em alguns casos. No entanto, se seu projeto for relativamente pequeno, você poderá usar opções de Refatoração do Visual Studio para renomear o método. Para obter mais informações, consulte [Refatorando classes e tipos \(Designer de Classe\)](#).

Como alternativa, você pode evitar o aviso usando a palavra-chave `new` na definição da classe derivada:

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
```

Usando a palavra-chave `new` informa ao compilador que sua definição oculta a definição que está contida na classe base. Esse é o comportamento padrão.

## Seleção de método e substituição

Quando um método é chamado em uma classe, o compilador C# seleciona o melhor método a ser chamado se mais de um método for compatível com a chamada, como quando há dois métodos com o mesmo nome e parâmetros que são compatíveis com o parâmetro passado. Os métodos a seguir seriam compatíveis:

```
C#  
  
public class Derived : Base  
{  
    public override void DoWork(int param) { }  
    public void DoWork(double param) { }  
}
```

Quando `DoWork` é chamado em uma instância do `Derived`, o compilador C# tenta primeiro tornar a chamada compatível com as versões do `DoWork` originalmente declarado em `Derived`. Os métodos de substituição não são considerados como declarados em uma classe, eles são novas implementações de um método declarado em uma classe base. Somente se o compilador C# não puder corresponder a chamada de método a um método original em `Derived` é que tentará corresponder a chamada a um método substituído com o mesmo nome e parâmetros compatíveis. Por exemplo:

```
C#  
  
int val = 5;  
Derived d = new Derived();  
d.DoWork(val); // Calls DoWork(double).
```

Como a variável `val` pode ser convertida para um duplo implicitamente, o compilador C# chama `DoWork(double)` em vez de `DoWork(int)`. Há duas formas de evitar isso. Primeiro, evite declarar novos métodos com o mesmo nome que os métodos virtuais. Segundo, você pode instruir o compilador C# para chamar o método virtual fazendo-o pesquisar a lista do método de classe base convertendo a instância do `Derived` para `Base`. Como o método é virtual, a implementação de `DoWork(int)` em `Derived` será chamada. Por exemplo:

```
C#
```

```
((Base)d).DoWork(val); // Calls DoWork(int) on Derived.
```

Para obter mais exemplos de `new` e `override`, consulte [Quando usar as palavras-chave override e new](#).

## Confira também

- [Guia de Programação em C#](#)
- [Sistema de tipos do C#](#)
- [Métodos](#)
- [Herança](#)

# Quando usar as palavras-chave override e new (Guia de Programação em C#)

Artigo • 07/04/2023

No C#, um método em uma classe derivada pode ter o mesmo nome que um método na classe base. É possível especificar a maneira como os métodos interagem usando as palavras-chave `new` e `override`. O modificador `override` estende o método `virtual` da classe base e o modificador `new` oculta um método de classe base acessível. A diferença é ilustrada nos exemplos deste tópico.

Em um aplicativo de console, declare as duas classes a seguir, `BaseClass` e `DerivedClass`. `DerivedClass` herda de `BaseClass`.

C#

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

No método `Main`, declare as variáveis `bc`, `dc` e `bcdc`.

- `bc` é do tipo `BaseClass` e seu valor é do tipo `BaseClass`.
- `dc` é do tipo `DerivedClass` e seu valor é do tipo `DerivedClass`.
- `bcdc` é do tipo `BaseClass` e seu valor é do tipo `DerivedClass`. Essa é a variável à qual você deve prestar atenção.

Como `bc` e `bcdc` têm o tipo `BaseClass`, eles podem ter acesso direto a `Method1`, a menos que você usa a conversão. A variável `dc` pode acessar `Method1` e `Method2`. Essas relações são mostradas no código a seguir.

C#

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Em seguida, adicione o seguinte método `Method2()` a `BaseClass`. A assinatura desse método corresponde à assinatura do método `Method2` em `DerivedClass`.

C#

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Como `BaseClass` agora tem um método `Method2`, uma segunda instrução de chamada pode ser adicionada para variáveis de `BaseClass` `bc` e `bcdc`, conforme mostrado no código a seguir.

C#

```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

Quando você compilar o projeto, verá que a adição do método `Method2` gera um aviso `BaseClass`. O aviso informa que o método `Method2` em `DerivedClass` oculta o método `Method2` em `BaseClass`. É recomendável usar a palavra-chave `new` na definição `Method2`

se você pretende gerar esse resultado. Como alternativa, seria possível renomear um dos métodos `Method2` para resolver o aviso, mas isso nem sempre é prático.

Antes de adicionar `new`, execute o programa para ver a saída produzida pelas outras instruções de chamada. Os seguintes resultados são exibidos.

C#

```
// Output:  
// Base - Method1  
// Base - Method2  
// Base - Method1  
// Derived - Method2  
// Base - Method1  
// Base - Method2
```

A palavra-chave `new` preserva as relações que produzem essa saída, mas suprime o aviso. As variáveis que têm o tipo `BaseClass` continuam acessando os membros de `BaseClass` e a variável que tem o tipo `DerivedClass` continua acessando os membros em `DerivedClass` primeiro e, em seguida, considera os membros herdados de `BaseClass`.

Para suprimir o aviso, adicione o modificador `new` para a definição de `Method2` em `DerivedClass`, conforme mostrado no código a seguir. O modificador pode ser adicionado antes ou depois de `public`.

C#

```
public new void Method2()  
{  
    Console.WriteLine("Derived - Method2");  
}
```

Execute o programa novamente para verificar se a saída não foi alterada. Verifique também se o aviso não é mais exibido. Usando `new`, você está declarando que está ciente de que o membro que ele modifica oculta um membro herdado da classe base. Para obter mais informações sobre a ocultação de nome por meio de herança, consulte [Novo modificador](#).

Para comparar esse comportamento com os efeitos de usar `override`, adicione o seguinte método a `DerivedClass`. O modificador `override` pode ser adicionado antes ou depois de `public`.

C#

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Adicione o modificador `virtual` à definição de `Method1` em `BaseClass`. O modificador `virtual` pode ser adicionado antes ou depois de `public`.

C#

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Execute o projeto novamente. Observe principalmente as duas últimas linhas da saída a seguir.

C#

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

O uso do modificador `override` permite que `bcdc` acesse o método `Method1` definido em `DerivedClass`. Normalmente, esse é o comportamento desejado em hierarquias de herança. Você quer objetos com valores criados da classe derivada para usar os métodos definidos na classe derivada. Obtenha esse comportamento usando `override` para estender o método da classe base.

O código a seguir contém o exemplo completo.

C#

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
```

```

BaseClass bc = new BaseClass();
DerivedClass dc = new DerivedClass();
BaseClass bcdc = new DerivedClass();

// The following two calls do what you would expect. They call
// the methods that are defined in BaseClass.
bc.Method1();
bc.Method2();
// Output:
// Base - Method1
// Base - Method2

// The following two calls do what you would expect. They call
// the methods that are defined in DerivedClass.
dc.Method1();
dc.Method2();
// Output:
// Derived - Method1
// Derived - Method2

// The following two calls produce different results, depending
// on whether override (Method1) or new (Method2) is used.
bcdc.Method1();
bcdc.Method2();
// Output:
// Derived - Method1
// Base - Method2
}

}

class BaseClass
{
    public virtual void Method1()
    {
        Console.WriteLine("Base - Method1");
    }

    public virtual void Method2()
    {
        Console.WriteLine("Base - Method2");
    }
}

class DerivedClass : BaseClass
{
    public override void Method1()
    {
        Console.WriteLine("Derived - Method1");
    }

    public new void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}

```

```
    }  
}
```

O exemplo a seguir ilustra um comportamento semelhante em um contexto diferente. O exemplo define três classes: uma classe base chamada `Car` e duas classes derivadas dela, `ConvertibleCar` e `Minivan`. A classe base contém um método `DescribeCar`. O método exibe uma descrição básica de um carro e, em seguida, chama `ShowDetails` para fornecer mais informações. Cada uma das três classes define um método `ShowDetails`. O modificador `new` é usado para definir `ShowDetails` na classe `ConvertibleCar`. O modificador `override` é usado para definir `ShowDetails` na classe `Minivan`.

C#

```
// Define the base class, Car. The class defines two methods,  
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each  
derived  
// class also defines a ShowDetails method. The example tests which version  
of  
// ShowDetails is selected, the base class method or the derived class  
method.  
class Car  
{  
    public void DescribeCar()  
    {  
        System.Console.WriteLine("Four wheels and an engine.");  
        ShowDetails();  
    }  
  
    public virtual void ShowDetails()  
    {  
        System.Console.WriteLine("Standard transportation.");  
    }  
}  
  
// Define the derived classes.  
  
// Class ConvertibleCar uses the new modifier to acknowledge that  
ShowDetails  
// hides the base class method.  
class ConvertibleCar : Car  
{  
    public new void ShowDetails()  
    {  
        System.Console.WriteLine("A roof that opens up.");  
    }  
}  
  
// Class Minivan uses the override modifier to specify that ShowDetails  
// extends the base class method.
```

```
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

O exemplo testa qual versão do `ShowDetails` é chamada. O método a seguir, `TestCars1`, declara uma instância de cada classe e, em seguida, chama `DescribeCar` em cada instância.

C#

```
public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("-----");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("-----");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("-----");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("-----");
}
```

`TestCars1` produz a saída a seguir. Observe principalmente os resultados para `car2`, que provavelmente não são o que você espera. O tipo do objeto é `ConvertibleCar`, mas `DescribeCar` não acessa a versão de `ShowDetails` definida na classe `ConvertibleCar`, porque esse método é declarado com o modificador `new`, e não com o modificador `override`. Em decorrência disso, um objeto `ConvertibleCar` exibe a mesma descrição que um objeto `Car`. Compare os resultados de `car3`, que é um objeto `Minivan`. Nesse caso, o método `ShowDetails` declarado na classe `Minivan` substitui o método `ShowDetails` declarado na classe `Car` e a descrição exibida descreve uma minivan.

C#

```
// TestCars1
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Carries seven people.
// -----
```

`TestCars2` cria uma lista de objetos que têm tipo `Car`. Os valores dos objetos são instanciados com base nas classes `Car`, `ConvertibleCar` e `Minivan`. `DescribeCar` é chamado em cada elemento da lista. O código a seguir mostra a definição de `TestCars2`.

C#

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("-----");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("-----");
    }
}
```

É exibida a saída a seguir. Observe que é a mesma que a saída exibida por `TestCars1`. O método `ShowDetails` da classe `ConvertibleCar` não é chamado, independentemente se o tipo do objeto é `ConvertibleCar`, como em `TestCars1` ou `Car`, como em `TestCars2`. Por outro lado, `car3` chama o método `ShowDetails` com base na classe `Minivan` nos dois casos, tendo ele o tipo `Minivan` ou o tipo `Car`.

C#

```
// TestCars2
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
```

```
// Four wheels and an engine.  
// Standard transportation.  
// -----  
// Four wheels and an engine.  
// Carries seven people.  
// -----
```

Os métodos `TestCars3` e `TestCars4` completam o exemplo. Esses métodos chamam `ShowDetails` diretamente, primeiro com base nos objetos declarados para ter o tipo `ConvertibleCar` e `Minivan` (`TestCars3`), em seguida, com base nos objetos declarados para ter o tipo `Car` (`TestCars4`). O código a seguir define esses dois métodos.

C#

```
public static void TestCars3()  
{  
    System.Console.WriteLine("\nTestCars3");  
    System.Console.WriteLine("-----");  
    ConvertibleCar car2 = new ConvertibleCar();  
    Minivan car3 = new Minivan();  
    car2.ShowDetails();  
    car3.ShowDetails();  
}  
  
public static void TestCars4()  
{  
    System.Console.WriteLine("\nTestCars4");  
    System.Console.WriteLine("-----");  
    Car car2 = new ConvertibleCar();  
    Car car3 = new Minivan();  
    car2.ShowDetails();  
    car3.ShowDetails();  
}
```

Os métodos produzem a saída a seguir, que corresponde aos resultados do primeiro exemplo neste tópico.

C#

```
// TestCars3  
// -----  
// A roof that opens up.  
// Carries seven people.  
  
// TestCars4  
// -----  
// Standard transportation.  
// Carries seven people.
```

O código a seguir mostra o projeto completo e sua saída.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("-----");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("-----");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("-----");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("-----");
        }
        // Output:
        // TestCars1
```

```
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Carries seven people.
// -----



public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("-----");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("-----");
    }
}

// Output:
// TestCars2
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Carries seven people.
// -----



public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("-----");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

// Output:
// TestCars3
// -----
// A roof that opens up.
// Carries seven people.



public static void TestCars4()
{
```

```
        System.Console.WriteLine("\nTestCars4");
        System.Console.WriteLine("-----");
        Car car2 = new ConvertibleCar();
        Car car3 = new Minivan();
        car2.ShowDetails();
        car3.ShowDetails();
    }
    // Output:
    // TestCars4
    // -----
    // Standard transportation.
    // Carries seven people.
}

// Define the base class, Car. The class defines two virtual methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each
derived
    // class also defines a ShowDetails method. The example tests which
version of
    // ShowDetails is used, the base class method or the derived class
method.

class Car
{
    public virtual void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that
ShowDetails
    // hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

```
        }  
    }  
  
}
```

## Confira também

- [Guia de Programação em C#](#)
- [Sistema de tipos do C#](#)
- [Controle de versão com as palavras-chave override e new](#)
- [base](#)
- [abstract](#)

# Como substituir o método `ToString` (Guia de Programação em C#)

Artigo • 07/04/2023

Cada classe ou struct no C# herda implicitamente a classe `Object`. Portanto, cada objeto no C# obtém o método `ToString`, que retorna uma representação de cadeia de caracteres desse objeto. Por exemplo, todas as variáveis do tipo `int` tem um método `ToString`, que permite retornar seus conteúdos como uma cadeia de caracteres:

C#

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

Ao criar uma classe ou struct personalizada, é necessário substituir o método `ToString` a fim de fornecer informações sobre o tipo ao código cliente.

Para obter informações sobre como usar cadeias de caracteres de formato e outros tipos de formatação personalizada com o método `ToString`, consulte [Tipos de Formatação](#).

## ⓘ Importante

Ao decidir quais informações devem ser fornecidas por meio desse método, considere se a classe ou struct será utilizado por código não confiável. Assegure-se de que nenhuma informação que possa ser explorada por código mal-intencionado seja fornecida.

Substituir o método `ToString` na classe ou struct:

1. Declare um método `ToString` com os seguintes modificadores e tipo retornado:

C#

```
public override string ToString(){}  
  
```

2. Implemente o método para que ele retorne uma cadeia de caracteres.

O exemplo a seguir retorna o nome da classe, além dos dados específicos de uma instância particular da classe.

```
C#  
  
class Person  
{  
    public string Name { get; set; }  
    public int Age { get; set; }  
  
    public override string ToString()  
    {  
        return "Person: " + Name + " " + Age;  
    }  
}
```

É possível testar o método `ToString`, conforme mostrado no exemplo de código a seguir:

```
C#  
  
Person person = new Person { Name = "John", Age = 12 };  
Console.WriteLine(person);  
// Output:  
// Person: John 12
```

## Confira também

- [IFormattable](#)
- [Guia de Programação em C#](#)
- [O sistema de tipos C#](#)
- [Cadeias de caracteres](#)
- [cadeia de caracteres](#)
- [override](#)
- [virtual](#)
- [Formatar tipos](#)

# Membros (Guia de Programação em C#)

Artigo • 07/04/2023

Classes e structs têm membros que representam seus dados e comportamento. Os membros de uma classe incluem todos os membros declarados na classe, juntamente com todos os membros (exceto construtores e finalizadores) declarados em todas as classes em sua hierarquia de herança. Os membros privados em classes base são herdados, mas não podem ser acessados de classes derivadas.

A tabela a seguir lista os tipos de membros que uma classe ou struct pode conter:

Membro	Descrição
Fields	Os campos são variáveis declaradas no escopo da classe. Um campo pode ser um tipo numérico interno ou uma instância de outra classe. Por exemplo, uma classe de calendário pode ter um campo que contém a data atual.
Constantes	Constantes são campos cujo valor é definido em tempo de compilação e não pode ser alterado.
Propriedades	As propriedades são métodos de uma classe acessados como se fossem campos dessa classe. Uma propriedade pode fornecer proteção para um campo de classe para evitar que ele seja alterado sem o conhecimento do objeto.
Métodos	Os métodos definem as ações que uma classe pode executar. Métodos podem usar parâmetros que fornecem dados de entrada e retornar dados de saída por meio de parâmetros. Os métodos também podem retornar um valor diretamente, sem usar um parâmetro.
Eventos	Os eventos fornecem notificações sobre ocorrências a outros objetos, como cliques de botão ou a conclusão bem-sucedida de um método. Eventos são definidos e disparados pelos delegados.
Operadores	Os operadores sobrecarregados são considerados membros de tipo. Ao sobrecarregar um operador, ele é definido como um método estático público em um tipo. Para obter mais informações, consulte <a href="#">Sobrecarga de operador</a> .
Indexadores	Os indexadores permitem que um objeto seja indexado de maneira semelhante às matrizes.
Construtores	Os construtores são os métodos chamados quando o objeto é criado pela primeira vez. Geralmente, eles são usados para inicializar os dados de um objeto.
Finalizadores	Os finalizadores raramente são usados no C#. Eles são métodos chamados pelo mecanismo de runtime quando o objeto está prestes a ser removido da memória. Geralmente, eles são usados para garantir que recursos que devem ser liberados sejam manipulados corretamente.

Membro	Descrição
Tipos aninhados	Os tipos aninhados são tipos declarados dentro de outro tipo. Geralmente, eles são usados para descrever objetos utilizados somente pelos tipos que os contêm.

## Confira também

- [Guia de Programação em C#](#)
- [Classes](#)

# Classes e membros de classes abstract e sealed (Guia de Programação em C#)

Artigo • 07/04/2023

A palavra-chave `abstract` permite que você crie classes e membros de `classe` que estão incompletos e devem ser implementados em uma classe derivada.

A palavra-chave `sealed` permite evitar a herança de uma classe ou de determinados membros de classe que foram marcados anteriormente com `virtual`.

## Classes abstratas e membros de classe

As classes podem ser declaradas como abstratas, colocando a palavra-chave `abstract` antes da definição de classe. Por exemplo:

```
C#  
  
public abstract class A  
{  
    // Class members here.  
}
```

Uma classe abstrata não pode ser instanciada. A finalidade de uma classe abstrata é fornecer uma definição comum de uma classe base que pode ser compartilhada por várias classes derivadas. Por exemplo, uma biblioteca de classes pode definir uma classe abstrata que serve como um parâmetro para muitas de suas funções e exige que os programadores que usam essa biblioteca forneçam sua própria implementação da classe, criando uma classe derivada.

As classes abstratas também podem definir métodos abstratos. Isso é realizado através da adição da palavra-chave `abstract` antes do tipo de retorno do método. Por exemplo:

```
C#  
  
public abstract class A  
{  
    public abstract void DoWork(int i);  
}
```

Os métodos abstratos não têm implementação, portanto, a definição do método é seguida por um ponto e vírgula, em vez de um bloco de método normal. As classes

derivadas da classe abstrata devem implementar todos os métodos abstratos. Quando uma classe abstrata herda um método virtual de uma classe base, a classe abstrata pode substituir o método virtual por um método abstrato. Por exemplo:

```
C#  
  
// compile with: -target:library  
public class D  
{  
    public virtual void DoWork(int i)  
    {  
        // Original implementation.  
    }  
}  
  
public abstract class E : D  
{  
    public abstract override void DoWork(int i);  
}  
  
public class F : E  
{  
    public override void DoWork(int i)  
    {  
        // New implementation.  
    }  
}
```

Se um método `virtual` for declarado `abstract`, ele ainda será virtual para qualquer classe que herdar da classe abstrata. Uma classe que herda um método abstrato não pode acessar a implementação original do método. No exemplo anterior, `DoWork` na classe F não pode chamar `DoWork` na classe D. Dessa forma, uma classe abstrata pode forçar classes derivadas a fornecerem novas implementações de método para métodos virtuais.

## Classes e membros de classes sealed

As classes podem ser declaradas como `sealed`, colocando a palavra-chave `sealed` antes da definição de classe. Por exemplo:

```
C#  
  
public sealed class D  
{  
    // Class members here.  
}
```

Uma classe sealed não pode ser usada como uma classe base. Por esse motivo, também não pode ser uma classe abstrata. As classes sealed impedem a derivação. Como elas nunca podem ser usadas como uma classe base, algumas otimizações em tempo de execução podem tornar a chamada a membros de classe sealed ligeiramente mais rápida.

Um método, um indexador, uma propriedade ou um evento em uma classe derivada que está substituindo um membro virtual da classe base, pode declarar esse membro como sealed. Isso anula o aspecto virtual do membro para qualquer outra classe derivada. Isso é realizado através da colocação da palavra-chave `sealed` antes da palavra-chave `override` na declaração de membro de classe. Por exemplo:

C#

```
public class D : C
{
    public sealed override void DoWork() { }
```

## Confira também

- [Guia de Programação em C#](#)
- [O sistema de tipos C#](#)
- [Herança](#)
- [Métodos](#)
- [Fields](#)
- [Como definir propriedades abstract](#)

# Classes static e membros de classes static (Guia de Programação em C#)

Artigo • 17/03/2023

Uma classe `static` é basicamente o mesmo que uma classe não estática, mas há uma diferença: uma classe estática não pode ser instanciada. Em outras palavras, você não pode usar o operador `new` para criar uma variável do tipo de classe. Como não há nenhuma variável de instância, você acessa os membros de uma classe estática usando o próprio nome da classe. Por exemplo, se houver uma classe estática chamada `UtilityClass` com um método público chamado `MethodA`, chame o método, como mostra o exemplo a seguir:

C#

```
UtilityClass.MethodA();
```

Uma classe estática pode ser usada como um contêiner conveniente para conjuntos de métodos que operam apenas em parâmetros de entrada e não precisam obter ou definir campos de instância internos. Por exemplo, na biblioteca de classes .NET, a classe estática `System.Math` contém métodos que executam operações matemáticas, sem a necessidade de armazenar ou recuperar dados que são exclusivos de uma determinada instância da classe `Math`. Ou seja, você aplica os membros da classe especificando o nome da classe e o nome do método, conforme mostrado no exemplo a seguir.

C#

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

Como é o caso com todos os tipos de classe, as informações de tipo de uma classe estática são carregadas pelo runtime do .NET quando o programa que faz referência à classe é carregado. O programa não pode especificar exatamente quando a classe é carregada. No entanto, é garantido que ela será carregada e terá seus campos inicializados e seu construtor estático chamado antes que a classe seja referenciada pela primeira vez em seu programa. Um construtor estático é chamado apenas uma vez e

uma classe estática permanece na memória pelo tempo de vida do domínio do aplicativo em que seu programa reside.

### ① Observação

Para criar uma classe não estática que permite que apenas uma instância de si mesma seja criada, consulte [Implementando singleton no C#](#).

A lista a seguir fornece os principais recursos de uma classe estática:

- Contém apenas membros estáticos.
- Não pode ser instanciada.
- É lacrada.
- Não pode conter [Construtores de instância](#).

Criar uma classe estática é, portanto, basicamente o mesmo que criar uma classe que contém apenas membros estáticos e um construtor particular. Um construtor particular impede que a classe seja instanciada. A vantagem de usar uma classe estática é que o compilador pode verificar se nenhum membro de instância foi adicionado acidentalmente. O compilador garantirá que as instâncias dessa classe não possam ser criadas.

Classes estáticas são lacradas e, portanto, não podem ser herdadas. Eles não podem herdar de nenhuma classe ou interface, exceto [Object](#). Classes estáticas não podem conter um construtor de instância. No entanto, eles podem conter um construtor estático. Classes não estáticas também devem definir um construtor estático se a classe contiver membros estáticos que exigem inicialização não trivial. Para obter mais informações, consulte [Construtores estáticos](#).

## Exemplo

Temos aqui um exemplo de uma classe estática que contém dois métodos que convertem a temperatura de Celsius em Fahrenheit e de Fahrenheit em Celsius:

C#

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
```

```

        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string? selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F =
TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine() ?? "0");
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C =
TemperatureConverter.FahrenheitToCelsius(Console.ReadLine() ?? "0");
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
    }
}

```

```
        Console.ReadKey();
    }
}

/* Example Output:
Please select the convertor direction
1. From Celsius to Fahrenheit.
2. From Fahrenheit to Celsius.
:2
Please enter the Fahrenheit temperature: 20
Temperature in Celsius: -6.67
Press any key to exit.
*/
```

## Membros Estáticos

Uma classe não estática não pode conter métodos, campos, propriedades ou eventos estáticos. O membro estático pode ser chamado em uma classe, mesmo quando nenhuma instância da classe foi criada. O membro estático sempre é acessado pelo nome de classe, não pelo nome da instância. Existe apenas uma cópia de um membro estático, independentemente de quantas instâncias da classe forem criadas.

Propriedades e métodos estáticos não podem acessar eventos e campos não estáticos no tipo que os contêm e não podem acessar uma variável de instância de nenhum objeto, a menos que ele seja passado explicitamente em um parâmetro de método.

É mais comum declarar uma classe não estática com alguns membros estáticos do que declarar uma classe inteira como estática. Dois usos comuns dos campos estáticos são manter uma contagem do número de objetos que foram instanciados ou armazenar um valor que deve ser compartilhado entre todas as instâncias.

Métodos estáticos podem ser sobre carregados, mas não substituídos, porque pertencem à classe e não a qualquer instância da classe.

Embora um campo não possa ser declarado como `static const`, um campo `const` é essencialmente estático em seu comportamento. Ele pertence ao tipo e não a instâncias do tipo. Portanto, campos `const` podem ser acessados usando a mesma notação `ClassName.MemberName` usada para campos estáticos. Nenhuma instância de objeto é necessária.

O C# não dá suporte a variáveis locais estáticas (variáveis que são declaradas no escopo do método).

Você declara membros de classe estática usando a palavra-chave `static` antes do tipo de retorno do membro, conforme mostrado no exemplo a seguir:

C#

```
public class Automobile
{
    public static int NumberOfWheels = 4;

    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }

    public static void Drive() { }

    public static event EventType? RunOutOfGas;

    // Other non-static fields and properties...
}
```

Membros estáticos são inicializados antes que o membro estático seja acessado pela primeira vez e antes que o construtor estático, se houver, seja chamado. Para acessar um membro de classe estática, use o nome da classe em vez de um nome de variável para especificar o local do membro, conforme mostrado no exemplo a seguir:

C#

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

Se sua classe contiver campos estáticos, forneça um construtor estático que os inicializa quando a classe é carregada.

Uma chamada para um método estático gera uma instrução de chamada em MSIL (Microsoft Intermediate Language), enquanto uma chamada para um método de instância gera uma instrução `callvirt`, que também verifica se há referências de objeto nulas. No entanto, na maioria das vezes, a diferença de desempenho entre os dois não é significativa.

## Especificação da Linguagem C#

Para saber mais, confira [Classes estáticas, Membros estáticos e de instância](#) e [Construtores estáticos](#) na [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Guia de Programação em C#](#)
- [static](#)
- [Classes](#)
- [class](#)
- [Construtores estáticos](#)
- [Construtores de instância](#)

# Modificadores de acesso (Guia de Programação em C#)

Artigo • 07/04/2023

Todos os tipos e membros de tipo têm um nível de acessibilidade. O nível de acessibilidade controla se eles podem ser usados em outro código no assembly ou em outros assemblies. Um **assembly** é um *.dll* ou *.exe* criado ao selecionar um ou mais arquivos *.cs* em uma única compilação. Use os modificadores de acesso a seguir para especificar a acessibilidade de um tipo ou membro quando você o declarar:

- **public**: o tipo ou membro pode ser acessado por qualquer outro código no mesmo assembly ou em outro assembly que faz referência a ele. O nível de acessibilidade de membros públicos de um tipo é controlado pelo nível de acessibilidade do próprio tipo.
- **private**: o tipo ou membro pode ser acessado somente pelo código na mesma `class` ou `struct`.
- **protected**: o tipo ou membro pode ser acessado somente pelo código na mesma `class` ou em uma `class` derivada dessa `class`.
- **internal**: o tipo ou membro pode ser acessado por qualquer código no mesmo assembly, mas não de outro assembly. Em outras palavras, tipos ou membros `internal` podem ser acessados no código que faz parte da mesma compilação.
- **protected internal**: o tipo ou membro pode ser acessado por qualquer código no assembly no qual ele é declarado ou de uma `class` derivada em outro assembly.
- **private protected**: o tipo ou membro pode ser acessado por tipos derivados do `class`, que são declarados no assembly relativo.

## Tabela de resumo

Local do chamador	public	protected internal	protected	internal	private protected	private
Dentro da classe	✓	✓	✓	✓	✓	✓
Classe derivada (mesmo assembly)	✓	✓	✓	✓	✓	✗
Classe não derivada (mesmo assembly)	✓	✓	✗	✓	✗	✗
Classe derivada (assembly diferente)	✓	✓	✓	✗	✗	✗

Local do chamador	public	protected internal	protected	internal	private	private protected
Classe não derivada (assembly diferente)	✓	✗	✗	✗	✗	✗

Os exemplos a seguir demonstram como especificar modificadores de acesso em um tipo e membro:

C#

```
public class Bicycle
{
    public void Pedal() { }
```

Nem todos os modificadores de acesso são válidos para todos os tipos ou membros em todos os contextos. Em alguns casos, a acessibilidade de um membro de tipo é restrita pela acessibilidade do tipo de conteúdo.

## Acessibilidade de classe, registro e struct

As classes, os registros e os structs declarados em um namespace (em outras palavras, que não estão aninhadas em outras classes ou structs) podem ser `public` ou `internal`. `internal` é o padrão, se nenhum modificador de acesso for especificado.

Os membros de struct, incluindo classes e structs aninhados, podem ser declarados como `public`, `internal` ou `private`. Os membros de classe, incluindo classes e structs aninhados, podem ser `public`, `protected internal`, `protected`, `internal`, `private protected` ou `private`. Os membros de classe e struct, incluindo classes e structs aninhados, têm acesso `private` por padrão. Tipos aninhados privados não são acessíveis de fora do tipo relativo.

As classes e registros derivados não podem ter uma acessibilidade maior do que os tipos base. Você não pode declarar uma classe pública `B` derivada de uma classe interna `A`. Se permitido, teria o efeito de tornar `A` público, pois todos os membros `protected` ou `internal` de `A` são acessíveis na classe derivada.

Você pode permitir que outros assemblies específicos acessem os tipos internos usando o `InternalsVisibleToAttribute`. Para obter mais informações, consulte [Assemblies amigáveis](#).

# Acessibilidade de membro de classe, registro e struct

Os membros de classe e de registro (incluindo as classes, os registros e os structs aninhados) podem ser declarados com qualquer um dos seis tipos de acesso. Os membros de struct não podem ser declarados como `protected`, `protected internal` ou `private protected`, pois os structs não permitem herança.

Normalmente, a acessibilidade de um membro não é maior que a acessibilidade do tipo que o contém. No entanto, um membro `public` de uma classe interna pode ser acessível de fora do assembly se o membro implementa os métodos de interface ou substitui os métodos virtuais que são definidos em uma classe base pública.

O tipo de qualquer campo, propriedade ou evento do membro deve ser pelo menos tão acessível quanto o próprio membro. Da mesma forma, o tipo de retorno e os tipos de parâmetro de qualquer método, indexador ou delegado devem ser pelo menos tão acessíveis quanto o próprio membro. Por exemplo, você não pode ter um método `public`, `M`, que retorna uma classe `C`, a menos que `C` também seja `public`. Da mesma forma, você não pode ter uma propriedade `protected` do tipo `A`, se `A` for declarada como `private`.

Os operadores definidos pelo usuário sempre devem ser declarados como `public` e `static`. Para obter mais informações, consulte [Sobrecarga de operador](#).

Os finalizadores não podem ter modificadores de acessibilidade.

Para definir o nível de acesso para um membro de `class`, `record` ou `struct`, adicione a palavra-chave apropriada à declaração do membro, como mostrado o exemplo a seguir.

C#

```
// public class:  
public class Tricycle  
{  
    // protected method:  
    protected void Pedal() { }  
  
    // private field:  
    private int _wheels = 3;  
  
    // protected internal property:  
    protected internal int Wheels  
    {  
        get { return _wheels; }  
    }  
}
```

```
}
```

## Outros tipos

As interfaces declaradas diretamente em um namespace podem ser `public` ou `internal` e, assim como classes e structs, o padrão das interfaces é o acesso `internal`. Membros de interface são `public` por padrão, pois a finalidade de uma interface é permitir que outros tipos acessem uma classe ou um struct. As declarações de membro da interface podem incluir qualquer modificador de acesso. Isso é mais útil para que os métodos estáticos forneçam as implementações comuns necessárias para todos os implementadores de uma classe.

Membros de enumeração sempre são `public` e nenhum modificador de acesso pode ser aplicado.

Delegados se comportam como classes e structs. Por padrão, eles têm acesso `internal` quando declarados diretamente dentro de um namespace e acesso `private` quando aninhados.

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Especificar a ordem do modificador \(regra de estilo IDE0036\)](#)
- [Guia de Programação em C#](#)
- [Sistema de tipos do C#](#)
- [Interfaces](#)
- [private](#)
- [público](#)
- [interno](#)
- [protected](#)
- [internos protegidos](#)
- [privado protegido](#)
- [class](#)
- [struct](#)
- [interface](#)



# Campos (Guia de Programação em C#)

Artigo • 02/06/2023

Um *campo* é uma variável de qualquer tipo que é declarada diretamente em uma [classe](#) ou [struct](#). Os campos são *membros* do tipo que os contém.

Uma classe ou um struct podem ter campos de instância, estáticos ou ambos. Os campos de instância são específicos a uma instância de um tipo. Se você tem uma classe `T`, com um campo de instância `F`, você pode criar dois objetos do tipo `T` e modificar o valor de `F` em cada objeto sem afetar o valor no outro objeto. Por outro lado, um campo estático pertence ao próprio tipo e é compartilhado entre todas as instâncias desse tipo. Você só pode acessar o campo estático usando o nome do tipo. Se você acessar o campo estático por meio de um nome de instância, receberá o erro em tempo de compilação [CS0176](#).

Em geral, você deve declarar a acessibilidade `private` ou `protected` para campos. Os dados que o tipo expõe para o código de cliente devem ser fornecidos por meio de [métodos](#), [propriedades](#) e [indexadores](#). Usando esses constructos para acesso indireto aos campos internos, você pode proteger contra valores de entrada inválidos. Um campo particular que armazena os dados expostos por uma propriedade pública é chamado de *repositório de backup* ou de *campo de suporte*. Você pode declarar campos `public`, mas não pode impedir que o código que usa seu tipo defina esse campo como um valor inválido ou altere os dados de um objeto.

Os campos normalmente armazenam os dados que devem estar acessíveis a mais de um método de tipo e devem ser armazenados por mais tempo que o tempo de vida de qualquer método único. Por exemplo, um tipo que representa uma data do calendário pode ter três campos de inteiros: um para o mês, um para o dia e outro para o ano. As variáveis que não são usadas fora do escopo de um método único devem ser declaradas como *variáveis locais* dentro do próprio corpo do método.

Os campos são declarados no bloco da classe ou do struct, especificando o nível de acesso, seguido pelo tipo e pelo nome do campo. Por exemplo:

C#

```
public class CalendarEntry
{
    // private field (Located near wrapping "Date" property).
    private DateTime _date;

    // Public property exposes _date field safely.
```

```
public DateTime Date
{
    get
    {
        return _date;
    }
    set
    {
        // Set some reasonable boundaries for likely birth dates.
        if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
        {
            _date = value;
        }
        else
        {
            throw new ArgumentOutOfRangeException("Date");
        }
    }
}

// public field (Generally not recommended).
public string? Day;

// Public method also exposes _date field safely.
// Example call: birthday.SetDate("1975, 6, 30");
public void SetDate(string dateString)
{
    DateTime dt = Convert.ToDateTime(dateString);

    // Set some reasonable boundaries for likely birth dates.
    if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
    {
        _date = dt;
    }
    else
    {
        throw new ArgumentOutOfRangeException("dateString");
    }
}

public TimeSpan GetTimeSpan(string dateString)
{
    DateTime dt = Convert.ToDateTime(dateString);

    if (dt.Ticks < _date.Ticks)
    {
        return _date - dt;
    }
    else
    {
        throw new ArgumentOutOfRangeException("dateString");
    }
}
```

Para acessar um campo em uma instância, adicione um ponto após o nome da instância, seguido pelo nome do campo, como em `instancename._fieldName`. Por exemplo:

C#

```
CalendarEntry birthday = new CalendarEntry();
birthday.Day = "Saturday";
```

Um campo pode receber um valor inicial, usando o operador de atribuição quando o campo é declarado. Para atribuir automaticamente o campo `Day` ao `"Monday"`, por exemplo, você poderia declarar `Day` como no exemplo a seguir:

C#

```
public class CalendarDateWithInitialization
{
    public string Day = "Monday";
    //...
}
```

Os campos são inicializados imediatamente antes do construtor para a instância do objeto ser chamado. Se o construtor atribuir o valor de um campo, ele substituirá qualquer valor fornecido durante a declaração do campo. Para obter mais informações, veja [Usando construtores](#).

### ⓘ Observação

Um inicializador de campo não pode fazer referência a outros campos de instância.

Os campos podem ser marcados como `public`, `private`, `protected`, `internal`, `protected internal` ou `private protected`. Esses modificadores de acesso definem como os usuários do tipo podem acessar os campos. Para obter mais informações, consulte [Modificadores de Acesso](#).

Opcionalmente, um campo pode ser declarado `static`. Os campos estáticos estão disponíveis para chamadores a qualquer momento, mesmo se não existir nenhuma instância do tipo. Para obter mais informações, consulte [Classes estáticas e membros de classes estáticas](#).

Um campo pode ser declarado `readonly`. Um valor só pode ser atribuído a um campo somente leitura durante a inicialização ou em um construtor. Um campo `static readonly` é semelhante a uma constante, exceto que o compilador C# não tem acesso

ao valor de um campo somente leitura estático em tempo de compilação, mas somente em tempo de execução. Para obter mais informações, consulte [Constantes](#).

Um campo pode ser declarado [required](#). Um campo obrigatório precisa ser inicializado pelo construtor ou por um [inicializador de objeto](#) quando um objeto é criado. Adicione o atributo [System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute](#) a qualquer declaração de construtor que initialize todos os membros necessários.

O modificador `required` não pode ser combinado com o modificador `readonly` no mesmo campo. No entanto, a [propriedade](#) pode ser `required` e `init` somente.

A partir do C# 12, os parâmetros do [construtor primário](#) são uma alternativa para declarar campos. Quando seu tipo tem dependências que devem ser fornecidas na inicialização, você pode criar um construtor primário que fornece essas dependências. Esses parâmetros podem ser capturados e usados no lugar de campos declarados em seus tipos. No caso de tipos [record](#), os parâmetros do construtor primário são exibidos como propriedades públicas.

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Guia de Programação em C#](#)
- [Sistema de tipos do C#](#)
- [Usando construtores](#)
- [Herança](#)
- [Modificadores de acesso](#)
- [Classes e membros de classes abstract e sealed](#)

# Constantes (Guia de Programação em C#)

Artigo • 07/04/2023

As constantes são valores imutáveis que são conhecidos no tempo de compilação e não são alterados durante a vida útil do programa. Constantes são declaradas com o modificador `const`. Apenas os [tipos internos](#) do C# (excluindo `System.Object`) podem ser declarados como `const`. Tipos definidos pelo usuário, incluindo classes, struct e matrizes, não podem ser `const`. Use o modificador `readonly` para criar uma classe, um struct ou uma matriz que sejam inicializados uma vez em tempo de execução (por exemplo, em um construtor) e não possam mais ser alterados depois disso.

O C# não dá suporte aos métodos `const`, propriedades ou eventos.

O tipo de enumeração permite que você defina constantes nomeadas para tipos internos inteiros (por exemplo `int`, `uint`, `long` e assim por diante). Para obter mais informações, consulte [enum](#).

As constantes devem ser inicializadas conforme elas são declaradas. Por exemplo:

```
C#  
  
class Calendar1  
{  
    public const int Months = 12;  
}
```

Neste exemplo, a constante `Months` sempre é 12 e não pode ser alterada até mesmo pela própria classe. Na verdade, quando o compilador encontra um identificador constante no código-fonte C# (por exemplo, `Months`), ele substitui o valor literal diretamente no código de IL (linguagem intermediária) que ele produz. Como não há nenhum endereço variável associado a uma constante em tempo de execução, os campos `const` não podem ser passados por referência e não podem aparecer como um l-value em uma expressão.

## ⓘ Observação

Tenha cuidado ao fazer referência a valores constantes definidos em outro código como DLLs. Se uma nova versão da DLL definir um novo valor para a constante, seu programa ainda conterá o valor literal antigo até que ele seja recompilado com a nova versão.

Várias constantes do mesmo tipo podem ser declaradas ao mesmo tempo, por exemplo:

C#

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

A expressão que é usada para inicializar uma constante poderá fazer referência a outra constante se ela não criar uma referência circular. Por exemplo:

C#

```
class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

As constantes podem ser marcadas como [public](#), [private](#), [protected](#), [internal](#), [protected internal](#) ou [private protected](#). Esses modificadores de acesso definem como os usuários da classe podem acessar a constante. Para obter mais informações, consulte [Modificadores de Acesso](#).

As constantes são acessadas como se fossem campos [static](#) porque o valor da constante é o mesmo para todas as instâncias do tipo. Você não usa a palavra-chave [static](#) para declará-las. As expressões que não estão na classe que define a constante devem usar o nome de classe, um período e o nome da constante para acessar a constante. Por exemplo:

C#

```
int birthstones = Calendar.Months;
```

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Guia de Programação em C#](#)
- [Propriedades](#)
- [Types](#)
- [readonly](#)
- [Immutability in C# Part One: Kinds of Immutability](#) (Imutabilidade no C#, parte um: tipos de imutabilidade)

# Como definir propriedades abstract (Guia de programação em C#)

Artigo • 07/04/2023

O exemplo a seguir mostra como definir propriedades `abstract`. Uma declaração de propriedade `abstract` não fornece uma implementação dos acessadores da propriedade – ela declara que a classe dá suporte às propriedades, mas deixa a implementação do acessador para classes derivadas. O exemplo a seguir demonstra como implementar as propriedades `abstract` herdadas de uma classe base.

Esse exemplo consiste em três arquivos, cada um deles é compilado individualmente e seu assembly resultante é referenciado pela próxima compilação:

- `abstractshape.cs`: a classe `Shape` que contém uma propriedade `abstract Area`.
- `shapes.cs`: as subclasses da classe `Shape`.
- `shapetest.cs`: um programa de teste para exibir as áreas de alguns objetos derivados de `Shape`.

Para compilar o exemplo, use o comando a seguir:

```
csc abstractshape.cs shapes.cs shapetest.cs
```

Isso criará o arquivo executável `shapetest.exe`.

## Exemplos

Esse arquivo declara a classe `Shape` que contém a propriedade `Area` do tipo `double`.

```
C#  
  
// compile with: csc -target:library abstractshape.cs  
public abstract class Shape  
{  
    private string name;  
  
    public Shape(string s)  
    {  
        // calling the set accessor of the Id property.  
        Id = s;  
    }  
  
    public string Id  
    {
```

```

    get
    {
        return name;
    }

    set
    {
        name = value;
    }
}

// Area is a read-only property - only a get accessor is needed:
public abstract double Area
{
    get;
}

public override string ToString()
{
    return $"{Id} Area = {Area:F2}";
}
}

```

- Os modificadores da propriedade são colocados na própria declaração de propriedade. Por exemplo:

C#

```
public abstract double Area
```

- Ao declarar uma propriedade abstract (como `Area` neste exemplo), você simplesmente indica quais acessadores de propriedade estão disponíveis, mas não os implementa. Neste exemplo, apenas um acessador `get` está disponível, assim, a propriedade é somente leitura.

O código a seguir mostra três subclasses de `Shape` e como elas substituem a propriedade `Area` para fornecer sua própria implementação.

C#

```

// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public int GetArea()
    {
        return side * side;
    }
}

```

```
}

public override double Area
{
    get
    {
        // Given the side, return the area of a square:
        return side * side;
    }
}

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}
```

O código a seguir mostra um programa de teste que cria uma quantidade de objetos derivados de `Shape` e imprime suas áreas.

C#

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
   Shapes Collection
   Square #1 Area = 25.00
   Circle #1 Area = 28.27
   Rectangle #1 Area = 20.00
*/
```

## Confira também

- [Guia de Programação em C#](#)
- [O sistema do tipo C#](#)
- [Classes e membros de classes abstract e sealed](#)
- [Propriedades](#)

# Como definir constantes em C#

Artigo • 07/04/2023

As constantes são campos cujos valores são definidos em tempo de compilação e nunca podem ser alterados. Use constantes para fornecer nomes significativos em vez de literais numéricos ("números mágicos") a valores especiais.

## ⓘ Observação

No C#, a diretiva de pré-processador `#define` não pode ser utilizada para definir constantes da mesma maneira que é normalmente usada no C e no C++.

Para definir valores de constantes de tipos integrais (`int`, `byte` e assim por diante), use um tipo enumerado. Para obter mais informações, consulte [enum](#).

Para definir constantes não integrais, uma abordagem é agrupá-las em uma única classe estática de nome `Constants`. Isso exigirá que todas as referências às constantes sejam precedidas com o nome de classe, conforme mostrado no exemplo a seguir.

## Exemplo

C#

```
static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}

class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
        Console.WriteLine(secsFromSun);
    }
}
```

O uso do qualificador de nome de classe ajuda a garantir que você e outras pessoas que usam a constante entendam que ele é constante e não pode ser modificado.

## Confira também

- Sistema de tipos do C#

# Propriedades (Guia de Programação em C#)

Artigo • 07/04/2023

Uma propriedade é um membro que oferece um mecanismo flexível para ler, gravar ou calcular o valor de um campo particular. As propriedades podem ser usadas como se fossem membros de dados públicos, mas são métodos realmente especiais chamados *acessadores*. Esse recurso permite que os dados sejam acessados facilmente e ainda ajuda a promover a segurança e a flexibilidade dos métodos.

## Visão geral das propriedades

- As propriedades permitem que uma classe exponha uma forma pública de obter e definir valores, enquanto oculta o código de implementação ou de verificação.
- Um acessador de propriedade `get` é usado para retornar o valor da propriedade e um acessador de propriedade `set` é usado para atribuir um novo valor. No C# 9 e versões posteriores, um acessador de propriedade `init` é usado para atribuir um novo valor somente durante a construção do objeto. Esses acessadores podem ter diferentes níveis de acesso. Para obter mais informações, consulte [Restringindo a acessibilidade aos acessadores](#).
- A palavra-chave `value` é usada para definir o valor que está sendo atribuído pelo acessador `set` ou `init`.
- As propriedades podem ser de *leitura/gravação* (elas têm um acessador `get` e `set`), *somente leitura* (elas têm um acessador `get`, mas nenhum `set`) ou *somente gravação* (elas têm um acessador `set`, mas nenhum `get`). As propriedades somente gravação são raras e são mais comumente usadas para restringir o acesso a dados confidenciais.
- As propriedades simples que não exigem nenhum código de acessador personalizado podem ser implementadas como definições de corpo da expressão ou como [propriedades autoimplementadas](#).

## Propriedades com campos de suporte

Um padrão básico para implementar uma propriedade envolve o uso de um campo de suporte particular da propriedade para definir e recuperar o valor da propriedade. O acessador `get` retorna o valor do campo particular e o acessador `set` pode realizar alguma validação de dados antes de atribuir um valor ao campo particular. Os dois

acessadores também podem realizar alguma conversão ou cálculo nos dados antes de eles serem armazenados ou retornados.

O exemplo a seguir ilustra esse padrão. Neste exemplo, a classe `TimePeriod` representa um intervalo de tempo. Internamente, a classe armazena o intervalo de tempo em segundos em um campo particular chamado `_seconds`. Uma propriedade de leitura/gravação chamada `Hours` permite que o cliente especifique o intervalo de tempo em horas. Tanto o acessador `get` quanto o `set` executam a conversão necessária entre horas e segundos. Além disso, o acessador `set` valida os dados e gera um `ArgumentOutOfRangeException` se o número de horas é inválido.

C#

```
public class TimePeriod
{
    private double _seconds;

    public double Hours
    {
        get { return _seconds / 3600; }
        set
        {
            if (value < 0 || value > 24)
                throw new ArgumentOutOfRangeException(nameof(value),
                    "The valid range is between 0 and 24.");
            _seconds = value * 3600;
        }
    }
}
```

Você pode acessar propriedades para obter e definir o valor, conforme mostrado no seguinte exemplo:

C#

```
TimePeriod t = new TimePeriod();
// The property assignment causes the 'set' accessor to be called.
t.Hours = 24;

// Retrieving the property causes the 'get' accessor to be called.
Console.WriteLine($"Time in hours: {t.Hours}");
// The example displays the following output:
//      Time in hours: 24
```

## Definições de corpo de expressão

Os acessadores de propriedade geralmente consistem em instruções de linha única que simplesmente atribuem ou retornam o resultado de uma expressão. Você pode implementar essas propriedades como membros aptos para expressão. As definições de corpo da expressão consistem no símbolo `=>` seguido pela expressão à qual atribuir ou recuperar da propriedade.

Propriedades somente leitura podem implementar o acessador `get` como um membro apto para expressão. Nesse caso, nem a palavra-chave do acessador `get` nem a palavra-chave `return` é usada. O exemplo a seguir implementa a propriedade `Name` somente leitura como um membro apto para expressão.

```
C#
```

```
public class Person
{
    private string _firstName;
    private string _lastName;

    public Person(string first, string last)
    {
        _firstName = first;
        _lastName = last;
    }

    public string Name => $"{_firstName} {_lastName}";
}
```

Os acessadores `get` e `set` podem ser implementados como membros aptos para expressão. Nesse caso, as palavras-chave `get` e `set` devem estar presentes. O exemplo a seguir ilustra o uso de definições de corpo de expressão para ambos os acessadores. A palavra-chave `return` não é usada com o acessador `get`.

```
C#
```

```
public class SaleItem
{
    string _name;
    decimal _cost;

    public SaleItem(string name, decimal cost)
    {
        _name = name;
        _cost = cost;
    }

    public string Name
    {
        get => _name;
    }
}
```

```
        set => _name = value;
    }

    public decimal Price
    {
        get => _cost;
        set => _cost = value;
    }
}
```

## Propriedades autoimplementadas

Em alguns casos, os acessadores `get` e `set` da propriedade apenas atribuem um valor ou recuperam um valor de um campo de suporte sem incluir nenhuma lógica extra. Usando propriedades autoimplementadas, você pode simplificar o código enquanto o compilador C# fornece de forma transparente o campo de suporte para você.

Se uma propriedade tiver tanto os acessadores `get` e `set` (ou `get` e `init`), ambos deverão ser autoimplementados. Você define uma propriedade autoimplementada usando as palavras-chave `get` e `set` sem fornecer qualquer implementação. O exemplo a seguir repete o anterior, exceto que `Name` e `Price` são propriedades autoimplementadas. O exemplo também remove o construtor parametrizado, de modo que objetos `SaleItem` agora sejam inicializados com uma chamada para o construtor sem parâmetros e um [inicializador de objeto](#).

C#

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

    public decimal Price
    { get; set; }
}
```

As propriedades implementadas automaticamente podem declarar diferentes acessibilidades para os acessadores `get` e `set`. Normalmente, você declara um acessador público `get` e um acessador privado `set`. Saiba mais no artigo sobre [como restringir a acessibilidade do acessador](#).

## Propriedades obrigatórias

Do C# 11 em diante, você pode adicionar o membro `required` para forçar o código do cliente a inicializar qualquer propriedade ou campo:

```
C#  
  
public class SaleItem  
{  
    public required string Name  
    { get; set; }  
  
    public required decimal Price  
    { get; set; }  
}
```

Para criar um `SaleItem`, você precisa definir as propriedades `Name` e `Price` usando [inicializadores de objeto](#), conforme mostrado no seguinte código:

```
C#  
  
var item = new SaleItem { Name = "Shoes", Price = 19.95m };  
Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
```

## Seções relacionadas

- [Usando propriedades](#)
- [Propriedades de interface](#)
- [Comparação entre propriedades e indexadores](#)
- [Restringindo a acessibilidade ao acessador](#)
- [Propriedades autoimplementadas](#)

## Especificação da Linguagem C#

Para obter mais informações, veja [Propriedades](#) na [Especificação da Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Indexadores](#)
- [Palavra-chave get](#)
- [Palavra-chave set](#)

# Usando propriedades (Guia de Programação em C#)

Artigo • 07/04/2023

As propriedades combinam aspectos de métodos e campos. Para o usuário de um objeto, uma propriedade parece ser um campo. Acessar a propriedade requer a mesma sintaxe. Para o implementador de uma classe, uma propriedade consiste em um ou dois blocos de código, que representam um acessador `get` e/ou um acessador `set`. O bloco de código para o acessador `get` é executado quando a propriedade é lida. O bloco de código para o acessador `set` é executado quando um valor é atribuído à propriedade. Uma propriedade sem um acessador `set` é considerada como somente leitura. Uma propriedade sem um acessador `get` é considerada como somente gravação. Uma propriedade que tem os dois acessadores é leitura/gravação. No C# 9 e versões posteriores, você pode usar um acessador `init` em vez de um acessador `set` para tornar a propriedade somente leitura.

Diferentemente dos campos, as propriedades não são classificadas como variáveis. Portanto, você não pode passar uma propriedade como um parâmetro `ref` ou `out`.

As propriedades têm muitos usos: elas podem validar os dados antes de permitir uma alteração; elas podem expor, de forma transparente, os dados em uma classe em que esses dados são recuperados de outra origem qualquer, como um banco de dados; elas podem executar uma ação quando os dados são alterados, como acionar um evento ou alterar o valor de outros campos.

As propriedades são declaradas no bloco de classe, especificando o nível de acesso do campo, seguido pelo tipo da propriedade, pelo nome da propriedade e por um bloco de código que declara um acessador `get` e/ou um acessador `set`. Por exemplo:

```
C#  
  
public class Date  
{  
    private int _month = 7; // Backing store  
  
    public int Month  
    {  
        get => _month;  
        set  
        {  
            if ((value > 0) && (value < 13))  
            {  
                _month = value;  
            }  
        }  
    }  
}
```

```
        }
    }
}
```

Neste exemplo, `Month` é declarado como uma propriedade de maneira que o acessador `set` possa garantir que o valor `Month` esteja definido entre 1 e 12. A propriedade `Month` usa um campo particular para rastrear o valor real. O local real dos dados de uma propriedade é frequentemente chamado de "repositório de backup" da propriedade. É comum que as propriedades usem campos privados como repositório de backup. O campo é marcado como particular para garantir que ele só pode ser alterado ao chamar a propriedade. Para obter mais informações sobre as restrições de acesso público e particular, consulte [Modificadores de acesso](#).

As propriedades autoimplementadas fornecem sintaxe simplificada para declarações de propriedade simples. Para obter mais informações, consulte [Propriedades autoimplementadas](#).

## O acessador get

O corpo do acessador `get` assemelha-se ao de um método. Ele deve retornar um valor do tipo de propriedade. A execução do acessador `get` é equivalente à leitura do valor do campo. Por exemplo, quando você estiver retornando a variável particular do acessador `get` e as otimizações estiverem habilitadas, a chamada ao método do acessador `get` será embutida pelo compilador, portanto, não haverá nenhuma sobrecarga de chamada de método. No entanto, um método do acessador `get` virtual não pode ser embutido porque o compilador não sabe, em tempo de compilação, qual método pode ser chamado em tempo de execução. O seguinte exemplo mostra um acessador `get` que retorna o valor de um campo particular `_name`:

C#

```
class Employee
{
    private string _name; // the name field
    public string Name => _name; // the Name property
}
```

Quando você referencia a propriedade, exceto como o destino de uma atribuição, o acessador `get` é invocado para ler o valor da propriedade. Por exemplo:

C#

```
var employee= new Employee();
//...

System.Console.WriteLine(employee.Name); // the get accessor is invoked here
```

O acessador `get` deve terminar em uma instrução `return` ou `throw` e o controle não pode fluir para fora do corpo do acessador.

### ⚠️ Aviso

Alterar o estado do objeto usando o acessador `get` é um estilo ruim de programação.

O acessador `get` pode ser usado para retornar o valor do campo ou para calculá-lo e retorná-lo. Por exemplo:

C#

```
class Manager
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

No segmento de código anterior, se você não atribuir um valor para a propriedade `Name`, ela retornará o valor `NA`.

## O acessador set

O acessador `set` é semelhante a um método cujo tipo de retorno é `void`. Ele usa uma parâmetro implícito chamado `value`, cujo tipo é o tipo da propriedade. No exemplo a seguir, uma acessador `set` é adicionado à propriedade `Name`:

C#

```
class Student
{
    private string _name; // the name field
    public string Name // the Name property
    {
        get => _name;
        set => _name = value;
    }
}
```

Quando você atribui um valor à propriedade, o acessador `set` é invocado por meio do uso de um argumento que fornece o novo valor. Por exemplo:

C#

```
var student = new Student();
student.Name = "Joe"; // the set accessor is invoked here

System.Console.WriteLine(student.Name); // the get accessor is invoked here
```

É um erro usar o nome de parâmetro implícito `value`, para uma declaração de variável local em um acessador `set`.

## O acessador `init`

O código para criar um acessador `init` é o mesmo que o código para criar um acessador `set`, exceto que você usa a palavra-chave `init` em vez de `set`. A diferença é que o acessador `init` só pode ser empregado no construtor ou usando um [inicializador de objeto](#).

## Comentários

As propriedades podem ser marcadas como `public`, `private`, `protected`, `internal`, `protected internal` ou `private protected`. Esses modificadores de acesso definem como os usuários da classe podem acessar a propriedade. Os acessadores `get` e `set` para a mesma propriedade podem ter modificadores de acesso diferentes. Por exemplo, o `get` pode ser `public` para permitir acesso somente leitura de fora do tipo e o `set` pode ser `private` ou `protected`. Para obter mais informações, consulte [Modificadores de Acesso](#).

Uma propriedade pode ser declarada como uma propriedade estática, usando a palavra-chave `static`. As propriedades estáticas estão disponíveis para chamadores a qualquer momento, mesmo se não existir nenhuma instância da classe. Para obter mais informações, consulte [Classes estáticas e membros de classes estáticas](#).

Uma propriedade pode ser marcada como uma propriedade virtual, usando a palavra-chave `virtual`. As propriedades virtuais permitem que classes derivadas substituam o comportamento da propriedade, usando a palavra-chave `override`. Para obter mais informações sobre essas opções, consulte [Herança](#).

Uma propriedade que substitui uma propriedade virtual também pode ser `sealed`, especificando que ela não é mais virtual para classes derivadas. Por fim, uma propriedade pode ser declarada `abstract`. As propriedades abstratas não definem nenhuma implementação na classe e as classes derivadas devem escrever sua própria implementação. Para obter mais informações sobre essas opções, consulte [Classes e membros de classes abstract e sealed](#).

#### ⓘ Observação

É um erro usar um modificador `virtual`, `abstract` ou `override` em um acessador de uma propriedade `static`.

## Exemplos

Este exemplo demonstra as propriedades instância, estática e somente leitura. Ele aceita o nome do funcionário digitado no teclado, incrementa `NumberOfEmployees` em 1 e exibe o nome e o número do funcionário.

C#

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the
    employee's number:
}
```

## Exemplo de propriedade oculta

Este exemplo demonstra como acessar uma propriedade em uma classe base que está oculta por outra propriedade que tem o mesmo nome em uma classe derivada:

```
C#  
  
public class Employee  
{  
    private string _name;  
    public string Name  
    {  
        get => _name;  
        set => _name = value;  
    }  
}  
  
public class Manager : Employee  
{  
    private string _name;  
  
    // Notice the use of the new modifier:  
    public new string Name  
    {  
        get => _name;  
        set => _name = value + ", Manager";  
    }  
}  
  
class TestHiding  
{  
    public static void Test()  
    {  
        Manager m1 = new Manager();  
  
        // Derived class property.  
        m1.Name = "John";  
  
        // Base class property.  
        ((Employee)m1).Name = "Mary";  
  
        System.Console.WriteLine("Name in the derived class is: {0}",  
m1.Name);  
        System.Console.WriteLine("Name in the base class is: {0}",  
((Employee)m1).Name);  
    }  
}  
/* Output:  
   Name in the derived class is: John, Manager  
   Name in the base class is: Mary  
*/
```

A seguir estão os pontos importantes do exemplo anterior:

- A propriedade `Name` na classe derivada oculta a propriedade `Name` na classe base.

Nesse caso, o modificador `new` é usado na declaração da propriedade na classe derivada:

C#

```
public new string Name
```

- A conversão `(Employee)` é usada para acessar a propriedade oculta na classe base:

C#

```
((Employee)m1).Name = "Mary";
```

Para obter mais informações sobre como ocultar membros, consulte o [Modificador new](#).

## Exemplo de substituição de propriedade

Neste exemplo, duas classes, `Cube` e `Square`, implementam uma classe abstrata `Shape` e substituem sua propriedade `Area` abstrata. Observe o uso do modificador `override` nas propriedades. O programa aceita o lado como uma entrada e calcula as áreas para o cubo e o quadrado. Ele também aceita a área como uma entrada e calcula o lado correspondente para o cubo e o quadrado.

C#

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}
```

```

}

class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
    }
}

/* Example Output:
   Enter the side: 4
   Area of the square = 16.00
   Area of the cube = 96.00

   Enter the area: 24
   Side of the square = 4.90
   Side of the cube = 2.00
*/

```

## Confira também

- [Propriedades](#)
- [Propriedades de interface](#)
- [Propriedades autoimplementadas](#)

# Propriedades de interface (Guia de Programação em C#)

Artigo • 07/04/2023

As propriedades podem ser declaradas em uma [interface](#). O exemplo a seguir declara um acessador de propriedade de interface:

C#

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

As propriedades da interface normalmente não têm um corpo. Os acessadores indicam se a propriedade é leitura/gravação, somente leitura ou somente gravação. Ao contrário de classes e structs, declarar os acessadores sem um corpo não declara uma [propriedade implementada automaticamente](#). Uma interface pode definir uma implementação padrão para membros, incluindo propriedades. Definir uma implementação padrão para uma propriedade em uma interface é raro porque as interfaces podem não definir campos de dados de instância.

## Exemplo

Neste exemplo, a interface `IEmployee` tem uma propriedade de leitura/gravação, `Name` e uma propriedade somente leitura, `Counter`. A classe `Employee` implementa a interface `IEmployee` e usa essas duas propriedades. O programa lê o nome de um novo funcionário e o número atual de funcionários e exibe o nome do funcionário e o número do funcionário computado.

Seria possível usar o nome totalmente qualificado da propriedade, que referencia a interface na qual o membro é declarado. Por exemplo:

C#

```
string IEmployee.Name
{
```

```
    get { return "Employee Name"; }
    set { }
}
```

O exemplo anterior demonstra a [Implementação explícita da interface](#). Por exemplo, se a classe `Employee` estiver implementando duas interfaces `ICitizen` e `IEmployee` e as duas interfaces tiverem a propriedade `Name`, será necessária a implementação explícita de membro da interface. Ou seja, a seguinte declaração de propriedade:

C#

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

implementa a propriedade `Name` na interface `IEmployee`, enquanto a seguinte declaração:

C#

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

implementa a propriedade `Name` na interface `ICitizen`.

C#

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
```

```
public static int numberOfEmployees;

private string _name;
public string Name // read-write instance property
{
    get => _name;
    set => _name = value;
}

private int _counter;
public int Counter // read-only instance property
{
    get => _counter;
}

// constructor
public Employee() => _counter = ++numberOfEmployees;
}
```

C#

```
System.Console.Write("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.Write("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine("Employee number: {0}", e1.Counter);
System.Console.WriteLine("Employee name: {0}", e1.Name);
```

## Saída de exemplo

Console

```
Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous
```

## Confira também

- [Guia de Programação em C#](#)
- [Propriedades](#)
- [Usando propriedades](#)

- Comparação entre propriedades e indexadores
- Indexadores
- Interfaces

# Restringindo a acessibilidade ao acessador (Guia de Programação em C#)

Artigo • 07/04/2023

As partes `get` e `set` de uma propriedade ou de um indexador são chamadas *acessadores*. Por padrão, esses acessadores têm a mesma visibilidade ou nível de acesso da propriedade ou do indexador aos quais pertencem. Para obter mais informações, consulte [níveis de acessibilidade](#). No entanto, às vezes é útil restringir o acesso a um desses acessadores. Normalmente, você restringe a acessibilidade do acessador `set`, mantendo o acessador `get` publicamente acessível. Por exemplo:

C#

```
private string _name = "Hello";

public string Name
{
    get
    {
        return _name;
    }
    protected set
    {
        _name = value;
    }
}
```

Neste exemplo, uma propriedade chamada `Name` define um acessador `get` e `set`. O acessador `get` recebe o nível de acessibilidade da propriedade em si, `public` nesse caso, embora o `set` acessador esteja restrito explicitamente ao aplicar o modificador de acesso `protegido` ao acessador em si.

## ⓘ Observação

Os exemplos neste artigo não usam *propriedades de implementação automática*. As *propriedades de implementação automática* fornecem uma sintaxe concisa para declarar propriedades quando um campo de suporte personalizado não é necessário.

# Restrições em modificadores de acesso nos acessadores

O uso dos modificadores de acesso em propriedades ou indexadores está sujeito a estas condições:

- Não é possível usar modificadores de acessador em uma interface nem uma implementação explícita de membro de [interface](#).
- É possível usar os modificadores de acessador somente se a propriedade ou o indexador tiver os acessadores `set` e `get`. Nesse caso, o modificador é permitido em apenas um dos dois acessadores.
- Se a propriedade ou o indexador tiver um modificador [substituir](#), o modificador de acessador deverá corresponder ao acessador substituído, se houver.
- O nível de acessibilidade do acessador deve ser mais restritivo do que o nível de acessibilidade na propriedade ou no indexador em si.

## Modificadores de acesso em acessadores de substituição

Quando você substitui uma propriedade ou indexador, os acessadores substituídos devem estar acessíveis ao código de substituição. Além disso, a acessibilidade da propriedade/indexador, e seus acessadores, devem corresponder à propriedade/indexador substituído e seus acessadores. Por exemplo:

```
C#  
  
public class Parent  
{  
    public virtual int TestProperty  
    {  
        // Notice the accessor accessibility level.  
        protected set { }  
  
        // No access modifier is used here.  
        get { return 0; }  
    }  
}  
public class Kid : Parent  
{  
    public override int TestProperty  
    {  
        // Use the same accessibility level as in the overridden accessor.  
        protected set { }  
  
        // Cannot use access modifier here.  
    }  
}
```

```
        get { return 0; }
    }
}
```

## Implementando interfaces

Quando você usa um acessador para implementar uma interface, o acessador pode não ter um modificador de acesso. No entanto, se você implementar a interface usando um acessador, como `get`, o outro acessador poderá ter um modificador de acesso, como no exemplo a seguir:

C#

```
public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}
```

## Domínio de acessibilidade do acessador

Se você usar um modificador de acesso no acessador, o [domínio de acessibilidade](#) do acessador será determinado por esse modificador.

Se você não usou um modificador de acesso no acessador, o domínio de acessibilidade do acessador será determinado pelo nível de acessibilidade da propriedade ou do indexador.

# Exemplo

O exemplo a seguir contém três classes, `BaseClass`, `DerivedClass` e `MainClass`. Há duas propriedades no `BaseClass`, `Name` e `Id` em ambas as classes. O exemplo demonstra como a propriedade `Id` no `DerivedClass` pode ser oculta pela propriedade `Id` no `BaseClass` quando você usa um modificador de acesso restritivo como `protegido` ou `privado`. Portanto, em vez disso, quando você atribui valores a essa propriedade, a propriedade na classe `BaseClass` é chamada. Substituindo o modificador de acesso por `público` tornará a propriedade acessível.

O exemplo também demonstra que um modificador de acesso restritivo, como `private` ou `protected`, no acessador `set` da propriedade `Name` em `DerivedClass` impede o acesso ao acessador na classe derivada. Ele gera um erro quando você faz uma atribuição a ele ou acessa a propriedade de classe base com o mesmo nome, quando acessível.

C#

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }
    }

    // Using "protected" would make the set accessor not accessible.
}
```

```

        set
    {
        _name = value;
    }
}

// Using private on the following property hides it in the Main Class.
// Any assignment to the property will use Id in BaseClass.
new private string Id
{
    get
    {
        return _id;
    }
    set
    {
        _id = value;
    }
}
}

class MainClass
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123"; // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   Base: Name-BaseClass, ID-BaseClass
   Derived: John, ID-BaseClass
*/

```

## Comentários

Observe que, se você substituir a declaração `new private string Id` por `new public string Id`, você obterá a saída:

```
Name and ID in the base class: Name-BaseClass, ID-BaseClass Name and ID in the  
derived class: John, John123
```

## Confira também

- Propriedades
- Indexadores
- Modificadores de acesso
- Somente propriedades de inicialização
- Propriedades obrigatórias

# Como declarar e usar propriedades de leitura e gravação (Guia de Programação em C#)

Artigo • 07/04/2023

As propriedades oferecem a conveniência de membros de dados públicos sem os riscos associados ao acesso sem proteção, sem controle e não verificado aos dados de um objeto. Propriedades declaram *acessadores*: métodos especiais que atribuem e recuperam valores do membro de dados subjacente. O acessador `set` habilita a atribuição de membros de dados e o acessador `get` recupera valores do membro de dados.

Este exemplo mostra uma classe `Person` que tem duas propriedades: `Name` (string) e `Age` (int). Ambas as propriedades fornecem acessadores `get` e `set`, portanto, são consideradas propriedades de leitura/gravação.

## Exemplo

```
C#  
  
class Person  
{  
    private string _name = "N/A";  
    private int _age = 0;  
  
    // Declare a Name property of type string:  
    public string Name  
    {  
        get  
        {  
            return _name;  
        }  
        set  
        {  
            _name = value;  
        }  
    }  
  
    // Declare an Age property of type int:  
    public int Age  
    {  
        get  
        {  
            return _age;  
        }  
    }  
}
```

```
        }

        set
        {
            _age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

public class Wrapper
{
    private string _name = "N/A";
    public string Name
    {
        get
        {
            return _name;
        }
        private set
        {
            _name = value;
        }
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();

        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

```
/* Output:  
Person details - Name = N/A, Age = 0  
Person details - Name = Joe, Age = 99  
Person details - Name = Joe, Age = 100  
*/
```

## Programação robusta

No exemplo anterior, as propriedades `Name` e `Age` são **públicas** e incluem os acessadores `get` e `set`. Os acessadores públicos permitem que qualquer objeto leia e grave essas propriedades. No entanto, às vezes é desejável excluir um ou os acessadores. Você pode omitir o acessador `set` para tornar a propriedade somente leitura:

C#

```
public string Name  
{  
    get  
    {  
        return _name;  
    }  
    private set  
    {  
        _name = value;  
    }  
}
```

Como alternativa, é possível expor um acessador publicamente, porém, tornando o outro privado ou protegido. Para obter mais informações, consulte [Acessibilidade do Acessador Assimétrico](#).

Depois de serem declaradas, as propriedades podem ser usadas como campos da classe. As propriedades permitem uma sintaxe natural na obtenção e configuração do valor de uma propriedade, conforme as instruções a seguir:

C#

```
person.Name = "Joe";  
person.Age = 99;
```

Em um método de propriedade `set`, uma variável especial `value` está disponível. Essa variável contém o valor que o usuário especificou, por exemplo:

C#

```
_name = value;
```

Observe a sintaxe normal para incrementar a propriedade `Age` em um objeto `Person`:

C#

```
person.Age += 1;
```

Se métodos `set` e `get` separados fossem usados para modelar propriedades, o código equivalente se pareceria com isto:

C#

```
person.SetAge(person.GetAge() + 1);
```

O método `ToString` é substituído neste exemplo:

C#

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Observe que `ToString` não é usado explicitamente no programa. Ele é invocado por padrão pelas chamadas `WriteLine`.

## Confira também

- [Propriedades](#)
- [Sistema de tipos do C#](#)

# Propriedades autoimplementadas (Guia de Programação em C#)

Artigo • 08/06/2023

As propriedades autoimplementadas tornam a declaração de propriedade mais concisa quando nenhuma lógica adicional for necessária nos acessadores de propriedade. Elas também habilitam o código do cliente a criar objetos. Ao declarar uma propriedade, como mostrado no exemplo a seguir, o compilador cria um campo de suporte privado e anônimo que pode ser acessado somente por meio dos acessadores `get` e `set` da propriedade. Em C# 9 e posterior, os acessadores `init` também podem ser declarados como propriedades implementadas automaticamente.

## Exemplo

O exemplo a seguir mostra uma classe simples que tem algumas propriedades autoimplementadas:

C#

```
// This class is mutable. Its data can be modified from
// outside the class.
public class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
```

```
{  
    // Initialize a new object.  
    Customer cust1 = new Customer(4987.63, "Northwind", 90108);  
  
    // Modify a property.  
    cust1.TotalPurchases += 499.99;  
}  
}
```

Você não pode declarar propriedades implementadas automaticamente em interfaces. As propriedades implementadas automaticamente declaram um campo de backup de instância privada e as interfaces podem não declarar campos de instância. Declarar uma propriedade em uma interface sem definir um corpo declara uma propriedade com acessadores que devem ser implementados por cada tipo que implementa essa interface.

É possível inicializar as propriedades autoimplementadas da mesma forma que os campos:

C#

```
public string FirstName { get; set; } = "Jane";
```

A classe mostrada no exemplo anterior é mutável. O código cliente pode alterar os valores nos objetos após a criação. Em classes complexas que contêm comportamento significativo (métodos), bem como dados, geralmente é necessário ter propriedades públicas. No entanto, para classes pequenas ou structs que encapsulam apenas um conjunto de valores (dados) e têm pouco ou nenhum comportamento, você deve usar uma das seguintes opções para tornar os objetos imutáveis:

- Declare apenas um acessador `get` (imutável em todos os lugares, exceto o construtor).
- Declare um acessador `get` e um acessador `init` (imutável em todos os lugares, exceto durante a construção do objeto).
- Declare o acessador `set` como `privado` (imutável para os consumidores).

Para obter mais informações, confira [Como Implementar uma classe leve com propriedades autoimplementadas](#).

## Confira também

- [Usar propriedades implementadas automaticamente \(regra de estilo IDE0032\)](#)
- [Propriedades](#)

- Modificadores

# Como implementar uma classe leve com propriedades autoimplementadas (Guia de Programação em C#)

Artigo • 07/04/2023

Este exemplo mostra como criar uma classe leve imutável que serve apenas para encapsular um conjunto de propriedades autoimplementadas. Use esse tipo de constructo em vez de um struct quando for necessário usar a semântica do tipo de referência.

Você pode tornar uma propriedade imutável das seguintes maneiras:

- Declare somente o acessador `get`, o que torna a propriedade imutável em todos os lugares, exceto no construtor do tipo.
- Declare um acessador `init` em vez de `set`, o que torna a propriedade configurável somente no construtor ou com o uso de um [inicializador de objeto](#).
- Declare o acessador `set` como `private`. A propriedade será configurável somente dentro do tipo, mas imutável para os consumidores.

Você pode adicionar o modificador `required` à declaração de propriedade para forçar os chamadores a definir a propriedade como parte da inicialização de um novo objeto.

O exemplo a seguir mostra como uma propriedade somente com o acessador `get` difere de outra com `get` e conjunto privado.

C#

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;

    // Address is assignable here.
    public void ChangeAddress(string newAddress) => Address = newAddress;
}
```

# Exemplo

O exemplo a seguir mostra duas maneiras de implementar uma classe imutável que tem propriedades autoimplementadas. Entre essas maneiras, uma declara uma das propriedades com um `set` privado e outra declara uma das propriedades somente com um `get`. A primeira classe usa um construtor somente para inicializar as propriedades e a segunda classe usa um método de fábrica estático que chama um construtor.

C#

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
{
    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }

    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
```

```

    {
        return new Contact2(name, address);
    }
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = {"Terry Adams", "Fadi Fakhouri", "Hanying Feng",
                           "Cesar Garcia", "Debra Garcia"};
        string[] addresses = {"123 Main St.", "345 Cypress Ave.", "678 1st
Ave",
                           "12 108th St.", "89 E. 42nd St."};

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                     select Contact2.CreateContact(names[i],
addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();

        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
 Terry Adams, 123 Main St.
 Fadi Fakhouri, 345 Cypress Ave.
 Hanying Feng, 678 1st Ave
 Cesar Garcia, 12 108th St.
 Debra Garcia, 89 E. 42nd St.
*/

```

O compilador cria campos de suporte para cada propriedade autoimplementada. Os campos não são acessíveis diretamente do código-fonte.

## Confira também

- [Propriedades](#)
- [struct](#)
- [Inicializadores de objeto e coleção](#)

# Métodos (Guia de Programação em C#)

Artigo • 07/04/2023

Um método é um bloco de código que contém uma série de instruções. Um programa faz com que as instruções sejam executadas chamando o método e especificando os argumentos de método necessários. No C#, todas as instruções executadas são realizadas no contexto de um método.

O método `Main` é o ponto de entrada para todos os aplicativos C# e é chamado pelo CLR (Common Language Runtime) quando o programa é iniciado. Em um aplicativo que usa [instruções de nível superior](#), o `Main` método é gerado pelo compilador e contém todas as instruções de nível superior.

## ⓘ Observação

Este artigo discute métodos nomeados. Para obter mais informações sobre funções anônimas, consulte [Expressões lambda](#).

## Assinaturas de método

Os métodos são declarados em uma [classe](#), [struct](#) ou [interface](#) especificando o nível de acesso, como `public` ou `private`, modificadores opcionais, como `abstract` ou `sealed`, o valor retornado, o nome do método e os parâmetros de método. Juntas, essas partes são a assinatura do método.

## ⓘ Importante

Um tipo de retorno de um método não faz parte da assinatura do método para fins de sobrecarga de método. No entanto, ele faz parte da assinatura do método ao determinar a compatibilidade entre um delegado e o método para o qual ele aponta.

Os parâmetros de método estão entre parênteses e separados por vírgulas. Parênteses vazios indicam que o método não requer parâmetros. Essa classe contém quatro métodos:

C#

```
abstract class Motorcycle
{
```

```

// Anyone can call this.
public void StartEngine() /* Method statements here */

// Only derived classes can call this.
protected void AddGas(int gallons) { /* Method statements here */

// Derived classes can override the base class implementation.
public virtual int Drive(int miles, int speed) { /* Method statements
here */ return 1; }

// Derived classes must implement this.
public abstract double GetTopSpeed();
}

```

## Acesso a método

Chamar um método em um objeto é como acessar um campo. Após o nome do objeto, adicione um ponto final, o nome do método e parênteses. Os argumentos são listados dentro dos parênteses e são separados por vírgulas. Os métodos da classe `Motorcycle` podem, portanto, ser chamados como no exemplo a seguir:

C#

```

class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}

```

## Parâmetros de método versus argumentos

A definição do método especifica os nomes e tipos de quaisquer parâmetros obrigatórios. Quando o código de chamada chama o método, ele fornece valores concretos, chamados argumentos, para cada parâmetro. Os argumentos devem ser compatíveis com o tipo de parâmetro, mas o nome do argumento (se algum) usado no código de chamada não precisa ser o mesmo que o parâmetro chamado definido no método. Por exemplo:

```
C#  
  
public void Caller()  
{  
    int numA = 4;  
    // Call with an int variable.  
    int productA = Square(numA);  
  
    int numB = 32;  
    // Call with another int variable.  
    int productB = Square(numB);  
  
    // Call with an integer literal.  
    int productC = Square(12);  
  
    // Call with an expression that evaluates to int.  
    productC = Square(productA * 3);  
}  
  
int Square(int i)  
{  
    // Store input argument in a local variable.  
    int input = i;  
    return input * input;  
}
```

## Passando por referência versus passando por valor

Por padrão, quando uma instância de um [tipo de valor](#) é passada para um método, sua cópia é passada em vez da própria instância. Portanto, as alterações no argumento não têm nenhum efeito sobre a instância original no método de chamada. Para passar uma instância de tipo de valor por referência, use a palavra-chave `ref`. Para obter mais informações, consulte [Passando parâmetros de tipo de valor](#).

Quando um objeto de tipo de referência é passado para um método, uma referência ao objeto é passada. Ou seja, o método recebe não o objeto em si, mas um argumento que indica o local do objeto. Se você alterar um membro do objeto usando essa referência, a

alteração será refletida no argumento no método de chamada, ainda que você passe o objeto por valor.

Crie um tipo de referência usando a palavra-chave `class`, como mostra o exemplo a seguir:

```
C#  
  
public class SampleRefType  
{  
    public int value;  
}
```

Agora, se você passar um objeto com base nesse tipo para um método, uma referência ao objeto será passada. O exemplo a seguir passa um objeto do tipo `SampleRefType` ao método `ModifyObject`:

```
C#  
  
public static void TestRefType()  
{  
    SampleRefType rt = new SampleRefType();  
    rt.value = 44;  
    ModifyObject(rt);  
    Console.WriteLine(rt.value);  
}  
  
static void ModifyObject(SampleRefType obj)  
{  
    obj.value = 33;  
}
```

O exemplo faz essencialmente a mesma coisa que o exemplo anterior, pois ele passa um argumento por valor para um método. No entanto, como um tipo de referência é usado, o resultado é diferente. A modificação feita em `ModifyObject` para o campo `value` do parâmetro, `obj`, também altera o campo `value` do argumento, `rt`, no método `TestRefType`. O método `TestRefType` exibe 33 como a saída.

Para obter mais informações sobre como passar tipos de referência por referência e por valor, consulte [Passando parâmetros de tipo de referência](#) e [Tipos de referência](#).

## Valores retornados

Os métodos podem retornar um valor para o chamador. Se o tipo de retorno (o tipo listado antes do nome do método) não for `void`, o método poderá retornar o valor

usando a [instrução return](#). Uma instrução com a palavra-chave `return` seguida por uma variável que corresponde ao tipo de retorno retornará esse valor ao chamador do método.

O valor pode ser retornado ao chamador por valor ou [por referência](#). Valores são retornados ao chamador por referência se a `ref` palavra-chave é usada na assinatura do método e segue cada palavra-chave `return`. Por exemplo, a instrução de retorno e a assinatura de método a seguir indicam que o método retorna uma variável chamada `estDistance` por referência para o chamador.

C#

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

A palavra-chave `return` também interrompe a execução do método. Se o tipo de retorno for `void`, uma instrução `return` sem um valor ainda será útil para interromper a execução do método. Sem a palavra-chave `return`, a execução do método será interrompida quando chegar ao final do bloco de código. Métodos com um tipo de retorno não nulo devem usar a palavra-chave `return` para retornar um valor. Por exemplo, esses dois métodos usam a palavra-chave `return` para retornar inteiros:

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

Para usar um valor retornado de um método, o método de chamada pode usar a chamada de método em si em qualquer lugar que um valor do mesmo tipo seria suficiente. Você também pode atribuir o valor retornado a uma variável. Por exemplo, os dois exemplos de código a seguir obtêm a mesma meta:

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Usar uma variável local, nesse caso, `result`, para armazenar um valor é opcional. Isso pode ajudar a legibilidade do código ou pode ser necessário se você precisar armazenar o valor original do argumento para todo o escopo do método.

Para usar o valor retornado de um método por referência, você deve declarar uma variável `ref local` se você pretende modificar seu valor. Por exemplo, se o método `Planet.GetEstimatedDistance` retorna um valor `Double` por referência, você pode defini-lo como uma variável `ref local` com código semelhante ao seguinte:

C#

```
ref double distance = ref Planet.GetEstimatedDistance();
```

Retornar uma matriz multidimensional de um método, `M`, que modifica o conteúdo da matriz, não é necessário se a função de chamada passou a matriz para `M`. Você pode retornar a matriz resultante de `M` para um bom estilo ou fluxo funcional de valores, mas isso não é necessário porque o C# passa todos os tipos de referência por valor e o valor de uma referência de matriz é o ponteiro para a matriz. No método `M`, as alterações do conteúdo da matriz podem ser observadas por qualquer código que tiver uma referência à matriz, conforme mostrado no exemplo a seguir:

C#

```
static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
```

```
        for (int j = 0; j < matrix.GetLength(1); j++)
    {
        matrix[i, j] = -1;
    }
}
```

## Métodos assíncronos

Usando o recurso `async`, você pode invocar métodos assíncronos sem usar retornos de chamada explícitos ou dividir manualmente seu código entre vários métodos ou expressões lambda.

Se marcar um método com o modificador `async`, você poderá usar o operador `await` no método. Quando o controle atinge uma expressão `await` no método assíncrono, ele retorna para o chamador e o progresso no método é suspenso até a tarefa aguardada ser concluída. Quando a tarefa for concluída, a execução poderá ser retomada no método.

### ⓘ Observação

Um método assíncrono retorna para o chamador quando encontra o primeiro objeto esperado que ainda não está completo ou chega ao final do método assíncrono, o que ocorre primeiro.

Um método assíncrono normalmente tem um tipo de retorno `Task<TResult>`, `Task`, `IAsyncEnumerable<T>` ou `void`. O tipo de retorno `void` é usado principalmente para definir manipuladores de eventos, nos quais o tipo de retorno `void` é necessário. Um método assíncrono que retorna `void` não pode ser aguardado e o chamador de um método de retorno nulo não pode capturar as exceções que esse método gera. Um método assíncrono pode ter [qualquer tipo de retorno semelhante a tarefas](#).

No exemplo a seguir, `DelayAsync` é um método assíncrono que tem um tipo de retorno de `Task<TResult>`. `DelayAsync` tem uma instrução `return` que retorna um número inteiro. Portanto, a declaração do método de `Task<int>` deve ter um tipo de retorno de `DelayAsync`. Como o tipo de retorno é `Task<int>`, a avaliação da expressão `await` em `DoSomethingAsync` produz um inteiro, como a instrução a seguir demonstra: `int result = await delayTask.`

O método `Main` é um exemplo de método assíncrono que tem um tipo de retorno `Task`. Ele vai para o método `DoSomethingAsync` e, como é expresso com uma única linha, ele

pode omitir as palavras-chave `async` e `await`. Como `DoSomethingAsync` é um método assíncrono, a tarefa para a chamada para `DoSomethingAsync` deve ser colocada em espera, como mostra a seguinte instrução: `await DoSomethingAsync();`.

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

Um método assíncrono não pode declarar nenhum parâmetro `ref` ou `out`, mas pode chamar métodos com tais parâmetros.

Para obter mais informações sobre os métodos assíncronos, consulte [Programação assíncrona com `async` e `await`](#) e [Tipos de retorno Async](#).

## Definições de corpo de expressão

É comum ter definições de método que simplesmente retornam imediatamente com o resultado de uma expressão ou que têm uma única instrução como o corpo do método. Há um atalho de sintaxe para definir esses métodos usando `=>`:

C#

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
```

```
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Se o método retornar `void` ou for um método assíncrono, o corpo do método deverá ser uma expressão de instrução (igual às lambdas). Para propriedades e indexadores, eles devem ser somente leitura e você não usa a palavra-chave do acessador `get`.

## Iterators

Um iterador realiza uma iteração personalizada em uma coleção, como uma lista ou uma matriz. Um iterador usa a instrução `yield return` para retornar um elemento de cada vez. Quando uma instrução `yield return` for atingida, o local atual no código será lembrado. A execução será reiniciada desse local quando o iterador for chamado na próxima vez.

Você chama um iterador de um código de cliente usando uma instrução `foreach`.

O tipo de retorno de um iterador pode ser `IEnumerable`, `IEnumerable<T>`, `IAsyncEnumerable<T>`, `IEnumerator` ou `IEnumerator<T>`.

Para obter mais informações, consulte [Iteradores](#).

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Guia de Programação em C#](#)
- [Sistema de tipos do C#](#)
- [Modificadores de acesso](#)
- [Classes static e membros de classes static](#)
- [Herança](#)
- [Classes e membros de classes abstract e sealed](#)
- [params](#)
- [out](#)
- [ref](#)
- [Parâmetros de método](#)

# Funções locais (Guia de Programação em C#)

Artigo • 27/04/2023

*Funções locais* são métodos de um tipo que estão aninhados em outro membro. Eles só podem ser chamados do membro que os contém. Funções locais podem ser declaradas em e chamadas de:

- Métodos, especialmente os métodos iteradores e os métodos assíncronos
- Construtores
- Acessadores de propriedades
- Acessadores de eventos
- Métodos anônimos
- Expressões lambda
- Finalizadores
- Outras funções locais

No entanto, as funções locais não podem ser declaradas dentro de um membro apto para expressão.

## ⓘ Observação

Em alguns casos, você pode usar uma expressão lambda para implementar uma funcionalidade que também tem suporte por uma função local. Para obter uma comparação, confira [Funções locais em comparação a expressões Lambda](#).

Funções locais tornam a intenção do seu código clara. Qualquer pessoa que leia o código poderá ver que o método não pode ser chamado, exceto pelo método que o contém. Para projetos de equipe, elas também impossibilitam que outro desenvolvedor chame o método por engano diretamente de qualquer outro lugar na classe ou no struct.

## Sintaxe de função local

Uma função local é definida como um método aninhado dentro de um membro recipiente. Sua definição tem a seguinte sintaxe:

C#

```
<modifiers> <return-type> <method-name> <parameter-list>
```

Você pode usar os seguintes modificadores com uma função local:

- `async`
- `unsafe`
- `static` Uma função local estática não pode capturar variáveis locais nem o estado da instância.
- `extern` Uma função local externa deve ser `static`.

Todas as variáveis locais definidas no membro relativo, incluindo os parâmetros do método, são acessíveis em uma função local não estática.

Ao contrário de uma definição de método, uma definição de função local não pode incluir o modificador de acesso de membro. Já que todas as funções locais são privadas, incluir um modificador de acesso como a palavra-chave `private` gera o erro do compilador CS0106, "O modificador 'private' não é válido para este item".

O exemplo a seguir define uma função local chamada `AppendPathSeparator` que é privada para um método chamado `GetText`:

C#

```
private static string GetText(string path, string filename)
{
    var reader = File.OpenText(${AppendPathSeparator(path)}{filename});
    var text = reader.ReadToEnd();
    return text;

    string AppendPathSeparator(string filepath)
    {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
    }
}
```

A partir do C# 9.0, você pode aplicar atributos a uma função local, seus parâmetros e parâmetros de tipo, como mostra o exemplo a seguir:

C#

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }
}
```

```

        }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}

```

O exemplo anterior usa um [atributo especial](#) para ajudar o compilador na análise estática em um contexto anulável.

## Funções locais e exceções

Um dos recursos úteis de funções locais é que elas podem permitir que exceções sejam apresentadas imediatamente. Para métodos de iteradores, as exceções são apresentadas somente quando a sequência retornada é enumerada e não quando o iterador é recuperado. Para métodos assíncronos, as exceções geradas em um método assíncrono são observadas quando a tarefa retornada é esperada.

O exemplo a seguir define um método `OddSequence` que enumera números ímpares em um intervalo especificado. Já que ele passa um número maior que 100 para o método enumerador `OddSequence`, o método gera uma [ArgumentOutOfRangeException](#). Assim como demonstrado pela saída do exemplo, a exceção é apresentada somente quando você itera os números e não quando você recupera o enumerador.

C#

```

public class IteratorWithoutLocalExample
{
    public static void Main()
    {
        IEnumerable<int> xs = OddSequence(50, 110);
        Console.WriteLine("Retrieved enumerator...");

        foreach (var x in xs) // line 11
        {
            Console.Write($"{x} ");
        }
    }

    public static IEnumerable<int> OddSequence(int start, int end)
    {
        if (start < 0 || start > 99)
            throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
        if (end > 100)
            throw new ArgumentOutOfRangeException(nameof(end), "end must be

```

```

less than or equal to 100.");
    if (start >= end)
        throw new ArgumentException("start must be less than end.");

    for (int i = start; i <= end; i++)
    {
        if (i % 2 == 1)
            yield return i;
    }
}

// The example displays the output like this:
//
//      Retrieved enumerator...
//      Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//      at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32
end)+MoveNext() in IteratorWithoutLocal.cs:line 22
//      at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line
11

```

Se você colocar a lógica do iterador em uma função local, as exceções de validação de argumento serão geradas quando você recuperar o enumerador, como mostra o exemplo a seguir:

C#

```

public class IteratorWithLocalExample
{
    public static void Main()
    {
        IEnumerable<int> xs = OddSequence(50, 110); // line 8
        Console.WriteLine("Retrieved enumerator...");

        foreach (var x in xs)
        {
            Console.Write($"{x} ");
        }
    }

    public static IEnumerable<int> OddSequence(int start, int end)
    {
        if (start < 0 || start > 99)
            throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
        if (end > 100)
            throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
        if (start >= end)
            throw new ArgumentException("start must be less than end.");

        return GetOddSequenceEnumerator();
    }
}

```

```

IEnumarable<int> GetOddSequenceEnumerator()
{
    for (int i = start; i <= end; i++)
    {
        if (i % 2 == 1)
            yield return i;
    }
}

// The example displays the output like this:
//
//      Unhandled exception. System.ArgumentOutOfRangeException: end must be
//      less than or equal to 100. (Parameter 'end')
//      at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in
//      IteratorWithLocal.cs:line 22
//      at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8

```

## Funções locais vs. expressões lambda

À primeira vista, funções locais e [expressões lambda](#) são muito semelhantes. Em muitos casos, a escolha entre usar expressões lambda e funções locais é uma [questão de estilo e preferência pessoal](#). No entanto, há diferenças reais nos casos em que você pode usar uma ou outra, e é importante conhecer essas diferenças.

Examinaremos as diferenças entre a função local e as implementações de expressão lambda do algoritmo fatorial. Esta é a versão que usa uma função local:

```
C#
public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2
        ? 1
        : number * nthFactorial(number - 1);
}
```

Esta versão usa expressões lambda:

```
C#
public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);
```

```
nthFactorial = number => number < 2
    ? 1
    : number * nthFactorial(number - 1);

    return nthFactorial(n);
}
```

## Nomenclatura

As funções locais são explicitamente nomeadas como métodos. As expressões lambda são métodos anônimos e precisam ser atribuídas a variáveis de um tipo `delegate`, normalmente os tipos `Action` ou `Func`. Quando você declara uma função local, o processo é como gravar um método normal. Você declara um tipo de retorno e uma assinatura de função.

## Assinaturas de função e tipos de expressão lambda

As expressões lambda dependem do tipo da variável `Action`/`Func` atribuída para determinar o argumento e os tipos de retorno. Em funções locais, como a sintaxe é muito parecida com gravar um método normal, os tipos de argumento e o tipo de retorno já fazem parte da declaração de função.

A partir do C# 10, algumas expressões lambda têm um *tipo natural*, o que permite que o compilador infira o tipo de retorno e os tipos de parâmetro da expressão lambda.

## Atribuição definida

As expressões lambda são objetos declarados e atribuídos em tempo de execução. Para que uma expressão lambda seja usada, ela precisa ser atribuída de maneira definitiva: a variável `Action`/`Func` à qual ela será atribuída deve ser declarada e a expressão lambda atribuída a ela. Observe que `LambdaFactorial` deve declarar e inicializar a expressão lambda `nthFactorial` antes de defini-la. Não fazer isso resulta em um erro em tempo de compilação para referenciar `nthFactorial` antes de atribuí-lo.

As funções locais são definidas em tempo de compilação. Como elas não são atribuídas a variáveis, podem ser referenciadas em qualquer local de código **em que esteja no escopo**. Em nosso primeiro exemplo `LocalFunctionFactorial`, podemos declarar nossa função local acima ou abaixo da instrução `return` e não disparar erros do compilador.

Essas diferenças significam que os algoritmos recursivos são mais fáceis de criar usando funções locais. Você pode declarar e definir uma função local que chame a si mesma. As

expressões lambda devem ser declaradas e atribuídas a um valor padrão antes que possam ser reatribuídas a um corpo que referencie a mesma expressão lambda.

## Implementação como delegado

As expressões lambda são convertidas em delegados quando declaradas. As funções locais são mais flexíveis, pois podem ser gravadas como método tradicional *ou* como delegado. As funções locais só são convertidas em delegados quando *usadas* como delegado.

Se você declarar uma função local e só referenciá-la ao chamá-la como um método, ela não será convertida em um delegado.

## Captura de variável

As regras de [atribuição definitiva](#) também afetam as variáveis capturadas pela função local ou pela expressão lambda. O compilador pode executar uma análise estática, que permite que as funções locais atribuam de maneira definitiva as variáveis capturadas no escopo delimitador. Considere este exemplo:

```
C#  
  
int M()  
{  
    int y;  
    LocalFunction();  
    return y;  
  
    void LocalFunction() => y = 0;  
}
```

O compilador pode determinar que `LocalFunction` definitivamente atribua `y` quando chamada. Como a `LocalFunction` é chamada antes da instrução `return`, `y` é atribuído definitivamente na instrução `return`.

Observe que quando uma função local captura variáveis no escopo delimitador, a função local é implementada como tipo de delegado.

## Alocações de heap

Dependendo do uso, as funções locais podem evitar as alocações de heap que são sempre necessárias nas expressões lambda. Se uma função local nunca é convertida em um delegado, e nenhuma das variáveis capturadas pela função local é capturada por

outras lambdas ou funções locais convertidas em delegados, o compilador pode evitar alocações de heap.

Considere este exemplo assíncrono:

C#

```
public async Task<string> PerformLongRunningWorkLambda(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    Func<Task<string>> longRunningWorkImplementation = async () =>
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    };

    return await longRunningWorkImplementation();
}
```

O fechamento desta expressão lambda contém as variáveis `address`, `index` e `name`. No caso de funções locais, o objeto que implementa o encerramento pode ser um tipo `struct`. Esse tipo de struct seria passado por referência à função local. Essa diferença na implementação poderia economizar em uma alocação.

A instanciação necessária para expressões lambda ocasiona alocações adicionais de memória, tornando-se um fator de desempenho em caminhos de código com tempo crítico. As funções locais não incorrem nessa sobrecarga. No exemplo acima, a versão das funções locais tem duas alocações a menos que a versão da expressão lambda.

Se você sabe que a função local não será convertida em um delegado e nenhuma das variáveis capturadas por ela será capturada por outras lambdas ou funções locais convertidas em delegados, você pode garantir que a função local evite ser alocada no heap, declarando-a como função local `static`.

 Dica

Habilite a regra de estilo de código do .NETIDE0062, para garantir que as funções locais sejam sempre marcadas como `static`.

### ① Observação

A função local equivalente desse método também usa uma classe para o fechamento. O fechamento de uma função local ser implementado como um `class` ou como um `struct`, trata-se de um detalhe de implementação. Uma função local pode usar um `struct`, enquanto uma lambda sempre usará um `class`.

C#

```
public async Task<string> PerformLongRunningWork(string address, int index,
string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    return await longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    }
}
```

## Uso da palavra-chave `yield`

Uma vantagem final não demonstrada neste exemplo é que as funções locais podem ser implementadas como iteradores, usando a sintaxe `yield return` para produzir uma sequência de valores.

C#

```
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)
{
```

```
if (!input.Any())
{
    throw new ArgumentException("There are no items to convert to
lowercase.");
}

return LowercaseIterator();

IEnumerable<string> LowercaseIterator()
{
    foreach (var output in input.Select(item => item.ToLower()))
    {
        yield return output;
    }
}
```

A instrução `yield return` não é permitida em expressões lambda, confira [erro do compilador CS1621](#).

Embora as funções locais possam parecer redundantes para expressões lambda, elas realmente têm finalidades e usos diferentes. As funções locais são mais eficientes para quando você deseja escrever uma função que é chamada apenas do contexto de outro método.

## Confira também

- Usar função local em vez do lambda (regra de estilo IDE0039)
- Métodos

# Instruções de declaração

Artigo • 23/06/2023

Uma instrução de declaração declara uma nova variável local, constante local ou [variável de referência local](#). Para declarar uma variável local, especifique seu tipo e forneça seu nome. Você pode declarar várias variáveis do mesmo tipo em uma instrução, como mostra o seguinte exemplo:

C#

```
string greeting;
int a, b, c;
List<double> xs;
```

Em uma instrução de declaração, você também pode inicializar uma variável com seu valor inicial:

C#

```
string greeting = "Hello";
int a = 3, b = 2, c = a + b;
List<double> xs = new();
```

Os exemplos anteriores especificam explicitamente o tipo de uma variável. Você também pode deixar o compilador inferir o tipo de uma variável de sua expressão de inicialização. Para fazer isso, use a palavra-chave `var` em vez do nome de um tipo. Para saber mais, confira a seção [Variáveis locais de tipo implícito](#).

Para declarar uma constante local, use a [palavra-chave const](#), como mostra o seguinte exemplo:

C#

```
const string Greeting = "Hello";
const double MinLimit = -10.0, MaxLimit = -MinLimit;
```

Ao declarar uma constante local, você também deve inicializá-la.

Para obter informações sobre variáveis de referência local, confira a seção [Variáveis de referência](#).

## Variáveis locais tipadas implicitamente

Ao declarar uma variável local, você pode deixar o compilador inferir o tipo da variável da expressão de inicialização. Para fazer isso, use a palavra-chave `var` em vez do nome de um tipo:

C#

```
var greeting = "Hello";
Console.WriteLine(greeting.GetType()); // output: System.String

var a = 32;
Console.WriteLine(a.GetType()); // output: System.Int32

var xs = new List<double>();
Console.WriteLine(xs.GetType()); // output:
System.Collections.Generic.List`1[System.Double]
```

Como mostra o exemplo anterior, variáveis locais de tipo implícito são fortemente tipadas.

### ① Observação

Quando você usa `var` no contexto com reconhecimento anulável habilitado e o tipo de uma expressão de inicialização é um tipo de referência, o compilador sempre infere um tipo de referência anulável mesmo que o tipo de uma expressão de inicialização não seja anulável.

Um uso comum de `var` é com uma expressão de invocação de construtor. O uso de `var` permite não repetir um nome de tipo em uma declaração de variável e instanciação de objeto, como mostra o exemplo a seguir:

C#

```
var xs = new List<int>();
```

A partir do C# 9.0, você pode usar uma expressão new do tipo destino como alternativa:

C#

```
List<int> xs = new();
List<int>? ys = new();
```

Ao trabalhar com tipos anônimos, você deve usar variáveis locais tipadas implicitamente. O exemplo a seguir mostra uma expressão de consulta que usa um tipo

anônimo para conter o nome e o número de telefone de um cliente:

C#

```
var fromPhoenix = from cust in customers
                  where cust.City == "Phoenix"
                  select new { cust.Name, cust.Phone };

foreach (var customer in fromPhoenix)
{
    Console.WriteLine($"Name={customer.Name}, Phone={customer.Phone}");
}
```

No exemplo anterior, você não pode especificar explicitamente o tipo da variável `fromPhoenix`. O tipo é `IEnumerable<T>`, mas nesse caso `T` é um tipo anônimo e você não pode fornecer seu nome. É por isso que você precisa usar `var`. Pelo mesmo motivo, você deve usar `var` ao declarar a variável de iteração `customer` na instrução `foreach`.

Para obter mais informações sobre variáveis locais tipadas implicitamente, confira [Variáveis locais tipadas implicitamente](#).

Na correspondência de padrões, a palavra-chave `var` é usada em um [var padrão](#).

## Variáveis de referência

Ao declarar uma variável local e adicionar a palavra-chave `ref` antes do tipo da variável, você declara uma *variável de referência* ou um local `ref`:

C#

```
ref int alias = ref variable;
```

Uma variável de referência é uma variável que se refere a outra variável, que é chamada de *referenciante*. Ou seja, uma variável de referência é um *alias* para seu referenciante. Quando você atribui um valor a uma variável de referência, esse valor é atribuído ao referenciador. Quando você lê o valor de uma variável de referência, o valor do referenciante é retornado. O exemplo a seguir demonstra esse comportamento:

C#

```
int a = 1;
ref int alias = ref a;
Console.WriteLine($"{(a, alias) is ({a}, {alias})}"); // output: (a, alias)
is (1, 1)
```

```
a = 2;
Console.WriteLine($"(a, alias) is ({a}, {alias})"); // output: (a, alias)
is (2, 2)

alias = 3;
Console.WriteLine($"(a, alias) is ({a}, {alias})"); // output: (a, alias)
is (3, 3)
```

Use o `ref` operador de atribuição `= ref` para alterar o referenciante de uma variável de referência, como mostra o exemplo a seguir:

C#

```
void Display(int[] s) => Console.WriteLine(string.Join(" ", s));

int[] xs = { 0, 0, 0 };
Display(xs);

ref int element = ref xs[0];
element = 1;
Display(xs);

element = ref xs[^1];
element = 3;
Display(xs);
// Output:
// 0 0 0
// 1 0 0
// 1 0 3
```

No exemplo anterior, a variável de referência `element` é inicializada como um alias para o primeiro elemento de matriz. Ela é reatribuída `ref` para fazer referência ao último elemento de matriz.

Você pode definir uma variável local `ref readonly`. Você não pode atribuir um valor a uma variável `ref readonly`. No entanto, você pode reatribuir `ref` essa variável de referência, como mostra o exemplo a seguir:

C#

```
int[] xs = { 1, 2, 3 };

ref readonly int element = ref xs[0];
// element = 100; error CS0131: The left-hand side of an assignment must be
// a variable, property or indexer
Console.WriteLine(element); // output: 1
```

```
element = ref xs[^1];
Console.WriteLine(element); // output: 3
```

Você pode atribuir um [retorno de referência](#) a uma variável de referência, como mostra o exemplo a seguir:

C#

```
using System;

public class NumberStore
{
    private readonly int[] numbers = { 1, 30, 7, 1557, 381, 63, 1027, 2550,
511, 1023 };

    public ref int GetReferenceToMax()
    {
        ref int max = ref numbers[0];
        for (int i = 1; i < numbers.Length; i++)
        {
            if (numbers[i] > max)
            {
                max = ref numbers[i];
            }
        }
        return ref max;
    }

    public override string ToString() => string.Join(" ", numbers);
}

public static class ReferenceReturnExample
{
    public static void Run()
    {
        var store = new NumberStore();
        Console.WriteLine($"Original sequence: {store.ToString()}");

        ref int max = ref store.GetReferenceToMax();
        max = 0;
        Console.WriteLine($"Updated sequence: {store.ToString()}");
        // Output:
        // Original sequence: 1 30 7 1557 381 63 1027 2550 511 1023
        // Updated sequence: 1 30 7 1557 381 63 1027 0 511 1023
    }
}
```

No exemplo anterior, o método `GetReferenceToMax` é um método *returns-by-ref*. Ele não retorna o valor máximo em si, mas um retorno de referência que é um alias para o elemento de matriz que contém o valor máximo. O método `Run` atribui um retorno de

referência à variável de referência `max`. Em seguida, atribuindo a `max`, ele atualiza o armazenamento interno da instância `store`. Você também pode definir um método `ref readonly`. Os chamadores de um método `ref readonly` não podem atribuir um valor ao seu retorno de referência.

A variável de iteração da instrução `foreach` pode ser uma variável de referência. Para obter mais informações, consulte a seção sobre a [instrução foreach](#) do artigo [Instruções de iteração](#).

Em cenários críticos de desempenho, o uso de variáveis de referência e retornos pode aumentar o desempenho evitando operações de cópia potencialmente caras.

O compilador garante que uma variável de referência não sobreviva ao seu referenciador e permaneça válida durante todo o tempo de vida. Para saber mais, confira a seção [Contextos seguros de referência](#) da [Especificação da linguagem C#](#).

Para obter informações sobre os campos `ref`, confira a seção [refcampos](#) do artigo [reftipos de estrutura](#).

## ref com escopo

A palavra-chave contextual `scoped` restringe o tempo de vida de um valor. O modificador `scoped` restringe o tempo de vida *ref-safe-to-escape* ou *safe-to-escape*, respectivamente, ao método atual. Efetivamente, adicionar o modificador `scoped` declara que seu código não estenderá o tempo de vida da variável.

Você pode aplicar `scoped` a um parâmetro ou variável local. O modificador `scoped` pode ser aplicado a parâmetros e locais quando o tipo for um `ref struct`. Caso contrário, o modificador `scoped` poderá ser aplicado somente a [variáveis de referência](#) locais. Isso inclui variáveis locais declaradas com o modificador `ref` e os parâmetros declarados com os modificadores `in`, `ref` ou `out`.

O modificador `scoped` é adicionado implicitamente a `this` em métodos declarados em parâmetros `struct`, `out` e `ref` quando o tipo for um `ref struct`.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Instruções de declaração](#)

- [Variáveis de referência e retornos](#)

Para obter mais informações sobre o modificador `scoped`, consulte a nota de proposta [Aprimoramentos de struct de baixo nível](#).

## Confira também

- [Referência de C#](#)
- [Inicializadores de objeto e de coleção](#)
- [ref keyword](#)
- [Reducir alocações de memória usando novos recursos do C#](#)
- [preferências 'var' \(regras de estilo IDE0007 e IDE0008\)](#)

# Parâmetros de método (Referência de C#)

Artigo • 07/04/2023

No C#, argumentos podem ser passados para parâmetros por valor ou por referência. Lembre-se de que os tipos C# podem ser tipos de referência (`class`) ou tipos de valor (`struct`):

- *Aprovação por valor* significa aprovar uma cópia da variável para o método.
- *Aprovação por referência* significa aprovar o acesso à variável para o método.
- Uma variável de um *tipo de referência* contém uma referência aos seus dados.
- Uma variável de um *tipo de valor* contém seu valor diretamente.

Como um struct é um *tipo de valor*, ao [passar um struct por valor](#) a um método, esse método receberá e operará em uma cópia do argumento do struct. O método não tem acesso ao struct original no método de chamada e, portanto, não é possível alterá-lo de forma alguma. O método pode alterar somente a cópia.

Uma instância de classe é um *tipo de referência* e não é um tipo de valor. Quando [um tipo de referência é passado por valor](#) a um método, esse método receberá uma cópia da referência para a instância da classe. Ou seja, o método chamado recebe uma cópia do endereço da instância e o método de chamada retém o endereço original da instância. A instância de classe no método de chamada tem um endereço, o parâmetro do método chamado tem uma cópia do endereço e os dois endereços se referem ao mesmo objeto. Como o parâmetro contém apenas uma cópia do endereço, o método chamado não pode alterar o endereço da instância de classe no método de chamada. No entanto, o método chamado pode usar a cópia do endereço para acessar os membros de classe que o endereço original e a cópia do endereço referenciam. Se o método chamado alterar um membro de classe, a instância da classe original no método de chamada também será alterada.

O resultado do exemplo a seguir ilustra a diferença. O valor do campo `willIChange` da instância da classe foi alterado pela chamada ao método `ClassTaker`, pois o método usa o endereço no parâmetro para localizar o campo especificado da instância da classe. O campo `willIChange` do struct no método de chamada não foi alterado pela chamada ao método `StructTaker`, pois o valor do argumento é uma cópia do próprio struct e não uma cópia de seu endereço. `StructTaker` altera a cópia e a cópia será perdida quando a chamada para `StructTaker` for concluída.

```

class TheClass
{
    public string? willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    public static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Class field = Changed
   Struct field = Not Changed
*/

```

A forma como um argumento é aprovado e se é um tipo de referência ou tipo de valor controla quais modificações feitas no argumento são visíveis pelo chamador.

## Aprovar um tipo de valor por valor

Quando se aprova um tipo de valor por valor:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **não serão** visíveis pelo chamador.

O exemplo a seguir demonstra a passagem de parâmetros de tipo de valor por valor. A variável `n` é passada por valor para o método `SquareIt`. Quaisquer alterações que ocorrem dentro do método não têm efeito sobre o valor original da variável.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(n); // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(int x)
// The parameter x is passed by value.
// Changes to x will not affect the original value of n.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 5
*/
```

A variável `n` é um tipo de valor. Ela contém seus dados, o valor `5`. Quando `SquareIt` é invocado, o conteúdo de `n` é copiado para o parâmetro `x`, que é elevado ao quadrado dentro do método. No entanto, em `Main`, o valor de `n` é o mesmo depois de chamar o método `SquareIt` como era antes. A alteração que ocorre dentro do método afeta apenas a variável local `x`.

## Aprovar um tipo de valor por referência

Quando se aprova um tipo de valor por referência:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir é o mesmo que o exemplo anterior, exceto que o argumento é passado como um parâmetro `ref`. O valor do argumento subjacente, `n`, é alterado quando `x` é alterado no método.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(ref n); // Passing the variable by reference.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(ref int x)
// The parameter x is passed by reference.
// Changes to x will affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 25
*/
```

Neste exemplo, não é o valor de `n` que é passado; em vez disso, é passada uma referência a `n`. O parâmetro `x` não é um `int`; é uma referência a um `int`, nesse caso, uma referência a `n`. Portanto, quando `x` é elevado ao quadrado dentro do método, o que é realmente elevado ao quadrado é aquilo a que `x` se refere, `n`.

## Aprovar um tipo de referência por valor

Quando se aprova um tipo de *referênciapor valor*:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.

- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir demonstra a passagem de um parâmetro de tipo de referência, `arr`, por valor, para um método, `Change`. Como o parâmetro é uma referência a `arr`, é possível alterar os valores dos elementos da matriz. No entanto, a tentativa de reatribuir o parâmetro para um local diferente de memória só funciona dentro do método e não afeta a variável original, `arr`.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(int[] pArray)
{
    pArray[0] = 888; // This change affects the original element.
    pArray = new int[5] { -3, -1, -2, -3, -4 }; // This change is local.
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: 888
*/
```

No exemplo anterior, a matriz, `arr`, que é um tipo de referência, é passada para o método sem o parâmetro `ref`. Nesse caso, uma cópia da referência, que aponta para `arr`, é passada para o método. A saída mostra que é possível para o método alterar o conteúdo de um elemento de matriz, nesse caso de `1` para `888`. No entanto, alocar uma nova parte da memória usando o operador `new` dentro do método `Change` faz a variável `pArray` referenciar uma nova matriz. Portanto, quaisquer alterações realizadas depois disso não afetarão a matriz original, `arr`, criada dentro de `Main`. Na verdade, duas matrizes são criadas neste exemplo, uma dentro de `Main` e outra dentro do método `Change`.

## Aprovar um tipo de referência por referência

Quando se aprova um tipo de *referênciapor referência*:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **ficarão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir é o mesmo que o exemplo anterior, exceto que a palavra-chave `ref` é adicionada ao cabeçalho e à chamada do método. Quaisquer alterações que ocorrem no método afetam a variável original no programa de chamada.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(ref int[] pArray)
{
    // Both of the following changes will affect the original variables:
    pArray[0] = 888;
    pArray = new int[5] { -3, -1, -2, -3, -4 };
    System.Console.WriteLine("Inside the method, the first element is: {0}",
    pArray[0]);
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: -3
*/
```

Todas as alterações que ocorrem dentro do método afetam a matriz original em `Main`. Na verdade, a matriz original é realocada usando o operador `new`. Portanto, depois de chamar o método `Change`, qualquer referência a `arr` aponta para a matriz de cinco elementos, criada no método `Change`.

## Escopo de referências e valores

Os métodos podem armazenar os valores de parâmetros em campos. Quando os parâmetros são aprovados por valor, o método é sempre seguro. Os valores são copiados e os tipos de referência podem ser acessados quando armazenados em um campo. Para aprovar parâmetros por referência com segurança, é necessário que o compilador defina quando é seguro atribuir uma referência a uma nova variável. Para

cada expressão, o compilador define um *escopo* que vincula o acesso a uma expressão ou variável. O compilador usa dois escopos: *safe\_to\_escape* e *ref\_safe\_to\_escape*.

- *safe\_to\_escape* define o escopo em que qualquer expressão pode ser acessada com segurança.
- *ref\_safe\_to\_escape* define o escopo em que uma *referência* a qualquer expressão pode ser acessada ou modificada com segurança.

Informalmente, você pode considerar esses escopos como o mecanismo para garantir que seu código nunca acesse ou modifique uma referência que não seja mais válida. Uma referência é válida desde que se refira a um objeto ou struct válido. O escopo *safe\_to\_escape* define quando uma variável pode ser atribuída ou reatribuída. O escopo *ref\_safe\_to\_escape* define quando uma variável *pode ser atribuída* ou reatribuída por *ref*. A atribuição concede a uma variável um novo valor. A *atribuição de referência* indica para a variável que ela deve *se referir a* um local de armazenamento diferente.

## Modificadores

Os parâmetros declarados para um método sem *in*, *ref* nem *out* são passados para o método chamado pelo valor. Os modificadores *ref*, *in* e *out* diferem nas regras de atribuição:

- O argumento para um parâmetro *ref* deve ser atribuído de maneira definitiva. O método chamado pode reatribuir esse parâmetro.
- O argumento para um parâmetro *in* deve ser atribuído de maneira definitiva. O método chamado não pode reatribuir esse parâmetro.
- O argumento para um parâmetro *out* não precisa ser atribuído de maneira definitiva. O método chamado deve atribuir o parâmetro.

Esta seção descreve as palavras-chave que podem ser usadas ao declarar parâmetros de método:

- *params* especifica que esse parâmetro pode receber um número variável de argumentos.
- *in* especifica que esse parâmetro é passado por referência, mas é lido apenas pelo método chamado.
- *ref* especifica que esse parâmetro é passado por referência e pode ser lido ou gravado pelo método chamado.
- *out* especifica que esse parâmetro é passado por referência e é gravado pelo método chamado.

## Confira também

- [Referência de C#](#)
- [Palavras-chave do C#](#)
- Listas de argumentos na [Especificação de Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# Parâmetros de método (Referência de C#)

Artigo • 07/04/2023

No C#, argumentos podem ser passados para parâmetros por valor ou por referência. Lembre-se de que os tipos C# podem ser tipos de referência (`class`) ou tipos de valor (`struct`):

- *Aprovação por valor* significa aprovar uma cópia da variável para o método.
- *Aprovação por referência* significa aprovar o acesso à variável para o método.
- Uma variável de um *tipo de referência* contém uma referência aos seus dados.
- Uma variável de um *tipo de valor* contém seu valor diretamente.

Como um struct é um *tipo de valor*, ao [passar um struct por valor](#) a um método, esse método receberá e operará em uma cópia do argumento do struct. O método não tem acesso ao struct original no método de chamada e, portanto, não é possível alterá-lo de forma alguma. O método pode alterar somente a cópia.

Uma instância de classe é um *tipo de referência* e não é um tipo de valor. Quando [um tipo de referência é passado por valor](#) a um método, esse método receberá uma cópia da referência para a instância da classe. Ou seja, o método chamado recebe uma cópia do endereço da instância e o método de chamada retém o endereço original da instância. A instância de classe no método de chamada tem um endereço, o parâmetro do método chamado tem uma cópia do endereço e os dois endereços se referem ao mesmo objeto. Como o parâmetro contém apenas uma cópia do endereço, o método chamado não pode alterar o endereço da instância de classe no método de chamada. No entanto, o método chamado pode usar a cópia do endereço para acessar os membros de classe que o endereço original e a cópia do endereço referenciam. Se o método chamado alterar um membro de classe, a instância da classe original no método de chamada também será alterada.

O resultado do exemplo a seguir ilustra a diferença. O valor do campo `willIChange` da instância da classe foi alterado pela chamada ao método `ClassTaker`, pois o método usa o endereço no parâmetro para localizar o campo especificado da instância da classe. O campo `willIChange` do struct no método de chamada não foi alterado pela chamada ao método `StructTaker`, pois o valor do argumento é uma cópia do próprio struct e não uma cópia de seu endereço. `StructTaker` altera a cópia e a cópia será perdida quando a chamada para `StructTaker` for concluída.

```

class TheClass
{
    public string? willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    public static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Class field = Changed
   Struct field = Not Changed
*/

```

A forma como um argumento é aprovado e se é um tipo de referência ou tipo de valor controla quais modificações feitas no argumento são visíveis pelo chamador.

## Aprovar um tipo de valor por valor

Quando se aprova um tipo de valor por valor:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **não serão** visíveis pelo chamador.

O exemplo a seguir demonstra a passagem de parâmetros de tipo de valor por valor. A variável `n` é passada por valor para o método `SquareIt`. Quaisquer alterações que ocorrem dentro do método não têm efeito sobre o valor original da variável.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(n); // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(int x)
// The parameter x is passed by value.
// Changes to x will not affect the original value of n.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 5
*/
```

A variável `n` é um tipo de valor. Ela contém seus dados, o valor `5`. Quando `SquareIt` é invocado, o conteúdo de `n` é copiado para o parâmetro `x`, que é elevado ao quadrado dentro do método. No entanto, em `Main`, o valor de `n` é o mesmo depois de chamar o método `SquareIt` como era antes. A alteração que ocorre dentro do método afeta apenas a variável local `x`.

## Aprovar um tipo de valor por referência

Quando se aprova um tipo de valor por referência:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir é o mesmo que o exemplo anterior, exceto que o argumento é passado como um parâmetro `ref`. O valor do argumento subjacente, `n`, é alterado quando `x` é alterado no método.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(ref n); // Passing the variable by reference.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(ref int x)
// The parameter x is passed by reference.
// Changes to x will affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 25
*/
```

Neste exemplo, não é o valor de `n` que é passado; em vez disso, é passada uma referência a `n`. O parâmetro `x` não é um `int`; é uma referência a um `int`, nesse caso, uma referência a `n`. Portanto, quando `x` é elevado ao quadrado dentro do método, o que é realmente elevado ao quadrado é aquilo a que `x` se refere, `n`.

## Aprovar um tipo de referência por valor

Quando se aprova um tipo de *referênciapor valor*:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.

- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir demonstra a passagem de um parâmetro de tipo de referência, `arr`, por valor, para um método, `Change`. Como o parâmetro é uma referência a `arr`, é possível alterar os valores dos elementos da matriz. No entanto, a tentativa de reatribuir o parâmetro para um local diferente de memória só funciona dentro do método e não afeta a variável original, `arr`.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(int[] pArray)
{
    pArray[0] = 888; // This change affects the original element.
    pArray = new int[5] { -3, -1, -2, -3, -4 }; // This change is local.
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: 888
*/
```

No exemplo anterior, a matriz, `arr`, que é um tipo de referência, é passada para o método sem o parâmetro `ref`. Nesse caso, uma cópia da referência, que aponta para `arr`, é passada para o método. A saída mostra que é possível para o método alterar o conteúdo de um elemento de matriz, nesse caso de `1` para `888`. No entanto, alocar uma nova parte da memória usando o operador `new` dentro do método `Change` faz a variável `pArray` referenciar uma nova matriz. Portanto, quaisquer alterações realizadas depois disso não afetarão a matriz original, `arr`, criada dentro de `Main`. Na verdade, duas matrizes são criadas neste exemplo, uma dentro de `Main` e outra dentro do método `Change`.

## Aprovar um tipo de referência por referência

Quando se aprova um tipo de *referênciapor referência*:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **ficarão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir é o mesmo que o exemplo anterior, exceto que a palavra-chave `ref` é adicionada ao cabeçalho e à chamada do método. Quaisquer alterações que ocorrem no método afetam a variável original no programa de chamada.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(ref int[] pArray)
{
    // Both of the following changes will affect the original variables:
    pArray[0] = 888;
    pArray = new int[5] { -3, -1, -2, -3, -4 };
    System.Console.WriteLine("Inside the method, the first element is: {0}",
    pArray[0]);
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: -3
*/
```

Todas as alterações que ocorrem dentro do método afetam a matriz original em `Main`. Na verdade, a matriz original é realocada usando o operador `new`. Portanto, depois de chamar o método `Change`, qualquer referência a `arr` aponta para a matriz de cinco elementos, criada no método `Change`.

## Escopo de referências e valores

Os métodos podem armazenar os valores de parâmetros em campos. Quando os parâmetros são aprovados por valor, o método é sempre seguro. Os valores são copiados e os tipos de referência podem ser acessados quando armazenados em um campo. Para aprovar parâmetros por referência com segurança, é necessário que o compilador defina quando é seguro atribuir uma referência a uma nova variável. Para

cada expressão, o compilador define um *escopo* que vincula o acesso a uma expressão ou variável. O compilador usa dois escopos: *safe\_to\_escape* e *ref\_safe\_to\_escape*.

- *safe\_to\_escape* define o escopo em que qualquer expressão pode ser acessada com segurança.
- *ref\_safe\_to\_escape* define o escopo em que uma *referência* a qualquer expressão pode ser acessada ou modificada com segurança.

Informalmente, você pode considerar esses escopos como o mecanismo para garantir que seu código nunca acesse ou modifique uma referência que não seja mais válida. Uma referência é válida desde que se refira a um objeto ou struct válido. O escopo *safe\_to\_escape* define quando uma variável pode ser atribuída ou reatribuída. O escopo *ref\_safe\_to\_escape* define quando uma variável *pode ser atribuída* ou reatribuída por *ref*. A atribuição concede a uma variável um novo valor. A *atribuição de referência* indica para a variável que ela deve *se referir a* um local de armazenamento diferente.

## Modificadores

Os parâmetros declarados para um método sem *in*, *ref* nem *out* são passados para o método chamado pelo valor. Os modificadores *ref*, *in* e *out* diferem nas regras de atribuição:

- O argumento para um parâmetro *ref* deve ser atribuído de maneira definitiva. O método chamado pode reatribuir esse parâmetro.
- O argumento para um parâmetro *in* deve ser atribuído de maneira definitiva. O método chamado não pode reatribuir esse parâmetro.
- O argumento para um parâmetro *out* não precisa ser atribuído de maneira definitiva. O método chamado deve atribuir o parâmetro.

Esta seção descreve as palavras-chave que podem ser usadas ao declarar parâmetros de método:

- *params* especifica que esse parâmetro pode receber um número variável de argumentos.
- *in* especifica que esse parâmetro é passado por referência, mas é lido apenas pelo método chamado.
- *ref* especifica que esse parâmetro é passado por referência e pode ser lido ou gravado pelo método chamado.
- *out* especifica que esse parâmetro é passado por referência e é gravado pelo método chamado.

## Confira também

- [Referência de C#](#)
- [Palavras-chave do C#](#)
- Listas de argumentos na [Especificação de Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# Parâmetros de método (Referência de C#)

Artigo • 07/04/2023

No C#, argumentos podem ser passados para parâmetros por valor ou por referência. Lembre-se de que os tipos C# podem ser tipos de referência (`class`) ou tipos de valor (`struct`):

- *Aprovação por valor* significa aprovar uma cópia da variável para o método.
- *Aprovação por referência* significa aprovar o acesso à variável para o método.
- Uma variável de um *tipo de referência* contém uma referência aos seus dados.
- Uma variável de um *tipo de valor* contém seu valor diretamente.

Como um struct é um *tipo de valor*, ao [passar um struct por valor](#) a um método, esse método receberá e operará em uma cópia do argumento do struct. O método não tem acesso ao struct original no método de chamada e, portanto, não é possível alterá-lo de forma alguma. O método pode alterar somente a cópia.

Uma instância de classe é um *tipo de referência* e não é um tipo de valor. Quando [um tipo de referência é passado por valor](#) a um método, esse método receberá uma cópia da referência para a instância da classe. Ou seja, o método chamado recebe uma cópia do endereço da instância e o método de chamada retém o endereço original da instância. A instância de classe no método de chamada tem um endereço, o parâmetro do método chamado tem uma cópia do endereço e os dois endereços se referem ao mesmo objeto. Como o parâmetro contém apenas uma cópia do endereço, o método chamado não pode alterar o endereço da instância de classe no método de chamada. No entanto, o método chamado pode usar a cópia do endereço para acessar os membros de classe que o endereço original e a cópia do endereço referenciam. Se o método chamado alterar um membro de classe, a instância da classe original no método de chamada também será alterada.

O resultado do exemplo a seguir ilustra a diferença. O valor do campo `willIChange` da instância da classe foi alterado pela chamada ao método `ClassTaker`, pois o método usa o endereço no parâmetro para localizar o campo especificado da instância da classe. O campo `willIChange` do struct no método de chamada não foi alterado pela chamada ao método `StructTaker`, pois o valor do argumento é uma cópia do próprio struct e não uma cópia de seu endereço. `StructTaker` altera a cópia e a cópia será perdida quando a chamada para `StructTaker` for concluída.

```

class TheClass
{
    public string? willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    public static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Class field = Changed
   Struct field = Not Changed
*/

```

A forma como um argumento é aprovado e se é um tipo de referência ou tipo de valor controla quais modificações feitas no argumento são visíveis pelo chamador.

## Aprovar um tipo de valor por valor

Quando se aprova um tipo de valor por valor:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **não serão** visíveis pelo chamador.

O exemplo a seguir demonstra a passagem de parâmetros de tipo de valor por valor. A variável `n` é passada por valor para o método `SquareIt`. Quaisquer alterações que ocorrem dentro do método não têm efeito sobre o valor original da variável.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(n); // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(int x)
// The parameter x is passed by value.
// Changes to x will not affect the original value of n.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 5
*/
```

A variável `n` é um tipo de valor. Ela contém seus dados, o valor `5`. Quando `SquareIt` é invocado, o conteúdo de `n` é copiado para o parâmetro `x`, que é elevado ao quadrado dentro do método. No entanto, em `Main`, o valor de `n` é o mesmo depois de chamar o método `SquareIt` como era antes. A alteração que ocorre dentro do método afeta apenas a variável local `x`.

## Aprovar um tipo de valor por referência

Quando se aprova um tipo de valor por referência:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir é o mesmo que o exemplo anterior, exceto que o argumento é passado como um parâmetro `ref`. O valor do argumento subjacente, `n`, é alterado quando `x` é alterado no método.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(ref n); // Passing the variable by reference.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(ref int x)
// The parameter x is passed by reference.
// Changes to x will affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 25
*/
```

Neste exemplo, não é o valor de `n` que é passado; em vez disso, é passada uma referência a `n`. O parâmetro `x` não é um `int`; é uma referência a um `int`, nesse caso, uma referência a `n`. Portanto, quando `x` é elevado ao quadrado dentro do método, o que é realmente elevado ao quadrado é aquilo a que `x` se refere, `n`.

## Aprovar um tipo de referência por valor

Quando se aprova um tipo de *referênciapor valor*:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.

- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir demonstra a passagem de um parâmetro de tipo de referência, `arr`, por valor, para um método, `Change`. Como o parâmetro é uma referência a `arr`, é possível alterar os valores dos elementos da matriz. No entanto, a tentativa de reatribuir o parâmetro para um local diferente de memória só funciona dentro do método e não afeta a variável original, `arr`.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(int[] pArray)
{
    pArray[0] = 888; // This change affects the original element.
    pArray = new int[5] { -3, -1, -2, -3, -4 }; // This change is local.
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: 888
*/
```

No exemplo anterior, a matriz, `arr`, que é um tipo de referência, é passada para o método sem o parâmetro `ref`. Nesse caso, uma cópia da referência, que aponta para `arr`, é passada para o método. A saída mostra que é possível para o método alterar o conteúdo de um elemento de matriz, nesse caso de `1` para `888`. No entanto, alocar uma nova parte da memória usando o operador `new` dentro do método `Change` faz a variável `pArray` referenciar uma nova matriz. Portanto, quaisquer alterações realizadas depois disso não afetarão a matriz original, `arr`, criada dentro de `Main`. Na verdade, duas matrizes são criadas neste exemplo, uma dentro de `Main` e outra dentro do método `Change`.

## Aprovar um tipo de referência por referência

Quando se aprova um tipo de *referênciapor referência*:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **ficarão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir é o mesmo que o exemplo anterior, exceto que a palavra-chave `ref` é adicionada ao cabeçalho e à chamada do método. Quaisquer alterações que ocorrem no método afetam a variável original no programa de chamada.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(ref int[] pArray)
{
    // Both of the following changes will affect the original variables:
    pArray[0] = 888;
    pArray = new int[5] { -3, -1, -2, -3, -4 };
    System.Console.WriteLine("Inside the method, the first element is: {0}",
    pArray[0]);
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: -3
*/
```

Todas as alterações que ocorrem dentro do método afetam a matriz original em `Main`. Na verdade, a matriz original é realocada usando o operador `new`. Portanto, depois de chamar o método `Change`, qualquer referência a `arr` aponta para a matriz de cinco elementos, criada no método `Change`.

## Escopo de referências e valores

Os métodos podem armazenar os valores de parâmetros em campos. Quando os parâmetros são aprovados por valor, o método é sempre seguro. Os valores são copiados e os tipos de referência podem ser acessados quando armazenados em um campo. Para aprovar parâmetros por referência com segurança, é necessário que o compilador defina quando é seguro atribuir uma referência a uma nova variável. Para

cada expressão, o compilador define um *escopo* que vincula o acesso a uma expressão ou variável. O compilador usa dois escopos: *safe\_to\_escape* e *ref\_safe\_to\_escape*.

- *safe\_to\_escape* define o escopo em que qualquer expressão pode ser acessada com segurança.
- *ref\_safe\_to\_escape* define o escopo em que uma *referência* a qualquer expressão pode ser acessada ou modificada com segurança.

Informalmente, você pode considerar esses escopos como o mecanismo para garantir que seu código nunca acesse ou modifique uma referência que não seja mais válida. Uma referência é válida desde que se refira a um objeto ou struct válido. O escopo *safe\_to\_escape* define quando uma variável pode ser atribuída ou reatribuída. O escopo *ref\_safe\_to\_escape* define quando uma variável *pode ser atribuída* ou reatribuída por *ref*. A atribuição concede a uma variável um novo valor. A *atribuição de referência* indica para a variável que ela deve *se referir a* um local de armazenamento diferente.

## Modificadores

Os parâmetros declarados para um método sem *in*, *ref* nem *out* são passados para o método chamado pelo valor. Os modificadores *ref*, *in* e *out* diferem nas regras de atribuição:

- O argumento para um parâmetro *ref* deve ser atribuído de maneira definitiva. O método chamado pode reatribuir esse parâmetro.
- O argumento para um parâmetro *in* deve ser atribuído de maneira definitiva. O método chamado não pode reatribuir esse parâmetro.
- O argumento para um parâmetro *out* não precisa ser atribuído de maneira definitiva. O método chamado deve atribuir o parâmetro.

Esta seção descreve as palavras-chave que podem ser usadas ao declarar parâmetros de método:

- *params* especifica que esse parâmetro pode receber um número variável de argumentos.
- *in* especifica que esse parâmetro é passado por referência, mas é lido apenas pelo método chamado.
- *ref* especifica que esse parâmetro é passado por referência e pode ser lido ou gravado pelo método chamado.
- *out* especifica que esse parâmetro é passado por referência e é gravado pelo método chamado.

## Confira também

- [Referência de C#](#)
- [Palavras-chave do C#](#)
- Listas de argumentos na [Especificação de Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# Parâmetros de método (Referência de C#)

Artigo • 07/04/2023

No C#, argumentos podem ser passados para parâmetros por valor ou por referência. Lembre-se de que os tipos C# podem ser tipos de referência (`class`) ou tipos de valor (`struct`):

- *Aprovação por valor* significa aprovar uma cópia da variável para o método.
- *Aprovação por referência* significa aprovar o acesso à variável para o método.
- Uma variável de um *tipo de referência* contém uma referência aos seus dados.
- Uma variável de um *tipo de valor* contém seu valor diretamente.

Como um struct é um *tipo de valor*, ao [passar um struct por valor](#) a um método, esse método receberá e operará em uma cópia do argumento do struct. O método não tem acesso ao struct original no método de chamada e, portanto, não é possível alterá-lo de forma alguma. O método pode alterar somente a cópia.

Uma instância de classe é um *tipo de referência* e não é um tipo de valor. Quando [um tipo de referência é passado por valor](#) a um método, esse método receberá uma cópia da referência para a instância da classe. Ou seja, o método chamado recebe uma cópia do endereço da instância e o método de chamada retém o endereço original da instância. A instância de classe no método de chamada tem um endereço, o parâmetro do método chamado tem uma cópia do endereço e os dois endereços se referem ao mesmo objeto. Como o parâmetro contém apenas uma cópia do endereço, o método chamado não pode alterar o endereço da instância de classe no método de chamada. No entanto, o método chamado pode usar a cópia do endereço para acessar os membros de classe que o endereço original e a cópia do endereço referenciam. Se o método chamado alterar um membro de classe, a instância da classe original no método de chamada também será alterada.

O resultado do exemplo a seguir ilustra a diferença. O valor do campo `willIChange` da instância da classe foi alterado pela chamada ao método `ClassTaker`, pois o método usa o endereço no parâmetro para localizar o campo especificado da instância da classe. O campo `willIChange` do struct no método de chamada não foi alterado pela chamada ao método `StructTaker`, pois o valor do argumento é uma cópia do próprio struct e não uma cópia de seu endereço. `StructTaker` altera a cópia e a cópia será perdida quando a chamada para `StructTaker` for concluída.

```

class TheClass
{
    public string? willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    public static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Class field = Changed
   Struct field = Not Changed
*/

```

A forma como um argumento é aprovado e se é um tipo de referência ou tipo de valor controla quais modificações feitas no argumento são visíveis pelo chamador.

## Aprovar um tipo de valor por valor

Quando se aprova um tipo de valor por valor:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **não serão** visíveis pelo chamador.

O exemplo a seguir demonstra a passagem de parâmetros de tipo de valor por valor. A variável `n` é passada por valor para o método `SquareIt`. Quaisquer alterações que ocorrem dentro do método não têm efeito sobre o valor original da variável.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(n); // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(int x)
// The parameter x is passed by value.
// Changes to x will not affect the original value of n.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 5
*/
```

A variável `n` é um tipo de valor. Ela contém seus dados, o valor `5`. Quando `SquareIt` é invocado, o conteúdo de `n` é copiado para o parâmetro `x`, que é elevado ao quadrado dentro do método. No entanto, em `Main`, o valor de `n` é o mesmo depois de chamar o método `SquareIt` como era antes. A alteração que ocorre dentro do método afeta apenas a variável local `x`.

## Aprovar um tipo de valor por referência

Quando se aprova um tipo de valor por referência:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir é o mesmo que o exemplo anterior, exceto que o argumento é passado como um parâmetro `ref`. O valor do argumento subjacente, `n`, é alterado quando `x` é alterado no método.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(ref n); // Passing the variable by reference.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(ref int x)
// The parameter x is passed by reference.
// Changes to x will affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 25
*/
```

Neste exemplo, não é o valor de `n` que é passado; em vez disso, é passada uma referência a `n`. O parâmetro `x` não é um `int`; é uma referência a um `int`, nesse caso, uma referência a `n`. Portanto, quando `x` é elevado ao quadrado dentro do método, o que é realmente elevado ao quadrado é aquilo a que `x` se refere, `n`.

## Aprovar um tipo de referência por valor

Quando se aprova um tipo de *referênciapor valor*:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.

- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir demonstra a passagem de um parâmetro de tipo de referência, `arr`, por valor, para um método, `Change`. Como o parâmetro é uma referência a `arr`, é possível alterar os valores dos elementos da matriz. No entanto, a tentativa de reatribuir o parâmetro para um local diferente de memória só funciona dentro do método e não afeta a variável original, `arr`.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(int[] pArray)
{
    pArray[0] = 888; // This change affects the original element.
    pArray = new int[5] { -3, -1, -2, -3, -4 }; // This change is local.
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: 888
*/
```

No exemplo anterior, a matriz, `arr`, que é um tipo de referência, é passada para o método sem o parâmetro `ref`. Nesse caso, uma cópia da referência, que aponta para `arr`, é passada para o método. A saída mostra que é possível para o método alterar o conteúdo de um elemento de matriz, nesse caso de `1` para `888`. No entanto, alocar uma nova parte da memória usando o operador `new` dentro do método `Change` faz a variável `pArray` referenciar uma nova matriz. Portanto, quaisquer alterações realizadas depois disso não afetarão a matriz original, `arr`, criada dentro de `Main`. Na verdade, duas matrizes são criadas neste exemplo, uma dentro de `Main` e outra dentro do método `Change`.

## Aprovar um tipo de referência por referência

Quando se aprova um tipo de *referênciapor referência*:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **ficarão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir é o mesmo que o exemplo anterior, exceto que a palavra-chave `ref` é adicionada ao cabeçalho e à chamada do método. Quaisquer alterações que ocorrem no método afetam a variável original no programa de chamada.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(ref int[] pArray)
{
    // Both of the following changes will affect the original variables:
    pArray[0] = 888;
    pArray = new int[5] { -3, -1, -2, -3, -4 };
    System.Console.WriteLine("Inside the method, the first element is: {0}",
    pArray[0]);
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: -3
*/
```

Todas as alterações que ocorrem dentro do método afetam a matriz original em `Main`. Na verdade, a matriz original é realocada usando o operador `new`. Portanto, depois de chamar o método `Change`, qualquer referência a `arr` aponta para a matriz de cinco elementos, criada no método `Change`.

## Escopo de referências e valores

Os métodos podem armazenar os valores de parâmetros em campos. Quando os parâmetros são aprovados por valor, o método é sempre seguro. Os valores são copiados e os tipos de referência podem ser acessados quando armazenados em um campo. Para aprovar parâmetros por referência com segurança, é necessário que o compilador defina quando é seguro atribuir uma referência a uma nova variável. Para

cada expressão, o compilador define um *escopo* que vincula o acesso a uma expressão ou variável. O compilador usa dois escopos: *safe\_to\_escape* e *ref\_safe\_to\_escape*.

- *safe\_to\_escape* define o escopo em que qualquer expressão pode ser acessada com segurança.
- *ref\_safe\_to\_escape* define o escopo em que uma *referência* a qualquer expressão pode ser acessada ou modificada com segurança.

Informalmente, você pode considerar esses escopos como o mecanismo para garantir que seu código nunca acesse ou modifique uma referência que não seja mais válida. Uma referência é válida desde que se refira a um objeto ou struct válido. O escopo *safe\_to\_escape* define quando uma variável pode ser atribuída ou reatribuída. O escopo *ref\_safe\_to\_escape* define quando uma variável *pode ser atribuída* ou reatribuída por *ref*. A atribuição concede a uma variável um novo valor. A *atribuição de referência* indica para a variável que ela deve *se referir a* um local de armazenamento diferente.

## Modificadores

Os parâmetros declarados para um método sem *in*, *ref* nem *out* são passados para o método chamado pelo valor. Os modificadores *ref*, *in* e *out* diferem nas regras de atribuição:

- O argumento para um parâmetro *ref* deve ser atribuído de maneira definitiva. O método chamado pode reatribuir esse parâmetro.
- O argumento para um parâmetro *in* deve ser atribuído de maneira definitiva. O método chamado não pode reatribuir esse parâmetro.
- O argumento para um parâmetro *out* não precisa ser atribuído de maneira definitiva. O método chamado deve atribuir o parâmetro.

Esta seção descreve as palavras-chave que podem ser usadas ao declarar parâmetros de método:

- *params* especifica que esse parâmetro pode receber um número variável de argumentos.
- *in* especifica que esse parâmetro é passado por referência, mas é lido apenas pelo método chamado.
- *ref* especifica que esse parâmetro é passado por referência e pode ser lido ou gravado pelo método chamado.
- *out* especifica que esse parâmetro é passado por referência e é gravado pelo método chamado.

## Confira também

- [Referência de C#](#)
- [Palavras-chave do C#](#)
- Listas de argumentos na [Especificação de Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# Variáveis locais de tipo implícito (Guia de Programação em C#)

Artigo • 14/03/2023

Variáveis locais podem ser declaradas sem fornecer um tipo explícito. A palavra-chave `var` instrui o compilador a inferir o tipo da variável da expressão no lado direito da instrução de inicialização. O tipo inferido pode ser um tipo interno, um tipo anônimo, um tipo definido pelo usuário ou um tipo definido na biblioteca de classes .NET. Para obter mais informações sobre como inicializar matrizes com `var`, consulte [Matrizes de tipo implícito](#).

Os exemplos a seguir mostram várias maneiras em que as variáveis locais podem ser declaradas com `var`:

C#

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

É importante entender que a palavra-chave `var` não significa "variante" e não indica que a variável é vagamente tipada ou de associação tardia. Isso apenas significa que o compilador determina e atribui o tipo mais apropriado.

A palavra-chave `var` pode ser usada nos seguintes contextos:

- Em variáveis locais (variáveis declaradas no escopo do método) conforme mostrado no exemplo anterior.
- Em uma instrução de inicialização `for`.

```
C#
```

```
for (var x = 1; x < 10; x++)
```

- Em uma instrução de inicialização `foreach`.

```
C#
```

```
foreach (var item in list) {...}
```

- Em uma instrução `using`.

```
C#
```

```
using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

Para obter mais informações, consulte [Como usar matrizes e variáveis locais de tipo implícito em uma expressão de consulta](#).

## Tipos `var` e anônimos

Em muitos casos, o uso de `var` é opcional e é apenas uma conveniência sintática. No entanto, quando uma variável é inicializada com um tipo anônimo você deve declarar a variável como `var` se precisar acessar as propriedades do objeto em um momento posterior. Esse é um cenário comum em expressões de consulta LINQ. Para obter mais informações, consulte [Tipos Anônimos](#).

Da perspectiva do código-fonte, um tipo anônimo não tem nome. Portanto, se uma variável de consulta tiver sido inicializada com `var`, a única maneira de acessar as propriedades na sequência retornada será usar `var` como o tipo da variável de iteração na instrução `foreach`.

```
C#
```

```
class ImplicitlyTypedLocals2
{
    static void Main()
    {
```

```

string[] words = { "aPPLE", "BLUeBeRrY", "cHeRry" };

// If a query produces a sequence of anonymous types,
// then use var in the foreach statement to access the properties.
var upperLowerWords =
    from w in words
    select new { Upper = w.ToUpper(), Lower = w.ToLower() };

// Execute the query
foreach (var ul in upperLowerWords)
{
    Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper,
ul.Lower);
}
/* Outputs:
   Uppercase: APPLE, Lowercase: apple
   Uppercase: BLUEBERRY, Lowercase: blueberry
   Uppercase: CHERRY, Lowercase: cherry
*/

```

## Comentários

As seguintes restrições se aplicam às declarações de variável de tipo implícito:

- `var` pode ser usado apenas quando uma variável local é declarada e inicializada na mesma instrução, a variável não pode ser inicializada como nula, um grupo de métodos ou uma função anônima.
- `var` não pode ser usado em campos no escopo da classe.
- Variáveis declaradas usando `var` não podem ser usadas na expressão de inicialização. Em outras palavras, essa expressão é válida: `int i = (i = 20);`, mas essa expressão gera um erro em tempo de compilação: `var i = (i = 20);`
- Diversas variáveis de tipo implícito não podem ser inicializadas na mesma instrução.
- Se um tipo nomeado `var` estiver no escopo, a palavra-chave `var` será resolvida para esse nome de tipo e não será tratada como parte de uma declaração de variável local de tipo implícito.

Tipagem implícita com a palavra-chave `var` só pode ser aplicada às variáveis no escopo do método local. Digitação implícita não está disponível para os campos de classe, uma vez que o compilador C# encontraria um paradoxo lógico ao processar o código: o compilador precisa saber o tipo do campo, mas não é possível determinar o tipo até

que a expressão de atribuição seja analisada. A expressão não pode ser avaliada sem saber o tipo. Considere o seguinte código:

```
C#
```

```
private var bookTitles;
```

`bookTitles` é um campo de classe dado o tipo `var`. Como o campo não tem nenhuma expressão para avaliar, é impossível para o compilador inferir que tipo `bookTitles` deveria ser. Além disso, também é insuficiente adicionar uma expressão ao campo (como você faria para uma variável local):

```
C#
```

```
private var bookTitles = new List<string>();
```

Quando o compilador encontra campos durante a compilação de código, ele registra cada tipo de campo antes de processar quaisquer expressões associadas. O compilador encontra o mesmo paradoxo ao tentar analisar `bookTitles`: ele precisa saber o tipo do campo, mas o compilador normalmente determinaria o tipo de `var` analisando a expressão, o que não é possível sem saber o tipo com antecedência.

Você pode descobrir que `var` também pode ser útil com expressões de consulta em que o tipo construído exato da variável de consulta é difícil de ser determinado. Isso pode ocorrer com operações de agrupamento e classificação.

A palavra-chave `var` também pode ser útil quando o tipo específico da variável é enfadonho de digitar no teclado, é óbvio ou não acrescenta à legibilidade do código. Um exemplo em que `var` é útil dessa maneira é com os tipos genéricos aninhados, como os usados com operações de grupo. Na consulta a seguir, o tipo da variável de consulta é `IEnumerable<IGrouping<string, Student>>`. Contanto que você e as outras pessoas que devem manter o código entendam isso, não há problema em usar a tipagem implícita por questões de conveniência e brevidade.

```
C#
```

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

O uso de `var` ajuda a simplificar o código, mas seu uso deve ser restrito a casos em que ele é necessário ou facilita a leitura do código. Para obter mais informações sobre quando usar `var` corretamente, confira a seção [Variáveis locais digitados implicitamente](#) no artigo Diretrizes de Codificação de C#.

## Confira também

- [Referência de C#](#)
- [Matrizes de tipo implícito](#)
- [Como usar matrizes e variáveis locais de tipo implícito em uma expressão de consulta](#)
- [Tipos anônimos](#)
- [Inicializadores de objeto e coleção](#)
- [var](#)
- [LINQ em C#](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Instruções de iteração](#)
- [Instrução using](#)

# Como usar variáveis locais de tipo implícito e matrizes em uma expressão de consulta (Guia de Programação em C#)

Artigo • 07/04/2023

Será possível usar variáveis locais de tipo implícito sempre que você desejar que o compilador determine o tipo de uma variável local. É necessário usar variáveis locais de tipo implícito para armazenar tipos anônimos, usados frequentemente em expressões de consulta. Os exemplos a seguir ilustram usos obrigatórios e opcionais de variáveis locais de tipo implícito em consultas.

As variáveis locais de tipo implícito são declaradas usando a palavra-chave contextual `var`. Para obter mais informações, consulte [Variáveis locais de tipo implícito](#) e [Matrizes de tipo implícito](#).

## Exemplos

O exemplo a seguir mostra um cenário comum em que a palavra-chave `var` é necessária: uma expressão de consulta que produz uma sequência de tipos anônimos. Nesse cenário, a variável de consulta e a variável de iteração na instrução `foreach` devem ser tipadas implicitamente usando `var`, porque você não tem acesso a um nome de tipo para o tipo anônimo. Para obter mais informações sobre tipos anônimos, consulte [Tipos anônimos](#).

C#

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName,
```

```
anonType.LastName);  
    }  
}
```

O exemplo a seguir usa a palavra-chave `var` em uma situação semelhante, mas na qual o uso de `var` é opcional. Como `student.LastName` é uma cadeia de caracteres, a execução da consulta retorna uma sequência de cadeias de caracteres. Portanto, o tipo de `queryId` poderia ser declarado como

`System.Collections.Generic.IEnumerable<string>` em vez de `var`. A palavra-chave `var` é usada por conveniência. No exemplo, a variável de iteração na instrução `foreach` tem tipo explícito como uma cadeia de caracteres, mas, em vez disso, poderia ser declarada usando `var`. Como o tipo da variável de iteração não é um tipo anônimo, o uso de `var` é opcional, não obrigatório. Lembre-se de que `var` por si só não é um tipo, mas uma instrução para o compilador inferir e atribuir o tipo.

C#

```
// Variable queryId could be declared by using  
// System.Collections.Generic.IEnumerable<string>  
// instead of var.  
var queryId =  
    from student in students  
    where student.Id > 111  
    select student.LastName;  
  
// Variable str could be declared by using var instead of string.  
foreach (string str in queryId)  
{  
    Console.WriteLine("Last name: {0}", str);  
}
```

## Confira também

- [Guia de Programação em C#](#)
- [Métodos de Extensão](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [LINQ em C#](#)

# Métodos de extensão (Guia de Programação em C#)

Artigo • 10/05/2023

Os métodos de extensão permitem que você "adicone" tipos existentes sem criar um novo tipo derivado, recompilar ou, caso contrário, modificar o tipo original. Os métodos de extensão são métodos estáticos, mas são chamados como se fossem métodos de instância no tipo estendido. No caso do código cliente gravado em C#, F# e Visual Basic, não há nenhuma diferença aparente entre chamar um método de extensão e os métodos definidos em um tipo.

Os métodos de extensão mais comuns são os operadores de consulta padrão LINQ que adicionam funcionalidade de consulta aos tipos `System.Collections.IEnumerable` e `System.Collections.Generic.IEnumerable<T>` existentes. Para usar os operadores de consulta padrão, traga-os primeiro ao escopo com uma diretiva `using System.Linq`. Em seguida, qualquer tipo que implemente `IEnumerable<T>` parece ter métodos de instância como `GroupBy`, `OrderBy`, `Average` e assim por diante. Você pode exibir esses métodos adicionais no preenchimento de declaração do IntelliSense ao digitar "ponto" após uma instância de um tipo `IEnumerable<T>` como `List<T>` ou `Array`.

## Exemplo de OrderBy

O exemplo a seguir mostra como chamar o método de consulta padrão `OrderBy` em qualquer matriz de inteiros. A expressão entre parênteses é uma expressão lambda. Vários operadores de consulta padrão obtêm expressões lambda como parâmetros, mas isso não é um requisito para métodos de extensão. Para obter mais informações, consulte [Expressões Lambda](#).

C#

```
class ExtensionMethods2
{
    static void Main()
    {
        int[] ints = { 10, 45, 15, 39, 21, 26 };
        var result = ints.OrderBy(g => g);
        foreach (var i in result)
        {
            System.Console.Write(i + " ");
        }
    }
}
```

```
}
```

```
//Output: 10 15 21 26 39 45
```

Os métodos de extensão são definidos como estáticos, mas são chamados usando a sintaxe do método de instância. Seu primeiro parâmetro especifica em qual tipo o método opera. O parâmetro é precedido pelo modificador `this`. Os métodos de extensão só estarão no escopo quando você importar explicitamente o namespace para seu código-fonte com uma diretiva `using`.

O exemplo a seguir mostra um método de extensão definido para a classe `System.String`. Isso é definido em uma classe estática não aninhada e não genérica:

C#

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this string str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                            StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

O método de extensão `WordCount` pode ser colocado no escopo com esta diretiva `using`:

C#

```
using ExtensionMethods;
```

E pode ser chamado a partir de um aplicativo usando esta sintaxe:

C#

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

Você chama o método de extensão em seu código com a sintaxe do método de instância. A IL (linguagem intermediária) gerada pelo compilador converte seu código em uma chamada no método estático. O princípio de encapsulamento não está sendo violado. Os métodos de extensão não podem acessar variáveis particulares no tipo que estão estendendo.

Tanto a classe `MyExtensions` quanto o método `WordCount` são `static`, e podem ser acessados como todos os outros membros `static`. O método `WordCount` pode ser invocado como outros métodos `static` da seguinte maneira:

C#

```
string s = "Hello Extension Methods";
int i = MyExtensions.WordCount(s);
```

O código anterior do C#:

- Declara e atribui um novo `string` chamado `s` com um valor de `"Hello Extension Methods"`.
- Chama `MyExtensions.WordCount` com o argumento `s`

Para obter mais informações, confira [Como implementar e chamar um método de extensão personalizado](#).

Em geral, provavelmente você chamará métodos de extensão com muito mais frequência do que implementará os seus próprios. Como os métodos de extensão são chamados com a sintaxe do método de instância, nenhum conhecimento especial é necessário para usá-los no código do cliente. Para habilitar métodos de extensão para um tipo específico, apenas adicione uma diretiva `using` para o namespace no qual os métodos estão definidos. Por exemplo, para usar os operadores de consulta padrão, adicione esta diretiva `using` ao seu código:

C#

```
using System.Linq;
```

(Talvez você também precise adicionar uma referência a `System.Core.dll`.) Você observará que os operadores de consulta padrão agora aparecem no IntelliSense como métodos adicionais disponíveis para a maioria dos tipos `IEnumerable<T>`.

## Associando Métodos de Extensão no Momento da Compilação

Você pode usar métodos de extensão para estender uma classe ou interface, mas não os substituir. Um método de extensão com o mesmo nome e assinatura que um método de interface ou classe nunca será chamado. No tempo de compilação, os métodos de extensão sempre têm menos prioridade que os métodos de instância

definidos no próprio tipo. Em outras palavras, se um tipo possuir um método chamado `Process(int i)` e se você tiver um método de extensão com a mesma assinatura, o compilador sempre se associará ao método de instância. Quando o compilador encontra uma invocação de método, primeiro ele procura uma correspondência nos métodos de instância do tipo. Se nenhuma correspondência for encontrada, ele irá procurar todos os métodos de extensão definidos para o tipo e associará o primeiro método de extensão que encontrar. O exemplo a seguir demonstra como o compilador determina a qual método de extensão ou método de instância associar.

## Exemplo

O exemplo a seguir demonstra as regras que o compilador C# segue ao determinar se deve associar uma chamada de método a um método de instância no tipo ou a um método de extensão. A classe estática `Extensions` contém métodos de extensão definidos para qualquer tipo que implementa `IMyInterface`. As classes `A`, `B` e `C` implementam a interface.

O método de extensão `MethodB` nunca é chamado porque seu nome e assinatura são exatamente iguais aos métodos já implementados pelas classes.

Quando o compilador não consegue localizar um método de instância com uma assinatura compatível, ele se associa a um método de extensão correspondente se houver.

C#

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    using System;

    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}

// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class ExtensionMethods
    {
        public static void MethodA(IMyInterface target)
        {
            Console.WriteLine("MethodA");
        }

        public static void MethodB(IMyInterface target)
        {
            Console.WriteLine("MethodB");
        }
    }
}
```

```
// class that implements IMyInterface.
public static class Extension
{
    public static void MethodA(this IMyInterface myInterface, int i)
    {
        Console.WriteLine
            ("Extension.MethodA(this IMyInterface myInterface, int i)");
    }

    public static void MethodA(this IMyInterface myInterface, string s)
    {
        Console.WriteLine
            ("Extension.MethodA(this IMyInterface myInterface, string
s)");
    }

    // This method is never called in ExtensionMethodsDemo1, because
each
    // of the three classes A, B, and C implements a method named
MethodB
    // that has a matching signature.
    public static void MethodB(this IMyInterface myInterface)
    {
        Console.WriteLine
            ("Extension.MethodB(this IMyInterface myInterface)");
    }
}

// Define three classes that implement IMyInterface, and then use them to
test
// the extension methods.
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }

    class B : IMyInterface
    {
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
    }

    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
        {
    }
```

```

        Console.WriteLine("C.MethodA(object obj)");
    }
}

class ExtMethodDemo
{
    static void Main(string[] args)
    {
        // Declare an instance of class A, class B, and class C.
        A a = new A();
        B b = new B();
        C c = new C();

        // For a, b, and c, call the following methods:
        //      -- MethodA with an int argument
        //      -- MethodA with a string argument
        //      -- MethodB with no argument.

        // A contains no MethodA, so each call to MethodA resolves to
        // the extension method that has a matching signature.
        a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
        a.MethodA("hello");     // Extension.MethodA(IMyInterface,
string)

        // A has a method that matches the signature of the following
call
        // to MethodB.
        a.MethodB();            // A.MethodB()

        // B has methods that match the signatures of the following
// method calls.
        b.MethodA(1);           // B.MethodA(int)
        b.MethodB();             // B.MethodB()

        // B has no matching method for the following call, but
// class Extension does.
        b.MethodA("hello");     // Extension.MethodA(IMyInterface,
string)

        // C contains an instance method that matches each of the
following
        // method calls.
        c.MethodA(1);           // C.MethodA(object)
        c.MethodA("hello");     // C.MethodA(object)
        c.MethodB();             // C.MethodB()
    }
}
/* Output:
Extension.MethodA(this IMyInterface myInterface, int i)
Extension.MethodA(this IMyInterface myInterface, string s)
A.MethodB()
B.MethodA(int i)
B.MethodB()
Extension.MethodA(this IMyInterface myInterface, string s)

```

```
C.MethodA(object obj)
C.MethodA(object obj)
C.MethodB()
*/
```

## Padrões comuns de uso

### Funcionalidade de coleção

No passado, era comum criar "Classes de Coleção" que implementavam a interface `System.Collections.Generic.IEnumerable<T>` para determinado tipo e continham funcionalidades que atuavam em coleções desse tipo. Embora não haja nada de errado em criar esse tipo de objeto de coleção, a mesma funcionalidade pode ser obtida usando uma extensão em `System.Collections.Generic.IEnumerable<T>`. As extensões têm a vantagem de permitir que a funcionalidade seja chamada de qualquer coleção, como um `System.Array` ou `System.Collections.Generic.List<T>` que implementa `System.Collections.Generic.IEnumerable<T>` nesse tipo. Um exemplo disso usando uma Matriz de Int32 pode ser encontrado [anteriormente neste artigo](#).

### Funcionalidade específica da camada

Ao usar uma Arquitetura de camadas tipo cebola ou outro design de aplicativo em camadas, é comum ter um conjunto de entidades de domínio ou objetos de transferência de dados que podem ser usados para se comunicar entre os limites do aplicativo. Esses objetos geralmente não contêm funcionalidade ou têm apenas funcionalidades mínimas que se aplica a todas as camadas do aplicativo. Os métodos de extensão podem ser usados para adicionar funcionalidades específicas a cada camada de aplicativo, sem carregar o objeto com métodos não necessários ou desejados em outras camadas.

C#

```
public class DomainEntity
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
```

```
=> $"{value.FirstName} {value.LastName}";  
}
```

## Estendendo tipos predefinidos

Em vez de criar objetos quando a funcionalidade reutilizável precisa ser criada, muitas vezes podemos estender um tipo existente, como um tipo .NET ou CLR. Por exemplo, se não usarmos métodos de extensão, podemos criar uma classe `Engine` ou `Query` para realizar o trabalho de executar uma consulta em um SQL Server que pode ser chamado de vários lugares do código. No entanto, podemos estender a classe `System.Data.SqlClient.SqlConnection` usando métodos de extensão para executar essa consulta de qualquer lugar que tenhamos uma conexão com um SQL Server. Outros exemplos podem ser adicionar funcionalidades comuns à classe `System.String`, estender os recursos de processamento de dados dos objetos `System.IO.File` e `System.IO.Stream`, e objetos `System.Exception` para funcionalidades específicas de tratamento de erros. Esses tipos de casos de uso são limitados apenas por sua imaginação e bom senso.

Estender tipos predefinidos pode ser difícil com tipos `struct` porque eles são passados por valor para os métodos. Isso significa que todas as alterações no `struct` são feitas em uma cópia do `struct`. Essas alterações não ficam visíveis quando o método de extensão é encerrado. Você pode adicionar o modificador `ref` ao primeiro argumento de um método de extensão. Adicionar o modificador `ref` significa que o primeiro argumento é passado por referência. Isso permite que você escreva métodos de extensão que alterem o estado do `struct` que está sendo estendido.

## Diretrizes gerais

Embora ainda seja considerado preferível adicionar funcionalidade modificando o código de um objeto ou derivando um novo tipo sempre que for razoável e possível fazer isso, os métodos de extensão se tornaram uma opção crucial para criar funcionalidades reutilizáveis em todo o ecossistema do .NET. Para aquelas ocasiões em que a origem não está sob seu controle, quando um objeto derivado é inadequado ou impossível de usar, ou quando a funcionalidade não deve ser exposta além do escopo aplicável, os métodos de extensão são uma excelente opção.

Para obter mais informações sobre tipos derivados, confira [Herança](#).

Ao usar um método de extensão para estender um tipo cujo código-fonte você não pode alterar, há o risco de uma alteração na implementação do tipo interromper o funcionamento do método de extensão.

Se você implementar métodos de extensão para um determinado tipo, lembre-se das seguintes considerações:

- Um método de extensão nunca será chamado se possuir a mesma assinatura que um método definido no tipo.
- Os métodos de extensão são trazidos para o escopo no nível do namespace. Por exemplo, se você tiver várias classes estáticas que contenham métodos de extensão em um só namespace chamado `Extensions`, todos eles serão trazidos para o escopo pela diretiva `using Extensions;`.

Para uma biblioteca de classes que você implemente, não use métodos de extensão para evitar incrementar o número de versão de um assembly. Se desejar adicionar funcionalidade significativa a uma biblioteca da qual você tenha o código-fonte, siga as diretrizes do .NET para controle de versão do assembly. Para obter mais informações, consulte [Controle de versão do assembly](#).

## Confira também

- [Guia de Programação em C#](#)
- [Exemplos de programação paralela \(incluem vários métodos de extensão de exemplo\)](#)
- [Expressões Lambda](#)
- [Visão geral de operadores de consulta padrão](#)
- [Regras de conversão para parâmetros de instância e seu impacto](#)
- [Interoperabilidade de métodos de extensão entre linguagens](#)
- [Métodos de extensão e representantes via currying](#)
- [Associação do método de extensão e relatório de erros](#)

# Como implementar e chamar um método de extensão personalizado (Guia de Programação em C#)

Artigo • 10/05/2023

Este tópico mostra como implementar seus próprios métodos de extensão para qualquer tipo do .NET. O código de cliente pode usar seus métodos de extensão, adicionando uma referência à DLL que os contém e adicionando uma diretiva `using` que especifica o namespace no qual os métodos de extensão são definidos.

## Para definir e chamar o método de extensão

1. Defina uma `classe` estática para conter o método de extensão.

A classe deve estar visível para o código de cliente. Para obter mais informações sobre regras de acessibilidade, consulte [Modificadores de acesso](#).

2. Implemente o método de extensão como um método estático com, pelo menos, a mesma visibilidade da classe que a contém.
3. O primeiro parâmetro do método especifica o tipo no qual o método opera. Ele deve ser precedido pelo modificador `this`.
4. No código de chamada, adicione uma diretiva `using` para especificar o `namespace` que contém a classe do método de extensão.
5. Chame os métodos como se fossem métodos de instância no tipo.

Observe que o primeiro parâmetro não é especificado pelo código de chamada porque ele representa o tipo no qual o operador está sendo aplicado e o compilador já conhece o tipo do objeto. Você só precisa fornecer argumentos para os parâmetros de 2 até o `n`.

## Exemplo

O exemplo a seguir implementa um método de extensão chamado `WordCount` na classe `CustomExtensions.StringExtension`. O método funciona na classe `String`, que é especificada como o primeiro parâmetro do método. O namespace `CustomExtensions` é

importado para o namespace do aplicativo e o método é chamado dentro do método `Main`.

```
C#  
  
using System.Linq;  
using System.Text;  
using System;  
  
namespace CustomExtensions  
{  
    // Extension methods must be defined in a static class.  
    public static class StringExtension  
    {  
        // This is the extension method.  
        // The first parameter takes the "this" modifier  
        // and specifies the type for which the method is defined.  
        public static int WordCount(this string str)  
        {  
            return str.Split(new char[] { ' ', '.', '?' },  
StringSplitOptions.RemoveEmptyEntries).Length;  
        }  
    }  
}  
namespace Extension_Methods_Simple  
{  
    // Import the extension method namespace.  
    using CustomExtensions;  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string s = "The quick brown fox jumped over the lazy dog.";  
            // Call the method as if it were an  
            // instance method on the type. Note that the first  
            // parameter is not specified by the calling code.  
            int i = s.WordCount();  
            System.Console.WriteLine("Word count of s is {0}", i);  
        }  
    }  
}
```

## Segurança do .NET

Os métodos de extensão não apresentam nenhuma vulnerabilidade de segurança específica. Eles nunca podem ser usados para representar os métodos existentes em um tipo, porque todos os conflitos de nome são resolvidos em favor da instância ou do método estático, definidos pelo próprio tipo. Os métodos de extensão não podem acessar nenhum dado particular na classe estendida.

## Confira também

- [Guia de Programação em C#](#)
- [Métodos de Extensão](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Classes static e membros de classes static](#)
- [protected](#)
- [interno](#)
- [público](#)
- [this](#)
- [namespace](#)

# Como criar um novo método para uma enumeração (Guia de Programação em C#)

Artigo • 07/04/2023

Você pode usar métodos de extensão para adicionar funcionalidades específica para um tipo de enumeração específico.

## Exemplo

No exemplo a seguir, a enumeração `Grades` representa as letras possíveis que um aluno pode receber em uma classe. Um método de extensão chamado `Passing` é adicionado ao tipo `Grades` de forma que cada instância desse tipo agora "sabe" se ele representa uma nota de aprovação ou não.

C#

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

namespace EnumExtension
{
    // Define an extension method in a non-nested static class.
    public static class Extensions
    {
        public static Grades minPassing = Grades.D;
        public static bool Passing(this Grades grade)
        {
            return grade >= minPassing;
        }
    }

    public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
    class Program
    {
        static void Main(string[] args)
        {
            Grades g1 = Grades.D;
            Grades g2 = Grades.F;
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ?
"is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ?
"is" : "is not");
        }
    }
}
```

```
Extensions.minPassing = Grades.C;
Console.WriteLine("\r\nRaising the bar!\r\n");
Console.WriteLine("First {0} a passing grade.", g1.Passing() ?
"is" : "is not");
    Console.WriteLine("Second {0} a passing grade.", g2.Passing() ?
"is" : "is not");
}
}
/* Output:
   First is a passing grade.
   Second is not a passing grade.

   Raising the bar!

   First is not a passing grade.
   Second is not a passing grade.
*/
```

Observe que a classe `Extensions` também contém uma variável estática atualizada dinamicamente e que o valor retornado do método de extensão reflete o valor atual dessa variável. Isso demonstra que, nos bastidores, os métodos de extensão são chamados diretamente na classe estática na qual eles são definidos.

## Confira também

- [Guia de Programação em C#](#)
- [Métodos de Extensão](#)

# Argumentos nomeados e opcionais (Guia de Programação em C#)

Artigo • 09/05/2023

Os *argumentos nomeados* permitem que você especifique um argumento para um parâmetro correspondendo o argumento com seu nome em vez de com sua posição na lista de parâmetros. *Argumentos opcionais* permitem omitir argumentos para alguns parâmetros. Ambas as técnicas podem ser usadas com os métodos, indexadores, construtores e delegados.

Quando você usa argumentos nomeados e opcionais, os argumentos são avaliados na ordem em que aparecem na lista de argumentos e não na lista de parâmetros.

Parâmetros nomeados e opcionais permitem que você forneça argumentos para parâmetros selecionados. Essa capacidade facilita bastante chamadas para interfaces COM como as APIs de Automação do Microsoft Office.

## Argumentos nomeados

Os argumentos nomeados liberam você da necessidade de combinar a ordem dos argumentos com a ordem dos parâmetros nas listas de parâmetros de métodos chamados. O argumento para cada parâmetro pode ser especificado pelo nome do parâmetro. Por exemplo, uma função que imprime detalhes de pedidos (como o nome do vendedor, nome do produto & número do pedido) pode ser chamada por meio do envio de argumentos por posição, na ordem definida pela função.

C#

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

Se não se lembrar da ordem dos parâmetros, mas souber os nomes, você poderá enviar os argumentos em qualquer ordem.

C#

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Os argumentos nomeados também melhoram a legibilidade do código identificando o que cada argumento representa. No método de exemplo abaixo, o `sellerName` não pode ser nulo ou um espaço em branco. Como `sellerName` e `productName` são tipos de cadeia de caracteres, em vez de enviar argumentos por posição, é melhor usar argumentos nomeados para remover a ambiguidade dos dois e reduzir a confusão para qualquer pessoa que leia o código.

Os argumentos nomeados, quando usados com argumentos posicionais, são válidos, desde que

- não sejam seguidos por argumentos posicionais ou,

```
C#
```

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- são usados na posição correta. No exemplo a seguir, o parâmetro `orderNum` está na posição correta, mas não está explicitamente nomeado.

```
C#
```

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

Argumentos posicionais que seguem argumentos nomeados fora de ordem são inválidos.

```
C#
```

```
// This generates CS1738: Named argument specifications must appear after  
// all fixed arguments have been specified.  
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

## Exemplo

O código a seguir implementa os exemplos desta seção, juntamente com outros exemplos.

```
C#
```

```
class NamedExample  
{  
    static void Main(string[] args)  
    {  
        // The method can be called in the normal way, by using positional
```

```

arguments.

    PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName:
"Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop",
orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red
Mug");
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string
productName)
{
    if (string.IsNullOrWhiteSpace(sellerName))
    {
        throw new ArgumentException(message: "Seller name cannot be null
or empty.", paramName: nameof(sellerName));
    }

    Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum},
Product: {productName}");
}
}

```

## Argumentos opcionais

A definição de um método, construtor, indexador ou delegado pode especificar que seus parâmetros são obrigatórios ou opcionais. Qualquer chamada deve fornecer argumentos para todos os parâmetros necessários, mas pode omitir argumentos para parâmetros opcionais.

Cada parâmetro opcional tem um valor padrão como parte de sua definição. Se nenhum argumento é enviado para esse parâmetro, o valor padrão é usado. Um valor padrão deve ser um dos seguintes tipos de expressões:

- uma expressão de constante;

- uma expressão da forma `new ValType()`, em que `ValType` é um tipo de valor, como um `enum` ou um `struct`;
- uma expressão da forma `default(ValType)`, em que `ValType` é um tipo de valor.

Os parâmetros opcionais são definidos no final da lista de parâmetros, depois de todos os parâmetros obrigatórios. Se o chamador fornecer um argumento para qualquer um de uma sucessão de parâmetros opcionais, ele deverá fornecer argumentos para todos os parâmetros opcionais anteriores. Não há suporte para intervalos separados por vírgula na lista de argumentos. Por exemplo, no código a seguir, método de instância `ExampleMethod` está definido com um parâmetro obrigatório e dois opcionais.

C#

```
public void ExampleMethod(int required, string optionalstr = "default
string",
    int optionalint = 10)
```

A chamada para `ExampleMethod` a seguir causa um erro do compilador, porque um argumento é fornecido para o terceiro parâmetro, mas não para o segundo.

C#

```
//anExample.ExampleMethod(3, ,4);
```

No entanto, se você souber o nome do terceiro parâmetro, poderá usar um argumento nomeado para realizar a tarefa.

C#

```
anExample.ExampleMethod(3, optionalint: 4);
```

O IntelliSense usa colchetes para indicar parâmetros opcionais, conforme mostrado na seguinte ilustração:

```
anExample.ExampleMethod(
    void ExampleClass.ExampleMethod(int required,
        [string optionalstr = "default string"],
        [int optionalint = 10])
```

### ⓘ Observação

Você também pode declarar parâmetros opcionais usando a classe `OptionalAttribute` do .NET. Os parâmetros `OptionalAttribute` não exigem um valor

padrão.

## Exemplo

No exemplo a seguir, o construtor para `ExampleClass` tem um parâmetro, que é opcional. O método de instância `ExampleMethod` tem um parâmetro obrigatório, `required` e dois parâmetros opcionais, `optionalstr` e `optionalint`. O código em `Main` mostra as diferentes maneiras em que o construtor e o método podem ser invocados.

C#

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the
constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the
constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);

            // You can use a named parameter to make the previous
            // statement work.
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }
}
```

```

class ExampleClass
{
    private string _name;

    // Because the parameter for the constructor, name, has a default
    // value assigned to it, it is optional.
    public ExampleClass(string name = "Default name")
    {
        _name = name;
    }

    // The first parameter, required, has no default value assigned
    // to it. Therefore, it is not optional. Both optionalstr and
    // optionalint have default values assigned to them. They are
    optional.
    public void ExampleMethod(int required, string optionalstr =
"default string",
                            int optionalint = 10)
    {
        Console.WriteLine(
            $"{_name}: {required}, {optionalstr}, and {optionalint}.");
    }
}

// The output from this example is the following:
// Default name: 1, One, and 1.
// Default name: 2, Two, and 10.
// Default name: 3, default string, and 10.
// Provided name: 1, One, and 1.
// Provided name: 2, Two, and 10.
// Provided name: 3, default string, and 10.
// Default name: 3, default string, and 4.
}

```

O código anterior mostra vários exemplos em que os parâmetros opcionais não estão aplicados corretamente. O primeiro ilustra que um argumento deve ser fornecido para o primeiro parâmetro, o que é necessário.

## Interfaces COM

Os argumentos nomeados e opcionais, juntamente com suporte para objetos dinâmicos e outros aprimoramentos, aprimoraram enormemente a interoperabilidade com APIs COM, como APIs de Automação do Office.

Por exemplo, o método [AutoFormat](#) na interface [Range](#) do Microsoft Office Excel tem sete parâmetros, todos opcionais. Esses parâmetros são mostrados na seguinte ilustração:

```
excelApp.get_Range("A1", "B4").AutoFormat(  
    dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],  
    [object Number = Type.Missing], [object Font = Type.Missing],  
    [object Alignment = Type.Missing], [object Border = Type.Missing],  
    [object Pattern = Type.Missing], [object Width = Type.Missing])
```

No entanto, você pode simplificar muito a chamada para `AutoFormat` usando argumentos nomeados e opcionais. Os argumentos nomeados e opcionais permitem que você omita o argumento para um parâmetro opcional se não desejar alterar o valor padrão do parâmetro. Na chamada a seguir, um valor é especificado para apenas um dos sete parâmetros.

C#

```
var excelApp = new Microsoft.Office.Interop.Excel.Application();  
excelApp.Workbooks.Add();  
excelApp.Visible = true;  
  
var myFormat =  
  
Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting  
1;  
  
excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

Para obter mais informações e exemplos, consulte [Como usar argumentos nomeados e opcionais na programação do Office](#) e [Como acessar objetos de interoperabilidade do Office usando recursos do Visual C#](#).

## Resolução de sobrecarga

O uso de argumentos nomeados e opcionais afeta a resolução de sobrecarga das seguintes maneiras:

- Um método, indexador ou construtor é um candidato para a execução se cada um dos parâmetros é opcional ou corresponde, por nome ou posição, a um único argumento na instrução de chamada e esse argumento pode ser convertido para o tipo do parâmetro.
- Se mais de um candidato for encontrado, as regras de resolução de sobrecarga de conversões preferenciais serão aplicadas aos argumentos que são especificados explicitamente. Os argumentos omitidos para parâmetros opcionais são ignorados.
- Se dois candidatos são considerados igualmente bons, a preferência vai para um candidato que não tenha parâmetros opcionais para os quais argumentos foram omitidos na chamada. A resolução de sobrecarga geralmente prefere candidatos com menos parâmetros.

# Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# Como usar argumentos nomeados e opcionais na programação do Office

Artigo • 09/03/2023

Os argumentos nomeados e opcionais aprimoram a conveniência, a flexibilidade e a legibilidade na programação em C#. Além disso, esses recursos facilitam bastante o acesso a interfaces COM, como as APIs de Automação do Microsoft Office.

## ⓘ Importante

O VSTO se baseia no [.NET Framework](#). Os suplementos COM também podem ser gravados com o .NET Framework. Os suplementos do Office não podem ser criados com o [.NET Core](#) e o [.NET 5+](#), as versões mais recentes do .NET. Isso ocorre porque o .NET Core/.NET 5+ não pode trabalhar em conjunto com o .NET Framework no mesmo processo e pode levar a falhas de carga de suplemento. Você pode continuar a usar o .NET Framework para escrever suplementos VSTO e COM para o Office. A Microsoft não atualizará o VSTO ou a plataforma de suplemento COM para usar o .NET Core ou o .NET 5+. Você pode aproveitar o .NET Core e o .NET 5+, incluindo o ASP.NET Core, para criar o lado do servidor dos [Suplementos Web do Office](#).

No exemplo a seguir, o método [ConvertToTable](#) tem 16 parâmetros que representam as características de uma tabela, como o número de colunas e linhas, formatação, bordas, fontes e cores. Todos os 16 parâmetros são opcionais, pois na maioria das vezes você não deseja especificar determinados valores para todos eles. No entanto, sem argumentos nomeados e opcionais, é preciso fornecer um valor ou um valor de espaço reservado. Com argumentos nomeados e opcionais, você especifica somente os valores para os parâmetros necessários para seu projeto.

Você deve ter o Microsoft Office Word instalado em seu computador para concluir esses procedimentos.

## ⓘ Observação

Seu computador pode mostrar diferentes nomes ou locais para alguns dos elementos de interface do usuário do Visual Studio nas instruções a seguir. A edição do Visual Studio que você possui e as configurações que você usa

determinam esses elementos. Para obter mais informações, consulte [Personalizando o IDE](#).

## Crie um novo aplicativo de console

Inicie o Visual Studio. No menu **Arquivo**, aponte para **Novo** e selecione **Projeto**. No painel **Categorias de Modelos**, expanda **C#** e selecione **Windows**. Observe a parte superior do painel **Modelos** para se certificar de que **.NET Framework 4** é exibido na caixa **Estrutura de Destino**. No painel **Modelos**, selecione **Aplicativo de Console**. Digite um nome para o projeto no campo **Nome**. Selecione **OK**. O novo projeto aparece no **Gerenciador de Soluções**.

## Adicionar uma referência

No **Gerenciador de Soluções**, clique com o botão direito do mouse no nome do projeto e, em seguida, selecione **Adicionar Referência**. A caixa de diálogo **Adicionar Referência** é exibida. Na página **.NET**, selecione **Microsoft.Office.Interop.Word** na lista **Nome do Componente**. Selecione **OK**.

## Adicionar as diretivas using necessárias

No **Gerenciador de Soluções**, clique com o botão direito do mouse no arquivo *Program.cs* e, em seguida, selecione **Exibir Código**. Adicione as seguintes diretivas `using` na parte superior do arquivo de código:

C#

```
using Word = Microsoft.Office.Interop.Word;
```

## Exibir texto em um documento do Word

Na classe `Program` em *Program.cs*, adicione o seguinte método para criar um aplicativo do Word e um documento do Word. O método `Add` tem quatro parâmetros opcionais. Este exemplo usa os valores padrão. Portanto, nenhum argumento é necessário na instrução de chamada.

C#

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

Adicione o código a seguir ao final do método para definir onde exibir o texto no documento e o texto a ser exibido:

C#

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

## Executar o aplicativo

Adicione a seguinte instrução a Main:

C#

```
DisplayInWord();
```

Pressione **CTRL + F5** para executar o projeto. É exibido um documento do Word contendo o texto especificado.

## Alterar o texto para uma tabela

Use o método `ConvertToTable` para colocar o texto em uma tabela. O método tem 16 parâmetros opcionais. O IntelliSense coloca os parâmetros opcionais entre colchetes, como mostrado na ilustração a seguir.

```
range.ConvertToTable()  
Word.Table Range.ConvertToTable([ref object Separator = Type.Missing], [ref object NumRows =  
Type.Missing], [ref object NumColumns = Type.Missing], [ref object InitialColumnWidth =  
Type.Missing], [ref object Format = Type.Missing], [ref object ApplyBorders = Type.Missing],  
[ref object ApplyShading = Type.Missing], [ref object ApplyFont = Type.Missing], [ref object  
ApplyColor = Type.Missing], [ref object ApplyHeadingsRows = Type.Missing], [ref object  
ApplyLastRow = Type.Missing], [ref object ApplyFirstColumn = Type.Missing], [ref object  
ApplyLastColumn = Type.Missing], [ref object AutoFit = Type.Missing], [ref object  
AutoFitBehavior = Type.Missing], [ref object DefaultTableBehavior = Type.Missing])
```

Os argumentos nomeados e opcionais permitem que você especifique valores apenas para os parâmetros que deseja alterar. Adicione o seguinte código ao final do método `DisplayInWord` para criar uma tabela. O argumento especifica que as vírgulas na cadeia de caracteres de texto em `range` separam as células da tabela.

C#

```
// Convert to a simple table. The table will have a single row with  
// three columns.  
range.ConvertToTable(Separator: ",");
```

Pressione `CTRL` + `F5` para executar o projeto.

## Experimentar outros parâmetros

Altere a tabela para que ela tenha uma coluna e três linhas, substitua a última linha em `DisplayInWord` pela instrução a seguir e digite `CTRL` + `F5`.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

Especifique um formato predefinido para a tabela, substitua a última linha em `DisplayInWord` pela instrução a seguir e digite `CTRL` + `F5`. O formato pode ser qualquer uma das constantes `WdTableFormat`.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,  
Format: Word.WdTableFormat.wdTableFormatElegant);
```

## Exemplo

O código a seguir inclui o exemplo completo:

C#

```
using System;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by
            specifying
            // a starting and ending character position. Currently, the
            document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of
            the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a
            table.
            // The method has 16 optional parameters. You only have to
            specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row
            with
            // three columns.
            range.ConvertToTable(Separator: ",,");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",,", AutoFit: true, NumColumns:
1);
```

```
// Format the table.  
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns:  
1,  
    Format: Word.WdTableFormat.wdTableFormatElegant);  
}  
}  
}
```

# Construtores (guia de programação em C#)

Artigo • 07/04/2023

Sempre que uma instância de uma [classe](#) ou um [struct](#) é criada, seu construtor é chamado. Uma classe ou struct pode ter vários construtores que usam argumentos diferentes. Os construtores permitem que o programador defina valores padrão, limite a instanciação e grave códigos flexíveis e fáceis de ler. Para obter mais informações e exemplos, consulte [Construtores de Instância](#) e [Usando Construtores](#).

Há várias ações que fazem parte da inicialização de uma nova instância. Essas ações ocorrem na seguinte ordem:

1. *Os campos de instância são definidos como 0.* Normalmente, isso é feito pelo runtime.
2. *Inicializadores de campo são executados.* Os inicializadores de campo no tipo mais derivado são executados.
3. *Inicializadores de campo de tipo base são executados.* Inicializadores de campo começando com a base direta por meio de cada tipo base para [System.Object](#).
4. *Construtores de instância base são executados.* Quaisquer construtores de instância, começando com [Object.Object](#) por meio de cada classe base para a classe base direta.
5. *O construtor de instância é executado.* O construtor de instância para o tipo é executado.
6. *Inicializadores de objeto são executados.* Se a expressão incluir qualquer inicializador de objeto, eles são executados após a execução do construtor de instância. Inicializadores de objeto são executados na ordem textual.

As ações anteriores ocorrem quando uma nova instância é inicializada. Se uma nova instância de um [struct](#) for definida como seu [default](#) valor, todos os campos de instância serão definidos como 0.

Se o [construtor estático](#) não tiver sido executado, o construtor estático será executado antes que qualquer uma das ações do construtor de instância ocorra.

## Sintaxe do construtor

Um construtor é um método cujo nome é igual ao nome de seu tipo. Sua assinatura do método inclui apenas um [modificador de acesso](#) opcional, o nome do método e sua

lista de parâmetros; ela não inclui tipo de retorno. O exemplo a seguir mostra o construtor para uma classe denominada `Person`.

```
C#  
  
public class Person  
{  
    private string last;  
    private string first;  
  
    public Person(string lastName, string firstName)  
    {  
        last = lastName;  
        first = firstName;  
    }  
  
    // Remaining implementation of Person class.  
}
```

Se um construtor puder ser implementado como uma única instrução, você poderá usar uma [definição de corpo da expressão](#). O exemplo a seguir define uma classe `Location` cujo construtor tem um único parâmetro de cadeia de caracteres chamado *nome*. A definição de corpo da expressão atribui o argumento ao campo `locationName`.

```
C#  
  
public class Location  
{  
    private string locationName;  
  
    public Location(string name) => Name = name;  
  
    public string Name  
    {  
        get => locationName;  
        set => locationName = value;  
    }  
}
```

## Construtores estáticos

Os exemplos anteriores têm todos os construtores de instância mostrado, que criam um novo objeto. Uma classe ou struct também pode ter um construtor estático, que inicializa membros estáticos do tipo. Construtores estáticos não têm parâmetros. Se você não fornecer um construtor estático para inicializar campos estáticos, o compilador

C# inicializará campos estáticos com seu valor padrão, conforme listado no artigo [Valores padrão de tipos C#](#).

O exemplo a seguir usa um construtor estático para inicializar um campo estático.

C#

```
public class Adult : Person
{
    private static int minimumAge;

    public Adult(string lastName, string firstName) : base(lastName,
firstName)
    { }

    static Adult()
    {
        minimumAge = 18;
    }

    // Remaining implementation of Adult class.
}
```

Você também pode definir um construtor estático com uma definição de corpo da expressão, como mostra o exemplo a seguir.

C#

```
public class Child : Person
{
    private static int maximumAge;

    public Child(string lastName, string firstName) : base(lastName,
firstName)
    { }

    static Child() => maximumAge = 18;

    // Remaining implementation of Child class.
}
```

Para obter mais informações e exemplos, consulte [Construtores Estáticos](#).

## Nesta seção

[Usando construtores](#)

[Construtores de instância](#)

[Construtores particulares](#)

[Construtores estáticos](#)

[Como escrever um construtor de cópia](#)

## Confira também

- [Guia de Programação em C#](#)
- [O sistema de tipos C#](#)
- [Finalizadores](#)
- [static](#)
- [Por que os inicializadores são executados na ordem oposta, como construtores?](#)  
[Parte 1](#)

# Usando construtores (Guia de Programação em C#)

Artigo • 02/06/2023

Quando uma [classe](#) ou [struct](#) é instanciada, seu construtor é chamado. Os construtores têm o mesmo nome que a classe ou struct e eles geralmente inicializam os membros de dados do novo objeto.

No exemplo a seguir, uma classe chamada `Taxi` é definida usando um construtor simples. A classe é então instanciada com o operador `new`. O construtor `Taxi` é invocado pelo operador `new` imediatamente após a memória ser alocada para o novo objeto.

C#

```
public class Taxi
{
    public bool IsInitialized;

    public Taxi()
    {
        IsInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        Console.WriteLine(t.IsInitialized);
    }
}
```

Um construtor que não tem nenhum parâmetro é chamado de *construtor sem parâmetros*. Os construtores sem parâmetros são invocados sempre que um objeto é instanciado usando o operador `new` e nenhum argumento é fornecido para `new`. O C# 12 apresenta *construtores primários*. Um construtor primário especifica parâmetros que devem ser fornecidos para inicializar um novo objeto. Para obter mais informações, consulte [Construtores de instâncias](#).

A menos que a classe seja [static](#), as classes sem construtores recebem um construtor sem parâmetros público pelo compilador C# para habilitar a instanciação de classe. Para obter mais informações, consulte [Classes estáticas e membros de classes estáticas](#).

Você pode impedir que uma classe seja instanciada tornando o construtor privado, da seguinte maneira:

```
C#  
  
class NLog  
{  
    // Private Constructor:  
    private NLog() { }  
  
    public static double e = Math.E; //2.71828...  
}
```

Para obter mais informações, consulte [Construtores particulares](#).

Construtores para tipos `struct` se assemelham a construtores de classe. Quando um tipo `struct` é instanciado com `new`, um construtor é invocado. Quando um `struct` é definido como seu valor `default`, o runtime inicializa toda a memória no `struct` como 0. Antes do C# 10, os `structs` não podia conter um construtor explícito sem parâmetros porque um é fornecido automaticamente pelo compilador. Para obter mais informações, consulte a seção [Inicialização de struct e valores padrão](#) do artigo [Tipos de estrutura](#).

O código a seguir usa o construtor sem parâmetros para `Int32`, para que você tenha certeza de que o inteiro é inicializado:

```
C#  
  
int i = new int();  
Console.WriteLine(i);
```

O código a seguir, no entanto, causa um erro do compilador, pois não usa `new` e tenta usar um objeto não inicializado:

```
C#  
  
int i;  
Console.WriteLine(i);
```

Como alternativa, os objetos com base em `structs` (incluindo todos os tipos numéricos internos) podem ser inicializados ou atribuídos e, em seguida, usados como no exemplo a seguir:

```
C#
```

```
int a = 44; // Initialize the value type...
int b;
b = 33; // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

Classes e structs podem definir construtores que recebem parâmetros, incluindo [construtores primários](#). Os construtores que usam parâmetros devem ser chamados por meio de uma instrução `new` ou uma instrução `base`. As classes e structs também podem definir vários construtores e nenhum deles precisa definir um construtor sem parâmetros. Por exemplo:

C#

```
public class Employee
{
    public int Salary;

    public Employee() { }

    public Employee(int annualSalary)
    {
        Salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        Salary = weeklySalary * numberOfWeeks;
    }
}
```

Essa classe pode ser criada usando qualquer uma das instruções a seguir:

C#

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

Um construtor pode usar a palavra-chave `base` para chamar o construtor de uma classe base. Por exemplo:

C#

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
```

```
//Add further instructions here.  
}  
}
```

Neste exemplo, o construtor da classe base é chamado antes de o bloco do construtor ser executado. A palavra-chave `base` pode ser usada com ou sem parâmetros. Os parâmetros para o construtor podem ser usados como parâmetros para `base` ou como parte de uma expressão. Para obter mais informações, consulte [base](#).

Em uma classe derivada, se um construtor de classe base não for chamado explicitamente usando a palavra-chave `base`, o construtor sem parâmetros, se houver, será chamado implicitamente. As seguintes declarações de construtor são efetivamente iguais:

```
C#  
  
public Manager(int initialData)  
{  
    //Add further instructions here.  
}
```

```
C#  
  
public Manager(int initialData)  
    : base()  
{  
    //Add further instructions here.  
}
```

Se uma classe base não oferecer um construtor sem parâmetros, a classe derivada deverá fazer uma chamada explícita para um construtor base usando `base`.

Um construtor pode invocar outro construtor no mesmo objeto usando a palavra-chave `this`. Como `base`, `this` pode ser usado com ou sem parâmetros e todos os parâmetros no construtor estão disponíveis como parâmetros para `this` ou como parte de uma expressão. Por exemplo, o segundo construtor no exemplo anterior pode ser reescrito usando `this`:

```
C#  
  
public Employee(int weeklySalary, int numberOfWeeks)  
    : this(weeklySalary * numberOfWeeks)  
{  
}
```

O uso da palavra-chave `this` no exemplo anterior faz com que esse construtor seja chamado:

```
C#  
  
public Employee(int annualSalary)  
{  
    Salary = annualSalary;  
}
```

Os construtores podem ser marcados como [public](#), [private](#), [protected](#), [internal](#), [protected internal](#) ou [private protected](#). Esses modificadores de acesso definem como os usuários da classe podem construir a classe. Para obter mais informações, consulte [Modificadores de Acesso](#).

Um construtor pode ser declarado estático usando a palavra-chave [static](#). Os construtores estáticos são chamados automaticamente, imediatamente antes de qualquer campo estático ser acessado e são usados para inicializar membros da classe estática. Para obter mais informações, consulte [Construtores estáticos](#).

## Especificação da Linguagem C#

Para obter mais informações, veja [Construtores de instância](#) e [Construtores estáticos](#) na [Especificação de Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Guia de Programação em C#](#)
- [O sistema de tipos C#](#)
- [Construtores](#)
- [Finalizadores](#)

# Construtores de instâncias (Guia de Programação em C#)

Artigo • 02/06/2023

Você declara um construtor de instância para especificar o código executado ao criar uma nova instância de um tipo com a [expressão new](#). Para inicializar uma classe [estática](#) ou variáveis estáticas em uma classe não estática, é possível definir um [construtor estático](#).

Como mostra o exemplo a seguir, você pode declarar vários construtores de instância em um tipo:

```
C#  
  
class Coords  
{  
    public Coords()  
        : this(0, 0)  
    { }  
  
    public Coords(int x, int y)  
    {  
        X = x;  
        Y = y;  
    }  
  
    public int X { get; set; }  
    public int Y { get; set; }  
  
    public override string ToString() => $"({X},{Y})";  
}  
  
class Example  
{  
    static void Main()  
    {  
        var p1 = new Coords();  
        Console.WriteLine($"Coords #1 at {p1}");  
        // Output: Coords #1 at (0,0)  
  
        var p2 = new Coords(5, 3);  
        Console.WriteLine($"Coords #2 at {p2}");  
        // Output: Coords #2 at (5,3)  
    }  
}
```

No exemplo anterior, o primeiro construtor, sem parâmetros, chama o segundo construtor com ambos os argumentos iguais a `0`. Para fazer isso, use a palavra-chave `this`.

Ao declarar um construtor de instância em uma classe derivada, você pode chamar um construtor de uma classe base. Para fazer isso, use a palavra-chave `base`, como mostra o exemplo a seguir:

C#

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }

    public override double Area() => pi * x * x;
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area() => (2 * base.Area()) + (2 * pi * x * y);
}

class Example
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        var ring = new Circle(radius);
```

```

        Console.WriteLine($"Area of the circle = {ring.Area():F2}");
        // Output: Area of the circle = 19.63

        var tube = new Cylinder(radius, height);
        Console.WriteLine($"Area of the cylinder = {tube.Area():F2}");
        // Output: Area of the cylinder = 86.39
    }
}

```

## Construtores sem parâmetros

Se uma *classe* não tiver construtores de instância explícita, o C# fornecerá um construtor sem parâmetros que você pode usar para instanciar uma instância dessa classe, como mostra o exemplo a seguir:

C#

```

public class Person
{
    public int age;
    public string name = "unknown";
}

class Example
{
    static void Main()
    {
        var person = new Person();
        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Output: Name: unknown, Age: 0
    }
}

```

Esse construtor inicializa os campos e as propriedades da instância de acordo com os inicializadores correspondentes. Se um campo ou propriedade não tiver um inicializador, seu valor será definido como o **valor padrão** do tipo do campo ou da propriedade. Se você declarar pelo menos um construtor de instância em uma classe, o C# não fornecerá um construtor sem parâmetros.

Um tipo de *estrutura* sempre fornece um construtor sem parâmetros da seguinte maneira:

- No C# 9.0 e anteriores, esse é um construtor implícito sem parâmetros que produz o **valor padrão** de um tipo.
- No C# 10 e posteriores, esse é um construtor implícito sem parâmetros que produz o valor padrão de um tipo ou um construtor explicitamente declarado sem

parâmetros. Para obter mais informações, consulte a seção [Inicialização de struct e valores padrão](#) do artigo [Tipos de estrutura](#).

## Construtores primários

A partir do C# 12, você pode declarar um *construtor primário* em classes e structs. Você coloca todos os parâmetros entre parênteses seguindo o nome do tipo:

```
C#  
  
public class NamedItem(string name)  
{  
    public string Name => name;  
}
```

Os parâmetros para um construtor primário estão no escopo em todo o corpo do tipo de declaração. Eles podem inicializar propriedades ou campos. Eles podem ser usados como variáveis em métodos ou funções locais. Eles podem ser passados para um construtor base.

Um construtor primário indica que esses parâmetros são necessários para qualquer instância de um tipo. Qualquer construtor escrito explicitamente deve usar a sintaxe do inicializador `this(...)` para invocar o construtor primário. Isso garante que os parâmetros do construtor primário sejam definitivamente atribuídos por todos os construtores. Para qualquer tipo `class`, incluindo tipos `record class`, o construtor implícito sem parâmetros não é emitido quando um construtor primário está presente. Para qualquer tipo `struct`, incluindo tipos `record struct`, o construtor implícito sem parâmetros sempre é emitido e sempre inicializa todos os campos, incluindo parâmetros de construtor primário, para o padrão de 0 bit. Se você escrever um construtor explícito sem parâmetros, ele deverá invocar o construtor primário. Nesse caso, você pode especificar um valor diferente para os parâmetros do construtor primário. O código a seguir mostra exemplos de construtores primários.

```
C#  
  
// name isn't captured in Widget.  
// width, height, and depth are captured as private fields  
public class Widget(string name, int width, int height, int depth) :  
    NamedItem(name)  
{  
    public Widget() : this("N/A", 1,1,1) {} // unnamed unit cube  
  
    public int WidthInCM => width;  
    public int HeightInCM => height;
```

```
public int DepthInCM => depth;

public int Volume => width * height * depth;
}
```

Nos tipos `class` e `struct`, os parâmetros do construtor primário estão disponíveis em qualquer lugar no corpo do tipo. Eles podem ser usados como campos de membro. Quando um parâmetro de construtor primário é usado, o compilador captura o parâmetro de construtor em um campo privado com um nome gerado pelo compilador. Se um parâmetro de construtor primário não for usado no corpo do tipo, nenhum campo privado será capturado. Essa regra impede a alocação acidental de duas cópias de um parâmetro de construtor primário que é passado para um construtor base.

Se o tipo incluir o modificador `record`, o compilador sintetizará uma propriedade pública com o mesmo nome que o parâmetro do construtor primário. Para tipos `record class`, se um parâmetro de construtor primário usar o mesmo nome que um construtor primário base, essa propriedade será uma propriedade pública do tipo base `record class`. Ele não é duplicado no tipo derivado `record class`. Essas propriedades não são geradas para tipos não `record`.

## Confira também

- [Guia de programação em C#](#)
- [Classes, structs e registros](#)
- [Construtores](#)
- [Finalizadores](#)
- [base](#)
- [this](#)
- [Especificação de recurso de construtores primários](#)

# Construtores particulares (Guia de Programação em C#)

Artigo • 07/04/2023

Um construtor particular é um construtor de instância especial. Normalmente, ele é usado em classes que contêm apenas membros estáticos. Se uma classe tiver um ou mais construtores particulares e nenhum construtor público, outras classes (exceto as classes aninhadas) não poderão criar instâncias dessa classe. Por exemplo:

```
C#  
  
class NLog  
{  
    // Private Constructor:  
    private NLog() { }  
  
    public static double e = Math.E; //2.71828...  
}
```

A declaração do construtor vazio impede a geração automática de um construtor sem parâmetro. Observe que, se você não usar um modificador de acesso com o construtor, ele ainda será privado por padrão. No entanto, o modificador `private` geralmente é usado explicitamente para deixar claro que a classe não pode ser instanciada.

Construtores particulares são usados para impedir a criação de instâncias de uma classe quando não há métodos ou campos de instância, como a classe `Math` ou quando um método é chamado para obter uma instância de uma classe. Se todos os métodos na classe forem estáticos, considere deixar toda a classe estática. Para obter mais informações, consulte [Classes Estáticas e Membros de Classes Estáticas](#).

## Exemplo

A seguir, temos um exemplo de uma classe usando um construtor particular.

```
C#  
  
public class Counter  
{  
    private Counter() { }  
  
    public static int currentCount;  
  
    public static int IncrementCount()
```

```

    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter(); // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101

```

Observe que se você remover a marca de comentário da seguinte instrução do exemplo, ela gerará um erro porque o construtor está inacessível devido a seu nível de proteção:

C#

```
// Counter aCounter = new Counter(); // Error
```

## Confira também

- [Guia de Programação em C#](#)
- [O sistema de tipos C#](#)
- [Construtores](#)
- [Finalizadores](#)
- [private](#)
- [público](#)

# Construtores estáticos (Guia de Programação em C#)

Artigo • 07/04/2023

Um construtor estático é usado para inicializar quaisquer dados [estáticos](#) ou para executar uma ação específica que precisa ser executada apenas uma vez. Ele é chamado automaticamente antes que a primeira instância seja criada ou que quaisquer membros estáticos sejam referenciados. Um construtor estático será chamado no máximo uma vez.

C#

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Há várias ações que fazem parte da inicialização estática. Essas ações ocorrem na seguinte ordem:

1. *Os campos estáticos são definidos como 0.* Normalmente, isso é feito pelo runtime.
2. *Inicializadores de campo estático são executados.* Os inicializadores de campo estático no tipo mais derivado são executados.
3. *Inicializadores de campo estático de tipo base são executados.* Inicializadores de campo estáticos começando com a base direta por meio de cada tipo base para [System.Object](#).
4. *Construtores estáticos base são executados.* Quaisquer construtores estáticos, começando com [Object.Object](#) por meio de cada classe base para a classe base direta.
5. *O construtor estático é executado.* O construtor estático para o tipo é executado.

## Comentários

Construtores estáticos têm as seguintes propriedades:

- Um construtor estático não usa modificadores de acesso nem tem parâmetros.
- Uma classe ou struct só pode ter um construtor estático.
- Os construtores estáticos não podem ser herdados ou sobre carregados.
- Um construtor estático não pode ser chamado diretamente e destina-se apenas a ser chamado pela Common Language Runtime (CLR). Ele é invocado automaticamente.
- O usuário não tem controle sobre quando o construtor estático é executado no programa.
- Um construtor estático é chamado automaticamente. Ele inicializa a [classe](#) antes de a primeira instância ser criada ou de os membros estáticos declarados nessa classe (não nas classes base) serem referenciados. Um construtor estático será executado antes de um construtor de instância. Se inicializadores de variável de campo estático estiverem presentes na classe do construtor estático, eles serão executados na ordem textual em que aparecem na declaração de classe. Os inicializadores são executados imediatamente antes da execução do construtor estático.
- Se você não fornecer um construtor estático para inicializar campos estáticos, todos os campos estáticos serão inicializados com seu valor padrão, conforme listado nos [Valores padrão de tipos C#](#).
- Se um construtor estático gerar uma exceção, o runtime não o invocará uma segunda vez, e o tipo permanecerá não inicializado durante o tempo de vida do domínio do aplicativo. Normalmente, uma exceção [TypeInitializationException](#) é lançada quando um construtor estático não consegue instanciar um tipo ou uma exceção sem tratamento que ocorre em um construtor estático. Para construtores estáticos que não são definidos explicitamente no código-fonte, a solução de problemas pode exigir inspeção do código de linguagem intermediária (IL).
- A presença de um construtor estático impede a adição do atributo do tipo [BeforeFieldInit](#). Isso limita a otimização do runtime.
- Um campo declarado como `static readonly` só pode ser atribuído como parte de sua declaração ou em um construtor estático. Quando um construtor estático explícito não for necessário, inicialize os campos estáticos na declaração, em vez de usar um construtor estático para melhorar a otimização do runtime.
- O runtime chama um construtor estático não mais do que uma vez em um único domínio do aplicativo. Essa chamada é feita em uma região bloqueada com base no tipo específico da classe. Nenhum mecanismo de bloqueio adicional é necessário no corpo de um construtor estático. Para evitar o risco de deadlocks, não bloqueie o thread atual em inicializadores e construtores estáticos. Por exemplo, não aguarde tarefas, threads, identificadores de espera ou eventos, não adquira bloqueios e não execute o bloqueio de operações paralelas, como loops paralelos [Parallel.Invoke](#) e consultas LINQ paralelas.

## ① Observação

Embora não seja diretamente acessível, a presença de um construtor estático explícito deve ser documentada para auxiliar na solução de problemas de exceções de inicialização.

## Uso

- Um uso típico de construtores estáticos é quando a classe está usando um arquivo de log e o construtor é usado para gravar entradas nesse arquivo.
- Construtores estáticos também são úteis ao criar classes wrapper para código não gerenciado quando o construtor pode chamar o método `LoadLibrary`.
- Os construtores estáticos também são um local conveniente para impor verificações em tempo de execução no parâmetro de tipo que não pode ser verificado em tempo de compilação por meio de restrições de parâmetro de tipo.

## Exemplo

Nesse exemplo, a classe `Bus` tem um construtor estático. Quando a primeira instância do `Bus` for criada (`bus1`), o construtor estático será invocado para inicializar a classe. O exemplo de saída verifica se o construtor estático é executado somente uma vez, mesmo se duas instâncias de `Bus` forem criadas e se é executado antes que o construtor da instância seja executado.

C#

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to");
    }
}
```

```
{0}",  
    globalStartTime.ToString("T");  
}  
  
// Instance constructor.  
public Bus(int routeNum)  
{  
    RouteNumber = routeNum;  
    Console.WriteLine("Bus #{0} is created.", RouteNumber);  
}  
  
// Instance method.  
public void Drive()  
{  
    TimeSpan elapsedTime = DateTime.Now - globalStartTime;  
  
    // For demonstration purposes we treat milliseconds as minutes to  
    simulate  
    // actual bus times. Do not do this in your actual bus schedule  
    program!  
    Console.WriteLine("{0} is starting its route {1:N2} minutes after  
global start time {2}.",  
                    this.RouteNumber,  
                    elapsedTime.Milliseconds,  
                    globalStartTime.ToShortTimeString());  
}  
}  
  
class TestBus  
{  
    static void Main()  
{  
        // The creation of this instance activates the static constructor.  
        Bus bus1 = new Bus(71);  
  
        // Create a second bus.  
        Bus bus2 = new Bus(72);  
  
        // Send bus1 on its way.  
        bus1.Drive();  
  
        // Wait for bus2 to warm up.  
        System.Threading.Thread.Sleep(25);  
  
        // Send bus2 on its way.  
        bus2.Drive();  
  
        // Keep the console window open in debug mode.  
        Console.WriteLine("Press any key to exit.");  
        Console.ReadKey();  
    }  
}  
/* Sample output:  
Static constructor sets global start time to 3:57:08 PM.  
Bus #71 is created.
```

```
Bus #72 is created.  
71 is starting its route 6.00 minutes after global start time 3:57 PM.  
72 is starting its route 31.00 minutes after global start time 3:57 PM.  
*/
```

## Especificação da linguagem C#

Para saber mais, confira a seção [Construtores estáticos](#) da [Especificação da linguagem C#](#).

## Confira também

- [Guia de Programação em C#](#)
- [O sistema de tipos C#](#)
- [Construtores](#)
- [Classes static e membros de classes static](#)
- [Finalizadores](#)
- [Diretrizes de design de construtor](#)
- [Aviso de segurança – CA2121: os construtores estáticos devem ser privados](#)
- [Inicializadores de módulo](#)

# Como escrever um construtor de cópia (Guia de Programação em C#)

Artigo • 02/06/2023

Os [registros](#) em C# fornecem um construtor de cópia para objetos, mas para classes você precisa escrever um por conta própria.

## ⓘ Importante

Escrever construtores de cópia que funcionam para todos os tipos derivados em uma hierarquia de classe pode ser difícil. Se a sua classe não for `sealed`, você deverá criar uma hierarquia de tipos `record class` para usar o construtor de cópia sintetizado pelo compilador.

## Exemplo

No exemplo a seguir, a `Person` define um construtor de cópia que usa, como seu argumento, uma instância de `Person`. Os valores das propriedades do argumento são atribuídos às propriedades da nova instância de `Person`. O código contém um construtor de cópia alternativa que envia as propriedades `Name` e `Age` da instância que você deseja copiar para o construtor de instância da classe. A classe `Person` é `sealed`, portanto, nenhum tipo derivado pode ser declarado que possa introduzir erros copiando apenas a classe base.

C#

```
public sealed class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    /// // Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
}
```

```

public Person(string name, int age)
{
    Name = name;
    Age = age;
}

public int Age { get; set; }

public string Name { get; set; }

public string Details()
{
    return Name + " is " + Age.ToString();
}
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41

```

## Confira também

- ICloneable
- Registros
- Guia de Programação em C#

- O sistema de tipos C#
- Construtores
- Finalizadores

# Finalizadores (Guia de Programação em C#)

Artigo • 14/03/2023

Os finalizadores (historicamente chamados de **destruidores**) são usados para executar a limpeza final necessária quando uma instância da classe está sendo coletada pelo coletor de lixo. Na maioria dos casos, você pode evitar escrever um finalizador usando [System.Runtime.InteropServices.SafeHandle](#) ou as classes derivadas para encapsular qualquer identificador não gerenciado.

## Comentários

- Os finalizadores não podem ser definidos em structs. Eles são usados somente com classes.
- Uma classe pode ter somente um finalizador.
- Os finalizadores não podem ser herdados ou sobreescritos.
- Os finalizadores não podem ser chamados. Eles são invocados automaticamente.
- Um finalizador não usa modificadores ou não tem parâmetros.

Por exemplo, o seguinte é uma declaração de um finalizador para a classe `Car`.

```
C#  
  
class Car  
{  
    ~Car() // finalizer  
    {  
        // cleanup statements...  
    }  
}
```

Um finalizador também pode ser implementado como uma definição do corpo da expressão, como mostra o exemplo a seguir.

```
C#  
  
public class Destroyer  
{  
    public override string ToString() => GetType().Name;  
  
    ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is  
executing.");  
}
```

O finalizador chama implicitamente `Finalize` na classe base do objeto. Portanto, uma chamada para um finalizador é convertida implicitamente para o código a seguir:

C#

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

Esse arranjo significa que o método `Finalize` é chamado de forma recursiva para todas as instâncias da cadeia de herança, da mais derivada à menos derivada.

#### ⚠ Observação

Finalizadores vazios não devem ser usados. Quando uma classe contém um finalizador, uma entrada é criada na fila `Finalize`. Essa fila é processada pelo coletor de lixo. Quando o coletor de lixo processa a fila, ele chama cada finalizador. Finalizadores desnecessários, incluindo finalizadores vazios, finalizadores que chamam apenas o finalizador de classe base ou finalizadores que chamam apenas métodos emitidos condicionalmente, causam uma perda desnecessária de desempenho.

O programador não tem controle sobre quando o finalizador é chamado porque isso é determinado pelo coletor de lixo. O coletor de lixo procura objetos que não estão mais sendo usados pelo aplicativo. Se considerar um objeto qualificado para finalização, ele chamará o finalizador (se houver) e recuperará a memória usada para armazenar o objeto. É possível forçar a coleta de lixo chamando `Collect`, mas na maioria das vezes essa chamada deve ser evitada, porque pode criar problemas de desempenho.

#### ⚠ Observação

Executar ou não os finalizadores como parte do encerramento do aplicativo é uma situação específica para cada **implementação do .NET**. Quando um aplicativo é encerrado, o .NET Framework faz todos os esforços razoáveis para chamar

finalizadores para objetos que ainda não foram coletados, a menos que essa limpeza tenha sido suprimida (por uma chamada ao método de biblioteca `GC.SuppressFinalize`, por exemplo). O .NET 5 (incluindo o .NET Core) e versões posteriores não chamam finalizadores como parte do encerramento do aplicativo. Para obter mais informações, consulte o problema do GitHub [dotnet/csharpstandard #291](#).

Se você precisar executar a limpeza de forma confiável quando um aplicativo for encerrado, registre um manipulador para o evento `System.AppDomain.ProcessExit`. Esse manipulador garantiria que `IDisposable.Dispose()` (ou) `IAsyncDisposable.DisposeAsync()` fosse chamado para todos os objetos que exigem limpeza antes da saída do aplicativo. Como você não pode chamar *Finalizar* diretamente e não pode garantir que o coletor de lixo chame todos os finalizadores antes de sair, você deve usar `Dispose` ou `DisposeAsync` garantir que os recursos sejam liberados.

## Usar finalizadores para liberar recursos

Em geral, o C# não requer tanto gerenciamento de memória por parte do desenvolvedor quanto linguagens que não têm como destino um runtime com coleta de lixo. Isso ocorre porque o coletor de lixo do .NET gerencia implicitamente a alocação e a liberação de memória para seus objetos. No entanto, quando seu aplicativo encapsula recursos não gerenciados, como janelas, arquivos e conexões de rede, você deve usar finalizadores para liberar esses recursos. Quando o objeto está qualificado para finalização, o coletor de lixo executa o método `Finalize` do objeto.

## Liberação explícita de recursos

Se seu aplicativo estiver usando um recurso externo caro, também será recomendável fornecer uma maneira de liberar explicitamente o recurso antes que o coletor de lixo libere o objeto. Para liberar o recurso, implemente um método `Dispose` da interface `IDisposable` que executa a limpeza necessária para o objeto. Isso pode melhorar consideravelmente o desempenho do aplicativo. Mesmo com esse controle explícito sobre os recursos, o finalizador se tornará uma proteção usada para limpar os recursos se a chamada para o método `Dispose` falhar.

Para obter mais informações sobre como limpar recursos, consulte os seguintes artigos:

- [Limando recursos não gerenciados](#)
- [Implementando um método Dispose](#)
- [Implementar um método DisposeAsync](#)

- Instrução using

## Exemplo

O exemplo a seguir cria três classes que compõem uma cadeia de herança. A classe `First` é a classe base, `Second` é derivado de `First` e `Third` é derivado de `Second`. Todas as três têm finalizadores. Em `Main`, uma instância da classe mais derivada é criada. A saída desse código depende de qual implementação do .NET o aplicativo tem como destino:

- .NET Framework: a saída mostra que os finalizadores das três classes são chamados automaticamente quando o aplicativo é encerrado, na ordem do mais derivado para o menos derivado.
- .NET 5 (incluindo o .NET Core) ou uma versão posterior: não há saída, porque essa implementação do .NET não chama finalizadores quando o aplicativo termina.

C#

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");
    }
}

/*
Test with code like the following:
Third t = new Third();
t = null;

When objects are finalized, the output would be:
Third's finalizer is called.

```

```
Second's finalizer is called.  
First's finalizer is called.  
*/
```

## Especificação da linguagem C#

Para obter mais informações, confira a seção [Finalizadores](#) da [Especificação da linguagem C#](#).

## Confira também

- [IDisposable](#)
- [Guia de Programação em C#](#)
- [Construtores](#)
- [Coleta de lixo](#)

# Inicializadores de objeto e coleção (Guia de Programação em C#)

Artigo • 20/07/2023

O C# permite criar uma instância de um objeto ou uma coleção e executar as atribuições de membro em uma única instrução.

## Inicializadores de objeto

Os inicializadores de objeto permitem atribuir valores a quaisquer campos ou propriedades acessíveis de um objeto na hora de criação sem que seja necessário invocar um construtor seguido por linhas de instruções de atribuição. A sintaxe do inicializador de objeto permite especificar argumentos para um construtor ou omitir os argumentos (e a sintaxe de parênteses). O exemplo a seguir mostra como usar um inicializador de objeto com um tipo nomeado, `Cat`, e como invocar o construtor sem parâmetros. Observe o uso de propriedades autoimplementadas na classe `Cat`. Para obter mais informações, consulte [Propriedades autoimplementadas](#).

C#

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string? Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

C#

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

A sintaxe dos inicializadores de objetos permite que você crie uma instância, e depois atribui o objeto recém-criado, com suas propriedades atribuídas, à variável na

atribuição.

Os inicializadores de objeto podem definir indexadores, além de atribuir campos e propriedades. Considere esta classe `Matrix` básica:

```
C#  
  
public class Matrix  
{  
    private double[,] storage = new double[3, 3];  
  
    public double this[int row, int column]  
    {  
        // The embedded array will throw out of range exceptions as  
        appropriate.  
        get { return storage[row, column]; }  
        set { storage[row, column] = value; }  
    }  
}
```

Você poderia inicializar a matriz de identidade com o código a seguir:

```
C#  
  
var identity = new Matrix  
{  
    [0, 0] = 1.0,  
    [0, 1] = 0.0,  
    [0, 2] = 0.0,  
  
    [1, 0] = 0.0,  
    [1, 1] = 1.0,  
    [1, 2] = 0.0,  
  
    [2, 0] = 0.0,  
    [2, 1] = 0.0,  
    [2, 2] = 1.0,  
};
```

Nenhum indexador acessível que contenha um setter acessível pode ser usado como uma das expressões no inicializador de objeto, independentemente do número ou dos tipos de argumentos. Os argumentos de índice formam o lado esquerdo da atribuição e o valor é o lado direito da expressão. Por exemplo, estes serão todos válidos se `IndexersExample` tiver os indexadores apropriados:

```
C#  
  
var thing = new IndexersExample  
{
```

```
name = "object one",
[1] = '1',
[2] = '4',
[3] = '9',
Size = Math.PI,
['C',4] = "Middle C"
}
```

Para que o código anterior seja compilado, o tipo `IndexersExample` precisará ter os seguintes membros:

C#

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] { set { ... }; }
```

## Inicializadores de objeto com tipos anônimos

Embora inicializadores de objetos possam ser usados em qualquer contexto, eles são especialmente úteis em expressões de consulta LINQ. Expressões de consulta fazem uso frequente de [tipos anônimos](#), que podem ser inicializados somente usando um inicializador de objeto, como mostrado na declaração a seguir.

C#

```
var pet = new { Age = 10, Name = "Fluffy" };
```

Os tipos anônimos habilitam a cláusula `select` em uma expressão de consulta LINQ a transformar objetos da sequência original em objetos cujo valor e forma podem diferir dos originais. Isso será útil se você desejar armazenar apenas uma parte das informações de cada objeto em uma sequência. No exemplo a seguir, suponha que um objeto de produto (`p`) contenha vários campos e métodos e que você esteja apenas interessado em criar uma sequência de objetos que contenha o nome do produto e o preço unitário.

C#

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

Quando essa consulta for executada, a variável `productInfos` conterá uma sequência de objetos que podem ser acessados em uma instrução `foreach` como mostrado neste exemplo:

C#

```
foreach(var p in productInfos){...}
```

Cada objeto no novo tipo anônimo tem duas propriedades públicas que recebem os mesmos nomes que as propriedades ou os campos no objeto original. Você também poderá renomear um campo quando estiver criando um tipo anônimo; o exemplo a seguir renomeia o campo `UnitPrice` como `Price`.

C#

```
select new {p.ProductName, Price = p.UnitPrice};
```

## Inicializadores de objeto com o modificador required

Use a palavra-chave `required` para forçar os chamadores a definir o valor de uma propriedade ou campo usando um inicializador de objeto. As propriedades necessárias não precisam ser definidas como parâmetros de construtor. O compilador garante que todos os chamadores inicializem esses valores.

C#

```
public class Pet
{
    public required int Age;
    public string Name;
}

// `Age` field is necessary to be initialized.
// You don't need to initialize `Name` property
var pet = new Pet() { Age = 10};

// Compiler error:
// Error CS9035 Required member 'Pet.Age' must be set in the object
// initializer or attribute constructor.
// var pet = new Pet();
```

É uma prática típica garantir que seu objeto seja inicializado corretamente, especialmente quando você tem vários campos ou propriedades para gerenciar e não

deseja incluir todos no construtor.

## Inicializadores de objeto com o acessador `init`

Garantir que ninguém altere o objeto projetado pode ser limitado usando um acessador `init`. Ele ajuda a restringir a configuração do valor da propriedade.

C#

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; init; }
}

// Last name property is necessary to be initialized.
// The `FirstName` property can be modified after initialization.
var pet = new Person() { FirstName = "Joe", LastName = "Doe" };

// You can assign the FirstName property to a different value.
pet.FirstName = "Jane";

// Compiler error:
// Error CS8852 Init - only property or indexer 'Person.LastName' can only
be assigned in an object initializer,
//                 or on 'this' or 'base' in an instance constructor or an
'init' accessor.
// pet.LastName = "Kowalski";
```

As propriedades somente inicialização necessárias dão suporte a estruturas imutáveis, permitindo a sintaxe natural para os usuários do tipo.

## Inicializadores de coleção

Os inicializadores de coleção permitem especificar um ou mais inicializadores de elemento quando você inicializa um tipo de coleção que implementa `IEnumerable` e tem `Add` com a assinatura apropriada como um método de instância ou um método de extensão. Os inicializadores de elemento podem ser um valor simples, uma expressão ou um inicializador de objeto. Ao usar um inicializador de coleção, você não precisa especificar várias chamadas. O compilador adiciona as chamadas automaticamente.

O exemplo a seguir mostra dois inicializadores de coleção simples:

C#

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

O inicializador de coleção a seguir usa inicializadores de objeto para inicializar objetos da classe `Cat` definida em um exemplo anterior. Observe que os inicializadores de objeto individuais são envolvidos por chaves e separados por vírgulas.

C#

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

Você poderá especificar `nulo` como um elemento em um inicializador de coleção se o método `Add` da coleção permitir.

C#

```
List<Cat?> moreCats = new List<Cat?>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

É possível especificar elementos indexados quando a coleção é compatível com indexação de leitura/gravação.

C#

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

O exemplo anterior gera o código que chama o `Item[TKey]` para definir os valores. Você também poderia inicializar dicionários e outros contêineres associativos usando a seguinte sintaxe. Observe que, em vez da sintaxe do indexador, com parênteses e uma atribuição, ele usa um objeto com vários valores:

C#

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

Este exemplo de inicializador chama `Add(TKey, TValue)` para adicionar os três itens no dicionário. Essas duas maneiras diferentes para inicializar coleções associativas tem um comportamento um pouco diferente devido às chamadas de método que o compilador gera. As duas variantes trabalham com a classe `Dictionary`. Outros tipos podem ser compatíveis apenas com uma ou com outra, dependendo da API pública deles.

## Inicializadores de objeto com inicialização de propriedade somente leitura da coleção

Algumas classes podem ter propriedades de coleção em que a propriedade é somente leitura, como a propriedade `Cats` de `CatOwner` no seguinte caso:

C#

```
public class CatOwner
{
    public IList<Cat> Cats { get; } = new List<Cat>();
```

Você não poderá usar a sintaxe do inicializador de coleção discutida até o momento, pois a propriedade não pode ser atribuída a uma nova lista:

C#

```
CatOwner owner = new CatOwner
{
    Cats = new List<Cat>
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

No entanto, novas entradas podem ser adicionadas a `Cats` usando a sintaxe de inicialização omitindo a criação da lista (`new List<Cat>`), conforme mostrado a seguir:

```
C#  
  
CatOwner owner = new CatOwner  
{  
    Cats =  
    {  
        new Cat{ Name = "Sylvester", Age=8 },  
        new Cat{ Name = "Whiskers", Age=2 },  
        new Cat{ Name = "Sasha", Age=14 }  
    }  
};
```

O conjunto de entradas a serem adicionadas simplesmente aparece cercado por chaves. O acima é idêntico à gravação:

```
C#  
  
CatOwner owner = new CatOwner();  
owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });  
owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });  
owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });
```

## Exemplos

O exemplo a seguir combina os conceitos de inicializadores de coleção e objeto.

```
C#  
  
public class InitializationSample  
{  
    public class Cat  
    {  
        // Auto-implemented properties.  
        public int Age { get; set; }  
        public string? Name { get; set; }  
  
        public Cat() {}  
  
        public Cat(string name)  
        {  
            Name = name;  
        }  
    }  
  
    public static void Main()
```

```

{
    Cat cat = new Cat { Age = 10, Name = "Fluffy" };
    Cat sameCat = new Cat("Fluffy"){ Age = 10 };

    List<Cat> cats = new List<Cat>
    {
        new Cat { Name = "Sylvester", Age = 8 },
        new Cat { Name = "Whiskers", Age = 2 },
        new Cat { Name = "Sasha", Age = 14 }
    };

    List<Cat?> moreCats = new List<Cat?>
    {
        new Cat { Name = "Furrytail", Age = 5 },
        new Cat { Name = "Peaches", Age = 4 },
        null
    };

    // Display results.
    System.Console.WriteLine(cat.Name);

    foreach (Cat c in cats)
        System.Console.WriteLine(c.Name);

    foreach (Cat? c in moreCats)
        if (c != null)
            System.Console.WriteLine(c.Name);
        else
            System.Console.WriteLine("List element has null value.");
}

// Output:
//Fluffy
//Sylvester
//Whiskers
//Sasha
//Furrytail
//Peaches
//List element has null value.
}

```

O exemplo a seguir mostra um objeto que implementa `IEnumerable` e contém um método `Add` com vários parâmetros. Ele usa um inicializador de coleção com vários elementos por item na lista que correspondem à assinatura do método `Add`.

C#

```

public class FullExample
{
    class FormattedAddresses : IEnumerable<string>
    {
        private List<string> internalList = new List<string>();
        public IEnumerator<string> GetEnumerator() =>

```

```

internalList.GetEnumerator();

    System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

    public void Add(string firstname, string lastname,
                    string street, string city,
                    string state, string zipcode) => internalList.Add(
$@"{firstname} {lastname}
{street}
{city}, {state} {zipcode}"
);
}

public static void Main()
{
    FormattedAddresses addresses = new FormattedAddresses()
    {
        {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
        {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
    };

    Console.WriteLine("Address Entries:");

    foreach (string addressEntry in addresses)
    {
        Console.WriteLine("\r\n" + addressEntry);
    }
}

/*
 * Prints:

Address Entries:

John Doe
123 Street
Topeka, KS 00000

Jane Smith
456 Street
Topeka, KS 00000
*/
}

```

Os métodos `Add` podem usar a palavra-chave `params` para obter um número variável de argumentos, como mostrado no seguinte exemplo. Este exemplo também demonstra a implementação personalizada de um indexador para inicializar uma coleção usando índices.

```

public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> :
    IEnumerable<KeyValuePair<TKey, List<TValue>>> where TKey : notnull
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new
        Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator()
=> internalDictionary.GetEnumerator();

        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();

        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key,
(IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue>?
storedValues))
                internalDictionary.Add(key, storedValues = new List<TValue>());
            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary1
        = new RudimentaryMultiValuedDictionary<string, string>()
        {
            {"Group1", "Bob", "John", "Mary" },
            {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
        };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary2
        = new RudimentaryMultiValuedDictionary<string, string>()
        {
            ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
            ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie",
"Jesse" }
        };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary3
    }
}

```

```

        = new RudimentaryMultiValuedDictionary<string, string>()
    {
        {"Group1", new string []{ "Bob", "John", "Mary" } },
        { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse" }
    }
};

Console.WriteLine("Using first multi-valued dictionary created with
a collection initializer:");

foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary1)
{
    Console.WriteLine($"\\r\\nMembers of group {group.Key}: ");

    foreach (string member in group.Value)
    {
        Console.WriteLine(member);
    }
}

Console.WriteLine("\\r\\nUsing second multi-valued dictionary created
with a collection initializer using indexing:");

foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary2)
{
    Console.WriteLine($"\\r\\nMembers of group {group.Key}: ");

    foreach (string member in group.Value)
    {
        Console.WriteLine(member);
    }
}

Console.WriteLine("\\r\\nUsing third multi-valued dictionary created
with a collection initializer using indexing:");

foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary3)
{
    Console.WriteLine($"\\r\\nMembers of group {group.Key}: ");

    foreach (string member in group.Value)
    {
        Console.WriteLine(member);
    }
}

/*
 * Prints:

    Using first multi-valued dictionary created with a collection
initializer:

```

```

    Members of group Group1:
    Bob
    John
    Mary

    Members of group Group2:
    Eric
    Emily
    Debbie
    Jesse

    Using second multi-valued dictionary created with a collection
    initializer using indexing:

    Members of group Group1:
    Bob
    John
    Mary

    Members of group Group2:
    Eric
    Emily
    Debbie
    Jesse

    Using third multi-valued dictionary created with a collection
    initializer using indexing:

    Members of group Group1:
    Bob
    John
    Mary

    Members of group Group2:
    Eric
    Emily
    Debbie
    Jesse
    */
}

```

## Confira também

- Usar inicializadores de objeto (regra de estilo IDE0017)
- Usar inicializadores de coleção (regra de estilo IDE0028)
- Guia de Programação em C#
- LINQ em C#
- Tipos anônimos

# Como inicializar objetos usando um inicializador de objeto (Guia de Programação em C#)

Artigo • 07/04/2023

Você pode usar os inicializadores de objeto para inicializar objetos de tipo de maneira declarativa, sem invocar explicitamente um construtor para o tipo.

Os exemplos a seguir mostram como usar os inicializadores de objeto com objetos nomeados. O compilador processa os inicializadores de objetos ao acessar, primeiro, o construtor de instância sem parâmetro e, em seguida, processar as inicializações de membro. Portanto, se o construtor sem parâmetros for declarado como `private` na classe, os inicializadores de objeto que exigem acesso público falharão.

Se você estiver definindo um tipo anônimo, é necessário usar um inicializador de objeto. Para obter mais informações, consulte [Como retornar subconjuntos de propriedades de elementos em uma consulta](#).

## Exemplo

O exemplo a seguir mostra como inicializar um novo tipo `StudentName`, usando inicializadores de objeto. Este exemplo define as propriedades de `StudentName` tipo:

C#

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two
        // parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and
        // sending
        // arguments for the first and last names. The parameterless
        // constructor is
        // invoked in processing this declaration, not the constructor that
        // has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead"
```

```

};

// Declare a StudentName by using an object initializer and sending
// an argument for only the ID property. No corresponding
constructor is
    // necessary. Only the parameterless constructor is used to process
object
    // initializers.
StudentName student3 = new StudentName
{
    ID = 183
};

// Declare a StudentName by using an object initializer and sending
// arguments for all three properties. No corresponding constructor
is
    // defined in the class.
StudentName student4 = new StudentName
{
    FirstName = "Craig",
    LastName = "Playstead",
    ID = 116
};

Console.WriteLine(student1.ToString());
Console.WriteLine(student2.ToString());
Console.WriteLine(student3.ToString());
Console.WriteLine(student4.ToString());
}

// Output:
// Craig 0
// Craig 0
// 183
// Craig 116

public class StudentName
{
    // This constructor has no parameters. The parameterless constructor
    // is invoked in the processing of object initializers.
    // You can test this by changing the access modifier from public to
    // private. The declarations in Main that use object initializers
will
    // fail.
    public StudentName() { }

    // The following constructor has parameters for two of the three
    // properties.
    public StudentName(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }

    // Properties.
    public string? FirstName { get; set; }
}

```

```

        public string? LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + " " + ID;
    }
}

```

Os inicializadores de objeto podem ser usados para definir indexadores em um objeto. O exemplo a seguir define uma classe `BaseballTeam` que usa um indexador para obter e definir jogadores em posições diferentes. O inicializador pode atribuir jogadores com base na abreviação da posição ou no número usado para cada scorecard de beisebol de posição:

C#

```

public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new
List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value; }
        }
    }
}

public static void Main()
{
    var team = new BaseballTeam
    {
        ["RF"] = "Mookie Betts",
        [4] = "Jose Altuve",
        ["CF"] = "Mike Trout"
    };

    Console.WriteLine(team["2B"]);
}
}

```

## Confira também

- [Guia de Programação em C#](#)
- [Inicializadores de objeto e coleção](#)

# Como inicializar um dicionário com um inicializador de coleção (Guia de Programação em C#)

Artigo • 29/06/2023

Um `Dictionary<TKey, TValue>` contém uma coleção de pares de chave-valor. Seu método `Add` recebe dois parâmetros, um para a chave e outro para o valor. Uma maneira de inicializar um `Dictionary<TKey, TValue>` ou qualquer coleção cujo método `Add` use vários parâmetros, é colocar cada conjunto de parâmetros entre chaves, conforme mostrado no exemplo a seguir. Outra opção é usar um inicializador de índice, também mostrado no exemplo a seguir.

## ⓘ Observação

A principal diferença entre essas duas maneiras de inicializar a coleção é que, no caso de ter chaves duplicadas, por exemplo:

```
C#  
  
{ 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211  
} },  
{ 111, new StudentName { FirstName="Dina", LastName="Salimzianova",  
ID=317 } },
```

O método `Add` gerará `ArgumentException`: 'An item with the same key has already been added. Key: 111', enquanto a segunda parte do exemplo, o método de indexador público de leitura/gravação, substituirá silenciosamente a entrada já existente com a mesma chave.

## Exemplo

No exemplo de código a seguir, um `Dictionary<TKey, TValue>` é inicializado com instâncias do tipo `StudentName`. A primeira inicialização usa o método `Add` com dois argumentos. O compilador gera uma chamada para `Add` para cada um dos pares de chaves `int` e valores `StudentName`. A segunda usa um método de indexador público de leitura/gravação da classe `Dictionary`:

```
C#
```

```

public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 } },
            { 112, new StudentName { FirstName="Dina",
LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth",
ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is
{students[index].FirstName} {students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 },
            [112] = new StudentName { FirstName="Dina",
LastName="Salimzianova", ID=317 },
            [113] = new StudentName { FirstName="Andy", LastName="Ruth",
ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is
{students2[index].FirstName} {students2[index].LastName}");
        }
    }
}

```

Observe os dois pares de chaves em cada elemento da coleção na primeira declaração. As chaves internas circundam o inicializador de objeto do `StudentName` e as chaves mais externas circundam o inicializador do par chave/valor que será adicionado ao `students Dictionary< TKey, TValue >`. Por fim, todo o inicializador de coleção do dicionário

é colocado entre chaves. Na segunda inicialização, o lado esquerdo da atribuição é a chave e o lado direito é o valor, usando um inicializador de objeto para `StudentName`.

## Confira também

- [Guia de Programação em C#](#)
- [Inicializadores de objeto e coleção](#)

# Tipos aninhados (Guia de Programação em C#)

Artigo • 07/04/2023

Um tipo definido em uma [classe](#), um [struct](#) ou uma [interface](#) é chamado de tipo aninhado. Por exemplo

```
C#  
  
public class Container  
{  
    class Nested  
    {  
        Nested() { }  
    }  
}
```

Independentemente de o tipo externo ser uma classe, uma interface ou uma struct, os tipos aninhados são [privados](#) por padrão, acessíveis somente por meio do tipo que contêm. No exemplo anterior, a classe `Nested` é inacessível para tipos externos.

Você também pode especificar um [modificador de acesso](#) para definir a acessibilidade de um tipo aninhado, da seguinte maneira:

- Os tipos aninhados de uma [classe](#) podem ser [public](#), [protected](#), [internal](#), [protected internal](#), [private](#) ou [private protected](#).

No entanto, a definição de uma classe aninhada `protected`, `protected internal` ou `private protected` dentro de uma [classe selada](#) gera um aviso do compilador [CS0628](#), "novo membro protegido declarado na classe selada".

Lembre-se também de que tornar um tipo aninhado visível externamente viola a regra de qualidade de código [CA1034](#) – "Tipos aninhados não devem ser visíveis".

- Tipos aninhados de um [struct](#) podem ser [públicos](#), [internos](#) ou [particulares](#).

O exemplo a seguir torna a classe `Nested` pública:

```
C#  
  
public class Container  
{  
    public class Nested  
    {  
    }
```

```
Nested() { }  
}  
}
```

O tipo aninhado ou interno pode acessar o tipo recipiente ou externo. Para acessar o tipo recipiente, passe-o como um argumento ao construtor do tipo aninhado. Por exemplo:

```
C#  
  
public class Container  
{  
    public class Nested  
    {  
        private Container? parent;  
  
        public Nested()  
        {  
        }  
        public Nested(Container parent)  
        {  
            this.parent = parent;  
        }  
    }  
}
```

Um tipo aninhado tem acesso a todos os membros acessíveis ao seu tipo recipiente. Ele pode acessar membros privados e protegidos do tipo recipiente, incluindo quaisquer membros protegidos herdados.

Na declaração anterior, o nome completo da classe `Nested` é `Container.Nested`. Este é o nome usado para criar uma nova instância da classe aninhada, da seguinte maneira:

```
C#  
  
Container.Nested nest = new Container.Nested();
```

## Confira também

- [Guia de Programação em C#](#)
- [Sistema de tipos do C#](#)
- [Modificadores de acesso](#)
- [Construtores](#)
- [Regra CA1034](#)

# Classes e métodos partial (Guia de Programação em C#)

Artigo • 07/04/2023

É possível dividir a definição de uma [classe](#) ou [struct](#), uma [interface](#) ou um método em dois ou mais arquivos de origem. Cada arquivo de origem contém uma seção da definição de tipo ou método e todas as partes são combinadas quando o aplicativo é compilado.

## Classes parciais

Há várias situações em que a divisão de uma definição de classe é desejável:

- Ao trabalhar em projetos grandes, dividir uma classe em arquivos separados permite que vários programadores trabalhem ao mesmo tempo.
- Ao trabalhar com código-fonte gerado automaticamente, o código pode ser adicionado à classe sem precisar recriar o arquivo de origem. O Visual Studio usa essa abordagem quando cria Windows Forms, código de wrapper de serviço Web e assim por diante. Você pode criar código que usa essas classes sem precisar modificar o arquivo que o Visual Studio cria.
- Ao usar [geradores de origem](#) para gerar funcionalidade adicional em uma classe.

Para dividir uma definição de classe, use o modificador de palavra-chave [partial](#), como mostrado aqui:

C#

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

A palavra-chave `partial` indica que outras partes da classe, struct ou interface podem ser definidas no namespace. Todas as partes devem usar a palavra-chave `partial`. Todas

as partes devem estar disponíveis em tempo de compilação para formar o tipo final. Todas as partes devem ter a mesma acessibilidade, tais como `public`, `private` e assim por diante.

Se alguma parte for declarada como abstrata, o tipo inteiro será considerado abstrato. Se alguma parte for declarada como lacrada, o tipo inteiro será considerado lacrado. Se alguma parte declarar um tipo base, o tipo inteiro herda dessa classe.

Todas as partes que especificam uma classe base devem concordar, mas partes que omitem uma classe base ainda herdam o tipo base. As partes podem especificar diferentes interfaces base e o tipo final implementa todas as interfaces listadas por todas as declarações parciais. Qualquer membro de classe, struct ou interface declarado em uma definição parcial está disponível para todas as outras partes. O tipo final é a combinação de todas as partes em tempo de compilação.

### ⓘ Observação

O modificador `partial` não está disponível em declarações de enumeração ou delegados.

O exemplo a seguir mostra que tipos aninhados podem ser parciais, mesmo se o tipo no qual eles estão aninhados não for parcial.

```
C#  
  
class Container  
{  
    partial class Nested  
    {  
        void Test() { }  
    }  
  
    partial class Nested  
    {  
        void Test2() { }  
    }  
}
```

Em tempo de compilação, atributos de definições de tipo parcial são mesclados. Por exemplo, considere as declarações a seguir:

```
C#  
  
[SerializableAttribute]  
partial class Moon { }
```

```
[ObsoleteAttribute]  
partial class Moon { }
```

Elas são equivalentes às seguintes declarações:

C#

```
[SerializableAttribute]  
[ObsoleteAttribute]  
class Moon { }
```

Os itens a seguir são mesclados de todas as definições de tipo parcial:

- Comentários XML
- interfaces
- atributos de parâmetro de tipo genérico
- atributos class
- membros

Por exemplo, considere as declarações a seguir:

C#

```
partial class Earth : Planet, IRotate { }  
partial class Earth : IRevolve { }
```

Elas são equivalentes às seguintes declarações:

C#

```
class Earth : Planet, IRotate, IRevolve { }
```

## Restrições

Há várias regras para seguir quando você está trabalhando com definições de classes parciais:

- Todas as definições de tipo parcial que devem ser partes do mesmo tipo devem ser modificadas com `partial`. Por exemplo, as seguintes declarações de classe geram um erro:

C#

```
public partial class A { }
//public class A { } // Error, must also be marked partial
```

- O modificador `partial` só pode aparecer imediatamente antes das palavras-chave `class`, `struct` ou `interface`.
- Tipos parciais aninhados são permitidos em definições de tipo parcial, conforme ilustrado no exemplo a seguir:

C#

```
partial class ClassWithNestedClass
{
    partial class NestedClass { }
}

partial class ClassWithNestedClass
{
    partial class NestedClass { }
}
```

- Todas as definições de tipo parcial que devem ser partes do mesmo tipo devem ser definidas no mesmo assembly e no mesmo módulo (arquivo .dll ou .exe). Definições parciais não podem abranger vários módulos.
- O nome de classe e os parâmetros de tipo genérico devem corresponder em todas as definições de tipo parcial. Tipos genéricos podem ser parciais. Cada declaração parcial deve usar os mesmos nomes de parâmetro na mesma ordem.
- As seguintes palavras-chave em uma definição de tipo parcial são opcionais, mas, se estiverem presentes em uma definição de tipo parcial, não podem entrar em conflito com as palavras-chave especificadas em outra definição parcial para o mesmo tipo:
  - `público`
  - `private`
  - `protected`
  - `interno`
  - `abstract`
  - `sealed`
  - classe base
  - modificador `new` (partes aninhadas)
  - restrições genéricas

Para obter mais informações, consulte [Restrições a parâmetros de tipo](#).

## Exemplos

No exemplo a seguir, os campos e o construtor da classe, `Coords`, são declarados em uma definição de classe parcial e o membro, `PrintCoords`, é declarado em outra definição de classe parcial.

C#

```
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

O exemplo a seguir mostra que você também pode desenvolver interfaces e structs parciais.

C#

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
```

```
{  
    void Interface_Test2();  
}  
  
partial struct S1  
{  
    void Struct_Test() { }  
}  
  
partial struct S1  
{  
    void Struct_Test2() { }  
}
```

## Métodos parciais

Uma classe ou struct parcial pode conter um método parcial. Uma parte da classe contém a assinatura do método. Uma implementação pode ser definida na mesma parte ou em outra parte. Se a implementação não for fornecida, o método e todas as chamadas para o método serão removidos em tempo de compilação. A implementação pode ser necessária dependendo da assinatura do método. Um método parcial não precisa ter uma implementação nos seguintes casos:

- Ele não tem modificadores de acessibilidade (incluindo o `private` padrão).
- Ele retorna `void`.
- Ele não tem parâmetros `out`.
- Ele não tem nenhum dos seguintes modificadores `virtual`, `override`, `sealed`, `new` ou `extern`.

Qualquer método que não esteja em conformidade com todas essas restrições (por exemplo, método `public virtual partial void`), deve fornecer uma implementação.

Essa implementação pode ser fornecida por um *gerador de origem*.

Métodos parciais permitem que o implementador de uma parte de uma classe declare um método. O implementador de outra parte da classe pode definir esse método. Há dois cenários em que isso é útil: modelos que geram código clichê e geradores de origem.

- **Código do modelo:** o modelo reserva um nome de método e uma assinatura para que o código gerado possa chamar o método. Esses métodos seguem as restrições que permitem a um desenvolvedor decidir se quer implementar o método. Se o método não for implementado, o compilador removerá a assinatura do método e todas as chamadas para o método. As chamadas para o método, incluindo qualquer resultado que ocorreria da avaliação de argumentos nas

chamadas, não têm efeito em tempo de execução. Portanto, qualquer código na classe parcial pode usar livremente um método parcial, mesmo que a implementação não seja fornecida. Nenhum erro de tempo de compilação ou tempo de execução ocorrerá se o método for chamado, mas não implementado.

- **Geradores de origem:** os geradores de origem fornecem uma implementação para métodos. O desenvolvedor humano pode adicionar a declaração de método (geralmente com atributos lidos pelo gerador de origem). O desenvolvedor pode gravar código que chama esses métodos. O gerador de origem é executado durante a compilação e fornece a implementação. Nesse cenário, as restrições para métodos parciais que podem não ser implementados geralmente não são seguidas.

```
C#
```

```
// Definition in file1.cs
partial void OnNameChanged();

// Implementation in file2.cs
partial void OnNameChanged()
{
    // method body
}
```

- As declarações de método parcial devem começar com a palavra-chave contextual **partial**.
- As assinaturas de método parcial em ambas as partes do tipo parcial devem ser correspondentes.
- Métodos parciais podem ter modificadores **static** e **unsafe**.
- Métodos parciais podem ser genéricos. Restrições são colocadas quanto à declaração de método parcial de definição e, opcionalmente, podem ser repetidas na de implementação. Nomes de parâmetro e de tipo de parâmetro não precisam ser iguais na declaração de implementação e na de definição.
- Você pode fazer um **delegado** para um método parcial que foi definido e implementado, mas não para um método parcial que só foi definido.

## Especificação da Linguagem C#

Para obter mais informações, veja [Tipos parciais](#) na [Especificação da Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- Guia de Programação em C#
- Classes
- Tipos de estrutura
- Interfaces
- partial (tipo)

# Como retornar subconjuntos de propriedades de elementos em uma consulta (Guia de Programação em C#)

Artigo • 07/04/2023

Use um tipo anônimo em uma expressão de consulta quando essas duas condições se aplicarem:

- Você deseja retornar apenas algumas das propriedades de cada elemento de origem.
- Você não precisa armazenar os resultados da consulta fora do escopo do método em que a consulta é executada.

Se você deseja retornar apenas uma propriedade ou campo de cada elemento de origem, use somente o operador de ponto na cláusula `select`. Por exemplo, para retornar somente a `ID` de cada `student`, escreva a cláusula `select` da seguinte maneira:

C#

```
select student.ID;
```

## Exemplo

O exemplo a seguir mostra como usar um tipo anônimo para retornar apenas um subconjunto das propriedades de cada elemento de origem que corresponda à condição especificada.

C#

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
```

```
// they have the same names as the Student properties.  
Console.WriteLine(obj.FirstName + ", " + obj.LastName);  
}  
}  
/* Output:  
Adams, Terry  
Fakhouri, Fadi  
Garcia, Cesar  
Omelchenko, Svetlana  
Zabokritski, Eugene  
*/
```

Observe que o tipo anônimo usa nomes do elemento de origem para suas propriedades, se nenhum nome for especificado. Para fornecer novos nomes para as propriedades no tipo anônimo, escreva a instrução `select` da seguinte maneira:

C#

```
select new { First = student.FirstName, Last = student.LastName };
```

Se você tentar fazer isso no exemplo anterior, a instrução `Console.WriteLine` também deve ser alterada:

C#

```
Console.WriteLine(student.First + " " + student.Last);
```

## Compilando o código

Para executar esse código, copie e cole a classe em um aplicativo de console em C# com uma diretiva `using` para `System.Linq`.

## Confira também

- [Guia de Programação em C#](#)
- [Tipos anônimos](#)
- [LINQ em C#](#)

# Implementação de interface explícita (Guia de Programação em C#)

Artigo • 07/04/2023

Caso uma classe implemente duas interfaces que contêm um membro com a mesma assinatura, logo, implementar esse membro na classe fará com que as duas interfaces usem tal membro como sua implementação. No exemplo a seguir, todas as chamadas para `Paint` invocam o mesmo método. Esse primeiro exemplo define os tipos:

C#

```
public interface IControl
{
    void Paint();
}

public interface ISurface
{
    void Paint();
}

public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

O exemplo a seguir chama os métodos:

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

Mas talvez você não queira que a mesma implementação seja chamada para ambas as interfaces. Para chamar uma implementação diferente dependendo de qual interface está em uso, você pode implementar um membro de interface explicitamente. Uma implementação de interface explícita é um membro de classe que só é chamado por meio da interface especificada. Nomeia o membro de classe prefixando-o com o nome da interface e um ponto. Por exemplo:

C#

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

O membro da classe `IControl.Paint` está disponível somente por meio da interface `IControl` e `ISurface.Paint` está disponível apenas por meio do `ISurface`. Ambas as implementações de método são separadas e nenhuma delas está diretamente disponível na classe. Por exemplo:

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
//sample.Paint(); // Compiler error.
control.Paint(); // Calls IControl.Paint on SampleClass.
surface.Paint(); // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

A implementação explícita também é utilizada para resolver casos em que duas interfaces declaram membros diferentes com o mesmo nome, como uma propriedade e um método. Para implementar as duas interfaces, é necessário que uma classe use a implementação explícita para a propriedade `P`, o método `P` ou ambos, a fim de evitar um erro do compilador. Por exemplo:

C#

```
interface ILeft
{
    int P { get; }
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

Uma implementação de interface explícita não tem um modificador de acesso, pois não está acessível como membro do tipo em que está definido. Em vez disso, ele só pode ser acessado quando chamado por meio de uma instância da interface. Se você especificar um modificador de acesso para uma implementação de interface explícita, receberá o erro do compilador [CS0106](#). Para obter mais informações, confira [interface \(referência C#\)](#).

Você pode definir uma implementação para membros declarados em uma interface. Se uma classe herdar uma implementação de método de uma interface, esse método só será acessível por meio de uma referência do tipo de interface. O membro herdado não aparece como parte da interface pública. O exemplo a seguir define uma implementação padrão para um método de interface:

C#

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

O exemplo a seguir invoca a implementação padrão:

C#

```
var sample = new SampleClass();
//sample.Paint(); // "Paint" isn't accessible.
```

```
var control = sample as IControl;  
control.Paint();
```

Qualquer classe que implementa a interface `IControl` pode substituir o método padrão `Paint`, seja como um método público, seja como uma implementação de interface explícita.

## Confira também

- [Guia de Programação em C#](#)
- [Programação orientada a objetos](#)
- [Interfaces](#)
- [Herança](#)

# Como implementar os membros da interface explicitamente (Guia de Programação em C#)

Artigo • 07/04/2023

Este exemplo declara uma interface, `IDimensions` e uma classe, `Box`, que implementa explicitamente os membros de interface `GetLength` e `GetWidth`. Os membros são acessados por meio da instância `dimensions` da interface.

## Exemplo

C#

```
interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;
```

```

IDimensions dimensions = box1;

// The following commented lines would produce compilation
// errors because they try to access an explicitly implemented
// interface member from a class instance:
//System.Console.WriteLine("Length: {0}", box1.GetLength());
//System.Console.WriteLine("Width: {0}", box1.GetWidth());

// Print out the dimensions of the box by calling the methods
// from an instance of the interface:
System.Console.WriteLine("Length: {0}", dimensions.GetLength());
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
}

/*
/* Output:
Length: 30
Width: 20
*/

```

## Programação robusta

- Observe que as seguintes linhas, no método `Main`, foram comentadas, pois produziriam erros de compilação. Um membro de interface implementado explicitamente não pode ser acessado de uma instância de `classe`:

C#

```

//System.Console.WriteLine("Length: {0}", box1.GetLength());
//System.Console.WriteLine("Width: {0}", box1.GetWidth());

```

- Observe também que as linhas a seguir, no método `Main`, imprimem com êxito as dimensões da caixa, pois os métodos estão sendo chamados de uma instância da interface:

C#

```

System.Console.WriteLine("Length: {0}", dimensions.GetLength());
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());

```

## Confira também

- [Guia de Programação em C#](#)
- [Programação orientada a objetos](#)
- [Interfaces](#)
- [Como implementar membros de duas interfaces explicitamente](#)



# Como implementar explicitamente membros de duas interfaces (Guia de Programação em C#)

Artigo • 07/04/2023

A implementação explícita da `interface` também permite ao programador implementar duas interfaces que têm os mesmos nomes de membro e implementar separadamente cada membro de interface. Este exemplo exibe as dimensões de uma caixa em unidades inglesas e no sistema métrico. A Caixa `classe` implementa duas interfaces, `IEnglishDimensions` e `IMetricDimensions`, que representam os diferentes sistemas de medida. As duas interfaces têm nomes de membro idênticos, `Comprimento` e `Largura`.

## Exemplo

C#

```
// Declare the English units interface:  
interface IEnglishDimensions  
{  
    float Length();  
    float Width();  
}  
  
// Declare the metric units interface:  
interface IMetricDimensions  
{  
    float Length();  
    float Width();  
}  
  
// Declare the Box class that implements the two interfaces:  
// IEnglishDimensions and IMetricDimensions:  
class Box : IEnglishDimensions, IMetricDimensions  
{  
    float lengthInches;  
    float widthInches;  
  
    public Box(float lengthInches, float widthInches)  
    {  
        this.lengthInches = lengthInches;  
        this.widthInches = widthInches;  
    }  
  
    // Explicitly implement the members of IEnglishDimensions:  
    float IEnglishDimensions.Length() => lengthInches;
```

```

float IEnglishDimensions.Width() => widthInches;

// Explicitly implement the members of IMetricDimensions:
float IMetricDimensions.Length() => lengthInches * 2.54f;

float IMetricDimensions.Width() => widthInches * 2.54f;

static void Main()
{
    // Declare a class instance box1:
    Box box1 = new Box(30.0f, 20.0f);

    // Declare an instance of the English units interface:
    IEnglishDimensions eDimensions = box1;

    // Declare an instance of the metric units interface:
    IMetricDimensions mDimensions = box1;

    // Print dimensions in English units:
    System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
    System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

    // Print dimensions in metric units:
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}

/* Output:
   Length(in): 30
   Width (in): 20
   Length(cm): 76.2
   Width (cm): 50.8
*/

```

## Programação robusta

Caso deseje que as medidas padrão estejam unidades inglesas, implemente os métodos Comprimento e Largura normalmente e implemente explicitamente os métodos Comprimento e Largura da interface IMetricDimensions:

C#

```

// Normal implementation:
public float Length() => lengthInches;
public float Width() => widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => lengthInches * 2.54f;
float IMetricDimensions.Width() => widthInches * 2.54f;

```

Nesse caso, é possível acessar as unidades inglesas da instância de classe e acessar as unidades métricas da instância da interface:

C#

```
public static void Test()
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

## Confira também

- [Guia de Programação em C#](#)
- [Programação orientada a objetos](#)
- [Interfaces](#)
- [Como implementar membros de interface explicitamente](#)

# Delegados (Guia de Programação em C#)

Artigo • 15/02/2023

Um [delegado](#) é um tipo que representa referências aos métodos com lista de parâmetros e tipo de retorno específicos. Ao instanciar um delegado, você pode associar sua instância a qualquer método com assinatura e tipo de retorno compatíveis. Você pode invocar (ou chamar) o método através da instância de delegado.

Delegados são usados para passar métodos como argumentos a outros métodos. Os manipuladores de eventos nada mais são do que métodos chamados por meio de delegados. Ao criar um método personalizado, uma classe como um controle do Windows poderá chamá-lo quando um determinado evento ocorrer. O seguinte exemplo mostra uma declaração de delegado:

C#

```
public delegate int PerformCalculation(int x, int y);
```

Qualquer método de qualquer classe ou struct acessível que corresponda ao tipo delegado pode ser atribuído ao delegado. O método pode ser estático ou de instância. Essa flexibilidade significa que você pode alterar programaticamente as chamadas de método ou conectar um novo código às classes existentes.

## ⓘ Observação

No contexto da sobrecarga de método, a assinatura de um método não inclui o valor retornado. No entanto, no contexto de delegados, a assinatura inclui o valor retornado. Em outras palavras, um método deve ter o mesmo tipo de retorno que o delegado.

Essa capacidade de se referir a um método como um parâmetro torna delegados ideais para definir métodos de retorno de chamada. Você pode escrever um método que compara dois objetos em seu aplicativo. Esse método pode ser usado em um delegado para um algoritmo de classificação. Como o código de comparação é separado da biblioteca, o método de classificação pode ser mais geral.

[Ponteiros de função](#) foram adicionados ao C# 9 para cenários semelhantes em que você precisa de mais controle sobre a convenção de chamada. O código associado a um

delegado é invocado usando um método virtual adicionado a um tipo delegado. Usando ponteiros de função, você pode especificar convenções diferentes.

## Visão geral de delegados

Os delegados possuem as seguintes propriedades:

- Representantes são semelhantes a ponteiros de função do C++, mas delegados são totalmente orientados a objeto e, ao contrário dos ponteiros de C++ para funções de membro, os delegados encapsulam uma instância do objeto e um método.
- Os delegados permitem que métodos sejam passados como parâmetros.
- Os delegados podem ser usados para definir métodos de retorno de chamada.
- Os delegados podem ser encadeados juntos; por exemplo, vários métodos podem ser chamados em um único evento.
- Os métodos não precisam corresponder exatamente ao tipo delegado. Para obter mais informações, confira [Usando a variação delegados](#).
- Expressões lambda são uma forma mais concisa de escrever blocos de código embutidos. As expressões lambda (em determinados contextos) são compiladas para tipos delegados. Para saber mais sobre expressões lambda, confira o artigo sobre [expressões lambda](#).

## Nesta seção

- [Usando delegados](#)
- [Quando usar delegados em vez de interfaces \(Guia de Programação em C#\)](#)
- [Delegados com Métodos Nomeados vs. Métodos anônimos](#)
- [Usando variação em delegados](#)
- [Como combinar delegados \(delegados multicast\)](#)
- [Como declarar e usar um delegado e criar uma instância dele](#)

## Especificação da Linguagem C#

Para obter mais informações, veja [Delegados na Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Capítulos do Livro em Destaque

- [Expressões lambda, eventos e delegados](#) em *C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers*

- Delegados e eventos em [Aprendendo sobre C# 3.0: conceitos básicos do C# 3.0](#)

## Confira também

- [Delegate](#)
- [Guia de Programação em C#](#)
- [Eventos](#)

# Usando delegados (Guia de Programação em C#)

Artigo • 31/07/2023

Um [delegado](#) é um tipo que encapsula com segurança um método, semelhante a um ponteiro de função em C e C++. No entanto, ao contrário dos ponteiros de função de C, delegados são orientados a objeto, fortemente tipados e seguros. O tipo de um delegado é definido pelo nome do delegado. O exemplo a seguir declara um delegado chamado `callback` que pode encapsular um método que usa uma [cadeia de caracteres](#) como um argumento e retorna [nulo](#):

C#

```
public delegate void Callback(string message);
```

Normalmente, um objeto delegado é construído fornecendo-se o nome do método que o delegado encapsulará ou com uma [expressão lambda](#). Quando um delegado é instanciado, uma chamada de método feita ao delegado será passada pelo delegado para esse método. Os parâmetros passados para o delegado pelo chamador são passados para o método e o valor de retorno, se houver, do método é retornado ao chamador pelo delegado. Isso é conhecido como invocar o delegado. Um delegado instanciado pode ser invocado como se fosse o método encapsulado em si. Por exemplo:

C#

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

C#

```
// Instantiate the delegate.
Callback handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Tipos de delegado são derivados da classe [Delegate](#) no .NET. Tipos de delegado são [lacrados](#) – não podem ser derivados de – e não é possível derivar classes personalizadas

de [Delegate](#). Como o delegado instanciado é um objeto, ele pode ser passado como um argumento ou atribuído a uma propriedade. Isso permite que um método aceite um delegado como um parâmetro e chame o delegado posteriormente. Isso é conhecido como um retorno de chamada assíncrono e é um método comum de notificação de um chamador quando um processo longo for concluído. Quando um delegado é usado dessa maneira, o código que usa o delegado não precisa de conhecimento algum da implementação do método que está sendo usado. A funcionalidade é semelhante ao encapsulamento que as interfaces fornecem.

Outro uso comum de chamadas de retorno é definir um método de comparação personalizada e passar esse delegado para um método de classificação. Ele permite que o código do chamador se torne parte do algoritmo de classificação. O exemplo a seguir usa o tipo `Del` como um parâmetro:

C#

```
public static void MethodWithCallback(int param1, int param2, Callback
callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

Em seguida, você pode passar o delegado criado acima para esse método:

C#

```
MethodWithCallback(1, 2, handler);
```

e receber a seguinte saída para o console:

Console

```
The number is: 3
```

Usando o delegado como uma abstração, `MethodWithCallback` não precisa chamar o console diretamente — ele não precisa ser criado com um console em mente. O que `MethodWithCallback` faz é simplesmente preparar uma cadeia de caracteres e passá-la para outro método. Isso é especialmente poderoso, uma vez que um método delegado pode usar qualquer número de parâmetros.

Quando um delegado é construído para encapsular um método de instância, o delegado faz referência à instância e ao método. Um delegado não tem conhecimento do tipo de instância além do método que ele encapsula, de modo que um delegado

pode se referir a qualquer tipo de objeto desde que haja um método nesse objeto que corresponda à assinatura do delegado. Quando um delegado é construído para encapsular um método estático, ele só faz referência ao método. Considere as seguintes declarações:

```
C#  
  
public class MethodClass  
{  
    public void Method1(string message) { }  
    public void Method2(string message) { }  
}
```

Além do `DelegateMethod` estático mostrado anteriormente, agora temos três métodos que podem ser encapsulados por uma instância `De1`.

Um delegado pode chamar mais de um método quando invocado. Isso é chamado de multicast. Para adicionar um método extra à lista de métodos do delegado — a lista de invocação — basta adicionar dois delegados usando os operadores de adição ou de atribuição de adição ('+' ou '+ ='). Por exemplo:

```
C#  
  
var obj = new MethodClass();  
Callback d1 = obj.Method1;  
Callback d2 = obj.Method2;  
Callback d3 = DelegateMethod;  
  
//Both types of assignment are valid.  
Callback allMethodsDelegate = d1 + d2;  
allMethodsDelegate += d3;
```

Nesse ponto, `allMethodsDelegate` contém três métodos em sua lista de invocação — `Method1`, `Method2` e `DelegateMethod`. Os três delegados originais, `d1`, `d2` e `d3`, permanecem inalterados. Quando `allMethodsDelegate` é invocado, os três métodos são chamados na ordem. Se o delegado usar parâmetros de referência, a referência será passada em sequência para cada um dos três métodos por vez, e quaisquer alterações em um método serão visíveis no próximo método. Quando algum dos métodos gerar uma exceção que não foi detectada dentro do método, essa exceção será passada ao chamador do delegado e nenhum método subsequente na lista de invocação será chamado. Se o delegado tiver um valor de retorno e/ou parâmetros de saída, ele retornará o valor de retorno e os parâmetros do último método invocado. Para remover um método da lista de invocação, use os [operadores de atribuição de subtração ou subtração](#) (`-` ou `-=`). Por exemplo:

C#

```
//remove Method1  
allMethodsDelegate -= d1;  
  
// copy AllMethodsDelegate while removing d2  
Callback oneMethodDelegate = allMethodsDelegate - d2;
```

Como os tipos de delegados são derivados de `System.Delegate`, os métodos e as propriedades definidos por essa classe podem ser chamados no delegado. Por exemplo, para localizar o número de métodos na lista de invocação do delegado, é possível escrever:

C#

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Delegados com mais de um método em sua lista de invocação derivam de `MulticastDelegate`, que é uma subclasse de `System.Delegate`. O código acima funciona em ambos os casos, pois as classes oferecem suporte à `GetInvocationList`.

Delegados multicast são amplamente usados na manipulação de eventos. Objetos de origem do evento enviam notificações de eventos aos objetos de destinatário que se registraram para receber esse evento. Para se registrar para um evento, o destinatário cria um método projetado para lidar com o evento, em seguida, cria um delegado para esse método e passa o delegado para a origem do evento. A origem chama o delegado quando o evento ocorre. O delegado chama então o método de manipulação de eventos no destinatário, fornecendo os dados do evento. O tipo de delegado de um determinado evento é definido pela origem do evento. Para saber mais, consulte [Eventos](#).

A comparação de delegados de dois tipos diferentes atribuídos no tempo de compilação resultará em um erro de compilação. Se as instâncias de delegado forem estaticamente do tipo `System.Delegate`, então a comparação será permitida, mas retornará false no tempo de execução. Por exemplo:

C#

```
delegate void Callback1();  
delegate void Callback2();  
  
static void method(Callback1 d, Callback2 e, System.Delegate f)  
{  
    // Compile-time error.  
    //Console.WriteLine(d == e);
```

```
// OK at compile-time. False if the run-time type of f  
// is not the same as that of d.  
Console.WriteLine(d == f);  
}
```

## Confira também

- [Guia de Programação em C#](#)
- [Representantes](#)
- [Usando variação em delegados](#)
- [Variação em delegações](#)
- [Usando Variação para Delegações Genéricas Func e Action](#)
- [Eventos](#)

# Delegados com métodos nomeados versus anônimos (Guia de Programação em C#)

Artigo • 07/04/2023

Um [delegado](#) pode ser associado a um método nomeado. Ao instanciar um delegado usando um método nomeado, o método é passado como um parâmetro, por exemplo:

C#

```
// Declare a delegate.  
delegate void Del(int x);  
  
// Define a named method.  
void DoWork(int k) { /* ... */ }  
  
// Instantiate the delegate using the method as a parameter.  
Del d = obj.DoWork;
```

Isso é chamado usando um método nomeado. Os delegados construídos com um método nomeado podem encapsular um método [estático](#) ou um método de instância. Métodos nomeados são a única maneira de instanciar um delegado nas versões anteriores do C#. No entanto, em uma situação em que a criação de um novo método for uma sobrecarga indesejada, o C# permite instanciar um delegado e especificar imediatamente um bloco de código que esse delegado processará quando for chamado. O bloco pode conter uma [expressão lambda](#) ou um [método anônimo](#).

O método passado como parâmetro delegado deve ter a mesma assinatura da declaração delegada. Uma instância de delegado pode encapsular o método estático ou de instância.

## Observação

Embora o delegado possa usar um parâmetro [out](#), não é recomendável utilizá-lo com delegados de evento multicast, pois não é possível saber qual delegado será chamado.

Do C# 10 em diante, os grupos de métodos com uma só sobrecarga têm um *tipo natural*. Isso significa que o compilador pode inferir o tipo de retorno e os tipos de parâmetro para o tipo de delegado:

C#

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

## Exemplos

Este é um exemplo simples de declaração usando um delegado. Observe que tanto o delegado, `Del` e o método associado, `MultiplyNumbers`, têm a mesma assinatura

C#

```
// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
    }
}
/* Output:
   Invoking the delegate using 'MultiplyNumbers':
   2 4 6 8 10
*/
```

No exemplo a seguir, um delegado é mapeado para métodos estáticos e de instância e retorna informações específicas sobre cada um.

C#

```
// Declare a delegate
delegate void Del();

class SampleClass
{
    public void InstanceMethod()
    {
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Del d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
   A message from the instance method.
   A message from the static method.
*/
```

## Confira também

- [Guia de Programação em C#](#)
- [Representantes](#)
- [Como combinar delegados \(delegados multicast\)](#)
- [Eventos](#)

# Como combinar delegados (delegados multicast) (Guia de Programação em C#)

Artigo • 07/04/2023

Este exemplo demonstra como criar delegados multicast. Uma propriedade útil de objetos [delegados](#) é que vários objetos podem ser atribuídos a uma instância delegada usando o operador `+`. O delegado multicast contém uma lista dos delegados atribuídos. Quando o delegado multicast é chamado, ele invoca os delegados da lista, em ordem. Apenas os delegados do mesmo tipo podem ser combinados.

O operador `-` pode ser usado para remover um delegado de componente de um delegado multicast.

## Exemplo

C#

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel.
    static void Hello(string s)
    {
        Console.WriteLine($"  Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($"  Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Initialize the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;
```

```

// Initialize the delegate object byeDel that references the
// method Goodbye.
byeDel = Goodbye;

// The two delegates, hiDel and byeDel, are combined to
// form multiDel.
multiDel = hiDel + byeDel;

// Remove hiDel from the multicast delegate, leaving byeDel,
// which calls only the method Goodbye.
multiMinusHiDel = multiDel - hiDel;

Console.WriteLine("Invoking delegate hiDel:");
hiDel("A");
Console.WriteLine("Invoking delegate byeDel:");
byeDel("B");
Console.WriteLine("Invoking delegate multiDel:");
multiDel("C");
Console.WriteLine("Invoking delegate multiMinusHiDel:");
multiMinusHiDel("D");
}

}

/* Output:
Invoking delegate hiDel:
Hello, A!
Invoking delegate byeDel:
Goodbye, B!
Invoking delegate multiDel:
Hello, C!
Goodbye, C!
Invoking delegate multiMinusHiDel:
Goodbye, D!
*/

```

## Confira também

- [MulticastDelegate](#)
- [Guia de Programação em C#](#)
- [Eventos](#)

# Como declarar, instanciar e usar um delegado (Guia de Programação em C#)

Artigo • 07/04/2023

Você pode declarar representantes usando qualquer um dos seguintes métodos:

- Declare um tipo de delegado e declare um método com uma assinatura correspondente:

C#

```
// Declare a delegate.  
delegate void Del(string str);  
  
// Declare a method with the same signature as the delegate.  
static void Notify(string name)  
{  
    Console.WriteLine($"Notification received for: {name}");  
}
```

C#

```
// Create an instance of the delegate.  
Del del1 = new Del(Notify);
```

- Atribuir um grupo de métodos a um tipo de delegado:

C#

```
// C# 2.0 provides a simpler way to declare an instance of Del.  
Del del2 = Notify;
```

- Declarar um método anônimo:

C#

```
// Instantiate Del by using an anonymous method.  
Del del3 = delegate(string name)  
    { Console.WriteLine($"Notification received for: {name}"); };
```

- Use uma expressão lambda:

C#

```
// Instantiate Del by using a lambda expression.  
Del del4 = name => { Console.WriteLine($"Notification received for:  
{name}"); };
```

Para obter mais informações, consulte [Expressões Lambda](#).

O exemplo a seguir ilustra a declaração, instanciação e o uso de um delegado. A classe `BookDB` encapsula um banco de dados de uma livraria que mantém um banco de dados de livros. Ela expõe um método, `ProcessPaperbackBooks`, que localiza todos os livros de bolso no banco de dados e chama um delegado para cada um. O tipo `delegate` usado tem o nome `ProcessBookCallback`. A classe `Test` usa essa classe para imprimir os títulos e o preço médio dos livros de bolso.

O uso de delegados promove uma boa separação de funcionalidade entre o banco de dados da livraria e o código de cliente. O código de cliente não tem conhecimento de como os livros são armazenados ou como o código da livraria localiza os livros de bolso. O código da livraria não tem conhecimento do processamento executado nos livros de bolso após a localização.

## Exemplo

C#

```
// A set of classes for handling a bookstore:  
namespace Bookstore  
{  
    using System.Collections;  
  
    // Describes a book in the book list:  
    public struct Book  
    {  
        public string Title;           // Title of the book.  
        public string Author;          // Author of the book.  
        public decimal Price;          // Price of the book.  
        public bool Paperback;         // Is it paperback?  
  
        public Book(string title, string author, decimal price, bool  
paperBack)  
        {  
            Title = title;  
            Author = author;  
            Price = price;  
            Paperback = paperBack;  
        }  
    }  
  
    // Declare a delegate type for processing a book:  
}
```

```

public delegate void ProcessBookCallback(Book book);

// Maintains a book database.
public class BookDB
{
    // List of all books in the database:
    ArrayList list = new ArrayList();

    // Add a book to the database:
    public void AddBook(string title, string author, decimal price, bool
paperBack)
    {
        list.Add(new Book(title, author, price, paperBack));
    }

    // Call a passed-in delegate on each paperback book to process it:
    public void ProcessPaperbackBooks(ProcessBookCallback processBook)
    {
        foreach (Book b in list)
        {
            if (b.Paperback)
                // Calling the delegate:
                processBook(b);
        }
    }
}

// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class Test
    {
        // Print the title of the book.

```

```

        static void PrintTitle(Book b)
    {
        Console.WriteLine($"    {b.Title}");
    }

    // Execution starts here.
    static void Main()
    {
        BookDB bookDB = new BookDB();

        // Initialize the database with some books:
        AddBooks(bookDB);

        // Print all the titles of paperbacks:
        Console.WriteLine("Paperback Book Titles:");

        // Create a new delegate object associated with the static
        // method Test.PrintTitle:
        bookDB.ProcessPaperbackBooks(PrintTitle);

        // Get the average price of a paperback by using
        // a PriceTotaller object:
        PriceTotaller totaller = new PriceTotaller();

        // Create a new delegate object associated with the nonstatic
        // method AddBookToTotal on the object totaller:
        bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

        Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
                          totaller.AveragePrice());
    }

    // Initialize the book database with some test books:
    static void AddBooks(BookDB bookDB)
    {
        bookDB.AddBook("The C Programming Language", "Brian W. Kernighan
and Dennis M. Ritchie", 19.95m, true);
        bookDB.AddBook("The Unicode Standard 2.0", "The Unicode
Consortium", 39.95m, true);
        bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m,
false);
        bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott
Adams", 12.00m, true);
    }
}

/* Output:
Paperback Book Titles:
    The C Programming Language
    The Unicode Standard 2.0
    Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/

```

# Programação robusta

- Declarando um delegado.

A instrução a seguir declara um novo tipo de delegado.

C#

```
public delegate void ProcessBookCallback(Book book);
```

Cada tipo de delegado descreve o número e os tipos dos argumentos e o tipo do valor retornado dos métodos que pode encapsular. Sempre que um novo conjunto de tipos de argumento ou tipo de valor retornado for necessário, um novo tipo de delegado deverá ser declarado.

- Instanciando um delegado.

Após a declaração do tipo de delegado, um objeto delegado deve ser criado e associado a um método específico. No exemplo anterior, faça isso passando o método `PrintTitle` para o método `ProcessPaperbackBooks`, como no exemplo a seguir:

C#

```
bookDB.ProcessPaperbackBooks(PrintTitle);
```

Isso cria um novo objeto delegado associado ao método `estático Test.PrintTitle`. Da mesma forma, o método não estático `AddBookToTotal` no objeto `totaller` é passado como no exemplo a seguir:

C#

```
bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
```

Em ambos os casos, um novo objeto delegado é passado para o método `ProcessPaperbackBooks`.

Após a criação de um delegado, o método ao qual ele está associado nunca se altera; objetos delegados são imutáveis.

- Chamando um delegado.

Normalmente, o objeto delegado, após sua criação, é passado para outro código que chamará o delegado. Um objeto delegado é chamado usando seu nome

seguido dos argumentos entre parênteses a serem passados para o delegado. A seguir, veja um exemplo de uma chamada de delegado:

```
C#  
  
processBook(b);
```

Um delegado pode ser chamado de forma síncrona, como neste exemplo ou de forma assíncrona, usando os métodos `BeginInvoke` e `EndInvoke`.

## Confira também

- [Guia de Programação em C#](#)
- [Eventos](#)
- [Representantes](#)

# Matrizes (Guia de Programação em C#)

Artigo • 07/04/2023

Você pode armazenar diversas variáveis do mesmo tipo em uma estrutura de dados de matriz. Você pode declarar uma matriz especificando o tipo de seus elementos. Se você quiser que a matriz armazene elementos de qualquer tipo, você pode especificar `object` como seu tipo. No sistema de tipos unificado do C#, todos os tipos, predefinidos e definidos pelo usuário, tipos de referência e tipos de valor, herdam direta ou indiretamente de `Object`.

C#

```
type[] arrayName;
```

## Exemplo

O exemplo a seguir cria matrizes unidimensionais, multidimensionais e denteadas:

C#

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array of 5 integers.
        int[] array1 = new int[5];

        // Declare and set array element values.
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax.
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array.
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values.
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array.
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure.
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

# Visão geral de matriz

Uma matriz tem as seguintes propriedades:

- Uma matriz pode ser [unidimensional](#), [multidimensional](#) ou [denteada](#).
- O número de dimensões e o tamanho de cada dimensão são estabelecidos quando a instância de matriz é criada. Esses valores não podem ser alterados durante o ciclo de vida da instância.
- Os valores padrão dos elementos de matriz numérica são definidos como zero, e os elementos de referência são definidos como `null`.
- Uma matriz denteada é uma matriz de matrizes e, portanto, seus elementos são tipos de referência e são inicializados para `null`.
- As matrizes são indexadas por zero: uma matriz com elementos `n` é indexada de `0` para `n-1`.
- Os elementos de matriz podem ser de qualquer tipo, inclusive um tipo de matriz.
- Os tipos de matriz são [tipos de referência](#) derivados do tipo base abstrato [Array](#). Todas as matrizes implementam [IList](#) e [IEnumerable](#). Você pode usar a instrução [foreach](#) para iterar por meio de uma matriz. Matrizes unidimensionais também implementam [IList<T>](#) e [IEnumerable<T>](#).

## Comportamento de valor padrão

- Para tipos de valor, os elementos da matriz são inicializados com o [valor padrão](#), o padrão de 0 bit; os elementos terão o valor `0`.
- Todos os tipos de referência (incluindo o [não anulável](#)), têm os valores `null`.
- Para tipos de valor anuláveis, `HasValue` é definido como `false` e os elementos seriam definidos como `null`.

## Matrizes como Objetos

No C#, as matrizes são objetos e não apenas regiões endereçáveis de memória contígua, como no C e no C++. [Array](#) é o tipo base abstrato de todos os tipos de matriz. É possível usar as propriedades e outros membros de classe do [Array](#). Um exemplo disso é o uso da propriedade [Length](#) para obter o comprimento de uma matriz. O código a seguir atribui o comprimento da matriz `numbers`, que é `5`, a uma variável denominada `lengthOfNumbers`:

C#

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

A classe [Array](#) fornece vários outros métodos e propriedades úteis para classificar, pesquisar e copiar matrizes. O exemplo a seguir usa a propriedade [Rank](#) para exibir a quantidade de dimensões de uma matriz.

C#

```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array.
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.",
theArray.Rank);
    }
}
// Output: The array has 2 dimensions.
```

## Confira também

- [Como usar matrizes unidimensionais](#)
- [Como usar matrizes multidimensionais](#)
- [Como usar matrizes denteadas](#)
- [Usar foreach com matrizes](#)
- [Passando matrizes como argumentos](#)
- [Matrizes de tipo implícito](#)
- [Guia de Programação em C#](#)
- [Coleções](#)

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# Matrizes unidimensionais (Guia de Programação em C#)

Artigo • 07/04/2023

Você criará uma matriz unidimensional usando o [novo](#) operador que especifica o tipo de elemento de matriz e o número de elementos. O exemplo a seguir declara uma matriz de cinco inteiros:

C#

```
int[] array = new int[5];
```

Essa matriz contém os elementos de `array[0]` a `array[4]`. Os elementos da matriz são inicializados para o [valor padrão](#) do tipo de elemento, `0` para inteiros.

Matrizes podem armazenar qualquer tipo de elemento que você especificar, como o exemplo a seguir que declara uma matriz de cadeias de caracteres:

C#

```
string[] stringArray = new string[6];
```

## Inicialização de Matriz

Você pode inicializar os elementos de uma matriz ao declarar a matriz. O especificador de comprimento não é necessário porque é inferido pelo número de elementos na lista de inicialização. Por exemplo:

C#

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

O ícone a seguir mostra uma declaração de uma matriz de cadeia de caracteres em que cada elemento da matriz é inicializado por um nome de um dia:

C#

```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",  
"Sat" };
```

Você pode evitar a expressão `new` e o tipo de matriz ao inicializar uma matriz após a declaração, conforme mostrado no código a seguir. Isso é chamado de [matriz tipada implicitamente](#):

C#

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Você pode declarar uma variável de matriz sem criá-la, mas deve usar o operador `new` ao atribuir uma nova matriz a essa variável. Por exemplo:

C#

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 };    // OK
//array3 = {1, 3, 5, 7, 9};    // Error
```

## Matrizes de tipo de valor e de tipo de referência

Considere a seguinte declaração de matriz:

C#

```
SomeType[] array4 = new SomeType[10];
```

O resultado dessa instrução depende se `SomeType` é um tipo de valor ou um tipo de referência. Se for um tipo de valor, a instrução criará uma matriz de 10 elementos, cada um deles tem o tipo `SomeType`. Se `SomeType` for um tipo de referência, a instrução criará uma matriz de 10 elementos, cada um deles é inicializado com uma referência nula. Em ambas as instâncias, os elementos são inicializados para o valor padrão do tipo de elemento. Para obter mais informações sobre tipos de valor e de referência, consulte [Tipos de valor](#) e [Tipos de referência](#).

## Recuperar dados da matriz

Você pode recuperar os dados de uma matriz usando um índice. Por exemplo:

C#

```
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

Console.WriteLine(weekDays2[0]);
Console.WriteLine(weekDays2[1]);
Console.WriteLine(weekDays2[2]);
Console.WriteLine(weekDays2[3]);
Console.WriteLine(weekDays2[4]);
Console.WriteLine(weekDays2[5]);
Console.WriteLine(weekDays2[6]);

/*Output:
Sun
Mon
Tue
Wed
Thu
Fri
Sat
*/
```

## Confira também

- [Array](#)
- [matrizes](#)
- [Matrizes multidimensionais](#)
- [Matrizes denteadas](#)

# Matrizes multidimensionais (Guia de Programação em C#)

Artigo • 07/04/2023

As matrizes podem ter mais de uma dimensão. Por exemplo, a declaração a seguir cria uma matriz bidimensional de quatro linhas e duas colunas.

C#

```
int[,] array = new int[4, 2];
```

A declaração a seguir cria uma matriz de três dimensões, 4, 2 e 3.

C#

```
int[, ,] array1 = new int[4, 2, 3];
```

## Inicialização de Matriz

É possível inicializar a matriz na declaração, conforme mostrado no exemplo a seguir.

C#

```
// Two-dimensional array.  
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };  
// The same array with dimensions specified.  
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };  
// A similar array with string elements.  
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four"  
,  
{ "five", "six" } };  
  
// Three-dimensional array.  
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },  
{ { 7, 8, 9 }, { 10, 11, 12 } } };  
// The same array with dimensions specified.  
int[,,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },  
{ { 7, 8, 9 }, { 10, 11, 12 } } };  
  
// Accessing array elements.  
System.Console.WriteLine(array2D[0, 0]);  
System.Console.WriteLine(array2D[0, 1]);  
System.Console.WriteLine(array2D[1, 0]);  
System.Console.WriteLine(array2D[1, 1]);  
System.Console.WriteLine(array2D[3, 0]);
```

```
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);

// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12
```

Também é possível inicializar a matriz sem especificar a classificação.

C#

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Caso você escolha declarar uma variável de matriz sem inicialização, será necessário usar o operador `new` ao atribuir uma matriz a essa variável. O uso de `new` é mostrado no exemplo a seguir.

C#

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } }; // OK
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}}; // Error
```

O exemplo a seguir atribui um valor a um elemento de matriz específico.

C#

```
array5[2, 1] = 25;
```

Da mesma forma, o exemplo a seguir obtém o valor de um elemento de matriz específico e o atribui à variável `elementValue`.

C#

```
int elementValue = array5[2, 1];
```

O exemplo de código a seguir inicializa os elementos da matriz com valores padrão (exceto em matrizes denteadas).

C#

```
int[,] array6 = new int[10, 10];
```

## Confira também

- [Guia de Programação em C#](#)
- [matrizes](#)
- [Matrizes unidimensionais](#)
- [Matrizes denteadas](#)

# Matrizes denteadas (Guia de Programação em C#)

Artigo • 07/04/2023

Uma matriz irregular é uma matriz cujos elementos são matrizes, possivelmente de tamanhos diferentes. Uma matriz irregular às vezes é chamada de "matriz de matrizes". Os exemplos a seguir mostram como declarar, inicializar e acessar matrizes irregulares.

A seguir, há uma declaração de uma matriz unidimensional que tem três elementos, cada um do qual é uma matriz de dimensão única de inteiros:

C#

```
int[][] jaggedArray = new int[3][];
```

Antes de usar `jaggedArray`, seus elementos devem ser inicializados. É possível inicializar os elementos dessa forma:

C#

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Cada um dos elementos é uma matriz unidimensional de inteiros. O primeiro elemento é uma matriz de 5 inteiros, o segundo é uma matriz de 4 inteiros e o terceiro é uma matriz de 2 inteiros.

Também é possível usar os inicializadores para preencher os elementos matriz com valores, caso em que não é necessário o tamanho da matriz. Por exemplo:

C#

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

Também é possível inicializar a matriz mediante uma declaração como esta:

C#

```
int[][] jaggedArray2 = new int[][]
{
```

```
new int[] { 1, 3, 5, 7, 9 },
new int[] { 0, 2, 4, 6 },
new int[] { 11, 22 }
};
```

É possível usar a seguinte forma abreviada. Observe que não é possível omitir o operador `new` da inicialização de elementos, porque não há nenhuma inicialização padrão para os elementos:

C#

```
int[][] jaggedArray3 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

Uma matriz denteada é uma matriz de matrizes e, portanto, seus elementos são tipos de referência e são inicializados para `null`.

É possível acessar elementos de matrizes individuais como estes exemplos:

C#

```
// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray3[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray3[2][1] = 88;
```

É possível misturar matrizes irregulares e multidimensionais. A seguir, há uma declaração e inicialização de uma matriz denteada unidimensional que contém três elementos de matriz bidimensional de tamanhos diferentes. Para obter mais informações, confira [Matrizes multidimensionais](#).

C#

```
int[,] jaggedArray4 = new int[3][,
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};
```

É possível acessar elementos individuais conforme mostrado neste exemplo, que exibe o valor do elemento `[1,0]` da primeira matriz (valor `5`):

C#

```
System.Console.WriteLine("{0}", jaggedArray4[0][1, 0]);
```

O método `Length` retorna inúmeros conjuntos contidos na matriz denteada. Por exemplo, supondo que você tenha declarado a matriz anterior, esta linha:

C#

```
System.Console.WriteLine(jaggedArray4.Length);
```

retorna um valor de `3`.

## Exemplo

Este exemplo cria uma matriz cujos elementos são matrizes. Cada um dos elementos da matriz tem um tamanho diferente.

C#

```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements.
        int[][] arr = new int[2][];

        // Initialize the elements.
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements.
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j ==
(arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
    }
}
```

```
        System.Console.ReadKey();
    }
}

/* Output:
   Element(0): 1 3 5 7 9
   Element(1): 2 4 6 8
*/
```

## Confira também

- [Array](#)
- [Guia de Programação em C#](#)
- [matrizes](#)
- [Matrizes unidimensionais](#)
- [Matrizes multidimensionais](#)

# Usar foreach com matrizes (Guia de Programação em C#)

Artigo • 07/04/2023

Essa instrução `foreach` fornece uma maneira simples e limpa de iterar através dos elementos de uma matriz.

Em matrizes unidimensionais, a instrução `foreach` processa elementos em ordem crescente de índice, começando com o índice 0 e terminando com índice `Length - 1`:

C#

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

Em matrizes multidimensionais, os elementos são percorridos de modo a que os índices da dimensão mais à direita sejam aumentados primeiro e, em seguida, da próxima dimensão à esquerda, e assim por diante seguindo para a esquerda:

C#

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

No entanto, com matrizes multidimensionais, usar um loop aninhado `for` oferece mais controle sobre a ordem na qual processar os elementos da matriz.

## Confira também

- [Array](#)
- [Guia de Programação em C#](#)
- [matrizes](#)

- Matrizes unidimensionais
- Matrizes multidimensionais
- Matrizes denteadas

# Passando matrizes como argumentos (Guia de Programação em C#)

Artigo • 07/04/2023

As matrizes podem ser passadas como argumentos para parâmetros de método. Como matrizes são tipos de referência, o método pode alterar o valor dos elementos.

## Passando matrizes unidimensionais como argumentos

É possível passar uma matriz unidimensional inicializada para um método. Por exemplo, a instrução a seguir envia uma matriz a um método de impressão.

C#

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

O código a seguir mostra uma implementação parcial do método de impressão.

C#

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

É possível inicializar e passar uma nova matriz em uma etapa, conforme mostrado no exemplo a seguir.

C#

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

## Exemplo

No exemplo a seguir, uma matriz de cadeia de caracteres é inicializada e passada como um argumento para um método `DisplayArray` para cadeias de caracteres. O método exibe os elementos da matriz. Em seguida, o método `ChangeArray` inverte os elementos

da matriz, e o método `ChangeArrayElements` modifica os três primeiros elementos da matriz. Depois que cada método retorna, o método `DisplayArray` mostra que passar uma matriz por valor não impede alterações nos elementos da matriz.

C#

```
using System;

class ArrayExample
{
    static void DisplayArray(string[] arr) =>
Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
"Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to
ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}

// The example displays the following output:
//      Sun Mon Tue Wed Thu Fri Sat
//
//      Array weekDays after the call to ChangeArray:
//      Sat Fri Thu Tue Mon Sun
```

```
//  
//      Array weekDays after the call to ChangeArrayElements:  
//      Mon Wed Fri Wed Tue Mon Sun
```

## Passando matrizes multidimensionais como argumentos

Você passa uma matriz multidimensional inicializada para um método da mesma forma que você passa uma matriz unidimensional.

C#

```
int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };  
Print2DArray(theArray);
```

O código a seguir mostra uma declaração parcial de um método de impressão que aceita uma matriz bidimensional como seu argumento.

C#

```
void Print2DArray(int[,] arr)  
{  
    // Method code.  
}
```

É possível inicializar e passar uma nova matriz em uma única etapa, conforme é mostrado no exemplo a seguir:

C#

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
```

## Exemplo

No exemplo a seguir, uma matriz bidimensional de inteiros é inicializada e passada para o método `Print2DArray`. O método exibe os elementos da matriz.

C#

```
class ArrayClass2D  
{  
    static void Print2DArray(int[,] arr)  
    {
```

```

// Display the array elements.
for (int i = 0; i < arr.GetLength(0); i++)
{
    for (int j = 0; j < arr.GetLength(1); j++)
    {
        System.Console.WriteLine("Element({0},{1})={2}", i, j,
arr[i, j]);
    }
}
static void Main()
{
    // Pass the array as an argument.
    Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
/* Output:
   Element(0,0)=1
   Element(0,1)=2
   Element(1,0)=3
   Element(1,1)=4
   Element(2,0)=5
   Element(2,1)=6
   Element(3,0)=7
   Element(3,1)=8
*/

```

## Confira também

- [Guia de Programação em C#](#)
- [matrizes](#)
- [Matrizes unidimensionais](#)
- [Matrizes multidimensionais](#)
- [Matrizes denteadas](#)

# Matrizes de tipo implícito (Guia de Programação em C#)

Artigo • 07/04/2023

É possível criar uma matriz de tipo implícito na qual o tipo da instância da matriz é inferido com base nos elementos especificados no inicializador de matriz. As regras para qualquer variável de tipo implícito também se aplicam a matrizes de tipo implícito. Para obter mais informações, consulte [Variáveis locais de tipo implícito](#).

Geralmente, as matrizes de tipo implícito são usadas em expressões de consulta juntamente com objetos e tipos anônimos e inicializadores de coleção.

Os exemplos a seguir mostram como criar uma matriz de tipo implícito:

```
C#  
  
class ImplicitlyTypedArraySample  
{  
    static void Main()  
    {  
        var a = new[] { 1, 10, 100, 1000 }; // int[]  
        var b = new[] { "hello", null, "world" }; // string[]  
  
        // single-dimension jagged array  
        var c = new[]  
        {  
            new[]{1,2,3,4},  
            new[]{5,6,7,8}  
        };  
  
        // jagged array of strings  
        var d = new[]  
        {  
            new[]{"Luca", "Mads", "Luke", "Dinesh"},  
            new[]{"Karen", "Suma", "Frances"}  
        };  
    }  
}
```

No exemplo anterior, observe que, com as matrizes de tipo implícito, não são usados colchetes do lado esquerdo da instrução de inicialização. Observe também que as matrizes denteadas são inicializadas usando `new []` assim como matrizes unidimensionais.

# Matrizes de tipo implícito em Inicializadores de objeto

Ao criar um tipo anônimo que contém uma matriz, ela deve ser de tipo implícito no inicializador de objeto do tipo. No exemplo a seguir, `contacts` é uma matriz de tipo implícito de tipos anônimos, cada um contém uma matriz denominada `PhoneNumbers`. Observe que a palavra-chave `var` não é usada dentro dos inicializadores de objeto.

C#

```
var contacts = new[]
{
    new {
        Name = " Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new {
        Name = " Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

## Confira também

- [Guia de Programação em C#](#)
- [Variáveis Locais Tipadas Implicitamente](#)
- [matrizes](#)
- [Tipos anônimos](#)
- [Inicializadores de objeto e coleção](#)
- [var](#)
- [LINQ em C#](#)

# Cadeias de caracteres e literais de cadeia de caracteres

Artigo • 15/02/2023

Uma cadeia de caracteres é um objeto do tipo [String](#) cujo valor é texto. Internamente, o texto é armazenado como uma coleção sequencial somente leitura de objetos [Char](#). Não há um caractere de finalização null ao fim de uma cadeia em C#. Portanto, uma cadeia de caracteres em C# pode ter qualquer número de caracteres nulos inseridos ('\0'). A propriedade [Char](#) de uma cadeia de caracteres representa o número de objetos [Length](#) que ela contém e não o número de caracteres Unicode. Para acessar os pontos de código Unicode individuais em uma cadeia de caracteres, use o objeto [StringInfo](#).

## Cadeia de caracteres versus `System.String`

Em C#, a palavra-chave `string` é um alias para [String](#). Portanto, `String` e `string` são equivalentes, independentemente de ser recomendável usar o alias `string` fornecido, pois ele funciona mesmo sem `using System;`. A classe `String` fornece vários métodos para criar, manipular e comparar cadeias de caracteres com segurança. Além disso, a linguagem C# sobrecarrega alguns operadores para simplificar operações comuns de cadeia de caracteres. Para saber mais sobre a palavra-chave, confira [cadeia de caracteres](#). Para obter mais informações sobre o tipo e seus métodos, consulte [String](#).

## Declaração e inicialização de cadeias de caracteres

Você pode declarar e inicializar cadeias de caracteres de várias maneiras, conforme mostrado no seguinte exemplo:

```
C#  
  
// Declare without initializing.  
string message1;  
  
// Initialize to null.  
string message2 = null;  
  
// Initialize as an empty string.  
// Use the Empty constant instead of the literal "".  
string message3 = System.String.Empty;  
  
// Initialize with a regular string literal.
```

```

string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\\Program Files\\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);

```

Você não usa o operador `new` para criar um objeto de cadeia de caracteres, exceto ao inicializar a cadeia de caracteres com uma matriz de caracteres.

Inicialize uma cadeia de caracteres com o valor constante `Empty` para criar um novo objeto `String` cuja cadeia de caracteres tem comprimento zero. A representação de cadeia de caracteres literal de uma cadeia de caracteres de comprimento zero é `""`. Ao inicializar cadeias de caracteres com o valor `Empty` em vez de `nulo`, você poderá reduzir as chances de uma `NullReferenceException` ocorrer. Use o método estático `IsNullOrEmpty(String)` para verificar o valor de uma cadeia de caracteres antes de tentar acessá-la.

## Imutabilidade das cadeias de caracteres

Objetos de cadeia de caracteres são *imutáveis*: não pode ser alterados após serem criados. Todos os métodos `String` e operadores C# que aparecem para modificar uma cadeia de caracteres retornam, na verdade, os resultados em um novo objeto de cadeia de caracteres. No exemplo a seguir, quando o conteúdo de `s1` e `s2` é concatenado para formar uma única cadeia de caracteres, as duas cadeias de caracteres originais são modificadas. O operador `+=` cria uma nova cadeia de caracteres que tem o conteúdo combinado. Esse novo objeto é atribuído à variável `s1`, e o objeto original que foi atribuído a `s1` é liberado para coleta de lixo, pois nenhuma outra variável contém uma referência a ele.

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Como uma cadeia de caracteres de "modificação" na verdade é uma nova criação de cadeia de caracteres, você deve ter cuidado ao criar referências em cadeias de caracteres. Se você criar uma referência a uma cadeia de caracteres e "modificar" a cadeia de caracteres original, a referência continuará apontar para o objeto original em vez do novo objeto que foi criado quando a cadeia de caracteres foi modificada. O código a seguir ilustra esse comportamento:

```
C#

string str1 = "Hello ";
string str2 = str1;
str1 += "World";

System.Console.WriteLine(str2);
//Output: Hello
```

Para saber mais sobre como criar novas cadeias de caracteres que se baseiam em modificações, como operações de pesquisa e substituição na cadeia de caracteres original, confira [Como modificar o conteúdo da cadeia de caracteres](#).

## Literais de cadeia de caracteres entre aspas

Os *literais de cadeia de caracteres entre aspas* são iniciar e terminar com um único caractere de aspas duplas ("") na mesma linha. Literais de cadeia de caracteres entre aspas são mais adequados para cadeias de caracteres que se encaixam em uma única linha e não incluem [sequências de escape](#). Um literal de cadeia de caracteres entre aspas deve inserir caracteres de escape, conforme demonstrado no exemplo a seguir:

```
C#

string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1      Column 2      Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
```

```

/* Output:
   Row 1
   Row 2
   Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The A\u00C6olian Harp", by Samuel Taylor Coleridge

```

## Literais de cadeia de caracteres textuais

*Literais de cadeia de caracteres verbatim* são mais convenientes para cadeias de caracteres de várias linhas, cadeias de caracteres que contêm caracteres de barra invertida ou aspas duplas inseridas. Cadeias de caracteres verbatim preservam caracteres de nova linha como parte do texto da cadeia de caracteres. Use aspas duplas para inserir uma marca de aspas simples dentro de uma cadeia de caracteres textual. O exemplo a seguir mostra alguns usos comuns para cadeias de caracteres textuais:

C#

```

string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,....
*/

string quote = @"Her name was ""Sara."";
//Output: Her name was "Sara."

```

## Literais de cadeia de caracteres bruta

A partir do C# 11, é possível usar *literais de cadeia de caracteres bruta* para criar cadeias de caracteres multilinha com mais facilidade ou usar quaisquer caracteres que exijam sequências de escape. *Literais de cadeia de caracteres bruta* removem a necessidade de usar sequências de escape. É possível gravar a cadeia de caracteres, incluindo a formatação de espaço em branco, da forma que ela deve aparecer na saída. Um *literal de cadeia de caracteres bruta*:

- Inicia e termina com uma sequência de pelo menos três caracteres de aspas duplas (""""). É permitido que mais de três caracteres consecutivos iniciem e terminem a sequência para dar suporte a literais de cadeia de caracteres que contêm três (ou mais) caracteres de aspas repetidos.
- Literais de cadeia de caracteres bruta de linha única exigem os caracteres de aspas de abertura e fechamento na mesma linha.
- Literais de cadeia de caracteres bruta multilinha exigem caracteres de aspas de abertura e fechamento em suas próprias linhas.
- Nos literais de cadeia de caracteres bruta multilinha, qualquer espaço em branco à esquerda das aspas de fechamento é removido.

Os exemplos a seguir demonstram essas regras:

```
C#  
  
string singleLine = """Friends say "hello" as they pass by.""";  
string multiLine = """  
    Hello World!" is typically the first program someone writes.  
    """;  
string embeddedXML = """  
    <element attr = "content">  
        <body style="normal">  
            Here is the main text  
        </body>  
        <footer>  
            Excerpts from "An amazing story"  
        </footer>  
    </element >  
    """;  
// The line "<element attr = "content">" starts in the first column.  
// All whitespace left of that column is removed from the string.  
  
string rawStringLiteralDelimiter = """  
    Raw string literals are delimited  
    by a string of at least three double quotes,  
    like this: """  
    """;
```

Os exemplos a seguir demonstram os erros do compilador relatados com base nessas regras:

```
C#  
  
// CS8997: Unterminated raw string literal.  
var multiLineStart = """This  
    is the beginning of a string  
    """;  
  
// CS9000: Raw string literal delimiter must be on its own line.
```

```
var multiLineEnd = """
    This is the beginning of a string """;

// CS8999: Line does not start with the same whitespace as the closing line
// of the raw string literal
var noOutdenting = """
    A line of text.

Trying to outdent the second line.
""";
```

Os dois primeiros exemplos são inválidos porque literais de cadeia de caracteres bruta multilinha exigem a sequência de aspas de abertura e fechamento em sua própria linha. O terceiro exemplo é inválido porque o texto é recuado para a esquerda da sequência das aspas de fechamento.

Você deve considerar literais de cadeia de caracteres bruta ao gerar texto que inclua caracteres que exigem [sequências de escape](#) ao usar literais de cadeia de caracteres entre aspas ou literais de cadeia de caracteres verbatim. Literais de cadeia de caracteres bruta podem ser lidos com mais facilidade por você e outras pessoas, pois serão mais parecidos com o texto de saída. Por exemplo, considere o seguinte código que inclui uma cadeia de caracteres de JSON formatado:

C#

```
string jsonString = """
{
    "Date": "2019-08-01T00:00:00-07:00",
    "TemperatureCelsius": 25,
    "Summary": "Hot",
    "DatesAvailable": [
        "2019-08-01T00:00:00-07:00",
        "2019-08-02T00:00:00-07:00"
    ],
    "TemperatureRanges": {
        "Cold": {
            "High": 20,
            "Low": -10
        },
        "Hot": {
            "High": 60,
            "Low": 20
        }
    },
    "SummaryWords": [
        "Cool",
        "Windy",
        "Humid"
    ]
}""";
```

Compare esse texto com o texto equivalente em nosso exemplo de [serialização JSON](#), que não usa esse novo recurso.

## Sequências de escape de cadeia de caracteres

Sequência de escape	Nome do caractere	Codificação Unicode
\'	Aspas simples	0x0027
\\"	Aspas duplas	0x0022
\`	Barra invertida	0x005C
\0	Nulo	0x0000
\a	Alerta	0x0007
\b	Backspace	0x0008
\f	Avanço de formulário	0x000C
\n	Nova linha	0x000A
\r	Retorno de carro	0x000D
\t	Guia horizontal	0x0009
\v	Guia vertical	0x000B
\u	Sequência de escape Unicode (UTF-16)	\uHHHH (intervalo: 0000 - FFFF; exemplo: \u00E7 = "ç")
\U	Sequência de escape Unicode (UTF-32)	\U00HHHHHH (intervalo: 000000 - 10FFFF; exemplo: \U0001F47D = "👽")
\x	Sequência de escape Unicode semelhante a "\u", exceto pelo comprimento variável	\xH[H][H][H] (intervalo: 0 - FFFF; exemplo: \x00E7 ou \xE7 ou \xE7 = "ç")

### Aviso

Ao usar a sequência de escape `\x` e especificar menos de quatro dígitos hexadecimais, se os caracteres que seguem imediatamente a sequência de escape são dígitos hexadecimais válidos (ou seja, 0 a 9, A-F e a-f), eles serão interpretados como sendo parte da sequência de escape. Por exemplo, `\xA1` produz "j", que é o ponto de código U+00A1. No entanto, se o próximo caractere é "A" ou "a", então a

sequência de escape será, em vez disso, interpretada como sendo `\xA1A` e produzirá "𠂇", que é o ponto de código U+0A1A. Nesses casos, especificar todos os quatro dígitos hexadecimais (por exemplo, `\x00A1`) impedirá qualquer interpretação errônea possível.

### ⓘ Observação

Em tempo de compilação, cadeias de caracteres textuais são convertidas em cadeias de caracteres comuns com as mesmas sequências de escape. Portanto, se exibir uma cadeia de caracteres textual na janela de observação do depurador, você verá os caracteres de escape que foram adicionados pelo compilador, não a versão textual do código-fonte. Por exemplo, a cadeia de caracteres textual `@"C:\\files.txt"` será exibida na janela de inspeção como "C:\\files.txt".

## Cadeias de caracteres de formato

Uma cadeia de caracteres de formato é aquela cujo conteúdo pode ser determinado dinamicamente no runtime. Cadeias de caracteres de formato são criadas incorporando *expressões interpoladas* ou espaços reservados dentro de chaves dentro em uma cadeia de caracteres. Tudo dentro das chaves `{...}` será resolvido para um valor e uma saída como uma cadeia de caracteres formatada no runtime. Há dois métodos para criar cadeias de caracteres de formato: cadeia de caracteres de interpolação e formatação de composição.

## Interpolação de cadeia de caracteres

Disponíveis no C# 6.0 e posterior, as *cadeias de caracteres interpoladas* são identificadas pelo caractere especial `$` e incluem expressões interpoladas entre chaves. Se você não estiver familiarizado com a interpolação de cadeia de caracteres, confira o tutorial [Interpolação de cadeia de caracteres – tutorial interativo do C#](#).

Use a interpolação de cadeia de caracteres para melhorar a legibilidade e a facilidade de manutenção do seu código. A interpolação de cadeia de caracteres alcança os mesmos resultados que o método `String.Format`, mas aumenta a facilidade de uso e a clareza embutida.

C#

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published:  
1761);
```

```

Console.WriteLine($"jh.firstName} {jh.lastName} was an African American
poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of
{jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) *
100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.

```

A partir do C# 10, você pode usar a interpolação de cadeia de caracteres para inicializar uma cadeia de caracteres constante quando todas as expressões usadas para espaços reservados também são cadeias de caracteres constantes.

A partir do C# 11, você pode combinar *literais de cadeia de caracteres bruta* com interpolações de cadeia de caracteres. Você inicia e termina a cadeia de caracteres de formato com três ou mais aspas duplas sucessivas. Se a cadeia de caracteres de saída precisar conter o caractere { ou }, você poderá usar caracteres extras \$ para especificar quantos caracteres { e } iniciam e encerram uma interpolação. Qualquer sequência com menos de { ou } caracteres é incluída na saída. O exemplo a seguir mostra como você pode usar esse recurso para exibir a distância de um ponto da origem e colocar o ponto dentro de chaves:

C#

```

int X = 2;
int Y = 3;

var pointMessage = $$""The point {{X}}, {{Y}} is {{Math.Sqrt(X * X + Y *
Y)}} from the origin.""";

Console.WriteLine(pointMessage);
// Output:
// The point {2, 3} is 3.605551275463989 from the origin.

```

## Formatação de composição

O [String.Format](#) utiliza os espaços reservados entre chaves para criar uma cadeia de caracteres de formato. Este exemplo resulta em uma saída semelhante para o método de interpolação de cadeia de caracteres usado acima.

C#

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.",
pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.",
pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

Para mais informações sobre formatação de tipos .NET, confira [Tipos de formatação em .NET](#).

## Subcadeias de caracteres

Uma subcadeia de caracteres é qualquer sequência de caracteres contida em uma cadeia de caracteres. Use o método [Substring](#) para criar uma nova cadeia de caracteres com base em uma parte da cadeia de caracteres original. Você pode pesquisar uma ou mais ocorrências de uma subcadeia de caracteres usando o método [IndexOf](#). Use o método [Replace](#) para substituir todas as ocorrências de uma subcadeia de caracteres especificada por uma nova cadeia de caracteres. Como o método [Substring](#), [Replace](#) retorna, na verdade, uma nova cadeia de caracteres e não a modifica a cadeia de caracteres original. Saiba mais em [Como pesquisar cadeias de caracteres](#) e [Como modificar o conteúdo da cadeia de caracteres](#).

C#

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

## Acesso a caracteres individuais

Você pode usar a notação de matriz com um valor de índice para adquirir acesso somente leitura a caracteres individuais, como no seguinte exemplo:

```
C#
```

```
string s5 = "Printing backwards";  
  
for (int i = 0; i < s5.Length; i++)  
{  
    System.Console.Write(s5[s5.Length - i - 1]);  
}  
// Output: "sdrawkcab gnitnirP"
```

Se os métodos [String](#) não fornecerem a funcionalidade necessária para modificar caracteres individuais em uma cadeia de caracteres, você poderá usar um objeto [StringBuilder](#) para modificar os caracteres individuais "in-loco" e criar uma nova cadeia de caracteres para armazenar os resultados usando os métodos [StringBuilder](#). No exemplo a seguir, suponha que você deva modificar a cadeia de caracteres original de uma maneira específica e armazenar os resultados para uso futuro:

```
C#
```

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS LOCK KEY?";  
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);  
  
for (int j = 0; j < sb.Length; j++)  
{  
    if (System.Char.IsLower(sb[j]) == true)  
        sb[j] = System.Char.ToUpper(sb[j]);  
    else if (System.Char.IsUpper(sb[j]) == true)  
        sb[j] = System.Char.ToLower(sb[j]);  
}  
// Store the new string.  
string corrected = sb.ToString();  
System.Console.WriteLine(corrected);  
// Output: How does Microsoft Word deal with the Caps Lock key?
```

## Cadeias de caracteres nulas e cadeias de caracteres vazias

Uma cadeia de caracteres vazia é uma instância de um objeto [System.String](#) que contém zero caractere. As cadeias de caracteres vazias geralmente são usadas em vários cenários de programação para representar um campo de texto em branco. Você pode chamar métodos em cadeias de caracteres vazias porque eles são objetos [System.String](#) válidos. As cadeias de caracteres vazias são inicializadas da seguinte maneira:

C#

```
string s = String.Empty;
```

Por outro lado, uma cadeia de caracteres nula não se refere a uma instância de um objeto [System.String](#) e qualquer tentativa de chamar um método em uma cadeia de caracteres nula provocará uma [NullReferenceException](#). No entanto, você pode usar cadeias de caracteres nulas em operações de comparação e concatenação com outras cadeias de caracteres. Os exemplos a seguir ilustram alguns casos em que uma referência a uma cadeia de caracteres nula faz e não faz com que uma exceção seja lançada:

C#

```
string str = "hello";
string nullStr = null;
string emptyStr = String.Empty;

string tempStr = str + nullStr;
// Output of the following line: hello
Console.WriteLine(tempStr);

bool b = (emptyStr == nullStr);
// Output of the following line: False
Console.WriteLine(b);

// The following line creates a new empty string.
string newStr = emptyStr + nullStr;

// Null strings and empty strings behave differently. The following
// two lines display 0.
Console.WriteLine(emptyStr.Length);
Console.WriteLine(newStr.Length);
// The following line raises a NullReferenceException.
//Console.WriteLine(nullStr.Length);

// The null character can be displayed and counted, like other chars.
string s1 = "\x0" + "abc";
string s2 = "abc" + "\x0";
// Output of the following line: * abc*
Console.WriteLine("*" + s1 + "*");
// Output of the following line: *abc *
Console.WriteLine("*" + s2 + "*");
// Output of the following line: 4
Console.WriteLine(s2.Length);
```

# Uso do stringBuilder para a criação rápida de cadeias de caracteres

As operações de cadeia de caracteres no .NET são altamente otimizadas e, na maioria dos casos, não afetam o desempenho de forma significativa. No entanto, em alguns cenários, como loops rígidos que são executados centenas ou milhares de vezes, as operações de cadeia de caracteres podem afetar o desempenho. A classe [StringBuilder](#) cria um buffer de cadeia de caracteres que oferece desempenho melhor se o programa executa várias manipulações de cadeia de caracteres. A cadeia de caracteres [StringBuilder](#) também permite reatribuir caracteres individuais, o que o tipo de dados String interno não dá suporte. Esse código, por exemplo, altera o conteúdo de uma cadeia de caracteres sem criar uma nova cadeia de caracteres:

C#

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal  
pet");  
sb[0] = 'C';  
System.Console.WriteLine(sb.ToString());  
//Outputs Cat: the ideal pet
```

Neste exemplo, um objeto [StringBuilder](#) é usado para criar uma cadeia de caracteres com base em um conjunto de tipos numéricos:

C#

```
var sb = new StringBuilder();  
  
// Create a string composed of numbers 0 - 9  
for (int i = 0; i < 10; i++)  
{  
    sb.Append(i.ToString());  
}  
Console.WriteLine(sb); // displays 0123456789  
  
// Copy one character of the string (not possible with a System.String)  
sb[0] = sb[9];  
  
Console.WriteLine(sb); // displays 9123456789
```

## Cadeias de caracteres, métodos de extensão e LINQ

Uma vez que o tipo `String` implementa `IEnumerable<T>`, você pode usar os métodos de extensão definidos na classe `Enumerable` em cadeias de caracteres. Para evitar a desordem visual, esses métodos são excluídos do IntelliSense para o tipo `String`, mas estão disponíveis mesmo assim. Você também pode usar a expressão de consulta LINQ em cadeias de caracteres. Para saber mais, confira [LINQ e cadeias de caracteres](#).

## Artigos relacionados

- [Como modificar o conteúdo de uma cadeia de caracteres](#): ilustra as técnicas para transformar cadeias de caracteres e modificar o conteúdo delas.
- [Como comparar cadeias de caracteres](#): mostra como executar comparações ordinais e específicas da cultura de cadeias de caracteres.
- [Como concatenar várias cadeias de caracteres](#): demonstra várias maneiras de unir diversas cadeias de caracteres em uma só.
- [Como analisar cadeias de caracteres usando String.Split](#): contém exemplos de código que descrevem como usar o método `String.Split` para analisar cadeias de caracteres.
- [Como pesquisar cadeias de caracteres](#): explica como usar a pesquisa para texto específico ou padrões em cadeias de caracteres.
- [Como determinar se uma cadeia de caracteres representa um valor numérico](#): mostra como analisar com segurança uma cadeia de caracteres para ver se ela tem um valor numérico válido.
- [Interpolação de cadeias de caracteres](#): descreve o recurso de interpolação de cadeia de caracteres que fornece uma sintaxe prática para cadeias de caracteres de formato.
- [Operações básicas de cadeias de caracteres](#): fornece links para artigos que usam os métodos `System.String` e `System.Text.StringBuilder` para executar operações básicas de cadeia de caracteres.
- [Analizando cadeias de caracteres](#): descreve como converter representações de cadeia de caracteres de tipos base do .NET em instâncias de tipos correspondentes.
- [Como analisar cadeias de caracteres de data e hora no .NET](#): mostra como converter uma cadeia de caracteres como "24/01/2008" em um objeto `System.DateTime`.
- [Comparando cadeias de caracteres](#): inclui informações sobre como comparar cadeias de caracteres e fornece exemplos em C# e Visual Basic.
- [Uso da classe StringBuilder](#): descreve como criar e modificar objetos de cadeias de caracteres dinâmicas usando a classe `StringBuilder`.
- [LINQ e Strings](#): fornece informações sobre como executar várias operações de cadeia de caracteres usando consultas LINQ.

# Como determinar se uma cadeia de caracteres representa um valor numérico (Guia de Programação em C#)

Artigo • 08/04/2023

Para determinar se uma cadeia de caracteres é uma representação válida de um tipo numérico especificado, use o método estático `TryParse` implementado por todos os tipos numéricos primitivos e também por tipos como `DateTime` e `IPAddress`. O exemplo a seguir mostra como determinar se "108" é um `int` válido.

C#

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

Se a cadeia de caracteres contiver caracteres não numéricos ou o valor numérico for muito grande ou muito pequeno para o tipo especificado, `TryParse` retornará false e definirá o parâmetro de saída como zero. Caso contrário, ele retornará true e definirá o parâmetro de saída como o valor numérico da cadeia de caracteres.

## ⓘ Observação

Uma cadeia de caracteres pode conter apenas caracteres numéricos e ainda não ser válida para o método `TryParse` do tipo usado. Por exemplo, "256" não é um valor válido para `byte`, mas é válido para `int`. "98,6" não é um valor válido para `int`, mas é válido para `decimal`.

## Exemplo

Os exemplos a seguir mostram como usar `TryParse` com representações de cadeia de caracteres dos valores `long`, `byte` e `decimal`.

C#

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
```

```
if (canConvert == true)
Console.WriteLine("number1 now = {0}", number1);
else
Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
Console.WriteLine("number2 now = {0}", number2);
else
Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; // "27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
Console.WriteLine("number3 now = {0}", number3);
else
Console.WriteLine("number3 is not a valid decimal");
```

## Programação robusta

Os tipos numéricos primitivos também implementam o método estático `Parse`, que lançará uma exceção se a cadeia de caracteres não for um número válido. Geralmente, `TryParse` é mais eficiente, pois retornará `false` apenas se o número não for válido.

## Segurança do .NET

Sempre use os métodos `TryParse` ou `Parse` para validar entradas de usuário em controles como caixas de texto e caixas de combinação.

## Confira também

- [Como converter uma matriz de bytes em um int](#)
- [Como converter uma cadeia de caracteres em um número](#)
- [Como converter entre cadeias de caracteres hexadecimais e tipos numéricos](#)
- [Analizando cadeias de caracteres numéricas](#)
- [Formatar tipos](#)

# Indexadores (Guia de Programação em C#)

Artigo • 15/02/2023

Os indexadores permitem que instâncias de uma classe ou struct sejam indexados como matrizes. O valor indexado pode ser definido ou recuperado sem especificar explicitamente um membro de instância ou tipo. Os indexadores parecem com [propriedades](#), a diferença é que seus acessadores usam parâmetros.

O exemplo a seguir define uma classe genérica com métodos de acesso [get](#) e [set](#) simples para atribuir e recuperar valores. A classe `Program` cria uma instância dessa classe para armazenar cadeias de caracteres.

```
C#  
  
using System;  
  
class SampleCollection<T>  
{  
    // Declare an array to store the data elements.  
    private T[] arr = new T[100];  
  
    // Define the indexer to allow client code to use [] notation.  
    public T this[int i]  
    {  
        get { return arr[i]; }  
        set { arr[i] = value; }  
    }  
}  
  
class Program  
{  
    static void Main()  
    {  
        var stringCollection = new SampleCollection<string>();  
        stringCollection[0] = "Hello, World";  
        Console.WriteLine(stringCollection[0]);  
    }  
}  
// The example displays the following output:  
//      Hello, World.
```

## ⓘ Observação

Para mais exemplos, consulte as seções relacionadas.

# Definições de corpo de expressão

É comum para um acessador get ou set de um indexador ser constituído de uma única instrução que retorna ou define um valor. Os membros de expressão fornecem uma sintaxe simplificada para dar suporte a esse cenário. Começando do C# 6, um indexador somente leitura pode ser implementado como um membro de expressão, como mostra o exemplo a seguir.

C#

```
using System;

class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];
    int nextIndex = 0;

    // Define the indexer to allow client code to use [] notation.
    public T this[int i] => arr[i];

    public void Add(T value)
    {
        if (nextIndex >= arr.Length)
            throw new IndexOutOfRangeException($"The collection can hold only
{arr.Length} elements.");
        arr[nextIndex++] = value;
    }
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection.Add("Hello, World");
        System.Console.WriteLine(stringCollection[0]);
    }
}
// The example displays the following output:
//      Hello, World.
```

Observe que `=>` apresenta o corpo da expressão e que a palavra-chave `get` não é usada.

Começando do C# 7.0, os acessadores get e set podem ser implementados como membros aptos para expressão. Nesse caso, as palavras-chave `get` e `set` devem ser usadas. Por exemplo:

C#

```
using System;

class SampleCollection<T>
{
    // Declare an array to store the data elements.
    private T[] arr = new T[100];

    // Define the indexer to allow client code to use [] notation.
    public T this[int i]
    {
        get => arr[i];
        set => arr[i] = value;
    }
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection[0] = "Hello, World.";
        Console.WriteLine(stringCollection[0]);
    }
}
// The example displays the following output:
//      Hello, World.
```

## Visão Geral dos Indexadores

- Os indexadores permitem que objetos sejam indexados de maneira semelhante às matrizes.
- Um acessador `get` retorna um valor. Um acessador `set` atribui um valor.
- A palavra-chave `this` é usada para definir o indexador.
- A palavra-chave `value` é usada para definir o valor que está sendo atribuído pelo acessador `set`.
- Os indexadores não precisam ser indexados por um valor inteiro. Você deve definir o mecanismo de pesquisa específico.
- Os indexadores podem ser sobrecarregados.
- Os indexadores podem ter mais de um parâmetro formal, por exemplo, ao acessar uma matriz bidimensional.

# Seções relacionadas

- [Usando indexadores](#)
- [Indexadores em interfaces](#)
- [Comparação entre propriedades e indexadores](#)
- [Restringindo a acessibilidade ao acessador](#)

## Especificação da Linguagem C#

Para obter mais informações, veja [Indexadores](#) na [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Guia de Programação em C#](#)
- [Propriedades](#)

# Usando indexadores (Guia de Programação em C#)

Artigo • 07/04/2023

Os indexadores são uma conveniência sintática que permitem criar uma [classe](#), um [struct](#) ou uma [interface](#) que os aplicativos clientes podem acessar como uma matriz. O compilador vai gerar uma propriedade `Item` (ou uma propriedade de nome alternativo, se [IndexerNameAttribute](#) estiver presente) e os métodos acessadores apropriados. Os indexadores são implementados em tipos cuja principal finalidade é encapsular uma coleção ou matriz interna. Por exemplo, suponha que você tenha uma classe `TempRecord` que representa a temperatura em Fahrenheit, conforme registrada em 10 momentos diferentes durante um período de 24 horas. A classe contém uma matriz `temps` do tipo `float[]` para armazenar os valores de temperatura. Ao implementar um indexador nessa classe, os clientes podem acessar as temperaturas em uma instância `TempRecord` como `float temp = tempRecord[4]`, e não como `float temp = tempRecord.temps[4]`. A notação do indexador não apenas simplifica a sintaxe para aplicativos clientes, mas também torna a classe e a finalidade dela mais intuitivas para que os outros desenvolvedores entendam.

Para declarar um indexador em uma classe ou struct, use a palavra-chave `this`, como mostra o seguinte exemplo:

```
C#  
  
// Indexer declaration  
public int this[int index]  
{  
    // get and set accessors  
}
```

## ⓘ Importante

Declarar um indexador vai gerar automaticamente uma propriedade chamada `Item` no objeto. A propriedade `Item` não pode ser acessada diretamente por meio da [expressão de acesso a membro](#) da instância. Além disso, se você adicionar a sua propriedade `Item` a um objeto com indexador, será gerado um [erro do compilador CS0102](#). Para evitar esse erro, use [IndexerNameAttribute](#) para renomear o indexador, conforme detalhado abaixo.

# Comentários

O tipo de um indexador e o tipo dos seus parâmetros devem ser pelo menos tão acessíveis quanto o próprio indexador. Para obter mais informações sobre níveis de acessibilidade, consulte [Modificadores de acesso](#).

Para obter mais informações sobre como usar indexadores com uma interface, consulte [Indexadores de Interface](#).

A assinatura de um indexador consiste do número e dos tipos de seus parâmetros formais. Ela não inclui o tipo de indexador nem os nomes dos parâmetros formais. Se você declarar mais de um indexador na mesma classe, eles terão diferentes assinaturas.

Um indexador não é classificado como uma variável; portanto, um valor indexador não pode ser passado por referência (como um `ref` ou `out` parâmetro), a menos que seu valor seja uma referência (ou seja, ele retorna por referência.)

Para fornecer o indexador com um nome que outras linguagens possam usar, use [System.Runtime.CompilerServices.IndexerNameAttribute](#), como mostra o seguinte exemplo:

C#

```
// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

Esse indexador terá o nome `TheItem`, pois ele é substituído pelo atributo de nome do indexador. Por padrão, o nome do indexador é `Item`.

## Exemplo 1

O exemplo a seguir mostra como declarar um campo de matriz privada, `temps` e um indexador. O indexador permite acesso direto à instância `tempRecord[i]`. A alternativa ao uso do indexador é declarar a matriz como um membro [público](#) e acessar seus membros, `tempRecord.temps[i]`, diretamente.

C#

```
public class TempRecord
{
```

```

// Array of temperature values
float[] temps = new float[10]
{
    56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
    61.3F, 65.9F, 62.1F, 59.2F, 57.5F
};

// To enable client code to validate input
// when accessing your indexer.
public int Length => temps.Length;

// Indexer declaration.
// If index is out of range, the temps array will throw the exception.
public float this[int index]
{
    get => temps[index];
    set => temps[index] = value;
}
}

```

Observe que, quando o acesso de um indexador é avaliado, por exemplo, em uma instrução `Console.WriteLine`, o acessador `get` é invocado. Portanto, se não existir nenhum acessador `get`, ocorrerá um erro em tempo de compilação.

C#

```

class Program
{
    static void Main()
    {
        var tempRecord = new TempRecord();

        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"Element #{i} = {tempRecord[i]}");
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
    /* Output:
        Element #0 = 56.2
        Element #1 = 56.7
        Element #2 = 56.5
        Element #3 = 58.3
        Element #4 = 58.8
    */
}

```

```
    Element #5 = 60.1
    Element #6 = 65.9
    Element #7 = 62.1
    Element #8 = 59.2
    Element #9 = 57.5
  */
}
```

## Indexando usando outros valores

O C# não limita o tipo de parâmetro do indexador ao inteiro. Por exemplo, talvez seja útil usar uma cadeia de caracteres com um indexador. Esse indexador pode ser implementado pesquisando a cadeia de caracteres na coleção e retornando o valor adequado. Como os acessadores podem ser sobre carregados, as versões do inteiro e da cadeia de caracteres podem coexistir.

## Exemplo 2

O exemplo a seguir declara uma classe que armazena os dias da semana. Um acessador `get` aceita uma cadeia de caracteres, o nome de um dia e retorna o inteiro correspondente. Por exemplo, "Sunday" retorna 0, "Monday" retorna 1 e assim por diante.

C#

```
// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

    // Indexer with only a get accessor with the expression-bodied
    definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }

        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the form
```

```
\\"Sun\\", \\"Mon\\", etc");  
    }  
}
```

## Exemplo de consumo 2

C#

```
class Program  
{  
    static void Main(string[] args)  
    {  
        var week = new DayCollection();  
        Console.WriteLine(week["Fri"]);  
  
        try  
        {  
            Console.WriteLine(week["Made-up day"]);  
        }  
        catch (ArgumentOutOfRangeException e)  
        {  
            Console.WriteLine($"Not supported input: {e.Message}");  
        }  
    }  
    // Output:  
    // 5  
    // Not supported input: Day Made-up day is not supported.  
    // Day input must be in the form "Sun", "Mon", etc (Parameter 'day')  
}
```

## Exemplo 3

O exemplo a seguir declara uma classe que armazena os dias da semana usando a enumeração [System.DayOfWeek](#). Um acessador `get` aceita `DayOfWeek`, o valor de um dia, e retorna o inteiro correspondente. Por exemplo, `DayOfWeek.Sunday` retorna 0, `DayOfWeek.Monday` retorna 1 e assim por diante.

C#

```
using Day = System.DayOfWeek;  
  
class DayOfWeekCollection  
{  
    Day[] days =  
    {  
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,  
        Day.Thursday, Day.Friday, Day.Saturday
```

```

};

// Indexer with only a get accessor with the expression-bodied
definition:
public int this[Day day] => FindDayIndex(day);

private int FindDayIndex(Day day)
{
    for (int j = 0; j < days.Length; j++)
    {
        if (days[j] == day)
        {
            return j;
        }
    }
    throw new ArgumentOutOfRangeException(
        nameof(day),
        $"Day {day} is not supported.\nDay input must be a defined
System.DayOfWeek value.");
}
}

```

## Exemplo de consumo 3

C#

```

class Program
{
    static void Main()
    {
        var week = new DayOfWeekCollection();
        Console.WriteLine(week[DayOfWeek.Friday]);

        try
        {
            Console.WriteLine(week[(DayOfWeek)43]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day 43 is not supported.
    // Day input must be a defined System.DayOfWeek value. (Parameter 'day')
}

```

## Programação robusta

Há duas maneiras principais nas quais a segurança e a confiabilidade de indexadores podem ser melhoradas:

- Certifique-se de incorporar algum tipo de estratégia de tratamento de erros para manipular a chance de passagem de código cliente em um valor de índice inválido. Anteriormente, no primeiro exemplo neste tópico, a classe TempRecord oferece uma propriedade Length que permite que o código cliente verifique a saída antes de passá-la para o indexador. Também é possível colocar o código de tratamento de erro dentro do próprio indexador. Certifique-se documentar para os usuários as exceções que você gera dentro de um acessador do indexador.
- Defina a acessibilidade dos acessadores `get` e `set` para que ela seja mais restritiva possível. Isso é importante para o acessador `set` em particular. Para obter mais informações, consulte [Restringindo a acessibilidade aos acessadores](#).

## Confira também

- [Guia de Programação em C#](#)
- [Indexadores](#)
- [Propriedades](#)

# Indexadores em interfaces (Guia de Programação em C#)

Artigo • 07/04/2023

Os indexadores podem ser declarados em uma [interface](#). Acessadores de indexadores de interface diferem dos acessadores de indexadores de [classe](#) das seguintes maneiras:

- Os acessadores de interface não usam modificadores.
- Um acessador de interface normalmente não tem corpo.

Portanto, a finalidade do acessador é indicar se o indexador é do tipo leitura/gravação, somente leitura ou somente gravação. Você pode fornecer uma implementação para um indexador definido em uma interface, mas isso é raro. Normalmente, os indexadores definem uma API para acessar campos de dados, que não podem ser definidos em uma interface.

Este é um exemplo de um acessador de indexador de interface:

```
C#  
  
public interface ISomeInterface  
{  
    //...  
  
    // Indexer declaration:  
    string this[int index]  
    {  
        get;  
        set;  
    }  
}
```

A assinatura de um indexador deve ser diferente das assinaturas de todos os outros indexadores declarados na mesma interface.

## Exemplo

O exemplo a seguir mostra como implementar indexadores de interface.

```
C#  
  
// Indexer on an interface:  
public interface IIndexInterface  
{
```

```

// Indexer declaration:
int this[int index]
{
    get;
    set;
}
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index]    // indexer declaration
    {
        // The arr object will throw IndexOutOfRangeException exception.
        get => arr[index];
        set => arr[index] = value;
    }
}

```

C#

```

IndexerClass test = new IndexerClass();
System.Random rand = System.Random.Shared;
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
Element #0 = 360877544
Element #1 = 327058047
Element #2 = 1913480832
Element #3 = 1519039937
Element #4 = 601472233
Element #5 = 323352310
Element #6 = 1422639981
Element #7 = 1797892494
Element #8 = 875761049
Element #9 = 393083859
*/

```

No exemplo anterior, é possível usar a implementação de membro de interface explícita usando o nome totalmente qualificado do membro de interface. Por exemplo

C#

```
string IIndexInterface.this[int index]
{
}
```

No entanto, o nome totalmente qualificado só será necessário para evitar ambiguidade quando a classe estiver implementando mais de uma interface com a mesma assinatura do indexador. Por exemplo, se uma classe `Employee` estiver implementando dois interfaces, `ICitizen` e `IEmployee`, e as duas interfaces tiverem a mesma assinatura de indexador, a implementação de membro de interface explícita é necessária. Ou seja, a seguinte declaração de indexador:

C#

```
string IEmployee.this[int index]
{
}
```

implementa o indexador na interface `IEmployee`, enquanto a seguinte declaração:

C#

```
string ICitizen.this[int index]
{
}
```

implementa o indexador na interface `ICitizen`.

## Confira também

- [Guia de Programação em C#](#)
- [Indexadores](#)
- [Propriedades](#)
- [Interfaces](#)

# Comparação entre propriedades e indexadores (Guia de Programação em C#)

Artigo • 07/04/2023

Os indexadores são como propriedades. Com exceção das diferenças mostradas na tabela a seguir, todas as regras definidas para acessadores de propriedade também se aplicam a acessadores de indexador.

Propriedade	Indexador
Permite que os métodos sejam chamados como se fossem membros de dados públicos.	Permite que elementos de uma coleção interna de um objeto sejam acessados usando uma notação de matriz no próprio objeto.
Acessado por meio de um nome simples.	Acessado por meio de um índice.
Pode ser estático ou um membro de instância.	Deve ser um membro da instância.
Um acessador <code>get</code> de uma propriedade não tem parâmetros.	Um acessador <code>get</code> de um indexador tem a mesma lista de parâmetro formal que o indexador.
Um acessador <code>set</code> de uma propriedade contém o parâmetro implícito <code>value</code> .	Um acessador <code>set</code> de um indexador tem a mesma lista de parâmetro formal que o indexador, bem como o mesmo parâmetro de <code>valor</code> .
Dá suporte a sintaxe reduzida com <a href="#">Propriedades Autoimplementadas</a> .	Dá suporte a membros aptos para expressão a fim de obter somente indexadores.

## Confira também

- [Guia de Programação em C#](#)
- [Indexadores](#)
- [Propriedades](#)

# Eventos (Guia de Programação em C#)

Artigo • 15/02/2023

Eventos permitem que uma [classe](#) ou objeto notifique outras classes ou objetos quando algo interessante ocorre. A classe que envia (ou *aciona*) o evento é chamada de *editor* e as classes que recebem (ou *manipulam*) os eventos são chamadas *assinantes*.

Em um aplicativo Windows Forms em C# ou Web típico, você assina eventos acionados pelos controles, como botões e caixas de listagem. Você pode usar o IDE (ambiente de desenvolvimento integrado) do Visual C# para procurar os eventos que um controle publica e selecionar aqueles que você deseja manipular. O IDE oferece uma maneira fácil de adicionar automaticamente um método de manipulador de eventos vazio e o código para assinar o evento. Para obter mais informações, confira [Como assinar e cancelar a assinatura de eventos](#).

## Visão geral sobre eventos

Os eventos têm as seguintes propriedades:

- O editor determina quando um evento é acionado. Os assinantes determinam a ação que é executada em resposta ao evento.
- Um evento pode ter vários assinantes. Um assinante pode manipular vários eventos de vários publicadores.
- Eventos que não têm assinantes nunca são acionados.
- Normalmente, os eventos são usados para sinalizar ações do usuário, como cliques de botão ou seleções de menu em interfaces gráficas do usuário.
- Quando um evento tem vários assinantes, os manipuladores de eventos são invocados sincronicamente quando um evento é acionado. Para invocar eventos de forma assíncrona, consulte [Chamando métodos síncronos assincronamente](#).
- Na biblioteca de classes do .NET, os eventos são baseados no delegado [EventHandler](#) e na classe base [EventArgs](#).

## Seções relacionadas

Para obter mais informações, consulte:

- [Como realizar e cancelar a assinatura de eventos](#)

- Como publicar eventos em conformidade com as diretrizes do .NET
- Como acionar eventos de classe base em classes derivadas
- Como implementar eventos de interface
- Como implementar acessadores de eventos personalizados

## Especificação da Linguagem C#

Para obter mais informações, veja [Eventos](#) na [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Capítulos do Livro em Destaque

Expressões lambda, eventos e delegados em [C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers](#)

Delegados e eventos em [Aprendendo sobre C# 3.0: conceitos básicos do C# 3.0](#)

## Confira também

- [EventHandler](#)
- [Guia de Programação em C#](#)
- [Representantes](#)
- [Criando manipuladores de eventos no Windows Forms](#)

# Como realizar e cancelar a assinatura de eventos (Guia de Programação em C#)

Artigo • 10/03/2023

Você assina um evento publicado por outra classe quando quer escrever um código personalizado que é chamado quando esse evento é gerado. Por exemplo, você pode assinar o evento `click` de um botão para fazer com que seu aplicativo faça algo útil quando o usuário clicar no botão.

## Para assinar eventos usando o IDE do Visual Studio

1. Se você não vir a janela **Propriedades**, no modo de exibição de **Design**, clique com o botão direito do mouse no formulário ou controle para o qual deseja criar um manipulador de eventos e selecione **Propriedades**.
2. Na parte superior da janela **Propriedades**, clique no ícone **Eventos**.
3. Clique duas vezes no evento que deseja criar, por exemplo, o evento `Load`.

O Visual C# cria um método de manipulador de eventos vazio e adiciona-o ao código. Como alternativa, você pode adicionar o código manualmente no modo de exibição **Código**. Por exemplo, as linhas de código a seguir declaram um método de manipulador de eventos que será chamado quando a classe `Form` gerar o evento `Load`.

```
C#  
  
private void Form1_Load(object sender, System.EventArgs e)  
{  
    // Add your form load event handling code here.  
}
```

A linha de código que é necessária para assinar o evento também é gerada automaticamente no método `InitializeComponent` no arquivo Form1.Designer.cs em seu projeto. Ele é semelhante a isto:

```
C#  
  
this.Load += new System.EventHandler(this.Form1_Load);
```

## Para assinar eventos de forma programática

1. Defina um método de manipulador de eventos cuja assinatura corresponda à assinatura do delegado do evento. Por exemplo, se o evento se basear no tipo de delegado [EventHandler](#), o código a seguir representará o stub do método:

C#

```
void HandleCustomEvent(object sender, CustomEventArgs a)
{
    // Do something useful here.
}
```

2. Use o operador de atribuição de adição (`+=`) para anexar um manipulador de eventos ao evento. No exemplo a seguir, suponha que um objeto chamado `publisher` tem um evento chamado `RaiseCustomEvent`. Observe que a classe do assinante precisa de uma referência à classe do editor para assinar seus eventos.

C#

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

Você também pode usar uma [expressão lambda](#) para especificar um manipulador de eventos:

C#

```
public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
    {
        MessageBox.Show(((MouseEventArgs)e).Location.ToString());
    };
}
```

## Para assinar eventos usando uma função anônima

Se não tiver que cancelar a assinatura de um evento posteriormente, você poderá usar o operador de atribuição de adição (`+=`) para anexar uma função anônima como um manipulador de eventos. No exemplo a seguir, suponha que um objeto chamado `publisher` tenha um evento chamado `RaiseCustomEvent` e que uma classe `CustomEventArgs` também tenha sido definida para conter algum tipo de informação de

evento específico. Observe que a classe do assinante precisa de uma referência a `publisher` para assinar seus eventos.

C#

```
publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>
{
    string s = o.ToString() + " " + e.ToString();
    Console.WriteLine(s);
};
```

Você não poderá cancelar facilmente a assinatura de um evento se tiver usado uma função anônima para assiná-lo. Para cancelar a assinatura nesse cenário, volte para o código em que você assina o evento, armazene a função anônima em uma variável de delegado e adicione o delegado ao evento. Recomendamos que você não use funções anônimas para assinar eventos se tiver que cancelar a assinatura do evento em algum momento posterior em seu código. Para obter mais informações sobre expressões lambda, consulte [Expressão lambda](#).

## Cancelando a assinatura

Para impedir que o manipulador de eventos seja invocado quando o evento for gerado, cancele a assinatura do evento. Para evitar perda de recursos, cancele a assinatura de eventos antes de descartar um objeto de assinante. Até que você cancele a assinatura de um evento, o delegado multicast subjacente ao evento no objeto de publicação terá uma referência ao delegado que encapsula o manipulador de eventos do assinante. Desde que o objeto de publicação contenha essa referência, a coleta de lixo não excluirá seu objeto de assinante.

### Para cancelar a assinatura de um evento

- Use o operador de atribuição de subtração (`-=`) para cancelar a assinatura de um evento:

C#

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

Quando todos os assinantes tiverem cancelado a assinatura de um evento, a instância do evento na classe do publicador será definida como `null`.

## Confira também

- [Eventos](#)
- [event](#)
- [Como publicar eventos em conformidade com as diretrizes do .NET](#)
- [Operadores - e -=](#)
- [Operadores + e +=](#)

# Como publicar eventos em conformidade com as diretrizes do .NET (Guia de Programação em C#)

Artigo • 09/05/2023

O procedimento a seguir demonstra como adicionar eventos que seguem o padrão .NET para classes e structs. Todos os eventos na biblioteca de classes do .NET se baseiam no delegado [EventHandler](#), que é definido da seguinte maneira:

C#

```
public delegate void EventHandler(object sender, EventArgs e);
```

## ⓘ Observação

O .NET Framework 2.0 apresenta uma versão genérica desse delegado, o [EventHandler<TEventArgs>](#). Os exemplos a seguir mostram como usar as duas versões.

Embora os eventos nas classes que você define possam ser baseados em qualquer tipo delegado válido, até mesmo nos delegados que retornam um valor, é geralmente recomendável que você baseie seus eventos no padrão .NET usando o [EventHandler](#), conforme mostrado no exemplo a seguir.

O nome [EventHandler](#) pode levar a um pouco de confusão, pois ele, na verdade, não lida com o evento. Os tipos [EventHandler](#) e os tipos [EventHandler<TEventArgs>](#) genéricos são delegados. Um método ou expressão lambda cuja assinatura corresponda à definição do delegado é o *manipulador de eventos* e será invocada quando o evento for gerado.

## Publicar eventos com base no padrão EventHandler

1. (Ignore esta etapa e vá para a 3ª Etapa se você não precisar enviar dados personalizados com o evento.) Declare a classe para seus dados personalizados em um escopo visível para suas classes de publicador e assinante. Em seguida,

adicone os membros necessários para manter seus dados de evento personalizados. Neste exemplo, uma cadeia de caracteres simples é retornada.

```
C#  
  
public class CustomEventArgs : EventArgs  
{  
    public CustomEventArgs(string message)  
    {  
        Message = message;  
    }  
  
    public string Message { get; set; }  
}
```

2. (Ignore esta etapa se você estiver usando a versão genérica do `EventHandler<TEventArgs>`.) Declare um delegado em sua classe de publicação. Dê um nome que termine com `EventHandler`. O segundo parâmetro especifica o tipo personalizado `EventArgs`.

```
C#  
  
public delegate void CustomEventHandler(object sender, CustomEventArgs args);
```

3. Declare o evento em sua classe de publicação, usando uma das etapas a seguir.

- a. Se você não tiver uma classe `EventArgs` personalizada, o tipo de evento será o delegado `EventHandler` não genérico. Você não precisa declarar o delegado porque ele já está declarado no namespace `System` que está incluído quando você cria seu projeto do C#. Adicione o seguinte código à sua classe publicadora.

```
C#  
  
public event EventHandler RaiseCustomEvent;
```

- b. Se você estiver usando a versão não genérica de `EventHandler` e você tem uma classe personalizada derivada de `EventArgs`, declare o evento dentro de sua classe de publicação e use o delegado da etapa 2 como o tipo.

```
C#  
  
public event CustomEventHandler RaiseCustomEvent;
```

c. Se você estiver usando a versão genérica, não é necessário um delegado personalizado. Em vez disso, na sua classe de publicação, especifique o tipo de evento como `EventHandler<CustomEventArgs>`, substituindo o nome da sua própria classe entre os colchetes angulares.

C#

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

## Exemplo

O exemplo a seguir demonstra as etapas anteriores, usando uma classe `EventArgs` personalizada e o `EventHandler<TEventArgs>` como o tipo de evento.

C#

```
using System;

namespace DotNetEvents
{
    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string message)
        {
            Message = message;
        }

        public string Message { get; set; }
    }

    // Class that publishes an event
    class Publisher
    {
        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Event triggered"));
        }

        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation
        behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
```

```

    {
        // Make a temporary copy of the event to avoid possibility of
        // a race condition if the last subscriber unsubscribes
        // immediately after the null check and before the event is
        raised.

        EventHandler<CustomEventArgs> raiseEvent = RaiseCustomEvent;

        // Event will be null if there are no subscribers
        if (raiseEvent != null)
        {
            // Format the string to send inside the CustomEventArgs
            parameter
            e.Message += $" at {DateTime.Now}";

            // Call to raise the event.
            raiseEvent(this, e);
        }
    }

    //Class that subscribes to an event
    class Subscriber
    {
        private readonly string _id;

        public Subscriber(string id, Publisher pub)
        {
            _id = id;

            // Subscribe to the event
            pub.RaiseCustomEvent += HandleCustomEvent;
        }

        // Define what actions to take when the event is raised.
        void HandleCustomEvent(object sender, CustomEventArgs e)
        {
            Console.WriteLine($"{_id} received this message: {e.Message}");
        }
    }

    class Program
    {
        static void Main()
        {
            var pub = new Publisher();
            var sub1 = new Subscriber("sub1", pub);
            var sub2 = new Subscriber("sub2", pub);

            // Call the method that raises the event.
            pub.DoSomething();

            // Keep the console window open
            Console.WriteLine("Press any key to continue...");
            Console.ReadLine();
        }
    }
}

```

```
    }  
}
```

## Confira também

- [Delegate](#)
- [Guia de Programação em C#](#)
- [Eventos](#)
- [Representantes](#)

# Como acionar eventos de classe base em classes derivadas (Guia de Programação em C#)

Artigo • 10/05/2023

O exemplo simples a seguir mostra o modo padrão para declarar os eventos em uma classe base para que eles também possam ser gerados das classes derivadas. Esse padrão é amplamente usado em classes do Windows Forms nas bibliotecas de classes do .NET.

Quando você cria uma classe que pode ser usada como uma classe base para outras classes, deve considerar o fato de que os eventos são um tipo especial de delegado que pode ser invocado apenas de dentro da classe que os declarou. As classes derivadas não podem invocar diretamente eventos declarados dentro da classe base. Embora, às vezes, você possa desejar um evento que possa ser gerado apenas pela classe base, na maioria das vezes você deve habilitar a classe derivada para invocar os eventos de classe base. Para fazer isso, você pode criar um método de invocação protegido na classe base que encapsula o evento. Chamando ou substituindo esse método de invocação, as classes derivadas podem invocar o evento diretamente.

## ⓘ Observação

Não declare eventos virtuais em uma classe base e substitua-os em uma classe derivada. O compilador C# não lida com eles corretamente e é imprevisível se um assinante do evento derivado realmente estará assinando o evento de classe base.

## Exemplo

C#

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }
    }
}
```

```

        public double NewArea { get; }

    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double _area;

        public double Area
        {
            get => _area;
            set => _area = value;
        }

        // The event. Note that by using the generic EventHandler<T> event
        type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Safely raise the event for all subscribers
            ShapeChanged?.Invoke(this, e);
        }
    }

    public class Circle : Shape
    {
        private double _radius;

        public Circle(double radius)
        {
            _radius = radius;
            _area = 3.14 * _radius * _radius;
        }

        public void Update(double d)
        {
            _radius = d;
            _area = 3.14 * _radius * _radius;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }
    }
}

```

```
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}

public class Rectangle : Shape
{
    private double _length;
    private double _width;

    public Rectangle(double length, double width)
    {
        _length = length;
        _width = width;
        _area = _length * _width;
    }

    public void Update(double length, double width)
    {
        _length = length;
        _width = width;
        _area = _length * _width;
        OnShapeChanged(new ShapeEventArgs(_area));
    }

    protected override void OnShapeChanged(ShapeEventArgs e)
    {
        // Do any rectangle-specific processing here.

        // Call the base class event invocation method.
        base.OnShapeChanged(e);
    }

    public override void Draw()
    {
        Console.WriteLine("Drawing a rectangle");
    }
}

// Represents the surface on which the shapes are drawn
// Subscribes to shape events so that it knows
// when to redraw a shape.
public class ShapeContainer
{
    private readonly List<Shape> _list;

    public ShapeContainer()
    {
        _list = new List<Shape>();
    }

    public void AddShape(Shape shape)
    {
        _list.Add(shape);
    }
}
```

```

        // Subscribe to the base class event.
        shape.ShapeChanged += HandleShapeChanged;
    }

    // ...Other methods to draw, resize, etc.

    private void HandleShapeChanged(object sender, ShapeEventArgs e)
    {
        if (sender is Shape shape)
        {
            // Diagnostic message for demonstration purposes.
            Console.WriteLine($"Received event. Shape area is now
{e.NewArea}");

            // Redraw the shape here.
            shape.Draw();
        }
    }
}

class Test
{
    static void Main()
    {
        //Create the event publishers and subscriber
        var circle = new Circle(54);
        var rectangle = new Rectangle(12, 9);
        var container = new ShapeContainer();

        // Add the shapes to the container.
        container.AddShape(circle);
        container.AddShape(rectangle);

        // Cause some events to be raised.
        circle.Update(57);
        rectangle.Update(7, 7);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to continue...");
        Console.ReadKey();
    }
}
/* Output:
   Received event. Shape area is now 10201.86
   Drawing a circle
   Received event. Shape area is now 49
   Drawing a rectangle
*/

```

## Confira também

- Guia de Programação em C#
- Eventos
- Representantes
- Modificadores de acesso
- Criando manipuladores de eventos no Windows Forms

# Como implementar eventos de interface (Guia de Programação em C#)

Artigo • 10/03/2023

Um [interface](#) pode declarar uma [evento](#). O exemplo a seguir mostra como implementar eventos de interface em uma classe. Basicamente, as regras são as mesmas aplicadas à implementação de qualquer método ou propriedade de interface.

## Implementar eventos de interface em uma classe

Declare o evento na classe e, em seguida, invoque-o nas áreas apropriadas.

C#

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event...
            OnShapeChanged(new MyEventArgs(/*arguments*/));
            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }
}
```

# Exemplo

O exemplo a seguir mostra como lidar com a situação menos comum, na qual a classe herda de duas ou mais interfaces e cada interface tem um evento com o mesmo nome. Nessa situação, é necessário fornecer uma implementação explícita da interface para pelo menos um dos eventos. Ao gravar uma implementação explícita da interface de um evento, também é necessário gravar os acessadores de evento `add` e `remove`. Normalmente, eles são fornecidos pelo compilador, mas nesse caso o compilador não pode fornecê-los.

Ao fornecer acessadores próprios, é possível especificar se os dois eventos são representados pelo mesmo evento na classe ou por eventos diferentes. Por exemplo, se os eventos forem gerados em horários diferentes, de acordo com as especificações da interface, será possível associar cada evento a uma implementação separada na classe. No exemplo a seguir, os assinantes determinam qual evento `OnDraw` receberão ao converter a referência de forma para um `IDrawingObject` ou um `IShape`.

C#

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }

    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
```

```

#region IDrawingObjectOnDraw
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
#endregion
// Explicit interface implementation required.
// Associate IShape's event with
// PostDrawEvent
event EventHandler IShape.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PostDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PostDrawEvent -= value;
        }
    }
}

// For the sake of simplicity this one method
// implements both interfaces.
public void Draw()
{
    // Raise IDrawingObject's event before the object is drawn.
    PreDrawEvent?.Invoke(this, EventArgs.Empty);

    Console.WriteLine("Drawing a shape.");

    // Raise IShape's event after the object is drawn.
    PostDrawEvent?.Invoke(this, EventArgs.Empty);
}
}

public class Subscriber1
{

```

```

// References the shape object as an IDrawingObject
public Subscriber1(Shape shape)
{
    IDrawingObject d = (IDrawingObject)shape;
    d.OnDraw += d_OnDraw;
}

void d_OnDraw(object sender, EventArgs e)
{
    Console.WriteLine("Sub1 receives the IDrawingObject event.");
}

// References the shape object as an IShape
public class Subscriber2
{
    public Subscriber2(Shape shape)
    {
        IShape d = (IShape)shape;
        d.OnDraw += d_OnDraw;
    }

    void d_OnDraw(object sender, EventArgs e)
    {
        Console.WriteLine("Sub2 receives the IShape event.");
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Shape shape = new Shape();
        Subscriber1 sub = new Subscriber1(shape);
        Subscriber2 sub2 = new Subscriber2(shape);
        shape.Draw();

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   Sub1 receives the IDrawingObject event.
   Drawing a shape.
   Sub2 receives the IShape event.
*/

```

## Confira também

- Guia de Programação em C#
- Eventos

- Representantes
- Implementação de interface explícita
- Como acionar eventos de classe base em classes derivadas

# Como implementar acessadores de eventos personalizados (Guia de Programação em C#)

Artigo • 10/03/2023

Um evento é um tipo especial de delegado multicast que só pode ser invocado dentro da classe em que ele está declarado. O código cliente assina o evento ao fornecer uma referência a um método que deve ser invocado quando o evento for disparado. Esses métodos são adicionados à lista de invocação do delegado por meio de acessadores de evento, que se assemelham aos acessadores de propriedade, com a exceção de que os acessadores de eventos são nomeados `add` e `remove`. Na maioria dos casos, não é necessário fornecer acessadores de eventos personalizados. Quando nenhum acessador de evento personalizado for fornecido no código, o compilador o adicionará automaticamente. No entanto, em alguns casos será necessário fornecer um comportamento personalizado. Um caso desse tipo é mostrado no tópico [Como implementar eventos de interface](#).

## Exemplo

O exemplo a seguir mostra como implementar os acessadores de eventos personalizados adicionar e remover. Embora seja possível substituir qualquer código dentro dos acessadores, é recomendável que você bloqueeie o evento antes de adicionar ou remover um novo manipulador de eventos.

C#

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

```
    }  
}
```

## Confira também

- [Eventos](#)
- [event](#)

# Parâmetros de tipo genérico (Guia de Programação em C#)

Artigo • 17/08/2023

Na definição de um tipo genérico ou método, parâmetros de tipo são um espaço reservado para um tipo específico que o cliente especifica ao criar uma instância do tipo genérico. Uma classe genérica, como `GenericList<T>`, listada em [Introdução aos Genéricos](#), não pode ser usada no estado em que se encontra porque não é realmente um tipo, mas um plano gráfico de um tipo. Para usar `GenericList<T>`, o código cliente deve declarar e instanciar um tipo construído, especificando um argumento de tipo entre colchetes. O argumento de tipo para essa classe específica pode ser qualquer tipo reconhecido pelo compilador. É possível criar qualquer quantidade de instâncias do tipo construído, cada uma usando um argumento de tipo diferente, da seguinte maneira:

C#

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

Em cada uma dessas instâncias de `GenericList<T>`, todas as ocorrências de `T` na classe são substituídas em tempo de execução com o argumento de tipo. Por meio dessa substituição, cria-se três objetos separados eficientes e fortemente tipados usando uma única definição de classe. Para obter mais informações sobre como essa substituição é executada pelo CLR, confira [Genéricos no Runtime](#).

Você pode aprender as convenções de nomenclatura para parâmetros de tipo genérico no artigo sobre [convenções de nomenclatura](#).

## Veja também

- [System.Collections.Generic](#)
- [Guia de Programação em C#](#)
- [Genéricos](#)
- [Diferenças entre modelos C++ e genéricos C#](#)

# Restrições a parâmetros de tipo (Guia de Programação em C#)

Artigo • 10/05/2023

Restrições informam o compilador sobre as funcionalidades que um argumento de tipo deve ter. Sem nenhuma restrição, o argumento de tipo poderia ser qualquer tipo. O compilador pode assumir somente os membros de `System.Object`, que é a classe base definitiva para qualquer tipo .NET. Para obter mais informações, consulte [Por que usar restrições](#). Se o código do cliente usa um tipo que não satisfaz uma restrição, o compilador emite um erro. Restrições são especificadas usando a palavra-chave contextual `where`. A tabela a seguir lista os sete tipos de restrições:

Constraint	Descrição
<code>where T : struct</code>	O argumento de tipo deve ser um <a href="#">tipo de valor</a> não anulável. Para obter informações sobre tipos que permitem valor nulo, consulte <a href="#">Tipos que permitem valor nulo</a> . Como todos os tipos de valor têm um construtor sem parâmetros acessível, a restrição <code>struct</code> implica a restrição <code>new()</code> e não pode ser combinada com a restrição <code>new()</code> . Você não pode combinar a restrição <code>struct</code> com a restrição <code>unmanaged</code> .
<code>where T : class</code>	O argumento de tipo deve ser um tipo de referência. Essa restrição se aplica também a qualquer classe, interface, delegado ou tipo de matriz. Em um contexto anulável, <code>T</code> deve ser um tipo de referência não anulável.
<code>where T : class?</code>	O argumento de tipo deve ser um tipo de referência, anulável ou não anulável. Essa restrição se aplica também a qualquer classe, interface, delegado ou tipo de matriz.
<code>where T : notnull</code>	O argumento de tipo deve ser um tipo não anulável. O argumento pode ser um tipo de referência não anulável ou um tipo de valor não anulável.
<code>where T : default</code>	Essa restrição resolve a ambiguidade quando você precisa especificar um parâmetro de tipo não treinado ao substituir um método ou fornecer uma implementação de interface explícita. A restrição <code>default</code> implica o método base sem a restrição <code>class</code> a <code>struct</code> . Para obter mais informações, consulte a proposta de especificação de restrição <a href="#">default</a> .
<code>where T : unmanaged</code>	O argumento de tipo deve ser um <a href="#">tipo não gerenciado</a> não anulável. A restrição <code>unmanaged</code> implica a restrição <code>struct</code> e não pode ser combinada com as restrições <code>struct</code> ou <code>new()</code> .
<code>where T : new()</code>	O argumento de tipo deve ter um construtor público sem parâmetros. Quando usado em conjunto com outras restrições, a restrição <code>new()</code> deve ser a última a ser especificada. A restrição <code>new()</code> não pode ser combinada com as restrições restrições <code>struct</code> e <code>unmanaged</code> .

Constraint	Descrição
<code>where T : &lt;nome da classe base&gt;</code>	O argumento de tipo deve ser ou derivar da classe base especificada. Em um contexto anulável, <code>T</code> deve ser um tipo de referência não anulável derivado da classe base especificada.
<code>where T : &lt;nome da classe base&gt;?</code>	O argumento de tipo deve ser ou derivar da classe base especificada. Em um contexto anulável, <code>T</code> pode ser um tipo anulável ou não anulável derivado da classe base especificada.
<code>where T : &lt;nome da interface&gt;</code>	O argumento de tipo deve ser ou implementar a interface especificada. Várias restrições de interface podem ser especificadas. A interface de restrição também pode ser genérica. Em um contexto anulável, <code>T</code> deve ser um tipo não anulável que implementa a interface especificada.
<code>where T : &lt;nome da interface&gt;?</code>	O argumento de tipo deve ser ou implementar a interface especificada. Várias restrições de interface podem ser especificadas. A interface de restrição também pode ser genérica. Em um contexto anulável, <code>T</code> pode ser um tipo de referência anulável, um tipo de referência não anulável ou um tipo de valor. <code>T</code> pode não ser um tipo de valor anulável.
<code>where T : U</code>	O argumento de tipo fornecido para <code>T</code> deve ser ou derivar do argumento fornecido para <code>U</code> . Em um contexto anulável, se <code>U</code> for um tipo de referência não anulável, <code>T</code> deve ser um tipo de referência não anulável. Se <code>U</code> for um tipo de referência anulável, <code>T</code> pode ser anulável ou não anulável.

## Por que usar restrições

As restrições especificam os recursos e as expectativas de um parâmetro de tipo. Declarar essas restrições significa que você pode usar as operações e as chamadas de método do tipo de restrição. Se sua classe ou método genérico usar qualquer operação nos membros genéricos além da atribuição simples ou chamar quaisquer métodos sem suporte por [System.Object](#), você aplicará restrições ao parâmetro de tipo. Por exemplo, a restrição de classe base informa ao compilador que somente os objetos desse tipo ou derivados desse tipo serão usados como argumentos de tipo. Uma vez que o compilador tiver essa garantia, ele poderá permitir que métodos desse tipo sejam chamados na classe genérica. O exemplo de código a seguir demonstra a funcionalidade que pode ser adicionada à classe `GenericList<T>` (em [Introdução aos Genéricos](#)) ao aplicar uma restrição de classe base.

C#

```
public class Employee
{
```

```
public Employee(string name, int id) => (Name, ID) = (name, id);
public string Name { get; set; }
public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node? Next { get; set; }
        public T Data { get; set; }
    }

    private Node? head;

    public void AddHead(T t)
    {
        Node n = new Node(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node? current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T? FindFirstOccurrence(string s)
    {
        Node? current = head;
        T? t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
                break;
            }
            else
            {
                current = current.Next;
            }
        }
        return t;
    }
}
```

A restrição permite que a classe genérica use a propriedade `Employee.Name`. A restrição especifica que todos os itens do tipo `T` são um objeto `Employee` ou um objeto que herda de `Employee`.

Várias restrições podem ser aplicadas ao mesmo parâmetro de tipo e as restrições em si podem ser tipos genéricos, da seguinte maneira:

C#

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>,  
new()  
{  
    // ...  
}
```

Ao aplicar a restrição `where T : class`, evite os operadores `==` e `!=` no parâmetro de tipo, pois esses operadores testarão somente a identidade de referência e não a igualdade de valor. Esse comportamento ocorrerá mesmo se esses operadores forem sobrecarregados em um tipo usado como argumento. O código a seguir ilustra esse ponto; a saída é `false`, muito embora a classe `String` sobrecarregue o operador `==`.

C#

```
public static void OpEqualsTest<T>(T s, T t) where T : class  
{  
    System.Console.WriteLine(s == t);  
}  
  
private static void TestStringEquality()  
{  
    string s1 = "target";  
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");  
    string s2 = sb.ToString();  
    OpEqualsTest<string>(s1, s2);  
}
```

O compilador sabe apenas que `T` é um tipo de referência no tempo de compilação e deve usar os operadores padrão válidos para todos os tipos de referência. Caso seja necessário testar a igualdade de valor, a maneira recomendada é também aplicar a restrição `where T : IEquatable<T>` ou `where T : IComparable<T>` e implementar a interface em qualquer classe que seja usada para construir a classe genérica.

## Restringindo vários parâmetros

É possível aplicar restrições a vários parâmetros e várias restrições a um único parâmetro, conforme mostrado no exemplo a seguir:

```
C#
```

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

## Parâmetros de tipo não associado

Os parâmetros de tipo que não têm restrições, como o T na classe pública `SampleClass<T>{}`, são denominados “parâmetros de tipo não associado”. Os parâmetros de tipo não associado têm as seguintes regras:

- Os operadores `!=` e `==` não podem ser usados, pois não há garantia de que o argumento de tipo concreto oferecerá suporte a eles.
- Eles podem ser convertidos para e de `System.Object` ou explicitamente convertidos para qualquer tipo de interface.
- Você pode compará-los com `nulo`. Se um parâmetro não associado for comparado a `null`, a comparação sempre retornará `false` se o argumento de tipo for um tipo de valor.

## Parâmetros de tipo como restrições

O uso de um parâmetro de tipo genérico como uma restrição será útil quando uma função membro com parâmetro de tipo próprio tiver que restringir esse parâmetro para o parâmetro de tipo do tipo recipiente, conforme mostrado no exemplo a seguir:

```
C#
```

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T /*...*/
}
```

No exemplo anterior, `T` é uma restrição de tipo no contexto do método `Add` e um parâmetro de tipo não associado no contexto da classe `List`.

Parâmetros de tipo também podem ser usados como restrições em definições de classe genérica. O parâmetro de tipo deve ser declarado entre colchetes angulares junto com quaisquer outros parâmetros de tipo:

C#

```
//Type parameter V is used as a type constraint.  
public class SampleClass<T, U, V> where T : V { }
```

A utilidade dos parâmetros de tipo como restrições com classes genéricas é limitada, pois o compilador não pode presumir nada sobre o parâmetro de tipo, exceto que ele deriva de `System.Object`. Use parâmetros de tipo como restrições em classes genéricas em cenários nos quais deseja impor uma relação de herança entre dois parâmetros de tipo.

## restrição de `notnull`

Você pode usar a restrição `notnull` para especificar que o argumento de tipo deve ser um tipo de valor não anulável ou um tipo de referência não anulável. Ao contrário da maioria das outras restrições, se um argumento de tipo violar a restrição `notnull`, o compilador gerará um aviso em vez de um erro.

A restrição `notnull` só tem efeito quando usada em um contexto anulável. Se você adicionar a restrição `notnull` em um contexto alheio anulável, o compilador não gerará avisos ou erros para violações da restrição.

## restrição de `class`

A restrição `class` em um contexto anulável especifica que o argumento de tipo deve ser um tipo de referência não anulável. Em um contexto anulável, quando um argumento de tipo é um tipo de referência anulável, o compilador gera um aviso.

## restrição de `default`

A adição de tipos de referência anuláveis complica o uso de `T?` em um tipo ou método genérico. `T?` pode ser usado com a restrição `struct` ou `class`, mas uma deve estar presente. Quando a restrição `class` era usada, `T?` referia-se ao tipo de referência anulável para `T`. A partir do C# 9, `T?` pode ser usado quando nenhuma restrição é aplicada. Nesse caso, `T?` é interpretado como `T?` para tipos de valor e tipos de

referência. No entanto, se `T` for uma instância de `Nullable<T>`, `T?` será o mesmo que `T`. Em outras palavras, ele não se torna `T??`.

Como `T?` agora pode ser usado sem a restrição `class` ou `struct`, as ambiguidades podem surgir em substituições ou implementações de interface explícitas. Em ambos os casos, a substituição não inclui as restrições, mas herda da classe base. Quando a classe base não aplica a restrição `class` ou `struct`, as classes derivadas precisam especificar de alguma forma uma substituição aplicada ao método base sem qualquer restrição. É quando o método derivado aplica a restrição `default`. A restrição `default` não esclarece a restrição *nem* a restrição `class` nem `struct`.

## Restrição não gerenciada

Você pode usar a restrição `unmanaged` para especificar que o parâmetro de tipo deve ser um [tipo não gerenciado](#) não anulável. A restrição `unmanaged` permite que você escreva rotinas reutilizáveis para trabalhar com tipos que podem ser manipulados como blocos de memória, conforme mostrado no exemplo a seguir:

C#

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

O método anterior deve ser compilado em um contexto `unsafe` porque ele usa o operador `sizeof` em um tipo não conhecido como um tipo interno. Sem a restrição `unmanaged`, o operador `sizeof` não está disponível.

A restrição `unmanaged` implica a restrição `struct` e não pode ser combinada com ela. Como a restrição `struct` implica a restrição `new()`, a restrição `unmanaged` não pode ser combinada com a restrição `new()` também.

## Restrições de delegado

Você pode usar [System.Delegate](#) ou [System.MulticastDelegate](#) como uma restrição de classe base. O CLR sempre permitia essa restrição, mas a linguagem C# não a permite. A restrição [System.Delegate](#) permite que você escreva código que funcione com delegados de uma maneira fortemente tipada. O código a seguir define um método de extensão que combina dois delegados fornecidos que são do mesmo tipo:

C#

```
public static TDelegate? TypeSafeCombine<TDelegate>(this TDelegate source,  
TDelegate target)  
    where TDelegate : System.Delegate  
    => Delegate.Combine(source, target) as TDelegate;
```

Você pode usar o método acima para combinar delegados que são do mesmo tipo:

C#

```
Action first = () => Console.WriteLine("this");  
Action second = () => Console.WriteLine("that");  
  
var combined = first.TypeSafeCombine(second);  
combined!();  
  
Func<bool> test = () => true;  
// Combine signature ensures combined delegates must  
// have the same type.  
//var badCombined = first.TypeSafeCombine(test);
```

Se você remover a marca de comentário na última linha, ela não será compilada. Tanto `first` quanto `test` são tipos de representante, mas são tipos diferentes de representantes.

## Restrições de enum

Você também pode especificar o tipo [System.Enum](#) como uma restrição de classe base. O CLR sempre permitia essa restrição, mas a linguagem C# não a permite. Genéricos usando [System.Enum](#) fornecem programação fortemente tipada para armazenar em cache os resultados do uso de métodos estáticos em [System.Enum](#). O exemplo a seguir localiza todos os valores válidos para um tipo enum e, em seguida, cria um dicionário que mapeia esses valores para sua representação de cadeia de caracteres.

C#

```
public static Dictionary<int, string> EnumNamedValues<T>() where T :  
System.Enum
```

```
{  
    var result = new Dictionary<int, string>();  
    var values = Enum.GetValues(typeof(T));  
  
    foreach (int item in values)  
        result.Add(item, Enum.GetName(typeof(T), item)!);  
    return result;  
}
```

`Enum.GetValues` e `Enum.GetName` usam reflexão, que tem implicações de desempenho. Você pode chamar `EnumNamedValues` para criar uma coleção que é armazenada em cache e reutilizada, em vez de repetir as chamadas que exigem reflexão.

Você pode usá-lo conforme mostrado no exemplo a seguir para criar uma enum e compilar um dicionário de seus nomes e valores:

```
C#  
  
enum Rainbow  
{  
    Red,  
    Orange,  
    Yellow,  
    Green,  
    Blue,  
    Indigo,  
    Violet  
}
```

```
C#  
  
var map = EnumNamedValues<Rainbow>();  
  
foreach (var pair in map)  
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

## Argumentos de tipo implementam interface declarada

Alguns cenários exigem que um argumento fornecido para um parâmetro de tipo implemente essa interface. Por exemplo:

```
C#  
  
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>  
{
```

```
    public abstract static T operator +(T left, T right);
    public abstract static T operator -(T left, T right);
}
```

Esse padrão permite que o compilador C# determine o tipo que contém os operadores sobrecarregados ou qualquer método `static virtual` ou `static abstract`. Ele fornece a sintaxe para que os operadores de adição e subtração possam ser definidos em um tipo que contém. Sem essa restrição, os parâmetros e argumentos seriam necessários para serem declarados como a interface, em vez do parâmetro de tipo:

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    public abstract static IAdditionSubtraction<T> operator +
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);

    public abstract static IAdditionSubtraction<T> operator -
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);
}
```

A sintaxe anterior exigiria que os implementadores usassem a [implementação de interface explícita](#) para esses métodos. Fornecer a restrição extra permite que a interface defina os operadores em termos dos parâmetros de tipo. Os tipos que implementam a interface podem implementar implicitamente os métodos de interface.

## Confira também

- [System.Collections.Generic](#)
- [Guia de Programação em C#](#)
- [Introdução aos genéricos](#)
- [Classes genéricas](#)
- [Restrição new](#)

# Classes genéricas (Guia de Programação em C#)

Artigo • 10/05/2023

As classes genéricas encapsulam operações que não são específicas de um determinado tipo de dados. O uso mais comum das classes genéricas é com coleções, como listas vinculadas, tabelas de hash, pilhas, filas, árvores e assim por diante. As operações como adicionar e remover itens da coleção são realizadas basicamente da mesma maneira, independentemente do tipo de dados que estão sendo armazenados.

Na maioria dos cenários que exigem classes de coleção, a abordagem recomendada é usar as que são fornecidas na biblioteca de classes do .NET. Para obter mais informações sobre o uso dessas classes, consulte [Coleções genéricas no .NET](#).

Em geral, você cria classes genéricas iniciando com uma classe concreta existente e alterando os tipos para parâmetros de tipo, um por vez, até alcançar o equilíbrio ideal de generalização e usabilidade. Ao criar suas próprias classes genéricas, observe as seguintes considerações importantes:

- Quais tipos generalizar em parâmetros de tipo.

Como uma regra, quanto mais tipos você puder parametrizar, mais flexível e reutilizável seu código se tornará. No entanto, generalização em excesso poderá criar um código que seja difícil de ser lido ou entendido por outros desenvolvedores.

- Quais restrições, se houver, aplicar aos parâmetros de tipo (consulte [Restrições a parâmetros de tipo](#)).

Uma boa regra é aplicar o máximo de restrições, de maneira que ainda seja possível manipular os tipos que você precisa manipular. Por exemplo, se você souber que a classe genérica é destinada a ser usada apenas com tipos de referência, aplique a restrição da classe. Isso impedirá o uso não intencional de sua classe com tipos de valor e permitirá que você use o operador `as` em `T` e verificar se há valores nulos.

- Se deve-se levar em consideração o comportamento genérico em subclasses e classes base.

Como as classes genéricas podem servir como classes base, as mesmas considerações de design aplicam-se nesse caso, como com as classes não

genéricas. Consulte as regras sobre heranças de classes base genéricas mais adiante neste tópico.

- Se implementar uma ou mais interfaces genéricas.

Por exemplo, se você estiver projetando uma classe que será usada para criar itens em uma coleção com base em classes genéricas, poderá ser necessário implementar uma interface como a `IComparable<T>`, em que `T` é o tipo de sua classe.

Para obter um exemplo de uma classe genérica simples, consulte [Introdução aos genéricos](#).

As regras para parâmetros de tipo e restrições têm várias implicações para o comportamento de classes genéricas, especialmente em relação à acessibilidade de membro e herança. Antes de prosseguir, você deve compreender alguns termos. Para um `Node<T>`, que é de classe genérica, o código cliente pode fazer referência à classe especificando um argumento de tipo – para criar um tipo construído fechado (`Node<int>`); ou deixando o parâmetro de tipo não especificado – por exemplo, quando você especifica uma classe base genérica, para criar um tipo construído aberto (`Node<T>`). As classes genéricas podem herdar de classes base construídas concretas, fechadas ou abertas:

C#

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

As classes não genéricas ou em outras palavras, classes concretas, podem herdar de classes base construídas fechadas, mas não de classes construídas abertas ou de parâmetros de tipo, porque não há maneiras de o código cliente fornecer o argumento de tipo necessário para instanciar a classe base em tempo de execução.

C#

```
//No error
class Node1 : BaseNodeGeneric<int> { }
```

```
//Generates an error  
//class Node2 : BaseNodeGeneric<T> {}  
  
//Generates an error  
//class Node3 : T {}
```

As classes genéricas que herdam de tipos construídos abertos devem fornecer argumentos de tipo para qualquer parâmetro de tipo de classe base que não é compartilhado pela classe herdeira, conforme demonstrado no código a seguir:

```
C#  
  
class BaseNodeMultiple<T, U> {}  
  
//No error  
class Node4<T> : BaseNodeMultiple<T, int> {}  
  
//No error  
class Node5<T, U> : BaseNodeMultiple<T, U> {}  
  
//Generates an error  
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

As classes genéricas que herdam de tipos construídos abertos devem especificar restrições que são um superconjunto ou sugerem, as restrições no tipo base:

```
C#  
  
class NodeItem<T> where T : System.IComparable<T>, new() {}  
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>,  
new() {}
```

Os tipos genéricos podem usar vários parâmetros de tipo e restrições, da seguinte maneira:

```
C#  
  
class SuperKeyType<K, V, U>  
    where U : System.IComparable<U>  
    where V : new()  
{ }
```

Tipos construídos abertos e construídos fechados podem ser usados como parâmetros de método:

```
C#
```

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

Se uma classe genérica implementa uma interface, todas as instâncias dessa classe podem ser convertidas nessa interface.

As classes genéricas são invariáveis. Em outras palavras, se um parâmetro de entrada especifica um `List<BaseClass>`, você receberá um erro em tempo de compilação se tentar fornecer um `List<DerivedClass>`.

## Confira também

- [System.Collections.Generic](#)
- [Guia de Programação em C#](#)
- [Genéricos](#)
- [Salvar o estado de enumeradores](#)
- [Um enigma de herança, parte 1](#)

# Interfaces genéricas (Guia de Programação em C#)

Artigo • 08/06/2023

Muitas vezes, é útil definir interfaces para classes de coleção genéricas ou para as classes genéricas que representam itens na coleção. Para evitar operações de conversão boxe e unboxing em tipos de valor, é melhor usar [interfaces genéricas](#), como `IComparable<T>`, em classes genéricas. A biblioteca de classes .NET define várias interfaces genéricas para uso com as classes de coleção no namespace `System.Collections.Generic`. Para obter mais informações sobre essas interfaces, consulte [Interfaces genéricas](#).

Quando uma interface é especificada como uma restrição em um parâmetro de tipo, somente os tipos que implementam a interface podem ser usados. O exemplo de código a seguir mostra uma classe `SortedList<T>` que deriva da classe `GenericList<T>`. Para obter mais informações, consulte [Introdução aos Genéricos](#). `SortedList<T>` adiciona a restrição `where T : IComparable<T>`. Essa restrição habilita o método `BubbleSort` em `SortedList<T>` a usar o método genérico `CompareTo` em elementos de lista. Neste exemplo, os elementos de lista são uma classe simples, `Person`, que implementa `IComparable<Person>`.

C#

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data; //T as private member datatype

        public Node(T t) //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }
    }
}
```

```

    }

    public T Data //T as return type of property
    {
        get { return data; }
        set { data = value; }
    }
}

public GenericList() //constructor
{
    head = null;
}

public void AddHead(T t) //T as method parameter type
{
    Node n = new Node(t);
    n.Next = head;
    head = n;
}

// Implementation of the iterator
public System.Collections.Generic.IEnumerator<T> GetEnumerator()
{
    Node current = head;
    while (current != null)
    {
        yield return current.Data;
        current = current.Next;
    }
}

// IEnumerable<T> inherits from IEnumerable, therefore this class
// must implement both the generic and non-generic versions of
// GetEnumerator. In most cases, the non-generic method can
// simply call the generic method.
System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

```

```

        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                // Because we need to call this method, the SortedList
                // class is constrained on IComparable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {

```

```
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

public class Program
{
    public static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names = new string[]
        {
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        };

        int[] ages = new int[] { 45, 19, 28, 23, 18, 9, 108, 72, 30, 35 };

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();
```

```

//Print out sorted list.
foreach (Person p in list)
{
    System.Console.WriteLine(p.ToString());
}
System.Console.WriteLine("Done with sorted list");
}
}

```

Várias interfaces podem ser especificadas como restrições em um único tipo, da seguinte maneira:

C#

```

class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}

```

Uma interface pode definir mais de um parâmetro de tipo, da seguinte maneira:

C#

```

interface IDictionary<K, V>
{
}

```

As regras de herança que se aplicam às classes também se aplicam às interfaces:

C#

```

interface IMonth<T> { }

interface IJanuary : IMonth<int> { } //No error
interface IFebruary<T> : IMonth<int> { } //No error
interface IMarch<T> : IMonth<T> { }      //No error
                                         //interface IApril<T> : IMonth<T, U>
{} //Error

```

Interfaces genéricas poderão herdar de interfaces não genéricas se a interface genérica for covariante, o que significa que ela usa apenas seu parâmetro de tipo como um valor retornado. Na biblioteca de classes do .NET, `IEnumerable<T>` herda de `IEnumerable` porque `IEnumerable<T>` usa apenas `T` no valor retornado de `GetEnumerator` e no getter de propriedade `Current`.

Classes concretas podem implementar interfaces construídas fechadas, da seguinte maneira:

C#

```
interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }
```

Classes genéricas podem implementar interfaces genéricas ou interfaces construídas fechadas, contanto que a lista de parâmetros de classe forneça todos os argumentos exigidos pela interface, da seguinte maneira:

C#

```
interface IBaseInterface1<T> { }

interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }           //No error
class SampleClass2<T> : IBaseInterface2<T, string> { } //No error
```

As regras que controlam a sobrecarga de método são as mesmas para métodos em classes genéricas, structs genéricos ou interfaces genéricas. Para obter mais informações, consulte [Métodos Genéricos](#).

A partir do C# 11, as interfaces podem declarar membros `static abstract` ou `static virtual`. As interfaces que declaram membros `static abstract` ou `static virtual` são quase sempre interfaces genéricas. O compilador deve resolver chamadas para métodos `static virtual` e `static abstract` em tempo de compilação. Os métodos `static virtual` e `static abstract` declarados em interfaces não têm um mecanismo de expedição de runtime análogo a métodos `virtual` ou `abstract` declarados em classes. Em vez disso, o compilador usa informações de tipo disponíveis em tempo de compilação. Normalmente, esses membros são declarados em interfaces genéricas. Além disso, a maioria das interfaces que declaram os métodos `static virtual` ou `static abstract` declaram que um dos parâmetros de tipo deve [implementar a interface declarada](#). Em seguida, o compilador usa os argumentos de tipo fornecidos para resolver o tipo do membro declarado.

## Confira também

- [Guia de Programação em C#](#)
- [Introdução aos genéricos](#)
- [interface](#)
- [Genéricos](#)

# Métodos genéricos (Guia de Programação em C#)

Artigo • 07/04/2023

Um método genérico é um método declarado com parâmetros de tipo, da seguinte maneira:

```
C#  
  
static void Swap<T>(ref T lhs, ref T rhs)  
{  
    T temp;  
    temp = lhs;  
    lhs = rhs;  
    rhs = temp;  
}
```

O exemplo de código a seguir mostra uma maneira de chamar o método usando `int` para o argumento de tipo:

```
C#  
  
public static void TestSwap()  
{  
    int a = 1;  
    int b = 2;  
  
    Swap<int>(ref a, ref b);  
    System.Console.WriteLine(a + " " + b);  
}
```

Também é possível omitir o argumento de tipo e o compilador o inferirá. Esta chamada para `Swap` é equivalente à chamada anterior:

```
C#  
  
Swap(ref a, ref b);
```

As mesmas regras de inferência de tipos se aplicam a métodos estáticos e métodos de instância. O compilador pode inferir os parâmetros de tipo com base nos argumentos de método passados; não é possível inferir os parâmetros de tipo somente de uma restrição ou valor retornado. Portanto, a inferência de tipos não funciona com métodos que não têm parâmetros. A inferência de tipos ocorre em tempo de compilação, antes

de o compilador tentar resolver assinaturas de método sobre carregadas. O compilador aplica a lógica da inferência de tipos a todos os métodos genéricos que compartilham o mesmo nome. Na etapa de resolução de sobrecarga, o compilador incluirá somente os métodos genéricos em que a inferência de tipos foi bem-sucedida.

Em uma classe genérica, métodos não genéricos podem acessar os parâmetros de tipo de nível de classe, da seguinte maneira:

```
C#  
  
class SampleClass<T>  
{  
    void Swap(ref T lhs, ref T rhs) { }  
}
```

Se um método genérico que usa os mesmos parâmetros de tipo da classe que o contém for definido, o compilador gerará um aviso [CS0693](#), pois, dentro do escopo do método, o argumento fornecido para o `T` interno oculta o argumento fornecido para o `T` externo. Caso seja necessária a flexibilidade de chamar um método de classe genérica com argumentos de tipo diferentes dos fornecidos quando a instância da classe foi criada, considere fornecer outro identificador ao parâmetro de tipo do método, conforme mostrado no `GenericList2<T>` do exemplo a seguir.

```
C#  
  
class GenericList<T>  
{  
    // CS0693  
    void SampleMethod<T>() { }  
}  
  
class GenericList2<T>  
{  
    //No warning  
    void SampleMethod<U>() { }  
}
```

Use restrições para permitir operações mais especializadas em parâmetros de tipo de métodos. Essa versão do `Swap<T>`, agora denominada `SwapIfGreater<T>`, pode ser usada somente com argumentos de tipo que implementam `IComparable<T>`.

```
C#  
  
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>  
{  
    T temp;
```

```
if (lhs.CompareTo(rhs) > 0)
{
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
}
```

Métodos genéricos podem ser sobreescarregados vários parâmetros de tipo. Por exemplo, todos os seguintes métodos podem ser localizados na mesma classe:

C#

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#).

## Confira também

- [System.Collections.Generic](#)
- [Guia de Programação em C#](#)
- [Introdução aos genéricos](#)
- [Métodos](#)

# Genéricos e matrizes (Guia de Programação em C#)

Artigo • 08/06/2023

As matrizes unidimensionais que têm um limite inferior a zero implementam `IList<T>` automaticamente. Isso permite a criação de métodos genéricos que podem usar o mesmo código para iterar por meio de matrizes e outros tipos de coleção. Essa técnica é útil principalmente para ler dados em coleções. A interface `IList<T>` não pode ser usada para adicionar ou remover elementos de uma matriz. Uma exceção será lançada se você tentar chamar um método `IList<T>` tal como `RemoveAt` em uma matriz neste contexto.

O exemplo de código a seguir demonstra como um único método genérico que usa um parâmetro de entrada `IList<T>` pode iterar por meio de uma lista e uma matriz, nesse caso, uma matriz de inteiros.

C#

```
class Program
{
    static void Main()
    {
        int[] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item?.ToString() + " ");
        }
    }
}
```

```
        System.Console.WriteLine();  
    }  
}
```

## Confira também

- [System.Collections.Generic](#)
- [Guia de Programação em C#](#)
- [Genéricos](#)
- [matrizes](#)
- [Genéricos](#)

# Delegados genéricos (Guia de Programação em C#)

Artigo • 07/04/2023

Um [delegado](#) pode definir seus próprios parâmetros de tipo. O código que referencia o delegado genérico pode especificar o argumento de tipo para criar um tipo construído fechado, assim como quando uma classe genérica é instanciada ou quando um método genérico é chamado, conforme mostrado no exemplo a seguir:

C#

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

A versão 2.0 do C# tem um novo recurso chamado conversão de grupo de método, que pode ser aplicada a tipos concretos e de delegado genérico e habilita a gravação da linha anterior com esta sintaxe simplificada:

C#

```
Del<int> m2 = Notify;
```

Os delegados definidos em uma classe genérica podem usar os parâmetros de tipo da classe genérica da mesma forma que os métodos da classe.

C#

```
class Stack<T>
{
    public delegate void StackDelegate(T[] items);
}
```

O código que referencia o delegado deve especificar o argumento de tipo da classe recipiente, da seguinte maneira:

C#

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
```

```
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

Os delegados genéricos são especialmente úteis na definição de eventos com base no padrão de design comum, pois o argumento do remetente pode ser fortemente tipado e não precisa ser convertido de e para `Object`.

C#

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs>? StackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        if (StackEvent is not null)
            StackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs
args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.StackEvent += o.HandleStackChange;
}
```

## Confira também

- [System.Collections.Generic](#)
- [Guia de Programação em C#](#)
- [Introdução aos genéricos](#)
- [Métodos genéricos](#)
- [Classes genéricas](#)
- [Interfaces genéricas](#)
- [Representantes](#)
- [Genéricos](#)



# Diferenças entre modelos C++ e genéricos C# (Guia de Programação em C#)

Artigo • 02/06/2023

Os modelos C++ e genéricos C# são recursos de linguagem que fornecem o suporte aos tipos parametrizados. No entanto, há várias diferenças entre os dois. No nível de sintaxe, os genéricos C# são uma abordagem mais simples para os tipos parametrizados sem a complexidade de modelos C++. Além disso, o C# não tenta fornecer toda a funcionalidade que os modelos C++ fornecem. No nível da implementação, a principal diferença é que as substituições do tipo genérico do C# são realizadas em runtime e as informações do tipo genérico são preservadas para objetos instanciados. Para obter mais informações, confira [Genéricos no runtime](#).

A seguir estão as principais diferenças entre modelos C++ e genéricos C#:

- Os genéricos C# não oferecem a mesma flexibilidade que os modelos C++. Por exemplo, não é possível chamar os operadores aritméticos em uma classe genérica C#, embora seja possível chamar operadores definidos pelo usuário.
- O C# não permite parâmetros de modelo sem tipo, como `template <int i> {}`.
- O C# não dá suporte à especialização explícita ou seja, uma implementação personalizada de um modelo para um tipo específico.
- O C# não dá suporte à especialização parcial: uma implementação personalizada para um subconjunto dos argumentos de tipo.
- O C# não permite que o parâmetro de tipo a ser usado como a classe base para o tipo genérico.
- O C# não permite que os parâmetros de tipo tenham tipos padrão.
- No C#, um parâmetro de tipo genérico não pode ser genérico, embora os tipos construídos possam ser usados como genéricos. O C++ permite parâmetros de modelo.
- O C++ permite o código que pode não ser válido para todos os parâmetros de tipo no modelo, que é então verificado para o tipo específico usado como o parâmetro de tipo. O C# requer código em uma classe a ser gravada de forma que ele funcionará com qualquer tipo que satisfaça as restrições. Por exemplo, em C++

é possível escrever uma função que usa os operadores aritméticos `+` e `-` em objetos do parâmetro de tipo, que produzirá um erro no momento da instanciação do modelo com um tipo que não dá suporte a esses operadores. O C# não permite isso. Os únicos constructos da linguagem permitidos são os que podem ser deduzidos das restrições.

## Confira também

- [Guia de Programação em C#](#)
- [Introdução aos genéricos](#)
- [Modelos](#)

# Genéricos em tempo de execução (Guia de Programação em C#)

Artigo • 07/04/2023

Um tipo genérico ou método compilado em Microsoft Intermediate Language (MSIL) conterá metadados que o identificarão como possuidor de parâmetros de tipo. O uso de MSIL em um tipo genérico será diferente de acordo com o parâmetro de tipo fornecido, ou seja, se ele é um tipo de valor ou um tipo de referência.

Quando um tipo genérico é construído pela primeira vez com um tipo de valor como parâmetro, o runtime cria um tipo genérico especializado com o(s) parâmetro(s) fornecido(s) substituído(s) nos locais apropriados do MSIL. Os tipos genéricos especializados são criados uma vez para cada tipo de valor único usado como parâmetro.

Por exemplo, caso o código do programa declarar uma pilha construída de inteiros:

C#

```
Stack<int> stack;
```

Neste ponto, o runtime gerará uma versão especializada da classe `Stack<T>` com o inteiro substituído corretamente, de acordo com seu parâmetro. Agora, sempre que o código do programa utilizar uma pilha de inteiros, o runtime reutilizará a classe especializada `Stack<T>` gerada. No exemplo a seguir, são criadas duas instâncias de uma pilha de inteiros e eles compartilham uma única instância do código `Stack<int>`:

C#

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

No entanto, suponha que outra classe `Stack<T>` com um tipo de valor diferente – como `long` ou uma estrutura definida pelo usuário como parâmetro – foi criada em outro ponto do código. Como resultado, o runtime gerará outra versão do tipo genérico e substituirá um `long` nos locais apropriados no MSIL. Conversões não são mais necessárias, pois cada classe genérica especializada contém o tipo de valor nativamente.

Os genéricos funcionam de outro modo nos tipos de referência. Na primeira vez em que um genérico é construído com qualquer tipo de referência, o runtime cria um tipo genérico especializado com referências de objeto substituídas por parâmetros no MSIL.

Em seguida, sempre que um tipo construído for instanciado com um tipo de referência como parâmetro, independentemente do tipo, o runtime reutilizará a versão especializada do tipo genérico criada anteriormente. Isso é possível porque todas as referências são do mesmo tamanho.

Por exemplo, suponha que há dois tipos de referência, uma classe `Customer` e uma classe `Order` e que uma pilha de tipos `Customer` foi criada:

C#

```
class Customer { }
class Order { }
```

C#

```
Stack<Customer> customers;
```

Neste ponto, o runtime gerará uma versão especializada da classe `Stack<T>` que armazenará referências de objeto que serão preenchidas posteriormente, em vez de armazenar dados. Suponha que a próxima linha de código crie uma pilha de outro tipo de referência, com o nome `Order`:

C#

```
Stack<Order> orders = new Stack<Order>();
```

Ao contrário dos tipos de valor, outra versão especializada da classe `Stack<T>` não será criada para o tipo `Order`. Em vez disso, uma instância da versão especializada da classe `Stack<T>` será criada e a variável `orders` será definida para referenciá-la. Imagine que uma linha de código foi encontrada para criar uma pilha de um tipo `Customer`:

C#

```
customers = new Stack<Customer>();
```

Assim como acontece com o uso anterior da classe `Stack<T>` criada usando o tipo `Order`, outra instância da classe especializada `Stack<T>` é criada. Os ponteiros contidos nela são definidos para referenciar uma área de memória do tamanho de um tipo `Customer`. Como a quantidade de tipos de referência pode variar muito entre os programas, a implementação de genéricos no C# reduz significativamente a quantidade de código ao diminuir para um o número de classes especializadas criadas pelo compilador para classes genéricas ou tipos de referência.

Além disso, quando uma classe genérica do C# é instanciada usando um tipo de valor ou parâmetro de tipo de referência, a reflexão pode consultá-la em runtime e o seu tipo real e parâmetro de tipo podem ser determinados.

## Confira também

- [System.Collections.Generic](#)
- [Guia de Programação em C#](#)
- [Introdução aos genéricos](#)
- [Genéricos](#)

# Genéricos e reflexão

Artigo • 20/03/2023

Como o tempo de execução do CLR (Common Language Runtime) tem acesso às informações de tipo genérico em tempo de execução, você pode usar a reflexão para obter informações sobre tipos genéricos da mesma forma como para tipos não genéricos. Para obter mais informações, confira [Genéricos no runtime](#).

O namespace [System.Reflection.Emit](#) também contém novos membros que dão suporte a genéricos. Consulte [Como definir um tipo genérico com a emissão de reflexão](#).

Para obter uma lista das condições invariáveis para termos usados na reflexão genérica, consulte os comentários da propriedade [IsGenericType](#):

- [IsGenericType](#): retornará true se um tipo for genérico.
- [GetGenericArguments](#): retorna uma matriz de objetos [Type](#) que representa os argumentos de tipo fornecidos para um tipo construído ou os parâmetros de tipo de uma definição de tipo genérico.
- [GetGenericTypeDefinition](#): retorna a definição de tipo genérico subjacente para o tipo construído atual.
- [GetGenericParameterConstraints](#): retorna uma matriz de objetos [Type](#) que representam as restrições no parâmetro de tipo genérico atual.
- [ContainsGenericParameters](#): retorna verdadeiro se o tipo ou qualquer um dos seus tipos ou métodos de delimitação contêm parâmetros de tipo para os quais não foram fornecidos tipos específicos.
- [GenericParameterAttributes](#): obtém uma combinação de sinalizadores [GenericParameterAttributes](#) que descrevem as restrições especiais do parâmetro de tipo genérico atual.
- [GenericParameterPosition](#): para um objeto [Type](#) que representa um parâmetro de tipo, obtém a posição do parâmetro de tipo na lista de parâmetros de tipo da definição de tipo genérico ou da definição de método genérico que declarou o parâmetro de tipo.
- [IsGenericParameter](#): obtém um valor que indica se o [Type](#) atual representa um parâmetro de tipo de um tipo genérico ou uma definição de método.
- [IsGenericTypeDefinition](#): obtém um valor que indica se o [Type](#) atual representa uma definição de tipo genérico, da qual outros tipos genéricos podem ser construídos. Retorna verdadeiro se o tipo representa a definição de um tipo genérico.
- [DeclaringMethod](#): retorna o método genérico que definiu o parâmetro de tipo genérico atual ou nulo, se o parâmetro de tipo não foi definido por um método

genérico.

- [MakeGenericType](#): substitui os elementos de uma matriz de tipos pelos parâmetros de tipo da definição de tipo genérico atual e retorna um objeto [Type](#) que representa o tipo construído resultante.

Além disso, membros da classe [MethodInfo](#) habilitam informações em tempo de execução para métodos genéricos. Consulte os comentários sobre a propriedade [IsGenericMethod](#) para obter uma lista das condições invariáveis para termos usados para refletir sobre os métodos genéricos:

- [IsGenericMethod](#): retorna true se um método é genérico.
- [GetGenericArguments](#): retorna uma matriz de objetos [Type](#) que representam os argumentos de tipo de um método genérico construído ou os parâmetros de tipo de uma definição de método genérico.
- [GetGenericMethodDefinition](#): retorna a definição de método genérico subjacente para o método construído atual.
- [ContainsGenericParameters](#): retorna true se o método ou qualquer um dos seus tipos de delimitação contêm parâmetros de tipo para os quais não foram fornecidos tipos específicos.
- [IsGenericMethodDefinition](#): retorna true se o [MethodInfo](#) atual representar a definição de um método genérico.
- [MakeGenericMethod](#): substitui os elementos de uma matriz de tipos pelos parâmetros de tipo da definição de método genérico atual e retorna um objeto [MethodInfo](#) que representa o método construído resultante.

## Confira também

- [Genéricos](#)
- [Reflexão e tipos genéricos](#)
- [Genéricos](#)

# Genéricos e atributos

Artigo • 20/03/2023

Atributos podem ser aplicados a tipos genéricos da mesma forma que a tipos não genéricos. No entanto, você pode aplicar atributos somente em *tipos genéricos abertos* e *tipos genéricos construídos fechados*, não em *tipos genéricos parcialmente construídos*. Um *tipo genérico aberto* é aquele em que nenhum dos argumentos de tipo é especificado, como `Dictionary< TKey, TValue >`. Um *tipo genérico construído fechado* especifica todos os argumentos de tipo, como `Dictionary<string, object>`. Um *tipo genérico parcialmente construído* especifica alguns argumentos de tipo, mas não todos. Um exemplo é `Dictionary<string, TValue>`.

Os exemplos a seguir usam esse atributo personalizado:

C#

```
class CustomAttribute : Attribute
{
    public object? info;
}
```

Um atributo pode referenciar um tipo genérico aberto:

C#

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Especifica vários parâmetros de tipo usando a quantidade apropriada de vírgulas. Neste exemplo, `GenericClass2` tem dois parâmetros de tipo:

C#

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<, >))]
class ClassB { }
```

Um atributo pode referenciar um tipo genérico construído fechado:

C#

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

Um atributo que referencia um parâmetro de tipo genérico causa um erro em tempo de compilação:

C#

```
[CustomAttribute(info = typeof(GenericClass3<int, T, string>))] //Error
CS0416
class ClassD<T> { }
```

A partir do C# 11, um tipo genérico pode herdar de [Attribute](#):

C#

```
public class CustomGenericAttribute<T> : Attribute { } //Requires C# 11
```

Para obter informações sobre um tipo genérico ou um parâmetro de tipo em tempo de execução, é possível usar os métodos do [System.Reflection](#). Para obter mais informações, confira [Genéricos e Reflexão](#).

## Confira também

- [Genéricos](#)
- [Atributos](#)

# E/S de arquivo e de fluxo

Artigo • 15/02/2023

E/S (entrada/saída) de arquivos e fluxos refere-se à transferência de dados de ou para uma mídia de armazenamento. No .NET, os namespaces `System.IO` contêm tipos que permitem a leitura e a gravação, de maneira síncrona e assíncrona, em fluxos de dados e arquivos. Esses namespaces também contêm tipos que executam compactação e descompactação em arquivos e tipos que possibilitam a comunicação por meio de pipes e portas seriais.

Um arquivo é uma coleção ordenada e nomeada de bytes com armazenamento persistente. Ao trabalhar com arquivos, você trabalha com caminhos de diretórios, armazenamento em disco e nomes de arquivos e diretórios. Por outro lado, um fluxo é uma sequência de bytes que você pode usar para ler e gravar em um repositório, o qual pode ser uma entre vários tipos de mídia de armazenamento (por exemplo, discos ou memória). Assim como há vários repositórios diferentes de discos, há vários tipos diferentes de fluxos diferentes de fluxos de arquivos, como os fluxos de rede, memória e pipes.

## Arquivos e diretórios

Você pode usar os tipos no namespace `System.IO` para interagir com arquivos e diretórios. Por exemplo, você pode obter e definir propriedades para arquivos e diretórios e recuperar coleções de arquivos e diretórios com base em critérios de pesquisa.

Para convenções de nomenclatura de caminhos e os modos de expressar um caminho de arquivo para sistemas Windows, incluindo a sintaxe de dispositivo DOS compatível com o .NET Core 1.1 e posterior e o .NET Framework 4.6.2 e posterior, confira [Formatos de caminho de arquivo em sistemas Windows](#).

Aqui estão algumas classes de arquivos e diretórios comumente usadas:

- [File](#) – Fornece métodos estáticos para criar, copiar, excluir, mover e abrir arquivos, além de ajudar na criação de um objeto [FileStream](#).
- [FileInfo](#) – Fornece métodos de instâncias para criar, copiar, excluir, mover e abrir arquivos, além de ajudar na criação de um objeto [FileStream](#).
- [Directory](#) – Fornece métodos estáticos para criar, mover e enumerar ao longo de diretórios e subdiretórios.

- [DirectoryInfo](#) – Fornece métodos de instância para criar, mover e enumerar ao longo de diretórios e subdiretórios.
- [Path](#) – Fornece métodos e propriedades para processar cadeias de caracteres de diretório de uma maneira compatível com várias plataformas.

Você sempre deve fornecer tratamento de exceção robusto ao chamar métodos de sistema de arquivos. Para obter mais informações, veja [Tratamento de erros de E/S](#).

Além de usar essas classes, os usuários do Visual Basic podem usar os métodos e as propriedades fornecidas pela classe [Microsoft.VisualBasic.FileIO.FileSystem](#) para E/S de arquivo.

Confira [Como copiar diretórios](#), [Como criar uma listagem de diretórios](#) e [Como enumerar diretórios e arquivos](#).

## Fluxos

A classe base abstrata [Stream](#) oferece suporte a leitura e gravação de bytes. Todas as classes que representam fluxos herdam da classe [Stream](#). A classe [Stream](#) e suas classes derivadas fornecem uma visão comum de fontes e repositórios de dados, isolando o programador de detalhes específicos do sistema operacional e dispositivos subjacentes.

Fluxos envolvem estas três operações fundamentais:

- Leitura – Transferência de dados de um fluxo para uma estrutura de dados, como uma matriz de bytes.
- Gravação – Transferência de dados para um fluxo a partir de uma fonte de dados.
- Busca – Consulta e modificação da posição atual em um fluxo.

Dependendo do repositório ou da fonte de dados subjacente, os fluxos podem oferecer suporte somente algumas dessas capacidades. Por exemplo, a classe [PipeStream](#) não oferece suporte à operação de busca. As propriedades [CanRead](#), [CanWrite](#) e [CanSeek](#) de um fluxo especificam as operações às quais o fluxo oferece suporte.

Algumas classes de fluxo comumente usadas são:

- [FileStream](#) – Para leitura e gravação em um arquivo.
- [IsolatedStorageFileStream](#) – Para leitura e gravação em um arquivo no armazenamento isolado.

- [MemoryStream](#) – Para leitura e gravação na memória como o repositório de backup.
- [BufferedStream](#) – Para melhorar o desempenho das operações de leitura e gravação.
- [NetworkStream](#) – Para leitura e gravação via soquetes de rede.
- [PipeStream](#) – Para leitura e gravação sobre pipes anônimos e nomeados.
- [CryptoStream](#) – Para vincular fluxos de dados a transformações criptográficas.

Para um exemplo de como trabalhar com fluxos de forma assíncrona, confira [E/S de arquivo assíncrono](#).

## Leitores e gravadores

O namespace [System.IO](#) também fornece tipos usados para ler caracteres codificados de fluxos e gravá-los em fluxos. Normalmente, os fluxos são criados para a entrada e a saída de bytes. Os tipos de leitor e de gravador tratam a conversão dos caracteres codificados de/para bytes para que o fluxo possa concluir a operação. Cada classe de leitor e gravador é associada a um fluxo, o qual pode ser recuperado pela propriedade `BaseStream` da classe.

Algumas classes de leitores e gravadores comumente usadas são:

- [BinaryReader](#) e [BinaryWriter](#) – Para leitura e gravação de tipos de dados primitivos como valores binários.
- [StreamReader](#) e [StreamWriter](#) – Para leitura e gravação de caracteres usando um valor de codificação para converter os caracteres para/de bytes.
- [StringReader](#) e [StringWriter](#) – Para leitura e gravação de caracteres e cadeias de caracteres.
- [TextReader](#) e [TextWriter](#) – Funcionam como as classes base abstratas para outros leitores e gravadores que leem e gravam caracteres e cadeias de caracteres, mas não dados binários.

Confira [Como ler texto de um arquivo](#), [Como gravar texto em um arquivo](#), [Como ler caracteres em uma cadeia de caracteres](#) e [Como gravar caracteres em uma cadeia de caracteres](#).

# Operações de E/S assíncronas

A leitura ou gravação de uma grande quantidade de dados pode consumir muitos recursos. Você deve executar essas tarefas de forma assíncrona se seu aplicativo precisar continuar respondendo ao usuário. Com as operações de E/S síncronas, o thread de interface do usuário é bloqueado até que a operação de uso intensivo seja concluída. Use operações de E/S assíncronas durante o desenvolvimento de aplicativos da Microsoft Store 8.x para evitar causar a impressão de que o aplicativo parou de funcionar.

Os membros assíncronas contêm `Async` em seus nomes, como os métodos `CopyToAsync`, `FlushAsync`, `ReadAsync` e `WriteAsync`. Você usa esses métodos com as palavras-chave `async` e `await`.

Para saber mais, confira [E/S de arquivo assíncrona](#).

## Compactação

A compactação refere-se ao processo de reduzir o tamanho de um arquivo para fins de armazenamento. A descompactação é o processo de extrair o conteúdo de um arquivo compactado para um formato utilizável. O namespace `System.IO.Compression` contém tipos para compactar e descompactar arquivos e fluxos.

As classes a seguir são frequentemente usadas para compactar e descompactar arquivos e fluxos:

- [ZipArchive](#) – Para criar e recuperar entradas em arquivos ZIP.
- [ZipArchiveEntry](#) – Para representar um arquivo compactado.
- [ZipFile](#) – para criar, extrair e abrir um pacote compactado.
- [ZipFileExtensions](#) – Para criar e extrair entradas em um pacote compactado.
- [DeflateStream](#) – Para compactar e descompactar fluxos usando o algoritmo de Deflate.
- [GZipStream](#) – Para compactar e descompactar fluxos no formato de dados GZIP.

Confira [How to: Compress and Extract Files](#) (Como compactar e extrair arquivos).

## Armazenamento isolado

Um armazenamento isolado é um mecanismo de armazenamento de dados que fornece isolamento e segurança ao definir maneiras padronizadas de associar códigos a dados salvos. O armazenamento fornece um sistema de arquivos virtual que é isolado por usuário, assembly e (opcionalmente) domínio. O armazenamento isolado é particularmente útil quando o aplicativo não tem permissão para acessar arquivos de usuários. Você pode salvar configurações ou arquivos para seu aplicativo de modo que ele seja controlado pela política de segurança do computador.

O armazenamento isolado não está disponível para aplicativos da Microsoft Store 8.x. Em vez disso, use as classes de dados do aplicativo no namespace [Windows.Storage](#). Para saber mais, veja [Dados de aplicativo](#).

As classes a seguir são usadas com frequência na implementação do armazenamento isolado:

- [IsolatedStorage](#) – Fornece a classe base para implementações de armazenamento isolado.
- [IsolatedStorageFile](#) – Fornece uma área de armazenamento isolado que contém arquivos e diretórios.
- [IsolatedStorageFileStream](#) – Expõe um arquivo no armazenamento isolado.

Confira [Armazenamentos isolado](#).

## Operações de E/S em aplicativos da Windows Store

O .NET para aplicativos da Microsoft Store 8.x contém muitos tipos para leitura e gravação em fluxos. No entanto, esse conjunto não inclui todos os tipos de E/S do .NET.

Algumas diferenças importantes que devem ser observadas ao usar operações de E/S em aplicativos da Microsoft Store 8.x:

- Tipos especificamente relacionados às operações de arquivo, como [File](#), [FileInfo](#), [Directory](#) e [DirectoryInfo](#), não estão incluídos no .NET para aplicativos da Microsoft Store 8.x. Em vez disso, use os tipos no namespace [Windows.Storage](#) do Windows Runtime, como [StorageFile](#) e [StorageFolder](#).
- O armazenamento isolado não está disponível. Use [dados de aplicativo](#).
- Use métodos assíncronos, como [ReadAsync](#) e [WriteAsync](#), para evitar o bloqueio do thread da interface do usuário.

- Os tipos de compactação com base em caminhos [ZipFile](#) e [ZipFileExtensions](#) não estão disponíveis. Em vez disso, use os tipos no namespace [Windows.Storage.Compression](#).

É possível converter entre fluxos do .NET Framework e fluxos do Windows Runtime, se necessário. Para mais informações, confira [Como converter entre fluxos do .NET Framework e fluxos do Windows Runtime](#) ou [WindowsRuntimeStreamExtensions](#).

Para saber mais sobre operações de E/S em um aplicativo da Microsoft Store 8.x, confira [Guia de início rápido: leitura e gravação de arquivos](#).

## E/S e segurança

Ao usar as classes no namespace [System.IO](#), você deve atender aos requisitos de segurança do sistema operacional, como ACLs (listas de controle de acesso) para controlar o acesso a arquivos e diretórios. Esse é um requisito adicional aos requisitos de [FileIOPermission](#). As ACLs podem ser gerenciadas por meio de programação. Para saber mais, confira [Como adicionar ou remover entradas da lista de controle de acesso](#).

As políticas de segurança padrão impedem que aplicativos da Internet ou intranet acessem arquivos no computador do usuário. Consequentemente, não use classes de E/S que exijam um caminho para um arquivo físico ao escrever código que será baixado via Internet ou intranet. Em vez disso, use [armazenamento isolado](#) para aplicativos .NET.

Uma verificação de segurança é executada somente quando o fluxo é construído. Consequentemente, não abra um fluxo para depois passá-lo para código ou domínios de aplicativos menos confiáveis.

## Tópicos relacionados

- [Tarefas comuns de E/S](#)

Fornece uma lista das tarefas de E/S associadas a arquivos, diretórios e fluxos, além de links para conteúdo e exemplos relevantes para cada tarefa.

- [E/S de arquivo assíncrona](#)

Descreve as vantagens de desempenho e a operação básica da E/S assíncrona.

- [Armazenamentos isolado](#)

Descreve um mecanismo de armazenamento isolado que fornece isolamento e segurança ao definir maneiras padronizadas de associar códigos aos dados salvos.

- [Pipes](#)

Descreve operações de pipes anônimos e nomeados no .NET.

- [Arquivos mapeados em memória](#)

Descreve arquivos mapeados na memória, os quais armazenam o conteúdo de arquivos do disco na memória virtual. Você pode usar arquivos mapeados na memória para editar arquivos muito grandes e para criar memória compartilhada para a comunicação entre processos.

# Como enumerar diretórios e arquivos

Artigo • 10/05/2023

Coleções enumeráveis fornecem um desempenho melhor do que matrizes ao trabalhar com coleções grandes de arquivos e diretórios. Para enumerar diretórios e arquivos, use métodos que retornam uma coleção enumerável de nomes de diretório ou arquivo ou seus objetos [DirectoryInfo](#), [FileInfo](#) ou [FileSystemInfo](#).

Caso deseje pesquisar e retornar somente os nomes de diretórios ou arquivos, use os métodos de enumeração da classe [Directory](#). Caso deseje pesquisar e retornar outras propriedades de diretórios ou arquivos, use as classes  [DirectoryInfo](#) e  [FileSystemInfo](#).

Use coleções enumeráveis desses métodos como o parâmetro [IEnumerable<T>](#) para construtores de classes de coleção como [List<T>](#).

A seguinte tabela resume os métodos que retornam coleções enumeráveis de arquivos e diretórios:

Para pesquisar e retornar	Use o método
Nomes de diretório	<a href="#">Directory.EnumerateDirectories</a>
Informações de diretório ( <a href="#"> DirectoryInfo</a> )	<a href="#"> DirectoryInfo.EnumerateDirectories</a>
Nomes de arquivo	<a href="#">Directory.EnumerateFiles</a>
Informações do arquivo ( <a href="#"> FileInfo</a> )	<a href="#"> DirectoryInfo.EnumerateFiles</a>
Nomes de entrada do sistema de arquivos	<a href="#">Directory.EnumerateFileSystemEntries</a>
Informações de entrada do sistema de arquivos ( <a href="#"> FileSystemInfo</a> )	<a href="#"> DirectoryInfo.EnumerateFileSystemInfos</a>
Nomes de diretório e arquivo	<a href="#">Directory.EnumerateFileSystemEntries</a>

## ⓘ Observação

Embora você possa enumerar imediatamente todos os arquivos nos subdiretórios de um diretório pai usando a opção [AllDirectories](#) da enumeração [SearchOption](#) opcional, os erros [UnauthorizedAccessException](#) podem tornar a enumeração incompleta. Capture essas exceções enumerando primeiro os diretórios e, em seguida, os arquivos.

# Exemplos: Usar a classe Directory

O exemplo a seguir usa o método [Directory.EnumerateDirectories\(String\)](#) para obter uma lista dos nomes de diretório de nível superior em um caminho especificado.

C#

```
using System;
using System.Collections.Generic;
using System.IO;

class Program
{
    private static void Main(string[] args)
    {
        try
        {
            // Set a variable to the My Documents path.
            string docPath =
                Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            List<string> dirs = new List<string>
                (Directory.EnumerateDirectories(docPath));

            foreach (var dir in dirs)
            {
                Console.WriteLine($"{dir.Substring(dir.LastIndexOf(Path.DirectorySeparatorChar) + 1)}");
            }
            Console.WriteLine($"{dirs.Count} directories found.");
        }
        catch (UnauthorizedAccessException ex)
        {
            Console.WriteLine(ex.Message);
        }
        catch (PathTooLongException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

O exemplo a seguir usa o método [Directory.EnumerateFiles\(String, String, SearchOption\)](#) para enumerar recursivamente todos os nomes de arquivo em um diretório e subdiretórios que correspondam a determinado padrão. Em seguida, ele lê cada linha de cada arquivo e exibe as linhas que contêm uma cadeia de caracteres especificada, com seus nomes de arquivo e caminhos.

C#

```

using System;
using System.IO;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        try
        {
            // Set a variable to the My Documents path.
            string docPath =
                Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            var files = from file in Directory.EnumerateFiles(docPath,
                "*.txt", SearchOption.AllDirectories)
                from line in File.ReadLines(file)
                where line.Contains("Microsoft")
                select new
                {
                    File = file,
                    Line = line
                };

            foreach (var f in files)
            {
                Console.WriteLine($"{f.File}\t{f.Line}");
            }
            Console.WriteLine($"{files.Count().ToString()} files found.");
        }
        catch (UnauthorizedAccessException uAEx)
        {
            Console.WriteLine(uAEx.Message);
        }
        catch (PathTooLongException pathEx)
        {
            Console.WriteLine(pathEx.Message);
        }
    }
}

```

## Exemplos: Usar a classe DirectoryInfo

O exemplo a seguir usa o método `DirectoryInfo.EnumerateDirectories` para listar uma coleção de diretórios de nível superior cuja `CreationTimeUtc` é anterior a determinado valor `DateTime`.

```

using System;
using System.IO;

namespace EnumDir
{
    class Program
    {
        static void Main(string[] args)
        {
            // Set a variable to the Documents path.
            string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            DirectoryInfo dirPrograms = new DirectoryInfo(docPath);
            DateTime StartOf2009 = new DateTime(2009, 01, 01);

            var dirs = from dir in dirPrograms.EnumerateDirectories()
                       where dir.CreationTimeUtc > StartOf2009
                       select new
                       {
                           ProgDir = dir,
                       };

            foreach (var di in dirs)
            {
                Console.WriteLine($"{di.PrgDir.Name}");
            }
        }
    }
// </Snippet1>

```

O exemplo a seguir usa o método `DirectoryInfo.EnumerateFiles` para listar todos os arquivos cujo `Length` excede 10 MB. Este exemplo enumera primeiro os diretórios de nível superior para capturar possíveis exceções de acesso não autorizado e, em seguida, enumera os arquivos.

C#

```

using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {
        // Set a variable to the My Documents path.
        string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        DirectoryInfo diTop = new DirectoryInfo(docPath);

```

```
try
{
    foreach (var fi in diTop.EnumerateFiles())
    {
        try
        {
            // Display each file over 10 MB;
            if (fi.Length > 10000000)
            {
                Console.WriteLine($"${fi.FullName}\t{fi.Length.ToString("N0")}");
            }
        }
        catch (UnauthorizedAccessException unAuthTop)
        {
            Console.WriteLine($"{unAuthTop.Message}");
        }
    }

    foreach (var di in diTop.EnumerateDirectories("*"))
    {
        try
        {
            foreach (var fi in di.EnumerateFiles("*",
SearchOption.AllDirectories))
            {
                try
                {
                    // Display each file over 10 MB;
                    if (fi.Length > 10000000)
                    {
                        Console.WriteLine($"${fi.FullName}\t{fi.Length.ToString("N0")}");
                    }
                }
                catch (UnauthorizedAccessException unAuthFile)
                {
                    Console.WriteLine($"unAuthFile:
{unAuthFile.Message}");
                }
            }
        }
        catch (UnauthorizedAccessException unAuthSubDir)
        {
            Console.WriteLine($"unAuthSubDir:
{unAuthSubDir.Message}");
        }
    }

    catch (FileNotFoundException dirNotFound)
    {
        Console.WriteLine($"{dirNotFound.Message}");
    }
    catch (UnauthorizedAccessException unAuthDir)
```

```
{  
    Console.WriteLine($"unAuthDir: {unAuthDir.Message}");  
}  
catch (PathTooLongException longPath)  
{  
    Console.WriteLine($"{longPath.Message}");  
}  
}  
}
```

## Confira também

- E/S de arquivo e de fluxo

# Tarefas comuns de E/S

Artigo • 10/05/2023

O namespace [System.IO](#) fornece várias classes que permitem que várias ações, como leitura e gravação, sejam realizadas em arquivos, diretórios e fluxos. Para obter mais informações, confira [E/S de arquivo e fluxo](#).

## Tarefas comuns de arquivos

Para fazer isso...	Veja o exemplo neste tópico...
Criar um arquivo de texto	Método <a href="#">File.CreateText</a>  Método <a href="#">FileInfo.CreateText</a>
	Método <a href="#">File.Create</a>  Método <a href="#">FileInfo.Create</a>
Gravar em um arquivo de texto	<a href="#">Como gravar texto em um arquivo</a>  <a href="#">Como escrever um arquivo de texto (C++/CLI)</a>
Ler de um arquivo de texto	<a href="#">Como ler texto de um arquivo</a>
Anexar texto em um arquivo	<a href="#">Como abrir e acrescentar a um arquivo de log</a>  Método <a href="#">File.AppendText</a>  Método <a href="#">FileInfo.AppendText</a>
Renomear ou mover um arquivo	Método <a href="#">File.Move</a>  Método <a href="#">FileInfo.MoveTo</a>
Excluir um arquivo	Método <a href="#">File.Delete</a>  Método <a href="#">FileInfo.Delete</a>
Copiar um arquivo	Método <a href="#">File.Copy</a>  Método <a href="#">FileInfo.CopyTo</a>
Obter o tamanho de um arquivo	Propriedade <a href="#">FileInfo.Length</a>
Obter os atributos de um arquivo	Método <a href="#">File.GetAttributes</a>

<b>Para fazer isso...</b>	<b>Veja o exemplo neste tópico...</b>
Definir os atributos de um arquivo	Método <a href="#">File.SetAttributes</a>
Determinar se um arquivo existe	Método <a href="#">File.Exists</a>
Ler de um arquivo binário	<a href="#">Como ler e gravar em um arquivo de dados recém-criado</a>
Gravar em um arquivo binário	<a href="#">Como ler e gravar em um arquivo de dados recém-criado</a>
Recuperar uma extensão de nome de arquivo	Método <a href="#">Path.GetExtension</a>
Recuperar o caminho totalmente qualificado de um arquivo	Método <a href="#">Path.GetFullPath</a>
Recuperar o nome e a extensão do arquivo de um caminho	Método <a href="#">Path.GetFileName</a>
Alterar a extensão de um arquivo	Método <a href="#">Path.ChangeExtension</a>

## Tarefas comuns de diretório

<b>Para fazer isso...</b>	<b>Veja o exemplo neste tópico...</b>
Acessar um arquivo em uma pasta especial, como Meus Documentos	<a href="#">Como gravar texto em um arquivo</a>
Criar um diretório	Método <a href="#">Directory.CreateDirectory</a>  Propriedade <a href="#">FileInfo.Directory</a>
Criar um subdiretório	Método <a href="#">DirectoryInfo.CreateSubdirectory</a>
Renomear ou mover um diretório	Método <a href="#">Directory.Move</a>  Método <a href="#">DirectoryInfo.MoveTo</a>
Copiar um diretório	<a href="#">Como: copiar diretórios</a>
Excluir um diretório	Método <a href="#">Directory.Delete</a>  Método <a href="#">DirectoryInfo.Delete</a>
Ver os arquivos e subdiretórios em um diretório	<a href="#">Como: enumerar diretórios e arquivos</a>
Descobrir o tamanho de um diretório	Classe <a href="#">System.IO.Directory</a>

**Para fazer isso...**

Determinar se um diretório existe

**Veja o exemplo neste tópico...**

Método [Directory.Exists](#)

## Confira também

- [E/S de arquivo e de fluxo](#)
- [Compor fluxos](#)
- [E/S de arquivo assíncrona](#)

# Como: ler e gravar em um arquivo de dados recém-criado

Artigo • 10/05/2023

As classes [System.IO.BinaryWriter](#) e [System.IO.BinaryReader](#) são usadas para gravar e ler dados que não sejam cadeias de caracteres. O exemplo a seguir mostra como criar um fluxo de arquivo vazio, gravar dados nele e ler dados dele.

O exemplo cria um arquivo de dados chamado *Test.data* no diretório atual, cria os objetos [BinaryWriter](#) e [BinaryReader](#) associados e usa o objeto [BinaryWriter](#) para gravar os inteiros de 0 a 10 em *Test.data*, o que deixa o ponteiro de arquivo no final do arquivo. Em seguida, o objeto [BinaryReader](#) define o ponteiro de arquivo novamente para a origem e lê o conteúdo especificado.

## ① Observação

Se *Test.data* já existir no diretório atual, uma exceção [IOException](#) será gerada. Use a opção de modo de arquivo  [FileMode.Create](#) em vez de  [FileMode.CreateNew](#) para sempre criar um arquivo sem gerar uma exceção.

## Exemplo

C#

```
using System;
using System.IO;

class MyStream
{
    private const string FILE_NAME = "Test.data";

    public static void Main()
    {
        if (File.Exists(FILE_NAME))
        {
            Console.WriteLine($"{FILE_NAME} already exists!");
            return;
        }

        using (FileStream fs = new FileStream(FILE_NAME,
FileMode.CreateNew))
        {
            using (BinaryWriter w = new BinaryWriter(fs))
```

```

    {
        for (int i = 0; i < 11; i++)
        {
            w.Write(i);
        }
    }

    using (FileStream fs = new FileStream(FILE_NAME, FileMode.Open,
FileAccess.Read))
    {
        using (BinaryReader r = new BinaryReader(fs))
        {
            for (int i = 0; i < 11; i++)
            {
                Console.WriteLine(r.ReadInt32());
            }
        }
    }
}

// The example creates a file named "Test.data" and writes the integers 0
// through 10 to it in binary format.
// It then writes the contents of Test.data to the console with each integer
// on a separate line.

```

## Confira também

- [BinaryReader](#)
- [BinaryWriter](#)
- [FileStream](#)
- [FileStream.Seek](#)
- [SeekOrigin](#)
- [Como: enumerar diretórios e arquivos](#)
- [Como: abrir um arquivo de log e acrescentar dados a ele](#)
- [Como ler texto de um arquivo](#)
- [Como gravar texto em um arquivo](#)
- [Como: ler caracteres de uma cadeia de caracteres](#)
- [Como: escrever caracteres em uma cadeia de caracteres](#)
- [E/S de arquivo e de fluxo](#)

# Como copiar diretórios

Artigo • 07/04/2023

Este artigo demonstra como usar classes de E/S para copiar de maneira síncrona o conteúdo de um diretório para outro local.

Para obter um exemplo de cópia assíncrona de arquivo, confira [E/S assíncrona de arquivo](#).

Este exemplo copia os subdiretórios definindo o parâmetro `recursive` do método `CopyDirectory` como `true`. O método `CopyDirectory` copia os subdiretórios recursivamente chamando a si próprio em cada subdiretório até não existir nada mais para copiar.

## Exemplo

C#

```
using System.IO;

CopyDirectory(@".\\", @".\copytest", true);

static void CopyDirectory(string sourceDir, string destinationDir, bool
recursive)
{
    // Get information about the source directory
    var dir = new DirectoryInfo(sourceDir);

    // Check if the source directory exists
    if (!dir.Exists)
        throw new DirectoryNotFoundException($"Source directory not found:
{dir.FullName}");

    // Cache directories before we start copying
    DirectoryInfo[] dirs = dir.GetDirectories();

    // Create the destination directory
    Directory.CreateDirectory(destinationDir);

    // Get the files in the source directory and copy to the destination
    // directory
    foreach (FileInfo file in dir.GetFiles())
    {
        string targetFilePath = Path.Combine(destinationDir, file.Name);
        file.CopyTo(targetFilePath);
    }

    // If recursive and copying subdirectories, recursively call this method
}
```

```
if (recursive)
{
    foreach (DirectoryInfo subDir in dirs)
    {
        string newDestinationDir = Path.Combine(destinationDir,
subDir.Name);
        CopyDirectory(subDir.FullName, newDestinationDir, true);
    }
}
```

## Confira também

- [FileInfo](#)
- [DirectoryInfo](#)
- [FileStream](#)
- [E/S de arquivo e de fluxo](#)
- [Tarefas comuns de E/S](#)
- [E/S de arquivo assíncrona](#)

# Registry Classe

Referência

## Definição

Namespace: [Microsoft.Win32](#)

Assembly: Microsoft.Win32.Registry.dll

Fornece objetos [RegistryKey](#) que representam as chaves raiz no Registro do Windows e os métodos `static` para acessar os pares chave/valor.

C#

```
public static class Registry
```

Herança [Object](#) → Registry

## Exemplos

Esta seção contém dois exemplos de código. O primeiro exemplo demonstra as chaves raiz e o segundo exemplo demonstra os `static` [GetValue](#) métodos e [SetValue](#).

### Exemplo 1

O exemplo de código a seguir demonstra como recuperar as subchaves da chave HKEY\_USERS e imprimir seus nomes na tela. Use o [OpenSubKey](#) método para criar uma instância da subchave específica de interesse. Em seguida, você pode usar outras operações no [RegistryKey](#) para manipular essa chave.

C#

```
using System;
using Microsoft.Win32;

class Reg {
    public static void Main() {

        // Create a RegistryKey, which will access the HKEY_USERS
        // key in the registry of this machine.
        RegistryKey rk = Registry.Users;

        // Print out the keys.
        PrintKeys(rk);
    }
}

void PrintKeys(RegistryKey rk) {
    foreach (string subkeyName in rk.GetSubKeyNames())
        Console.WriteLine(subkeyName);
}
```

```

    }

    static void PrintKeys(RegistryKey rkey) {

        // Retrieve all the subkeys for the specified key.
        string [] names = rkey.GetSubKeyNames();

        int icount = 0;

        Console.WriteLine("Subkeys of " + rkey.Name);
        Console.WriteLine("-----");
    });

        // Print the contents of the array to the console.
        foreach (string s in names) {
            Console.WriteLine(s);

            // The following code puts a limit on the number
            // of keys displayed. Comment it out to print the
            // complete list.
            icount++;
            if (icount >= 10)
                break;
        }
    }
}

```

## Exemplo 2

O exemplo de código a seguir armazena valores de vários tipos de dados em uma chave de exemplo, criando a chave à medida que faz isso e, em seguida, recupera e exibe os valores. O exemplo demonstra como armazenar e recuperar o par nome/valor padrão (sem nome) e o uso de `defaultValue` quando um par nome/valor não existe.

C#

```

using System;
using Microsoft.Win32;

public class Example
{
    public static void Main()
    {
        // The name of the key must include a valid root.
        const string userRoot = "HKEY_CURRENT_USER";
        const string subkey = "RegistrySetValueExample";
        const string keyName = userRoot + "\\\" + subkey;

        // An int value can be stored without specifying the
        // registry data type, but long values will be stored
        // as strings unless you specify the type. Note that
        // the int is stored in the default name/value
    }
}

```

```
// pair.
Registry.SetValue(keyName, "", 5280);
Registry.SetValue(keyName, "TestLong", 12345678901234,
    RegistryValueKind.QWord);

// Strings with expandable environment variables are
// stored as ordinary strings unless you specify the
// data type.
Registry.SetValue(keyName, "TestExpand", "My path: %path%");
Registry.SetValue(keyName, "TestExpand2", "My path: %path%",
    RegistryValueKind.ExpandString);

// Arrays of strings are stored automatically as
// MultiString. Similarly, arrays of Byte are stored
// automatically as Binary.
string[] strings = {"One", "Two", "Three"};
Registry.SetValue(keyName, "TestArray", strings);

// Your default value is returned if the name/value pair
// does not exist.
string noSuch = (string) Registry.GetValue(keyName,
    "NoSuchName",
    "Return this default if NoSuchName does not exist.");
Console.WriteLine("\r\nNoSuchName: {0}", noSuch);

// Retrieve the int and long values, specifying
// numeric default values in case the name/value pairs
// do not exist. The int value is retrieved from the
// default (nameless) name/value pair for the key.
int tInteger = (int) Registry.GetValue(keyName, "", -1);
Console.WriteLine("(Default): {0}", tInteger);
long tLong = (long) Registry.GetValue(keyName, "TestLong",
    long.MinValue);
Console.WriteLine("TestLong: {0}", tLong);

// When retrieving a MultiString value, you can specify
// an array for the default return value.
string[] tArray = (string[]) Registry.GetValue(keyName,
    "TestArray",
    new string[] {"Default if TestArray does not exist."});
for(int i=0; i<tArray.Length; i++)
{
    Console.WriteLine("TestArray({0}): {1}", i, tArray[i]);
}

// A string with embedded environment variables is not
// expanded if it was stored as an ordinary string.
string tExpand = (string) Registry.GetValue(keyName,
    "TestExpand",
    "Default if TestExpand does not exist.");
Console.WriteLine("TestExpand: {0}", tExpand);

// A string stored as ExpandString is expanded.
string tExpand2 = (string) Registry.GetValue(keyName,
    "TestExpand2",
```

```

        "Default if TestExpand2 does not exist.");
Console.WriteLine("TestExpand2: {0}...", 
    tExpand2.Substring(0, 40));

        Console.WriteLine("\r\nUse the registry editor to examine the
key.");
        Console.WriteLine("Press the Enter key to delete the key.");
        Console.ReadLine();
        Registry.CurrentUser.DeleteSubKey(subkey);
    }
}
// This code example produces output similar to the following:
//
//NoSuchName: Return this default if NoSuchName does not exist.
//(Default): 5280
//TestLong: 12345678901234
//TestArray(0): One
//TestArray(1): Two
//TestArray(2): Three
//TestExpand: My path: %path%
//TestExpand2: My path: D:\Program Files\Microsoft.NET\...
//
//Use the registry editor to examine the key.
//Press the Enter key to delete the key.

```

## Comentários

Essa classe fornece o conjunto de chaves raiz padrão encontradas no Registro em computadores que executam o Windows. O registro é um recurso de armazenamento para obter informações sobre aplicativos, usuários e configurações padrão do sistema. Por exemplo, os aplicativos podem usar o registro para armazenar informações que precisam ser preservadas depois que o aplicativo é fechado e acessar essas mesmas informações quando o aplicativo é recarregado. Por exemplo, você pode armazenar preferências de cor, locais de tela ou o tamanho da janela. Você pode controlar esses dados para cada usuário armazenando as informações em um local diferente no Registro.

As instâncias base ou raiz [RegistryKey](#) expostas pela [Registry](#) classe delineam o mecanismo de armazenamento básico para subchaves e valores no Registro. Todas as chaves são somente leitura porque o registro depende de sua existência. As chaves expostas por [Registry](#) são:

[CurrentUser](#) Armazena informações sobre as preferências do usuário.

[LocalMachine](#) Armazena informações de configuração para o computador local.

[ClassesRoot](#) Armazena informações sobre tipos (e classes) e suas propriedades.

[Users](#) Armazena informações sobre a configuração de usuário padrão.

[PerformanceData](#) Armazena informações de desempenho para componentes de software.

[CurrentConfig](#) Armazena informações de hardware não específicas do usuário.

[DynData](#) Armazena dados dinâmicos.

Depois de identificar a chave raiz sob a qual você deseja armazenar/recuperar informações do Registro, você pode usar a [RegistryKey](#) classe para adicionar ou remover subchaves e manipular os valores de uma determinada chave.

Os dispositivos de hardware podem colocar informações no registro automaticamente usando a interface Plug and Play. O software para instalar drivers de dispositivo pode colocar informações no registro gravando em APIs padrão.

## Métodos estáticos para obter e definir valores

A [Registry](#) classe também contém `static GetValue` métodos e `SetValue` para definir e recuperar valores de chaves do Registro. Esses métodos abrem e fecham chaves do Registro sempre que são usadas, para que não sejam executados tão bem quanto métodos análogos na [RegistryKey](#) classe , quando você acessa um grande número de valores.

A [RegistryKey](#) classe também fornece métodos que permitem definir a segurança do controle de acesso do Windows para chaves do Registro, testar o tipo de dados de um valor antes de recuperá-lo e excluir chaves.

## Campos

<a href="#">ClassesRoot</a>	Define os tipos (ou classes) de documentos e as propriedades associadas a esses tipos. Este campo lê a chave base do Registro HKEY_CLASSES_ROOT do Windows.
<a href="#">CurrentConfig</a>	Contém informações de configuração relacionadas ao hardware que não é específico do usuário. Este campo lê a chave base HKEY_CURRENT_CONFIG do Registro do Windows.
<a href="#">CurrentUser</a>	Contém informações sobre as preferências do usuário atual. Este campo lê a chave base de Registro HKEY_CURRENT_USER do Windows.

<a href="#">LocalMachine</a>	Contém os dados de configuração para o computador local. Este campo lê a chave de base de Registro HKEY_LOCAL_MACHINE do Windows.
<a href="#">PerformanceData</a>	Contém informações de desempenho de componentes de software. Esse campo lê a chave base do Registro HKEY_PERFORMANCE_DATA do Windows.
<a href="#">Users</a>	Contém informações sobre a configuração de usuário padrão. Este campo lê a chave base do Registro do Windows HKEY_USERS.

## Métodos

<a href="#">GetValue(String, String, Object)</a>	Recupera o valor associado ao nome especificado, na chave do Registro especificada. Se o nome não for encontrado na chave especificada, retornará um valor padrão que você fornecer, ou <code>null</code> se a chave especificada não existir.
<a href="#">SetValue(String, String, Object)</a>	Define o par nome-valor especificado na chave do Registro especificada. Se a chave especificada não existir, ela será criada.
<a href="#">SetValue(String, String, Object, RegistryValueKind)</a>	Define o par nome-valor na chave do Registro especificada, usando o tipo de dados do Registro especificado. Se a chave especificada não existir, ela será criada.

## Aplica-se a

Produto	Versões
<b>.NET</b>	Core 1.0, Core 1.1, 6, 7, 8
<b>.NET Framework</b>	1.1, 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
<b>.NET Platform Extensions</b>	2.1, 2.2, 3.0, 3.1, 5
<b>Windows Desktop</b>	3.0, 3.1, 5
<b>Xamarin.iOS</b>	10.8
<b>Xamarin.Mac</b>	3.0

## Confira também

- RegistryHive
- RegistryKey

# Interoperability Overview

Article • 02/25/2023

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is *managed code*, and code that runs outside the CLR is *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the [System.Runtime.InteropServices](#) namespace, C++ interoperability, and COM interoperability (COM interop).

## Platform Invoke

*Platform invoke* is a service that enables managed code to call unmanaged functions implemented in dynamic link libraries (DLLs), such as the Microsoft Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see [Consuming Unmanaged DLL Functions](#) and [How to use platform invoke to play a WAV file](#).

### ⓘ Note

The **Common Language Runtime** (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see [Security in .NET](#).

## C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class. C++ interop enables code authored in C# or another .NET language to access it. You write C++ code to wrap a native DLL or COM component. Unlike other .NET languages, Visual C++ has interoperability support that enables managed and unmanaged code in the same application and even in the same file. You then build the C++ code by using the `/clr` compiler switch to produce a managed assembly. Finally, you add a reference to

the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

## Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use `regsvr32.exe` to register or un-register a COM DLL.
2. Add to the project a reference to the COM component or type library. When you add the reference, Visual Studio uses the [Tlbimp.exe \(Type Library Importer\)](#), which takes a type library as input, to output a .NET interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.
3. Create an instance of a class defined in the RCW. Creating an instance of that class creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see [Exposing COM Components to the .NET Framework](#).

## Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project. You can make an assembly COM visible by modifying C# project properties. For more information, see [Assembly Information Dialog Box](#).
2. Generate a COM type library and register it for COM usage. You can modify C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the [Regasm.exe \(Assembly Registration Tool\)](#), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see [Exposing .NET Framework Components to COM](#) and [Example COM Class](#).

## See also

- [Improving Interop Performance](#)
- [Introduction to Interoperability between COM and .NET](#)
- [Introduction to COM Interop in Visual Basic](#)
- [Marshaling between Managed and Unmanaged Code](#)
- [Interoperating with Unmanaged Code](#)

# Interoperability Overview

Article • 02/25/2023

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is *managed code*, and code that runs outside the CLR is *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the [System.Runtime.InteropServices](#) namespace, C++ interoperability, and COM interoperability (COM interop).

## Platform Invoke

*Platform invoke* is a service that enables managed code to call unmanaged functions implemented in dynamic link libraries (DLLs), such as the Microsoft Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see [Consuming Unmanaged DLL Functions](#) and [How to use platform invoke to play a WAV file](#).

### ⓘ Note

The **Common Language Runtime** (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see [Security in .NET](#).

## C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class. C++ interop enables code authored in C# or another .NET language to access it. You write C++ code to wrap a native DLL or COM component. Unlike other .NET languages, Visual C++ has interoperability support that enables managed and unmanaged code in the same application and even in the same file. You then build the C++ code by using the `/clr` compiler switch to produce a managed assembly. Finally, you add a reference to

the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

## Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use `regsvr32.exe` to register or un-register a COM DLL.
2. Add to the project a reference to the COM component or type library. When you add the reference, Visual Studio uses the [Tlbimp.exe \(Type Library Importer\)](#), which takes a type library as input, to output a .NET interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.
3. Create an instance of a class defined in the RCW. Creating an instance of that class creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see [Exposing COM Components to the .NET Framework](#).

## Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project. You can make an assembly COM visible by modifying C# project properties. For more information, see [Assembly Information Dialog Box](#).
2. Generate a COM type library and register it for COM usage. You can modify C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the [Regasm.exe \(Assembly Registration Tool\)](#), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see [Exposing .NET Framework Components to COM](#) and [Example COM Class](#).

## See also

- [Improving Interop Performance](#)
- [Introduction to Interoperability between COM and .NET](#)
- [Introduction to COM Interop in Visual Basic](#)
- [Marshaling between Managed and Unmanaged Code](#)
- [Interoperating with Unmanaged Code](#)

# Como acessar objetos de interoperabilidade do Office

Artigo • 09/03/2023

O C# tem recursos que simplificam o acesso a objetos de API do Office. Os novos recursos incluem argumentos nomeados e opcionais, um novo tipo chamado `dynamic` e a capacidade de passar argumentos para parâmetros de referência em métodos COM como se fossem parâmetros de valor.

Neste artigo, você usará os novos recursos para escrever código que cria e exibe uma planilha do Microsoft Office Excel. Escreva o código para adicionar um documento do Office Word que contenha um ícone que esteja vinculado à planilha do Excel.

Para concluir este passo a passo, você deve ter o Microsoft Office Excel 2007 e o Microsoft Office Word 2007 ou versões posteriores, instaladas no computador.

## ⓘ Observação

Seu computador pode mostrar diferentes nomes ou locais para alguns dos elementos de interface do usuário do Visual Studio nas instruções a seguir. A edição do Visual Studio que você possui e as configurações que você usa determinam esses elementos. Para obter mais informações, consulte [Personalizando o IDE](#).

## ⓘ Importante

O VSTO se baseia no [.NET Framework](#). Os suplementos COM também podem ser gravados com o .NET Framework. Os suplementos do Office não podem ser criados com o [.NET Core](#) e o [.NET 5+](#), as versões mais recentes do .NET. Isso ocorre porque o .NET Core/.NET 5+ não pode trabalhar em conjunto com o .NET Framework no mesmo processo e pode levar a falhas de carga de suplemento. Você pode continuar a usar o .NET Framework para escrever suplementos VSTO e COM para o Office. A Microsoft não atualizará o VSTO ou a plataforma de suplemento COM para usar o .NET Core ou o .NET 5+. Você pode aproveitar o .NET Core e o .NET 5+, incluindo o ASP.NET Core, para criar o lado do servidor dos [Suplementos Web do Office](#).

## Para criar um novo aplicativo de console

1. Inicie o Visual Studio.
2. No menu **Arquivo**, aponte para **Novo** e selecione **Projeto**. A caixa de diálogo **Novo Projeto** aparecerá.
3. No painel **Modelos Instalados**, expanda **C#** e selecione **Windows**.
4. Observe a parte superior da caixa de diálogo **Novo Projeto** para selecionar **.NET Framework 4** (ou versão posterior) como uma estrutura de destino.
5. No painel **Modelos**, selecione **Aplicativo de Console**.
6. Digite um nome para o projeto no campo **Nome**.
7. Selecione **OK**.

O novo projeto aparece no **Gerenciador de Soluções**.

## Para adicionar referências

1. No **Gerenciador de Soluções**, clique com o botão direito do mouse no nome do projeto e, em seguida, selecione **Adicionar Referência**. A caixa de diálogo **Adicionar Referência** é exibida.
2. Na página **Assemblies**, selecione **Microsoft.Office.Interop.Word** na lista **Nome do Componente** e, mantendo a tecla CTRL pressionada, selecione **Microsoft.Office.Interop.Excel**. Se você não vir os assemblies, talvez seja necessário instalá-los. Confira [Como instalar assemblies de interoperabilidade primários do Office](#).
3. Selecione **OK**.

## Para adicionar as diretivas using necessárias

No **Gerenciador de Soluções**, clique com o botão direito do mouse no arquivo **Program.cs** e, em seguida, selecione **Exibir Código**. Adicione as seguintes diretivas **using** na parte superior do arquivo de código:

```
C#  
  
using Excel = Microsoft.Office.Interop.Excel;  
using Word = Microsoft.Office.Interop.Word;
```

## Para criar uma lista de contas bancárias

Cole a seguinte definição de classe em **Program.cs**, na classe **Program**.

```
C#
```

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

Adicione o seguinte código ao método `Main` para criar uma lista `bankAccounts` que contenha duas contas.

C#

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
        ID = 345678,
        Balance = 541.27
    },
    new Account {
        ID = 1230221,
        Balance = -127.44
    }
};
```

## Para declarar um método que exporta as informações de conta para o Excel

1. Adicione o método a seguir à classe `Program` para configurar uma planilha do Excel. O método `Add` tem um parâmetro opcional para especificar um modelo específico. Os parâmetros opcionais permitem omitir o argumento para esse parâmetro, se você deseja usar o valor padrão do parâmetro. Como você não forneceu um argumento, `Add` usará o modelo padrão e criará uma nova pasta de trabalho. A instrução equivalente em versões anteriores do C# requer um argumento de espaço reservado: `ExcelApp.Workbooks.Add(Type.Missing)`.

C#

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a particular template.
```

```
// Because no argument is sent in this example, Add creates a new
// workbook.
excelApp.Workbooks.Add();

// This example uses a single workSheet. The explicit type casting is
// removed in a later procedure.
Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

Adicione o código a seguir no final de `DisplayInExcel`. O código insere valores nas duas primeiras colunas da primeira linha da planilha.

C#

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

Adicione o código a seguir no final de `DisplayInExcel`. O loop `foreach` coloca as informações da lista de contas nas duas primeiras colunas de sucessivas linhas da planilha.

C#

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

Adicione o seguinte código no final de `DisplayInExcel` para ajustar as larguras das colunas para adequar o conteúdo.

C#

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

As versões anteriores do C# exigem a conversão explícita para essas operações, porque `ExcelApp.Columns[1]` retorna um `Object`, e `AutoFit` é um método `Range` do Excel. As linhas a seguir mostram a conversão.

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();  
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

O C# converte o `Object` retornado em `dynamic` automaticamente, se o assembly for referenciado pela opção do compilador `EmbedInteropTypes` ou, de maneira equivalente, se a propriedade `Inserir Tipos de Interoperabilidade` do Excel for true. True é o valor padrão para essa propriedade.

## Para executar o projeto

Adicione a seguinte linha no final de `Main`.

C#

```
// Display the list in an Excel spreadsheet.  
DisplayInExcel(bankAccounts);
```

Pressione CTRL+F5. Uma planilha do Excel é exibida contendo os dados das duas contas.

## Para adicionar um documento do Word

O código a seguir abre um aplicativo do Word e cria um ícone que é vinculado à planilha do Excel. Cole o método `CreateIconInWordDoc`, fornecido posteriormente nesta etapa, na classe `Program`. O `CreateIconInWordDoc` usa argumentos nomeados e opcionais para reduzir a complexidade das chamadas de método a `Add` e `PasteSpecial`. Essas chamadas incorporam dois recursos que simplificam as chamadas para métodos COM que possuem parâmetros de referência. Primeiro, você pode enviar argumentos para os parâmetros de referência como se fossem parâmetros de valor. Ou seja, você pode enviar valores diretamente, sem criar uma variável para cada parâmetro de referência. O compilador gera variáveis temporárias para conter os valores de argumento e descarta as variáveis quando você retornar da chamada. Em segundo lugar, você pode omitir a palavra-chave `ref` na lista de argumentos.

O método `Add` tem quatro parâmetros de referência que são opcionais. Você poderá omitir argumentos para alguns ou todos os parâmetros se desejar usar os valores padrão.

O método `PasteSpecial` insere o conteúdo da área de transferência. O método tem sete parâmetros de referência que são opcionais. O código a seguir especifica argumentos

para dois deles: `Link`, para criar um link para a origem do conteúdo da área de transferência, e `DisplayAsIcon`, para exibir o link como um ícone. Você pode usar argumentos nomeados para esses dois argumentos e omitir os outros. Embora esses argumentos sejam parâmetros de referência, você não precisa usar a palavra-chave `ref` ou criar variáveis para os enviar como argumentos. Você pode enviar os valores diretamente.

C#

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);
}
```

Adicione a instrução a seguir no final de `Main`.

C#

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

Adicione a instrução a seguir no final de `DisplayInExcel`. O método `Copy` adiciona a planilha na área de transferência.

C#

```
// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

Pressione CTRL+F5. Um documento do Word é exibido contendo um ícone. Clique duas vezes no ícone para colocar a planilha no primeiro plano.

# Para definir a propriedade Inserir Tipos Interop

Outras melhorias são possíveis quando você chama um tipo COM que não requer um assembly de interoperabilidade primário (PIA) no tempo de execução. A remoção da dependência nos PIAs resulta na independência de versão e em uma implantação mais fácil. Para obter mais informações sobre as vantagens da programação sem PIAs, consulte [Passo a passo: inserindo tipos de assemblies gerenciados](#).

Além disso, a programação é mais fácil porque o tipo `dynamic` representa os tipos necessários e retornados declarados em métodos COM. Variáveis com o tipo `dynamic` não são avaliadas até o tempo de execução, o que elimina a necessidade de conversão explícita. Para obter mais informações, veja [Usando o tipo dynamic](#).

A inserção de informações de tipo, em vez do uso de PIAs, é o comportamento padrão. Devido a esse padrão, vários dos exemplos anteriores são simplificados. Você não precisa de nenhuma conversão explícita. Por exemplo, a declaração de `worksheet` em `DisplayInExcel` é escrita como `Excel._Worksheet workSheet = excelApp.ActiveSheet`, em vez de `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. As chamadas para `AutoFit` no mesmo método também exigem conversão explícita sem o padrão, pois `ExcelApp.Columns[1]` retorna um `Object`, e `AutoFit` é um método do Excel. O código a seguir mostra a conversão.

```
C#
```

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

Para alterar o padrão e usar PIAs em vez de inserir informações de tipo, expanda o nó Referências no Gerenciador de Soluções e, em seguida, selecione `Microsoft.Office.Interop.Excel` ou `Microsoft.Office.Interop.Word`. Se você não conseguir ver a janela Propriedades, pressione F4. Localize **Inserir Tipos Interop** na lista de propriedades e altere seu valor para **False**. De maneira equivalente, você pode compilar usando a opção do compilador [Referências](#), em vez de `EmbedInteropTypes` em um prompt de comando.

## Para adicionar formatação adicional à tabela

Substitua as duas chamadas para `AutoFit` em `DisplayInExcel` pela instrução a seguir.

```
C#
```

```
// Call to AutoFormat in Visual C# 2010.  
workSheet.Range["A1", "B3"].AutoFormat(  
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

O método [AutoFormat](#) tem sete parâmetros de valor, que são opcionais. Argumentos nomeados e opcionais permitem que você forneça argumentos para nenhum, alguns ou todos eles. Na instrução anterior, você fornece um argumento para apenas um dos parâmetros, [Format](#). Como [Format](#) é o primeiro parâmetro na lista de parâmetros, você não precisará fornecer o nome do parâmetro. No entanto, poderá ser mais fácil entender a instrução se você incluir o nome do parâmetro, conforme mostrado no código a seguir.

C#

```
// Call to AutoFormat in Visual C# 2010.  
workSheet.Range["A1", "B3"].AutoFormat(Format:  
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

Pressione CTRL + F5 para ver o resultado. Você pode encontrar outros formatos na lista na enumeração [XlRangeAutoFormat](#).

## Exemplo

O código a seguir mostra um exemplo completo.

C#

```
using System.Collections.Generic;  
using Excel = Microsoft.Office.Interop.Excel;  
using Word = Microsoft.Office.Interop.Word;  
  
namespace OfficeProgrammingWalkthruComplete  
{  
    class Walkthrough  
    {  
        static void Main(string[] args)  
        {  
            // Create a list of accounts.  
            var bankAccounts = new List<Account>  
            {  
                new Account {  
                    ID = 345678,  
                    Balance = 541.27  
                },  
                new Account {  
                    ID = 1230221,
```

```

                Balance = -127.44
            }
        };

        // Display the list in an Excel spreadsheet.
        DisplayInExcel(bankAccounts);

        // Create a Word document that contains an icon that links to
        // the spreadsheet.
        CreateIconInWordDoc();
    }

    static void DisplayInExcel(IEnumerable<Account> accounts)
    {
        var excelApp = new Excel.Application();
        // Make the object visible.
        excelApp.Visible = true;

        // Create a new, empty workbook and add it to the collection
        returned
        // by property Workbooks. The new workbook becomes the active
        workbook.
        // Add has an optional parameter for specifying a particular
        template.
        // Because no argument is sent in this example, Add creates a
        new workbook.
        excelApp.Workbooks.Add();

        // This example uses a single workSheet.
        Excel._Worksheet workSheet = excelApp.ActiveSheet;

        // Earlier versions of C# require explicit casting.
        //Excel._Worksheet workSheet =
        (Excel.Worksheet)excelApp.ActiveSheet;

        // Establish column headings in cells A1 and B1.
        workSheet.Cells[1, "A"] = "ID Number";
        workSheet.Cells[1, "B"] = "Current Balance";

        var row = 1;
        foreach (var acct in accounts)
        {
            row++;
            workSheet.Cells[row, "A"] = acct.ID;
            workSheet.Cells[row, "B"] = acct.Balance;
        }

        workSheet.Columns[1].AutoFit();
        workSheet.Columns[2].AutoFit();

        // Call to AutoFormat in Visual C#. This statement replaces the
        // two calls to AutoFit.
        workSheet.Range["A1", "B3"].AutoFormat(
            Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
    }
}

```

```

        // Put the spreadsheet contents on the clipboard. The Copy
method has one
            // optional parameter for specifying a destination. Because no
argument
                // is sent, the destination is the Clipboard.
                workSheet.Range["A1:B3"].Copy();
            }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of which
are
                // optional. Visual C# allows you to omit arguments for them if
                // the default values are what you want.
                wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of which are
            // optional. This example uses named arguments to specify values
            // for two of the parameters. Although these are reference
            // parameters, you do not need to use the ref keyword, or to
create
                // variables to send in as arguments. You can send the values
directly.
                wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
            }
        }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
    }
}

```

## Confira também

- [Type.Missing](#)
- [dinâmico](#)
- [Argumentos nomeados e opcionais](#)
- [Como usar argumentos nomeados e opcionais na programação do Office](#)

# Como usar propriedades indexadas na programação para interoperabilidade COM

Artigo • 09/03/2023

As propriedades indexadas trabalham juntamente com outras funcionalidades no C#, como [argumentos nomeados e opcionais](#), um novo tipo ([dinâmico](#)) e [informações de tipo inseridas](#) para melhorar a programação do Microsoft Office.

## Importante

O VSTO se baseia no [.NET Framework](#). Os suplementos COM também podem ser gravados com o .NET Framework. Os suplementos do Office não podem ser criados com o [.NET Core](#) e o [.NET 5+](#), as versões mais recentes do .NET. Isso ocorre porque o .NET Core/.NET 5+ não pode trabalhar em conjunto com o .NET Framework no mesmo processo e pode levar a falhas de carga de suplemento. Você pode continuar a usar o .NET Framework para escrever suplementos VSTO e COM para o Office. A Microsoft não atualizará o VSTO ou a plataforma de suplemento COM para usar o .NET Core ou o .NET 5+. Você pode aproveitar o .NET Core e o .NET 5+, incluindo o ASP.NET Core, para criar o lado do servidor dos [Suplementos Web do Office](#).

Nas versões anteriores do C#, os métodos são acessíveis como propriedades apenas se o método `get` não tem parâmetros e o método `set` tem apenas um parâmetro de valor. No entanto, nem todas as propriedades COM atendem a essas restrições. Por exemplo, a propriedade [Range\[\]](#) do Excel tem um acessador `get` que requer um parâmetro para o nome do intervalo. No passado, como não era possível acessar a propriedade `Range` diretamente, era necessário usar o método `get_Range` em vez disso, conforme mostrado no exemplo a seguir.

```
{language}
```

```
// Visual C# 2008 and earlier.  
var excelApp = new Excel.Application();  
// . . .  
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Em vez disso, as propriedades indexadas permitem escrever o seguinte:

```
{language}
```

```
// Visual C# 2010.  
var excelApp = new Excel.Application();  
// . . .  
Excel.Range targetRange = excelApp.Range["A1"];
```

O exemplo anterior também usa o recurso [argumentos opcionais](#), que permite que você omita `Type.Missing`.

As propriedades indexadas permitem escrever o código a seguir.

```
{language}
```

```
// Visual C# 2010.  
targetRange.Value = "Name";
```

Não é possível criar propriedades indexadas de sua preferência. O recurso dá suporte apenas ao consumo de propriedades indexadas existentes.

## Exemplo

O código a seguir mostra um exemplo completo. Para obter mais informações sobre como configurar um projeto que acessa a API do Office, consulte [Como acessar objetos de interoperabilidade do Office usando recursos do Visual C#](#).

```
{language}
```

```
// You must add a reference to Microsoft.Office.Interop.Excel to run  
// this example.  
using System;  
using Excel = Microsoft.Office.Interop.Excel;  
  
namespace IndexedProperties  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            CSharp2010();  
        }  
  
        static void CSharp2010()  
        {  
            var excelApp = new Excel.Application();  
            excelApp.Workbooks.Add();  
            excelApp.Visible = true;
```

```
        Excel.Range targetRange = excelApp.Range["A1"];
        targetRange.Value = "Name";
    }

    static void CSharp2008()
    {
        var excelApp = new Excel.Application();
        excelApp.Workbooks.Add(Type.Missing);
        excelApp.Visible = true;

        Excel.Range targetRange = excelApp.get_Range("A1",
Type.Missing);
        targetRange.set_Value(Type.Missing, "Name");
        // Or
        //targetRange.Value2 = "Name";
    }
}
```

## Confira também

- [Argumentos nomeados e opcionais](#)
- [dinâmico](#)
- [Usando o tipo dynamic](#)

# Como usar invocação de plataforma para executar um arquivo WAV

Artigo • 07/04/2023

O exemplo de código C# a seguir ilustra como usar os serviços de invocação de plataforma para reproduzir um arquivo de som WAV no sistema operacional Windows.

## Exemplo

Esse código de exemplo usa [DllImportAttribute](#) para importar o ponto de entrada de método `PlaySound` da `winmm.dll` como `Form1 PlaySound()`. O exemplo tem um Windows Form simples com um botão. Ao clicar no botão, abre-se uma caixa de diálogo padrão [OpenFileDialog](#) do Windows para que você possa abrir o arquivo para reprodução. Quando um arquivo wave é selecionado, ele é executado usando o método `PlaySound()` da biblioteca `winmm.dll`. Para obter mais informações sobre esse método, consulte [Usando a função PlaySound com arquivos de áudio Waveform](#). Procure e selecione um arquivo que tenha uma extensão .wav e, em seguida, selecione **Abrir** para reproduzir o arquivo wave usando a invocação de plataforma. Uma caixa de texto exibe o caminho completo do arquivo selecionado.

C#

```
using System.Runtime.InteropServices;

namespace WinSound;

public partial class Form1 : Form
{
    private TextBox textBox1;
    private Button button1;

    public Form1() // Constructor.
    {
        InitializeComponent();
    }

    [DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true,
    CharSet = CharSet.Unicode, ThrowOnUnmappableChar = true)]
    private static extern bool PlaySound(string szSound, System.IntPtr hMod,
    PlaySoundFlags flags);

    [System.Flags]
    public enum PlaySoundFlags : int
    {
        SND_SYNC = 0x0000,
```

```

        SND_ASYNC = 0x0001,
        SND_NODEFAULT = 0x0002,
        SND_LOOP = 0x0008,
        SND_NOSTOP = 0x0010,
        SND_NOWAIT = 0x00002000,
        SND_FILENAME = 0x00020000,
        SND_RESOURCE = 0x00040004
    }

    private void button1_Click(object sender, System.EventArgs e)
    {
        var dialog1 = new OpenFileDialog();

        dialog1.Title = "Browse to find sound file to play";
        dialog1.InitialDirectory = @"c:\";
        //<Snippet5>
        dialog1.Filter = "Wav Files (*.wav)|*.wav";
        //</Snippet5>
        dialog1.FilterIndex = 2;
        dialog1.RestoreDirectory = true;

        if (dialog1.ShowDialog() == DialogResult.OK)
        {
            textBox1.Text = dialog1.FileName;
            PlaySound(dialog1.FileName, new System.IntPtr(),
PlaySoundFlags.SND_SYNC);
        }
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        // Including this empty method in the sample because in the IDE,
        // when users click on the form, generates code that looks for a
default method
        // with this name. We add it here to prevent confusion for those
using the samples.
    }
}

```

A caixa de diálogo **Abrir Arquivos** é filtrada por meio das configurações de filtro para mostrar somente os arquivos que têm uma extensão .wav.

## Compilando o código

Crie um novo projeto de aplicativos do Windows Forms em C# no Visual Studio e nomeie-o como **WinSound**. Copie o código anterior e cole-o sobre o conteúdo do arquivo *Form1.cs*. Copie o código a seguir e cole-o no arquivo *Form1.Designer.cs*, no método `InitializeComponent()`, após qualquer código existente.

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "File path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

Compile e execute o código.

## Confira também

- Um olhar detalhado sobre invocação de plataforma
- Marshaling de dados com invocação de plataforma

# Passo a passo: programação do Office em C#

Artigo • 09/03/2023

O C# oferece recursos que melhoram a programação do Microsoft Office. As funcionalidades úteis do C# incluem argumentos nomeados e opcionais e valores retornados do tipo `dynamic`. Na programação COM, você pode omitir a palavra-chave `ref` e obter acesso a propriedades indexadas.

Ambas as linguagens permitem incorporar as informações de tipo, que permitem a implantação de assemblies que interagem com componentes COM sem implantar assemblies de interoperabilidade primários (PIAs) no computador do usuário. Para obter mais informações, consulte [Instruções passo a passo: Inserindo tipos de assemblies gerenciados](#).

Este passo a passo demonstra essas funcionalidades no contexto de programação do Office, mas muitos deles também são úteis na programação em geral. No passo a passo, você usa um aplicativo Suplemento do Excel para criar uma pasta de trabalho do Excel. Em seguida, você cria um documento do Word que contém um link para a pasta de trabalho. Por fim, você vê como habilitar e desabilitar a dependência de PIA.

## Importante

O VSTO se baseia no [.NET Framework](#). Os suplementos COM também podem ser gravados com o .NET Framework. Os suplementos do Office não podem ser criados com o [.NET Core](#) e o [.NET 5+](#), as versões mais recentes do .NET. Isso ocorre porque o .NET Core/.NET 5+ não pode trabalhar em conjunto com o .NET Framework no mesmo processo e pode levar a falhas de carga de suplemento. Você pode continuar a usar o .NET Framework para escrever suplementos VSTO e COM para o Office. A Microsoft não atualizará o VSTO ou a plataforma de suplemento COM para usar o .NET Core ou o .NET 5+. Você pode aproveitar o .NET Core e o .NET 5+, incluindo o ASP.NET Core, para criar o lado do servidor dos [Suplementos Web do Office](#).

## Pré-requisitos

Você deve ter o Microsoft Office Excel e o Microsoft Office Word instalados no computador para concluir esse passo a passo.

### (!) Observação

Seu computador pode mostrar diferentes nomes ou locais para alguns dos elementos de interface do usuário do Visual Studio nas instruções a seguir. A edição do Visual Studio que você possui e as configurações que você usa determinam esses elementos. Para obter mais informações, consulte [Personalizando o IDE](#).

## Configurar um aplicativo Suplemento do Excel

1. Inicie o Visual Studio.
2. No menu **Arquivo**, aponte para **Novoe selecione Projeto**.
3. No painel **Modelos Instalados**, expanda **C#**, expanda **Office** e, em seguida, selecione o ano da versão do produto do Office.
4. No painel **Modelos**, selecione **Suplemento do <Excel> versão**.
5. Observe a parte superior do painel **Modelos** para se certificar de que **.NET Framework 4** ou uma versão posterior, é exibido na caixa **Estrutura de Destino**.
6. Digite um nome para seu projeto na caixa **Nome**, se desejar.
7. Selecione **OK**.
8. O novo projeto aparece no **Gerenciador de Soluções**.

## Adicionar referências

1. No **Gerenciador de Soluções**, clique com o botão direito do mouse no nome do projeto e, em seguida, selecione **Adicionar Referência**. A caixa de diálogo **Adicionar Referência** é exibida.
2. Na guia **Assemblies**, selecione **Microsoft.Office.Interop.Excel**, versão `<version>.0.0.0` (para uma chave para os números de versão de produto do Office, consulte [Versões da Microsoft](#)), na lista **Nome do componente** e mantenha a tecla CTRL pressionada e selecione **Microsoft.Office.Interop.Word**, `version <version>.0.0.0`. Se você não vir os assemblies, talvez seja necessário verificar se eles estão instalados (confira [Como: Instalar assemblies de interoperabilidade primária do Office](#)).
3. Selecione **OK**.

## Adicionar instruções Imports necessárias ou diretivas de uso

No Gerenciador de Soluções, clique com o botão direito do mouse no arquivo `ThisAddIn.cs` e, em seguida, selecione **Exibir Código**. Adicione as seguintes diretivas `using` (C#) na parte superior do arquivo de código se elas ainda não estiverem presentes.

```
C#
```

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

## Criar uma lista de contas bancárias

No Gerenciador de Soluções, clique com o botão direito do mouse no nome do projeto e, em seguida, selecione **Adicionar e Classe**. Nomeie a classe `Account.cs`. Selecione **Adicionar**. Substitua a definição da classe `Account` pelo código a seguir. As definições de classe usam *propriedades autoimplementadas*.

```
C#
```

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

Para criar uma lista `bankAccounts` que contém duas contas, adicione o seguinte código ao método `ThisAddIn_Startup` em `ThisAddIn.cs`. As declarações de lista usam *inicializadores de coleção*.

```
C#
```

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

# Exportar dados para o Excel

No mesmo arquivo, adicione o método a seguir para a classe `ThisAddIn`. O método configura uma planilha do Excel e exporta dados para ela.

```
C#  
  
void DisplayInExcel(IEnumerable<Account> accounts,  
                     Action<Account, Excel.Range> DisplayFunc)  
{  
    var excelApp = this.Application;  
    // Add a new Excel workbook.  
    excelApp.Workbooks.Add();  
    excelApp.Visible = true;  
    excelApp.Range["A1"].Value = "ID";  
    excelApp.Range["B1"].Value = "Balance";  
    excelApp.Range["A2"].Select();  
  
    foreach (var ac in accounts)  
    {  
        DisplayFunc(ac, excelApp.ActiveCell);  
        excelApp.ActiveCell.Offset[1, 0].Select();  
    }  
    // Copy the results to the Clipboard.  
    excelApp.Range["A1:B3"].Copy();  
}
```

- O método `Add` tem um *parâmetro opcional* para especificar um modelo específico. Os parâmetros opcionais permitem omitir o argumento para esse parâmetro, se você deseja usar o valor padrão do parâmetro. Como o exemplo anterior não tem argumentos, `Add` usa o modelo padrão e cria uma pasta de trabalho. A instrução equivalente em versões anteriores do C# requer um argumento de espaço reservado: `excelApp.Workbooks.Add(Type.Missing)`. Para obter mais informações, consulte [Argumentos nomeados e opcionais](#).
- As propriedades `Range` e `Offset` do objeto `Range` usam o recurso de *propriedades indexadas*. Este recurso permite consumir essas propriedades de tipos COM usando a sintaxe típica do C# a seguir. Propriedades indexadas também permitem que você use a propriedade `value` do objeto `Range`, eliminando a necessidade de usar a propriedade `value2`. A propriedade `value` é indexada, mas o índice é opcional. Argumentos opcionais e propriedades indexadas trabalham juntos no exemplo a seguir.

```
C#  
  
// Visual C# 2010 provides indexed properties for COM programming.  
excelApp.Range["A1"].Value = "ID";
```

```
excelApp.ActiveCell.Offset[1, 0].Select();
```

Não é possível criar propriedades indexadas de sua preferência. O recurso dá suporte apenas ao consumo de propriedades indexadas existentes.

Adicione o seguinte código no final de `DisplayInExcel` para ajustar as larguras das colunas para adequar o conteúdo.

C#

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

Essas adições demonstram outro recurso no C#: tratar valores `Object` retornados de hosts COM como o Office, como se eles tivessem o tipo [dinâmico](#). Os objetos COM são tratados como `dynamic` automaticamente quando os [Inserir Tipos de Interoperabilidade](#) têm seu valor padrão, `True`, ou, de modo equivalente, quando você faz referência ao assembly com a opção do compilador [EmbedInteropTypes](#). Para obter mais informações sobre como inserir tipos de interoperabilidade, consulte os procedimentos "Localizar a referência de PIA" e "Restaurar a dependência de PIA" posteriormente neste artigo. Para obter mais informações sobre `dynamic`, consulte [dynamic](#) ou [Usando o tipo dynamic](#).

## Invocar `DisplayInExcel`

Adicione o código a seguir no final do método `ThisAddIn_StartUp`. A chamada para `DisplayInExcel` contém dois argumentos. O primeiro argumento é o nome da lista de contas a ser processada. O segundo argumento é uma expressão lambda de várias linhas que define como processar os dados. Os valores `ID` e `balance` de cada conta serão exibidos em células adjacentes e a linha será exibida em vermelho se o equilíbrio for menor do que zero. Para obter mais informações, consulte [Expressões Lambda](#).

C#

```
DisplayInExcel(bankAccounts, (account, cell) =>
    // This multiline lambda expression sets custom processing rules
    // for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
```

```
});
```

Para executar o programa, pressione F5. Uma planilha do Excel é exibida contendo os dados das contas.

## Adicionar um documento do Word

Adicione o código a seguir ao final do método `ThisAddIn_StartUp` para criar um documento do Word que contém um link para a pasta de trabalho do Excel.

C#

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

Esse código demonstra vários dos recursos em C#: a capacidade de omitir a palavra-chave `ref` na programação COM, argumentos nomeados e argumentos opcionais. O método `PasteSpecial` tem sete parâmetros, todos os quais são parâmetros de referência opcionais. Argumentos opcionais e nomeados permitem designar os parâmetros que você deseja acessar pelo nome e enviar argumentos apenas para esses parâmetros. Neste exemplo, os argumentos indicam que deve ser criado um link para a pasta de trabalho na área de transferência (parâmetro `Link`), e que o link será exibido no documento do Word como um ícone (parâmetro `DisplayAsIcon`). O C# permite também que você omita a palavra-chave `ref` nesses argumentos.

## Executar o aplicativo

Pressione F5 para executar o aplicativo. O Excel é iniciado e exibe uma tabela que contém as informações das duas contas em `bankAccounts`. Em seguida, um documento do Word é exibido contendo um link para a tabela do Excel.

## Limpar o projeto concluído

No Visual Studio, clique em **Limpar Solução** no menu **Compilar**. Caso contrário, o suplemento será executado toda vez que você abrir o Excel no seu computador.

## Localizar a referência de PIA

1. Execute o aplicativo novamente, mas não selecione **Limpar Solução**.
2. Selecione **Iniciar**. Localize **Microsoft Visual Studio <versão>** e abra o prompt de comando do desenvolvedor.
3. Digite `ildasm` na janela Prompt de Comando do Desenvolvedor para Visual Studio e pressione Enter. A janela IL DASM é exibida.
4. No menu **Arquivo** na janela IL DASM, selecione **Arquivo>Abrir**. Clique duas vezes em **Visual Studio <versão>** e clique duas vezes em **Projetos**. Abra a pasta do seu projeto e procure na pasta bin/Debug por *nome do projeto.dll*. Clique duas vezes em *nome do projeto.dll*. Uma nova janela exibe os atributos do projeto, além das referências a outros módulos e assemblies. O assembly inclui os namespaces `Microsoft.Office.Interop.Excel` e `Microsoft.Office.Interop.Word`. Por padrão, no Visual Studio, o compilador importa os tipos que você precisa de um PIA referenciado para o seu assembly. Para obter mais informações, consulte [Como exibir o conteúdo de um assembly](#).
5. Clique duas vezes no ícone **MANIFEST**. Uma janela será exibida contendo uma lista de assemblies que contêm itens referenciados pelo projeto.  
`Microsoft.Office.Interop.Excel` e `Microsoft.Office.Interop.Word` não estão na lista. Como você importou os tipos de que seu projeto precisa para o assembly, não é necessário instalar referências a um PIA. Importar os tipos no assembly facilita a implantação. Os PIAs não precisam estar presentes no computador do usuário. Um aplicativo não requer a implantação de uma versão específica de um PIA. Os aplicativos podem funcionar com várias versões do Office, desde que as APIs necessárias existam em todas as versões. Como a implantação de PIAs não é mais necessária, você pode criar um aplicativo em cenários avançados que funcione com várias versões do Office, incluindo versões anteriores. Seu código não pode usar nenhuma API que não esteja disponível na versão do Office com a qual você está trabalhando. Nem sempre fica claro se uma API específica estava disponível em uma versão anterior. Não é recomendável trabalhar com versões anteriores do Office.
6. Feche a janela do manifesto e a janela do assembly.

## Restaurar a dependência de PIA

1. No **Gerenciador de Soluções**, selecione o botão **Mostrar Todos os Arquivos**. Expanda a pasta **Referências** e selecione **Microsoft.Office.Interop.Excel**. Pressione F4 para exibir a janela **Propriedades**.
2. Na janela **Propriedades**, altere a propriedade **Inserir Tipos de Interoperabilidade** de **True** para **False**.
3. Repita as etapas 1 e 2 deste procedimento para `Microsoft.Office.Interop.Word`.

4. Em C#, comente as duas chamadas para `Autofit` no final do método `DisplayInExcel`.
5. Pressione F5 para verificar se o projeto ainda é executado corretamente.
6. Repita as etapas de 1 a 3 do procedimento anterior para abrir a janela do assembly. Observe que `Microsoft.Office.Interop.Word` e `Microsoft.Office.Interop.Excel` não estão mais na lista de assemblies inseridos.
7. Clique duas vezes no ícone **MANIFEST** e role a lista de assemblies referenciados. Ambos `Microsoft.Office.Interop.Word` e `Microsoft.Office.Interop.Excel` estão na lista. Como o aplicativo faz referência aos PIAs do Excel e do Word e a propriedade **Inserir Tipos de Interoperabilidade** está definida como **False**, ambos os assemblies devem existir no computador do usuário final.
8. No Visual Studio, selecione **Limpar Solução** no menu **Compilar** para limpar o projeto concluído.

## Confira também

- [Propriedades autoimplementadas \(C#\)](#)
- [Inicializadores de objeto e coleção](#)
- [Argumentos nomeados e opcionais](#)
- [dinâmico](#)
- [Usando o tipo dynamic](#)
- [Expressões lambda \[C#\]](#)
- [Passo a passo: inserindo informações de tipo dos Microsoft Office Assemblies no Visual Studio](#)
- [Instruções passo a passo: Inserindo tipos de assemblies gerenciados](#)
- [Passo a passo: criando o primeiro suplemento do VSTO para Excel](#)

# Exemplo de classe COM

Artigo • 10/05/2023

O código a seguir é um exemplo de uma classe que você poderia expor como um objeto COM. Depois de posicionar o código em um arquivo .cs adicionado a seu projeto, defina a propriedade **Registrar para interoperabilidade COM** como **True**. Para obter mais informações, consulte [Como registrar um componente para interoperabilidade COM](#).

A exposição de objetos do C# para COM requer a declaração de uma interface de classe, de uma “interface de eventos” se necessário e da própria classe. Os membros de classe devem seguir estas regras para ficarem visíveis ao COM:

- A classe deve ser pública.
- As propriedades, os métodos e os eventos devem ser públicos.
- As propriedades e os métodos devem ser declarados na interface de classe.
- Os eventos devem ser declarados na interface de eventos.

Outros membros públicos da classe que não são declarados nessas interfaces não estarão visíveis para COM, mas eles ficarão visíveis para outros objetos .NET. Para expor propriedades e métodos ao COM, você deve declará-los na interface de classe e marcá-los com um atributo `DispId` e implementá-los na classe. A ordem de declaração dos membros na interface é a ordem usada para a vtable do COM. Para expor eventos de sua classe, você deve declará-los na interface de eventos e marcá-los com um atributo `DispId`. A classe não deve implementar essa interface.

A classe implementa a interface de classe. Ela pode implementar mais de uma interface, mas a primeira implementação é a interface de classe padrão. Implemente os métodos e propriedades expostos ao COM aqui. Ela deve ser marcada como pública e deve corresponder às declarações na interface de classe. Além disso, declare aqui os eventos acionados pela classe. Ela deve ser pública e deve corresponder às declarações na interface de eventos.

## Exemplo

C#

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
}
```

```
public interface ComClass1Interface
{
}

[Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
 InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
public interface ComClass1Events
{
}

[Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
 ClassInterface(ClassInterfaceType.None),
 ComSourceInterfaces(typeof(ComClass1Events))]
public class ComClass1 : ComClass1Interface
{
}
}
```

## Confira também

- Interoperabilidade
- Página de Build, Designer de Projeto (C#)

# Referência de C#

Artigo • 15/02/2023

Esta seção fornece o material de referência sobre palavras-chave do C#, operadores, caracteres especiais, diretivas de pré-processador, opções de compilador e erros de compilador e avisos.

## Nesta seção

### [Palavras-chave do C#](#)

Fornece links para informações sobre a sintaxe e as palavras-chave do C#.

### [Operadores do C#](#)

Fornece links para informações sobre a sintaxe e os operadores do C#.

### [Caracteres especiais de C#](#)

Fornece links para informações sobre caracteres especiais contextuais em C# e seu uso.

### [Diretivas do pré-processador do C#](#)

Fornece links para informações sobre os comandos do compilador para inserir no código-fonte do C#.

### [Opções do compilador de C#](#)

Inclui informações sobre as opções do compilador e como usá-las.

### [Erros do Compilador do C#](#)

Inclui snippets de código que demonstram a causa e a correção de erros do compilador do C# e avisos.

### [C# Language Specification \(Especificação da linguagem C#\)](#)

Especificação de linguagem do C# 6.0. Este é um projeto de proposta da linguagem C# 6.0. Este documento será refinado por meio do trabalho com o comitê de padrões ECMA C#. A versão 5.0 foi lançada em dezembro de 2017 como o documento [Padrão ECMA-334 – 5<sup>a</sup> Edição](#).

Os recursos que foram implementados nas versões do C# depois da 6.0 são representados em propostas de especificação de linguagem. Esses documentos descrevem os deltas para a especificação da linguagem a fim de adicionar os novos recursos. Eles estão em formato de rascunho de proposta. Essas especificações serão refinadas e enviadas ao comitê de padrões ECMA para revisão formal e incorporação em uma versão futura do Padrão C#.

## [Propostas de especificação do C# 7.0](#)

Implementamos diversos recursos novos no C# 7.0. Entre eles estão correspondência de padrões, funções locais, declarações de variável out, expressões throw, literais binários e separadores de dígito. Esta pasta contém as especificações de cada um desses recursos.

## [Propostas de especificação do C# 7.1](#)

Há novos recursos adicionados no C# 7.1. Primeiramente, você pode gravar um método `Main` que retorna `Task` ou `Task<int>`. Isso permite que você adicione o modificador `async` ao `Main`. A expressão `default` pode ser usada sem um tipo em locais onde o tipo pode ser inferido. Além disso, os nomes dos membros de tupla podem ser inferidos. Por fim, a correspondência de padrões pode ser usada com genéricos.

## [Propostas de especificação do C# 7.2](#)

O C# 7.2 adicionou a uma série de recursos pequenos. Você pode passar argumentos por referência de somente leitura usando a palavra-chave `in`. Há uma série de alterações de nível inferior para dar suporte à segurança de tempo de compilação para `Span` e tipos relacionados. Você pode usar argumentos nomeados nos quais os argumentos posteriores são posicionais, em algumas situações. O modificador de acesso `private protected` permite que você especifique que os chamadores são limitados aos tipos derivados, implementados no mesmo assembly. O operador `?:` pode resolver em uma referência a uma variável. Você também pode formatar números hexadecimais e binários usando um separador de dígito à esquerda.

## [Propostas de especificação do C# 7.3](#)

C# 7.3 é outra versão de ponto que inclui várias atualizações pequenas. Você pode usar novas restrições em parâmetros de tipo genérico. Outras alterações facilitam trabalhar com campos `fixed`, incluindo o uso de alocações `stackalloc`. As variáveis locais declaradas com a palavra-chave `ref` podem ser reatribuídas para se referir ao novo armazenamento. Você pode colocar os atributos em propriedades autoimplementadas que direcionam o campo de suporte gerado pelo compilador. As variáveis de expressão podem ser usadas em inicializadores. As tuplas podem ser comparadas quanto à igualdade (ou desigualdade). Também houve algumas melhorias para a resolução de sobrecarga.

## [Propostas de especificação do C# 8.0](#)

O C# 8.0 está disponível com o .NET Core 3.0. Os recursos incluem tipos de referência que permitem valor nulo, correspondência de padrões recursiva, métodos de interface padrão, fluxos assíncronos, intervalos e índices, uso com base em padrão e declarações de uso, atribuição de avaliação de nulo e membros de instância somente leitura.

## [Propostas de especificação do C# 9](#)

O C# 9 está disponível com o .NET 5. Os recursos incluem registros, instruções de nível

superior, aprimoramentos de correspondência de padrões, setters somente init, novas expressões de tipo de destino, inicializadores de módulo, extensão de métodos parciais, funções anônimas estáticas, expressões condicionais com tipo de destino, tipos de retorno covariantes, extensão GetEnumerator em loops foreach, parâmetros de descarte lambda, atributos em funções locais, inteiros de tamanho nativo, ponteiros de função, suprimir a emissão de sinalizador localsinit e anotações de parâmetro de tipo sem restrição.

### [Propostas de especificação do C# 10](#)

O C# 10 está disponível com o .NET Core 6. Os recursos incluem structs de registro, construtores de struct sem parâmetros, diretivas de uso global, namespaces com escopo de arquivo, padrões de propriedade estendidos, cadeias de caracteres interpoladas aprimoradas, cadeias de caracteres interpoladas constantes, melhorias de lambda, expressão de argumento de chamador, diretivas `#line` aprimoradas, atributos genéricos, análise de atribuição definitiva aprimorada e substituição

`AsyncMethodBuilder`.

## Seções relacionadas

### [Usando o ambiente de desenvolvimento do Visual Studio para C#](#)

Fornece links para tópicos conceituais e de tarefas que descrevem o IDE e o Editor.

### [Guia de Programação em C#](#)

Inclui informações sobre como usar a linguagem de programação do C#.

# Controle de versão da linguagem C#

Artigo • 10/05/2023

O compilador do C# mais recente determina uma versão da linguagem padrão com base nas estruturas de destino do projeto. O Visual Studio não fornece uma interface do usuário para alterar o valor, mas você pode alterá-lo editando o arquivo `csproj`. A escolha do padrão garante que você use a versão de idioma mais recente compatível com sua estrutura de destino. Você se beneficia do acesso aos recursos de linguagem mais recentes compatíveis com o destino do projeto. Essa opção padrão também garante que você não use uma linguagem que exija tipos ou comportamento de runtime não disponível em sua estrutura de destino. Escolher uma versão de idioma mais recente do que o padrão pode causar erros de tempo de compilação e de runtime difíceis de diagnosticar.

O [C# 11](#) tem suporte apenas no .NET 7 e em versões mais recentes. O [C# 10](#) tem suporte apenas no .NET 6 e em versões mais recentes. O [C# 9](#) tem suporte apenas no .NET 5 e em versões mais recentes.

Verifique a página de [compatibilidade da plataforma do Visual Studio](#) para obter detalhes sobre quais versões do .NET têm suporte por versões do Visual Studio. Verifique a página de [compatibilidade da plataforma do Visual Studio para Mac](#) para obter detalhes sobre quais versões do .NET têm suporte por versões do Visual Studio para Mac. Verifique a [página do Mono para o C# ↗](#) a respeito da compatibilidade do Mono com versões C#.

## Padrões

O compilador determina um padrão com base nestas regras:

Destino	Versão	Padrão da versão da linguagem C#
.NET	7.x	C# 11
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0

Destino	Versão	Padrão da versão da linguagem C#
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	all	C# 7.3

Quando seu projeto se destina a uma estrutura de visualização que tem uma versão da linguagem correspondente da visualização, a versão de linguagem usada é a de visualização. Você usa os recursos mais recentes com essa versão prévia em qualquer ambiente, sem afetar projetos destinados a uma versão lançada do .NET Core.

### ⓘ Importante

O novo modelo de projeto do Visual Studio 2017 adicionou uma entrada `<LangVersion>latest</LangVersion>` a novos arquivos de projeto. Se você atualizar a estrutura de destino para esses projetos, eles **substituirão o comportamento padrão**. Você deve remover o `<LangVersion>latest</LangVersion>` do seu arquivo de projeto ao atualizar o SDK do .NET. Em seguida, seu projeto usará a versão do compilador recomendada para sua estrutura de destino. Você pode atualizar a estrutura de destino para acessar recursos de linguagem mais recentes.

## Substituir um padrão

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

- Edite manualmente o [arquivo de projeto](#).
- Definir a versão da linguagem para vários projetos em um subdiretório.
- Configurar a [opção do compilador LangVersion](#).

### 💡 Dica

Você pode ver a versão do idioma no Visual Studio na página de propriedades do projeto. Na guia *Compilar*, o painel *Avançado* exibe a versão selecionada.

Para saber qual versão da linguagem você está usando no momento, coloque `#error version` (diferencia maiúsculas de minúsculas) em seu código. Isso torna o relatório do compilador um erro do compilador, CS8304, com uma mensagem contendo a versão do compilador em uso e a versão atual da linguagem selecionada. Confira [#erro \(Referência C#\)](#) para obter mais informações.

## 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:

XML

```
<PropertyGroup>
    <LangVersion>preview</LangVersion>
</PropertyGroup>
```

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

## Configurar vários projetos

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

XML

```
<Project>
    <PropertyGroup>
        <LangVersion>preview</LangVersion>
    </PropertyGroup>
</Project>
```

Agora, os builds de todos os subdiretórios do diretório que contém esse arquivo usarão a versão prévia do C#. Para obter mais informações, confira [Personalizar seu build](#).

## Referência à versão da linguagem C#

A tabela a seguir mostra todas as versões atuais da linguagem C#. Seu compilador talvez não entenda necessariamente todo valor se for mais antigo. Se você instalar o SDK mais recente do .NET, terá acesso a tudo o que for listado.

Valor	Significado
-------	-------------

<b>Valor</b>	<b>Significado</b>
<code>preview</code>	O compilador aceita todas as sintaxes de linguagem válidas da versão prévia mais recente.
<code>latest</code>	O compilador aceita a sintaxe da versão lançada mais recente do compilador (incluindo a versão secundária).
<code>latestMajor</code> ou <code>default</code>	O compilador aceita a sintaxe da versão principal mais recente lançada do compilador.
<code>11.0</code>	O compilador aceita somente a sintaxe incluída no C# 11 ou inferior.
<code>10.0</code>	O compilador aceita somente a sintaxe incluída no C# 10 ou inferior.
<code>9.0</code>	O compilador aceita somente a sintaxe incluída no C# 9 ou inferior.
<code>8.0</code>	O compilador aceita somente a sintaxe incluída no C# 8.0 ou inferior.
<code>7.3</code>	O compilador aceita somente a sintaxe incluída no C# 7.3 ou inferior.
<code>7.2</code>	O compilador aceita somente a sintaxe incluída no C# 7.2 ou inferior.
<code>7.1</code>	O compilador aceita somente a sintaxe incluída no C# 7.1 ou inferior.
<code>7</code>	O compilador aceita somente a sintaxe incluída no C# 7.0 ou inferior.
<code>6</code>	O compilador aceita somente a sintaxe incluída no C# 6.0 ou inferior.
<code>5</code>	O compilador aceita somente a sintaxe incluída no C# 5.0 ou inferior.
<code>4</code>	O compilador aceita somente a sintaxe incluída no C# 4.0 ou inferior.
<code>3</code>	O compilador aceita somente a sintaxe incluída no C# 3.0 ou inferior.
<code>ISO-2</code> ou <code>2</code>	O compilador aceita somente a sintaxe incluída no ISO/IEC 23270:2006 C# (2.0).
<code>ISO-1</code> ou <code>1</code>	O compilador aceita somente a sintaxe incluída no ISO/IEC 23270:2003 C# (1.0/1.2).

# Tipos de valor – (referência de C#)

Artigo • 07/04/2023

*Tipos de valor* e [tipos de referência](#) são as duas categorias principais de tipos C#. Uma variável de um tipo de valor contém uma instância do tipo. Isso é diferente de uma variável de um tipo de referência, que contém uma referência a uma instância do tipo. Por padrão, na [atribuição](#), ao passar um argumento para um método e retornar um resultado de método os valores de variável são copiados. No caso de variáveis de tipo de valor, as instâncias de tipo correspondentes são copiadas. O exemplo a seguir demonstra esse comportamento:

```
C#  
  
using System;  
  
public struct MutablePoint  
{  
    public int X;  
    public int Y;  
  
    public MutablePoint(int x, int y) => (X, Y) = (x, y);  
  
    public override string ToString() => $"({X}, {Y})";  
}  
  
public class Program  
{  
    public static void Main()  
    {  
        var p1 = new MutablePoint(1, 2);  
        var p2 = p1;  
        p2.Y = 200;  
        Console.WriteLine($"{nameof(p1)} after {nameof(p2)} is modified:  
{p1}");  
        Console.WriteLine($"{nameof(p2)}: {p2}");  
  
        MutateAndDisplay(p2);  
        Console.WriteLine($"{nameof(p2)} after passing to a method: {p2}");  
    }  
  
    private static void MutateAndDisplay(MutablePoint p)  
    {  
        p.X = 100;  
        Console.WriteLine($"Point mutated in a method: {p}");  
    }  
}  
// Expected output:  
// p1 after p2 is modified: (1, 2)  
// p2: (1, 200)
```

```
// Point mutated in a method: (100, 200)
// p2 after passing to a method: (1, 200)
```

Como mostra o exemplo anterior, as operações em uma variável de tipo de valor afetam apenas essa instância do tipo de valor armazenada na variável.

Se um tipo de valor contiver um membro de dados de um tipo de referência, somente a referência à instância do tipo de referência será copiada quando uma instância de tipo de valor for copiada. A instância de cópia e de tipo de valor original tem acesso à mesma instância de tipo de referência. O exemplo a seguir demonstra esse comportamento:

C#

```
using System;
using System.Collections.Generic;

public struct TaggedInteger
{
    public int Number;
    private List<string> tags;

    public TaggedInteger(int n)
    {
        Number = n;
        tags = new List<string>();
    }

    public void AddTag(string tag) => tags.Add(tag);

    public override string ToString() => $"{Number} [{string.Join(", ", tags)}]";
}

public class Program
{
    public static void Main()
    {
        var n1 = new TaggedInteger(0);
        n1.AddTag("A");
        Console.WriteLine(n1); // output: 0 [A]

        var n2 = n1;
        n2.Number = 7;
        n2.AddTag("B");

        Console.WriteLine(n1); // output: 0 [A, B]
        Console.WriteLine(n2); // output: 7 [A, B]
    }
}
```

## ⓘ Observação

Para tornar seu código menos propenso a erros e mais robusto, defina e use tipos de valor imutáveis. Este artigo usa tipos de valor mutáveis somente para fins de demonstração.

# Modalidades de tipos de valor e restrições de tipo

Um tipo de valor pode ser de uma das duas seguintes modalidades:

- um [tipo de estrutura](#), que encapsula dados e funcionalidade relacionada
- um [tipo de enumeração](#), que é definido por um conjunto de constantes nomeadas e representa uma escolha ou uma combinação de opções

Um [tipo de valor anulável](#) `T?` representa todos os valores de seu tipo de valor `T` subjacente e um valor [nulo](#) adicional. Você não pode atribuir `null` a uma variável de um tipo de valor, a menos que seja um tipo de valor anulável.

Você pode usar a [struct restrição](#) para especificar que um parâmetro de tipo é um tipo de valor não anulável. Os tipos de estrutura e enumeração atendem à restrição `struct`.

Você pode usar `System.Enum` em uma restrição de classe base (que é conhecida como [restrição de enumeração](#)) para especificar que um parâmetro de tipo é um tipo de enumeração.

## Tipos de valor internos

O C# fornece os seguintes tipos de valor internos, também conhecidos como *tipos simples*:

- [Tipos numéricos inteiros](#)
- [Tipos numéricos de ponto flutuante](#)
- `bool`, que representa um valor booleano
- `char`, que representa um caractere UTF-16 Unicode

Todos os tipos simples são tipos de estrutura, e diferem de outros tipos de estrutura por permitirem determinadas operações adicionais:

- Você pode usar literais para fornecer um valor de um tipo simples. Por exemplo, `'A'` é um literal do tipo `char` e `2001` é um literal do tipo `int`.

- Você pode declarar constantes dos tipos simples com a palavra-chave `const`. Não é possível ter constantes de outros tipos de estrutura.
- As expressões constantes, cujos operandos são todos constantes de tipo simples, são avaliadas em tempo de compilação.

Uma [tupla de valor](#) é um tipo de valor, mas não um tipo simples.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Tipos de valor](#)
- [Tipos simples](#)
- [Variáveis](#)

## Confira também

- [Referência de C#](#)
- [System.ValueType](#)
- [Tipos de referência](#)

# Tipos numéricos integrais (Referência C#)

Artigo • 10/05/2023

Os *tipos numéricos integrais* representam números inteiros. Todos os tipos numéricos integrais são [tipos de valor](#). Eles também são [tipos simples](#) e podem ser inicializados com [literais](#). Todos os tipos numéricos integrais dão suporte a operadores [aritméticos](#), [bit a bit lógicos](#), de [comparação](#) e [igualdade](#).

## Características dos tipos integrais

O C# é compatível com os seguintes tipos integrais predefinidos:

palavra-chave/tipo	Intervalo	Tamanho	Tipo .NET
<code>C#</code>			
<code>sbyte</code>	-128 a 127	Inteiro de 8 bits com sinal	<a href="#">System.SByte</a>
<code>byte</code>	0 a 255	Inteiro de 8 bits sem sinal	<a href="#">System.Byte</a>
<code>short</code>	-32.768 a 32.767	Inteiro de 16 bits com sinal	<a href="#">System.Int16</a>
<code>ushort</code>	0 a 65.535	Inteiro de 16 bits sem sinal	<a href="#">System.UInt16</a>
<code>int</code>	-2.147.483.648 a 2.147.483.647	Inteiro assinado de 32 bits	<a href="#">System.Int32</a>
<code>uint</code>	0 a 4.294.967.295	Inteiro de 32 bits sem sinal	<a href="#">System.UInt32</a>
<code>long</code>	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	Inteiro assinado de 64 bits	<a href="#">System.Int64</a>
<code>ulong</code>	0 a 18.446.744.073.709.551.615	Inteiro de 64 bits sem sinal	<a href="#">System.UInt64</a>
<code>nint</code>	Depende da plataforma (computada em runtime)	Inteiro de 32 bits ou de 64 bits com sinal	<a href="#">System.IntPtr</a>
<code>nuint</code>	Depende da plataforma (computada em runtime)	Inteiro de 32 bits ou de 64 bits sem sinal	<a href="#">System.UIntPtr</a>

Em todas as linhas da tabela, exceto as duas últimas, cada palavra-chave do tipo C# na coluna à esquerda é um alias do tipo .NET correspondente. A palavra-chave e o nome do tipo .NET são intercambiáveis. Por exemplo, as declarações a seguir declaram variáveis do mesmo tipo:

```
C#
```

```
int a = 123;  
System.Int32 b = 123;
```

Os tipos `nint` e `nuint` nas duas últimas linhas da tabela são inteiros de tamanho nativo. A partir do C# 9.0, você pode usar as palavras-chave `nint` e `nuint` para definir *inteiros de tamanho nativo*. São inteiros de 32 bits ao serem executados em um processo de 32 bits ou inteiros de 64 bits durante a execução em um processo de 64 bits. Eles podem ser usados para cenários de interoperabilidade, bibliotecas de baixo nível e para otimizar o desempenho em cenários em que a matemática inteiro é usada extensivamente.

Os tipos inteiros de tamanho nativo são representados internamente como os tipos .NET `System.IntPtr` e `System.UIntPtr`. A partir do C# 11, os tipos `nint` e `nuint` são aliases para os tipos subjacentes.

O valor padrão de cada tipo integral é zero, `0`.

Cada um dos tipos integrais possui as propriedades `MinValue` e `MaxValue` que fornecem o valor mínimo e máximo desse tipo. Essas propriedades são constantes em tempo de compilação, exceto para o caso dos tipos de tamanho nativo (`nint` e `nuint`). As propriedades `MinValue` e `MaxValue` são calculadas em runtime para tipos de tamanho nativo. Os tamanhos desses tipos dependem das configurações do processo.

Use a estrutura `System.Numerics.BigInteger` para representar um inteiro com sinal sem nenhum limite superior ou inferior.

## Literais inteiros

Os literais inteiros podem ser

- *decimal*: sem prefixos
- *hexadecimal*: com o prefixo `0x` ou `0X`
- *binário*: com o prefixo `0b` ou `0B`

O seguinte código demonstra um exemplo de cada um:

C#

```
var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

O exemplo anterior também mostra o uso de `_` como *separador de dígitos*. Você pode usar o separador de dígitos com todos os tipos de literais numéricos.

O tipo de literal inteiro é determinado pelo sufixo da seguinte maneira:

- Se o literal não tiver sufixo, seu tipo será o primeiro dos tipos a seguir em que seu valor pode ser representado: `int`, `uint`, `long`, `ulong`.

#### ➊ Observação

Os literais são interpretados como valores positivos. Por exemplo, o literal `0xFF_FF_FF_FF` representa o número `4294967295` do tipo `uint`, embora tenha a mesma representação de bit que o número `-1` do tipo `int`. Se você precisar de um valor de um determinado tipo, converta um literal nesse tipo. Use o operador `unchecked`, se um valor literal não puder ser representado no tipo de destino. Por exemplo, `unchecked((int)0xFF_FF_FF)` produz `-1`.

- Se o literal tiver o sufixo `U` ou `u`, seu tipo será o primeiro dos tipos a seguir em que seu valor pode ser representado: `uint`, `ulong`.
- Se o literal tiver o sufixo `L` ou `l`, seu tipo será o primeiro dos tipos a seguir em que seu valor pode ser representado: `long`, `ulong`.

#### ➊ Observação

Você pode usar a letra minúscula `l` como sufixo. No entanto, isso gera um aviso do compilador porque a letra `l` pode ser confundida com o dígito `1`. Use `L` para fins de clareza.

- Se o literal tiver o sufixo `UL`, `U1`, `uL`, `u1`, `LU`, `Lu`, `lU` ou `lu`, seu tipo será `ulong`.

Se o valor representado por um literal inteiro exceder `UInt64.MaxValue`, ocorrerá um erro de compilador [CS1021](#).

Se o tipo determinado de um literal inteiro for `int` e o valor representado pelo literal estiver dentro do intervalo do tipo de destino, o valor poderá ser convertido implicitamente em `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, `nint` ou `nuint`:

C#

```
byte a = 17;
byte b = 300; // CS0031: Constant value '300' cannot be converted to a
               'byte'
```

Como mostra o exemplo anterior, se o valor do literal não estiver dentro do intervalo do tipo de destino, ocorrerá um erro do compilador [CS0031](#).

Você também pode usar uma coerção para converter o valor representado por um literal inteiro em um tipo diferente do que foi determinado para o literal:

C#

```
var signedByte = (sbyte)42;
var longVariable = (long)42;
```

## Conversões

Você pode converter qualquer tipo numérico integral em qualquer outro tipo numérico integral. Se o tipo de destino puder armazenar todos os valores do tipo de origem, a conversão será implícita. Caso contrário, você deve usar uma [expressão de coerção](#) para executar uma conversão explícita. Para obter mais informações, confira [Conversões numéricas internas](#).

## Inteiros de tamanho nativo

Os tipos inteiros de tamanho nativo têm um comportamento especial, pois o armazenamento é determinado pelo tamanho do inteiro natural no computador de destino.

- Para obter o tamanho de um inteiro de tamanho nativo em tempo de execução, você pode usar `sizeof()`. No entanto, o código deve ser compilado em um contexto não seguro. Por exemplo:

C#

```

Console.WriteLine($"size of nint = {sizeof(nint)}");
Console.WriteLine($"size of nuint = {sizeof(nuint)}");

// output when run in a 64-bit process
//size of nint = 8
//size of nuint = 8

// output when run in a 32-bit process
//size of nint = 4
//size of nuint = 4

```

Você também pode obter o valor equivalente das propriedades [IntPtr.Size](#) e [UIntPtr.Size](#) estáticas.

- Para obter os valores mínimos e máximos de inteiros de tamanho nativo em tempo de execução, use `MinValue` e `.MaxValue` como propriedades estáticas com as palavras-chave `nint` e `nuint`, conforme o seguinte exemplo:

C#

```

Console.WriteLine($"nint.MinValue = {nint.MinValue}");
Console.WriteLine($"nint.MaxValue = {nint.MaxValue}");
Console.WriteLine($"nuint.MinValue = {nuint.MinValue}");
Console.WriteLine($"nuint.MaxValue = {nuint.MaxValue}");

// output when run in a 64-bit process
//nint.MinValue = -9223372036854775808
//nint.MaxValue = 9223372036854775807
//nuint.MinValue = 0
//nuint.MaxValue = 18446744073709551615

// output when run in a 32-bit process
//nint.MinValue = -2147483648
//nint.MaxValue = 2147483647
//nuint.MinValue = 0
//nuint.MaxValue = 4294967295

```

- Você pode usar valores constantes nos seguintes intervalos:
  - Para `nint`: [Int32.MinValue](#) para [Int32.MaxValue](#).
  - Para `nuint`: [UInt32.MinValue](#) para [UInt32.MaxValue](#).
- O compilador fornece conversões implícitas e explícitas para outros tipos numéricos. Para obter mais informações, confira [Conversões numéricas internas](#).
- Não há sintaxe direta para literais inteiros de tamanho nativo. Não há sufixo para indicar que um literal é um inteiro de tamanho nativo, como `L` para indicar um

`long`. Você pode usar conversões implícitas ou explícitas de outros valores inteiros.

Por exemplo:

C#

```
nint a = 42
nint a = (nint)42;
```

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Tipos integrais](#)
- [Literais inteiros](#)
- [C# 9 – Tipos integrais de tamanho nativo](#)
- [C# 11 – IntPtr numérico e 'UIntPtr](#)

## Confira também

- [Referência de C#](#)
- [Tipos de valor](#)
- [Tipos de ponto flutuante](#)
- [Cadeias de caracteres de formato numérico padrão](#)
- [Numéricos no .NET](#)

# Tipos numéricos de ponto flutuante (Referência de C#)

Artigo • 09/05/2023

Os *tipos numéricos de ponto flutuante* representam números reais. Todos os tipos numéricos de ponto flutuante são [tipos de valor](#). Eles também são [tipos simples](#) e podem ser inicializados com [literais](#). Todos os tipos numéricos de ponto flutuante dão suporte a operadores [aritméticos](#), [de comparação](#) e [de igualdade](#).

## Características dos tipos de ponto flutuante

O C# é compatível com os seguintes tipos de pontos flutuantes predefinidos:

palavra-chave/tipo C#	Intervalo aproximado	Precisão	Tamanho	Tipo .NET
<code>float</code>	$\pm 1,5 \times 10^{-45}$ para $\pm 3,4 \times 10^{38}$	~6 a 9 dígitos	4 bytes	<a href="#">System.Single</a>
<code>double</code>	$\pm 5,0 \times 10^{-324}$ to $\pm 1,7 \times 10^{308}$	~15 a 17 dígitos	8 bytes	<a href="#">System.Double</a>
<code>decimal</code>	$\pm 1,0 \times 10^{-28}$ para $\pm 7,9228 \times 10^{28}$	28 a 29 dígitos	16 bytes	<a href="#">System.Decimal</a>

Na tabela anterior, cada palavra-chave do tipo C# da coluna mais à esquerda é um alias do tipo .NET correspondente. Eles são intercambiáveis. Por exemplo, as declarações a seguir declaram variáveis do mesmo tipo:

```
C#  
  
double a = 12.3;  
System.Double b = 12.3;
```

O valor padrão de cada tipo de ponto flutuante é zero, `0`. Cada um dos tipos de ponto flutuante possui as constantes `MinValue` e `MaxValue` que fornecem o valor mínimo e máximo desse tipo. Os tipos `float` e `double` também fornecem constantes que representam valores não numéricos e infinitos. Por exemplo, o tipo `double` fornece as seguintes constantes: [Double.NaN](#), [Double.NegativeInfinity](#) e [Double.PositiveInfinity](#).

O tipo `decimal` é apropriado quando o grau de precisão necessário é determinado pelo número de dígitos à direita do ponto decimal. Esses números são comumente usados

em aplicativos financeiros, para valores de moeda (por exemplo, US\$ 1,00), taxas de juros (por exemplo, 2,625%) e assim por diante. Mesmo os números com apenas uma casa decimal são tratados com mais precisão pelo tipo `decimal`: 0,1, por exemplo, pode ser representado com exatidão por meio de uma instância `decimal`, ao passo que não há instância `double` ou `float` que representem 0,1 com exatidão. Devido a essa diferença em tipos numéricos, erros de arredondamento inesperados podem ocorrer em cálculos aritméticos quando você usa `double` ou `float` para dados decimais. Você pode usar `double` em vez de `decimal` quando otimizar o desempenho for mais importante do que garantir a precisão. No entanto, qualquer diferença no desempenho passaria despercebida por todos, exceto pelos aplicativos com maior uso de cálculo. Outro motivo possível a ser evitado `decimal` é minimizar os requisitos de armazenamento. Por exemplo, [ML.NET](#) usa `float` porque a diferença entre 4 bytes e 16 bytes se soma para cada conjuntos de dados muito grandes. Para obter mais informações, consulte [System.Decimal](#).

Você pode misturar tipos `integrais` e os tipos `float` e `double` em uma expressão. Nesse caso, os tipos integrais são convertidos implicitamente em um dos tipos de ponto flutuante e, se necessário, o tipo `float` é convertido implicitamente em `double`. A expressão é avaliada como segue:

- Se houver os tipo `double` na expressão, ela será avaliada como `double`, ou como `bool` em comparações relacionais e de igualdade.
- Se não houver nenhum tipo `double` na expressão, ela será avaliada como `float`, ou como `bool` em comparações relacionais e de igualdade.

Você também pode misturar tipos integrais e o tipo `decimal` em uma expressão. Nesse caso, os tipos integrais são convertidos implicitamente no tipo `decimal` e a expressão é avaliada como `decimal`, ou como `bool` em comparações relacionais e de igualdade.

Não é possível misturar o tipo `decimal` com os tipos `float` e `double` em uma expressão. Nesse caso, se você quiser executar operações aritméticas, de comparação ou de igualdade, deverá converter explicitamente os operandos de/para o tipo `decimal`, como mostra o seguinte exemplo:

C#

```
double a = 1.0;
decimal b = 2.1m;
Console.WriteLine(a + (double)b);
Console.WriteLine((decimal)a + b);
```

É possível usar [cadeias de caracteres de formato numérico padrão](#) ou [cadeias de caracteres de formato numérico personalizado](#) para formatar um valor de ponto flutuante.

## Literais reais

O tipo de literal real é determinado pelo sufixo da seguinte maneira:

- O literal sem sufixo ou com os sufixos `d` ou `D` é do tipo `double`
- O literal com os sufixos `f` ou `F` é do tipo `float`
- O literal com os sufixos `m` ou `M` é do tipo `decimal`

O seguinte código demonstra um exemplo de cada um:

C#

```
double d = 3D;  
d = 4d;  
d = 3.934_001;  
  
float f = 3_000.5F;  
f = 5.4f;  
  
decimal myMoney = 3_000.5m;  
myMoney = 400.75M;
```

O exemplo anterior também mostra o uso de `_` como *separador de dígitos*. Você pode usar o separador de dígitos com todos os tipos de literais numéricos.

Você também pode usar notação científica, ou seja, especificar uma parte expoente de um literal real, como mostra o seguinte exemplo:

C#

```
double d = 0.42e2;  
Console.WriteLine(d); // output 42  
  
float f = 134.45E-2f;  
Console.WriteLine(f); // output: 1.3445  
  
decimal m = 1.5E6m;  
Console.WriteLine(m); // output: 1500000
```

## Conversões

Há apenas uma conversão implícita entre tipos numéricos de ponto flutuante: de `float` para `double`. No entanto, você pode converter qualquer tipo de ponto flutuante em qualquer outro tipo de ponto flutuante com a [conversão explícita](#). Para obter mais informações, confira [Conversões numéricas internas](#).

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Tipos de ponto flutuante](#)
- [O tipo decimal](#)
- [Literais reais](#)

## Confira também

- [Referência de C#](#)
- [Tipos de valor](#)
- [Tipos inteiros](#)
- [Cadeias de caracteres de formato numérico padrão](#)
- [Numéricos no .NET](#)
- [System.Numerics.Complex](#)

# Conversões numéricas internas (referência em C#)

Artigo • 07/04/2023

O C# fornece um conjunto de tipos numéricos [integral](#) e [ponto flutuante](#). Existe uma conversão entre dois tipos numéricos, implícitos ou explícitos. Você deve usar uma [expressão de conversão](#) para executar uma conversão explícita.

## Conversões numéricas implícitas

A tabela a seguir mostra as conversões implícitas predefinidas entre tipos numéricos do .NET.

De	Para
<code>sbyte</code>	<code>short</code> , <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , <code>decimal</code> OU <code>nint</code>
<code>byte</code>	<code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> , <code>decimal</code> , <code>nint</code> OU <code>nuint</code>
<code>short</code>	<code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , OU <code>decimal</code> OU <code>nint</code>
<code>ushort</code>	<code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> OR <code>decimal</code> , <code>nint</code> OU <code>nuint</code>
<code>int</code>	<code>long</code> , <code>float</code> , <code>double</code> OU <code>decimal</code> , <code>nint</code>
<code>uint</code>	<code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> OR <code>decimal</code> , OU <code>nuint</code>
<code>longo</code>	<code>float</code> , <code>double</code> OU <code>decimal</code>
<code>ulong</code>	<code>float</code> , <code>double</code> OU <code>decimal</code>
<code>float</code>	<code>double</code>
<code>nint</code>	<code>long</code> , <code>float</code> , <code>double</code> OU <code>decimal</code>
<code>nuint</code>	<code>ulong</code> , <code>float</code> , <code>double</code> OU <code>decimal</code>

### ⓘ Observação

As conversões implícitas de `int`, `uint`, `long`, `ulong`, `nint` OU `nuint` para `float` e de `long`, `ulong`, `nint` OU `nuint` para `double` podem causar uma perda de precisão, mas nunca uma perda de uma ordem de magnitude. As outras conversões numéricas implícitas nunca perdem nenhuma informação.

Além disso, note que:

- Qualquer [tipo numérico integral](#) pode ser implicitamente convertido em qualquer [tipo numérico de ponto flutuante](#).
- Não há nenhuma conversão implícita para os tipos `byte`, `sbyte` e . Não há nenhuma conversão implícita dos tipos `double` e `decimal`.
- Não há nenhuma conversão implícita entre os tipos `decimal` e `float` ou os tipos `double`.
- Um valor de uma expressão de constante de tipo `int` (por exemplo, um valor representado por um literal integral) pode ser convertido em `sbyte`, `byte`, `short`, `ushort`, `uint` ou `ulong`, `nint` ou `nuint`, se estiver dentro do intervalo do tipo de destino:

C#

```
byte a = 13;
byte b = 300; // CS0031: Constant value '300' cannot be converted to a
               'byte'
```

Como mostra o exemplo anterior, se o valor constante não estiver dentro do intervalo do tipo de destino, ocorrerá um erro do compilador [CS0031](#) .

## Conversões numéricas explícitas

A tabela a seguir mostra as conversões explícitas predefinidas entre os tipos numéricos embutidos para os quais não há nenhuma [conversão implícita](#):

De	Para
<code>sbyte</code>	<code>byte</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code> OU <code>nuint</code>
<code>byte</code>	<code>sbyte</code>
<code>short</code>	<code>sbyte</code> , <code>byte</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code> OU <code>nuint</code>
<code>ushort</code>	<code>sbyte</code> , <code>byte</code> OU <code>short</code>
<code>int</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code> OU <code>nuint</code>
<code>uint</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> OU <code>nint</code>
<code>longo</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>ulong</code> , <code>nint</code> OU <code>nuint</code>

De	Para
<code>ulong</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>nint</code> ou <code>nuint</code>
<code>float</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>decimal</code> , <code>nint</code> ou <code>nuint</code>
<code>double</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>decimal</code> , <code>nint</code> ou <code>nuint</code>
<code>decimal</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> , <code>double</code> , <code>nint</code> ou <code>nuint</code>
<code>nint</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>ulong</code> ou <code>nuint</code>
<code>nuint</code>	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> ou <code>nint</code>

### ① Observação

Uma conversão numérica explícita pode resultar em perda de dados ou gerar uma exceção, normalmente um `OverflowException`.

Além disso, note que:

- Quando você converte um valor de um tipo integral em outro tipo integral, o resultado depende do [contexto de verificação](#) do estouro. Em um contexto verificado, a conversão terá êxito se o valor de origem estiver dentro do intervalo do tipo de destino. Caso contrário, um `OverflowException` será gerado. Em um contexto não verificado, a conversão sempre terá êxito e procederá da seguinte maneira:
  - Se o tipo de origem for maior do que o tipo de destino, então o valor de origem será truncado descartando seus bits "extra" mais significativos. O resultado é então tratado como um valor do tipo de destino.
  - Se o tipo de origem for menor do que o tipo de destino, então o valor de origem será estendido por sinal ou por zero para que tenha o mesmo tamanho que o tipo de destino. A extensão por sinal será usada se o tipo de origem tiver sinal; a extensão por zero será usada se o tipo de origem não tiver sinal. O resultado é então tratado como um valor do tipo de destino.
  - Se o tipo de origem tiver o mesmo tamanho que o tipo de destino, então o valor de origem será tratado como um valor do tipo de destino.
- Ao converter um valor `decimal` para um tipo integral, esse valor será arredondado para zero, para o valor integral mais próximo. Se o valor integral resultante estiver fora do intervalo do tipo de destino, uma `OverflowException` será lançada.

- Ao converter um valor `double` ou `float` em um tipo integral, esse valor será arredondado em direção a zero para o valor integral mais próximo. Se o valor integral resultante estiver fora do intervalo do tipo de destino, o resultado dependerá do contexto de verificação de estouro. Em um contexto verificado, uma `OverflowException` será lançada, ao passo que em um contexto não verificado, o resultado será um valor não especificado do tipo de destino.
- Ao converter `double` para `float`, o valor `double` será arredondado para o valor `float` mais próximo. Se o valor `double` for muito pequeno ou muito grande para caber no tipo `float`, o resultado será zero ou infinito.
- Ao converter `float` ou `double` para `decimal`, o valor de origem será convertido para uma representação `decimal` e arredondado para o número mais próximo após a 28.<sup>a</sup> casa decimal, se necessário. De acordo com o valor do valor de origem, um dos resultados a seguir podem ocorrer:
  - Se o valor de origem for muito pequeno para ser representado como um `decimal`, o resultado será zero.
  - Se o valor de origem for um NaN (não for um número), infinito ou muito grande para ser representado como um `decimal`, uma `OverflowException` será lançada.
- Ao converter `decimal` para `float` ou `double`, o valor de origem será arredondado para o valor `float` ou `double` mais próximo.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Conversões numéricas implícitas](#)
- [Conversões numéricas explícitas](#)

## Confira também

- [Referência de C#](#)
- [Conversão e conversões de tipo](#)

# bool (referência de C#)

Artigo • 10/05/2023

A palavra-chave de tipo `bool` é um alias para o tipo de estrutura `System.Boolean` do .NET que representa um valor booleano, que pode ser `true` ou `false`.

Para executar operações lógicas com valores do tipo `bool`, use operadores [lógicos booleanos](#). O tipo `bool` é o tipo de resultado de operadores de [comparação](#) e [igualdade](#). Uma expressão `bool` pode ser uma expressão condicional de controle nas instruções `if`, `do`, `while` e `for` e no [operador condicional ?:](#).

O valor padrão do tipo `bool` é `false`.

## Literais

Você pode usar os literais `true` e `false` para inicializar uma variável `bool` ou passar um valor `bool`:

C#

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked"); // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked"); // output: Not
checked
```

## Lógica booleana de três valores

Use o tipo `bool?` se você precisar oferecer suporte à lógica de três valores, por exemplo, ao trabalhar com bancos de dados que dão suporte a um tipo booleano de três valores. Para os operandos `bool?`, os operadores `&` e `|` predefinidos oferecem suporte à lógica de três valores. Para obter mais informações, confira a seção [Operadores lógicos booleanos anuláveis](#) do artigo [Operadores lógicos booleanos](#).

Para obter mais informações sobre tipos que permitem valor nulo, consulte [Tipos que permitem valor nulo](#).

## Conversões

O C# fornece apenas duas conversões que envolvem o tipo `bool`. Esse tipo é uma conversão implícita para o tipo que permite valor nulo `bool?` correspondente e uma conversão explícita do tipo `bool?`. No entanto, o .NET fornece métodos adicionais que você pode usar para converter de ou para o tipo `bool`. Para obter mais informações, consulte a seção [Convertendo de e para valores booleanos](#) da página de referência da API `System.Boolean`.

## Especificação da linguagem C#

Para obter mais informações, confira a seção [O tipo bool](#) da [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Tipos de valor](#)
- [operadores true e false](#)

# char (Referência de C#)

Artigo • 09/05/2023

A palavra-chave de tipo `char` é um alias para o tipo de estrutura `System.Char` em .NET que representa um caractere Unicode UTF-16.

Tipo	Intervalo	Tamanho	Tipo .NET
<code>char</code>	U+0000 a U+FFFF	16 bits	<code>System.Char</code>

O valor padrão do tipo `char` é `\0`, ou seja, U+0000.

O tipo `char` dá suporte a operadores de [comparação](#), [igualdade](#), [incremento](#) e [decremento](#). Além disso, para operandos `char`, operadores lógicos [aritméticos](#) e de [bits](#) executam uma operação nos códigos de caractere correspondentes e produzem o resultado do tipo `int`.

O tipo de [cadeia de caracteres](#) representa o texto como uma sequência de valores `char`.

## Literais

Você pode especificar um valor `char` com:

- um literal de caractere.
- uma sequência de escape Unicode, que é `\u` seguida pela representação hexadecimal de quatro símbolos de um código de caractere.
- uma sequência de escape hexadecimal, que é `\x` seguida pela representação hexadecimal de um código de caractere.

```
C#  
  
var chars = new[]  
{  
    'j',  
    '\u006A',  
    '\x006A',  
    (char)106,  
};  
Console.WriteLine(string.Join(" ", chars)); // output: j j j j
```

Como mostra o exemplo anterior, você também pode converter o valor de um código de caractere no valor correspondente `char`.

## ⓘ Observação

No caso de uma sequência de escape Unicode, você deve especificar todos os quatro dígitos hexadecimal. Ou seja, `\u006A` é uma sequência de escape válida, enquanto `\u06A` e `\u6A` não são válidas.

No caso de uma sequência de escape hexadecimal, você pode omitir os zeros à esquerda. Ou seja, as sequências de escape `\x006A`, `\x06A` e `\x6A` são válidas e correspondem ao mesmo caractere.

## Conversões

O tipo `char` é implicitamente conversível para os seguintes tipos [integrais](#): `ushort`, `int`, `uint`, `long` e `ulong`. Ele também é implicitamente conversível para os tipos numéricos de [ponto flutuante](#) internos: `float`, `double` e `decimal`. É explicitamente conversível para os tipos integrais `sbyte`, `byte` e `short`.

Não há conversões implícitas de outros tipos para o tipo `char`. No entanto, qualquer tipo numérico de [integral](#) ou de [ponto flutuante](#) é explicitamente conversível para `char`.

## Especificação da linguagem C#

Para saber mais, confira a seção [Tipos integrais](#) na [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Tipos de valor](#)
- [Cadeias de caracteres](#)
- [System.Text.Rune](#)
- [Codificação de caracteres no .NET](#)

# Tipos de enumeração (referência de C#)

Artigo • 07/04/2023

Um *tipo de enumeração* (ou *tipo enum*) é um [tipo de valor](#) definido por um conjunto de constantes nomeadas do tipo [numérico integral](#) subjacente. Para definir um tipo de enumeração, use a palavra-chave `enum` e especifique os nomes dos *membros de enumeração*:

```
C#  
  
enum Season  
{  
    Spring,  
    Summer,  
    Autumn,  
    Winter  
}
```

Por padrão, os valores constantes associados de enumeração de membros são do tipo `int`; eles começam com zero e aumentam em um seguindo a ordem de texto de definição. Você pode especificar explicitamente qualquer outro tipo [numérico integral](#) como um tipo subjacente de um tipo de enumeração. Você também pode especificar explicitamente os valores constantes associados, como mostra o exemplo a seguir:

```
C#  
  
enum ErrorCode : ushort  
{  
    None = 0,  
    Unknown = 1,  
    ConnectionLost = 100,  
    OutlierReading = 200  
}
```

Não é possível definir um método dentro da definição de um tipo de enumeração. Para adicionar funcionalidade a um tipo de enumeração, crie um [método de extensão](#).

O valor padrão de um tipo de enumeração `E` é o valor produzido pela expressão `(E)0`, mesmo que zero não tenha o membro de enumeração correspondente.

Você usa um tipo de enumeração para representar uma escolha de um conjunto de valores mutuamente exclusivos ou uma combinação de opções. Para representar uma combinação de opções, defina um tipo de enumeração como sinalizadores de bit.

# Tipos de Enumeração como Sinalizadores de Bit

Se você quiser que um tipo de enumeração represente uma combinação de opções, defina membros de enumeração para essas opções de forma que uma escolha individual seja um campo de bits. Ou seja, os valores associados desses membros de enumeração devem ser as potências de dois. Em seguida, você pode usar os [operadores lógicos bit a bit](#) | ou & para combinar opções ou cruzar combinações de opções, respectivamente. Para indicar que um tipo de enumeração declara campos de bit, aplique o atributo [Flags](#) a ele. Como mostra o exemplo a seguir, você também pode incluir algumas combinações típicas na definição de um tipo de enumeração.

C#

```
[Flags]
public enum Days
{
    None      = 0b_0000_0000, // 0
    Monday    = 0b_0000_0001, // 1
    Tuesday   = 0b_0000_0010, // 2
    Wednesday = 0b_0000_0100, // 4
    Thursday  = 0b_0000_1000, // 8
    Friday    = 0b_0001_0000, // 16
    Saturday  = 0b_0010_0000, // 32
    Sunday    = 0b_0100_0000, // 64
    Weekend   = Saturday | Sunday
}

public class FlagsEnumExample
{
    public static void Main()
    {
        Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
        Console.WriteLine(meetingDays);
        // Output:
        // Monday, Wednesday, Friday

        Days workingFromHomeDays = Days.Thursday | Days.Friday;
        Console.WriteLine($"Join a meeting by phone on {meetingDays & workingFromHomeDays}");
        // Output:
        // Join a meeting by phone on Friday

        bool isMeetingOnTuesday = (meetingDays & Days.Tuesday) ==
Days.Tuesday;
        Console.WriteLine($"Is there a meeting on Tuesday:
{isMeetingOnTuesday}");
        // Output:
        // Is there a meeting on Tuesday: False
```

```
    var a = (Days)37;
    Console.WriteLine(a);
    // Output:
    // Monday, Wednesday, Saturday
}
}
```

Para obter mais informações e exemplos, consulte a página de referência da API [System.FlagsAttribute](#) e a seção [Membros não exclusivos e o atributo Flags](#) da página de referência da API [System.Enum](#).

## O tipo System.Enum e a restrição enum

O tipo [System.Enum](#) é a classe base abstrata de todos os tipos de enumeração. Ele fornece vários métodos para obter informações sobre um tipo de enumeração e seus valores. Para obter mais informações e exemplos, consulte a página de referência da API [System.Enum](#).

Você pode usar `System.Enum` em uma restrição de classe base (conhecida como [restrição de enumeração](#)) para especificar que um parâmetro de tipo é um tipo de enumeração. Qualquer tipo de enumeração também satisfaz a restrição `struct`, que é usada para especificar que um parâmetro de tipo é um tipo de valor não anulável.

## Conversões

Para qualquer tipo de enumeração, existem conversões explícitas entre o tipo de enumeração e seu tipo integral subjacente. Se você [converter](#) um valor de enumeração para seu tipo subjacente, o resultado será o valor integral associado de um membro de enumeração.

C#

```
public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
```

```
        Console.WriteLine($"Integral value of {a} is {(int)a}"); // output:  
Integral value of Autumn is 2  
  
        var b = (Season)1;  
        Console.WriteLine(b); // output: Summer  
  
        var c = (Season)4;  
        Console.WriteLine(c); // output: 4  
    }  
}
```

Use o método [Enum.IsDefined](#) para determinar se um tipo de enumeração contém um membro de enumeração com determinado valor associado.

Para qualquer tipo de enumeração, existem as conversões [boxing](#) e [unboxing](#) de e para o tipo [System.Enum](#), respectivamente.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Enumerações](#)
- [Operações e valores de enum](#)
- [Operadores lógicos de enumeração](#)
- [Operadores de comparação de enumeração](#)
- [Conversões de enumeração explícitas](#)
- [Conversões de enumeração implícitas](#)

## Confira também

- [Referência de C#](#)
- [Cadeias de caracteres de formato de enumeração](#)
- [Diretrizes de design – design de enumeração](#)
- [Diretrizes de design – convenções de nomenclatura de enumeração](#)
- [Expressão switch](#)
- [instrução switch](#)

# Tipos de estrutura (referência de C#)

Artigo • 13/07/2023

Um *tipo de estrutura* (ou *tipo de struct*) é um [tipo de valor](#) que pode encapsular dados e funcionalidades relacionadas. Você usa a palavra-chave `struct` para definir um tipo de estrutura:

C#

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

Para obter informações sobre os tipos `ref struct` e `readonly ref struct`, confira o artigo [tipos de estrutura ref](#).

Os tipos de estrutura têm *semântica de valor*. Ou seja, uma variável de um tipo de estrutura contém uma instância do tipo. Por padrão, os valores das variáveis são copiados na atribuição, passando um argumento para um método e retornando um resultado de método. Para variáveis de tipo de estrutura, uma instância do tipo é copiada. Para obter mais informações, confira [Tipos de valor](#).

Normalmente, você usa tipos de estrutura para criar pequenos tipos centrados em dados que fornecem pouco ou nenhum comportamento. Por exemplo, o .NET usa tipos de estrutura para representar um número ([inteiro](#) e [real](#)), um [valor booleano](#), um [caractere Unicode](#), uma [instância de tempo](#). Se você estiver focado no comportamento de um tipo, considere definir uma [classe](#). Os tipos de classe têm *semântica de referência*. Ou seja, uma variável de um tipo de classe contém uma referência a uma instância do tipo, não à instância em si.

Como os tipos de estrutura têm semântica de valor, recomendamos que você defina tipos de estrutura *imutáveis*.

# Struct `readonly`

Use o modificador `readonly` para declarar que um tipo de estrutura é imutável. Todos os membros de dados de um struct `readonly` devem ser somente leitura da seguinte maneira:

- Qualquer declaração de campo deve ter o [modificador `readonly`](#)
- Qualquer propriedade, incluindo as implementadas automaticamente, deve ser somente leitura. No C# 9.0 e posterior, uma propriedade pode ter um [acessador `init`](#).

Isso garante que nenhum membro de um struct `readonly` modifique o estado do struct. Isso significa que outros membros de instância, exceto constructos, são implicitamente [readonly](#).

## ⓘ Observação

Em um struct `readonly`, um membro de dados de um tipo de referência mutável ainda pode alterar seu próprio estado. Por exemplo, você não pode substituir uma instância `List<T>`, mas pode adicionar novos elementos a ela.

O código a seguir define um struct `readonly` com setters de propriedade somente init, disponíveis no C# 9.0 e posterior:

C#

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}
```

## Membros da instância `readonly`

Você também pode usar o modificador `readonly` para declarar que um membro de instância não modifica o estado de um struct. Se você não puder declarar todo o tipo de estrutura como `readonly`, use o modificador `readonly` para marcar os membros da instância que não modificam o estado do struct.

Em um membro de instância `readonly`, você não pode atribuir aos campos de instância da estrutura. No entanto, um membro `readonly` pode chamar um não membro `readonly`. Nesse caso, o compilador cria uma cópia da instância da estrutura e chama o membro não `readonly` nessa cópia. Como resultado, a instância de estrutura original não é modificada.

Normalmente, você aplica o modificador `readonly` aos seguintes tipos de membros de instância:

- métodos:

```
C#  
  
public readonly double Sum()  
{  
    return X + Y;  
}
```

Você também pode aplicar o modificador `readonly` aos métodos que substituem os métodos declarados em [System.Object](#):

```
C#  
  
public readonly override string ToString() => $"({X}, {Y});
```

- propriedades e indexadores:

```
C#  
  
private int counter;  
public int Counter  
{  
    readonly get => counter;  
    set => counter = value;  
}
```

Se você precisar aplicar o modificador `readonly` a ambos os acessadores de uma propriedade ou indexador, aplique-o na declaração da propriedade ou do indexador.

## (!) Observação

O compilador declara um acessador `get` de uma **propriedade autoimplementada** como `readonly`, independentemente da presença do modificador `readonly` em uma declaração de propriedade.

No C# 9.0 e posterior, você pode aplicar o modificador `readonly` a uma propriedade ou indexador com um acessador `init`:

C#

```
public readonly double X { get; init; }
```

Você pode aplicar o modificador `readonly` a campos estáticos de um tipo de estrutura, mas não a outros membros estáticos, como propriedades ou métodos.

O compilador pode usar o modificador `readonly` para otimizações de desempenho. Para obter mais informações, confira [Como evitar alocações](#).

## Mutação não destrutiva

A partir do C# 10, você pode usar a [expressão with](#) para produzir uma cópia de uma instância do tipo estrutura com as propriedades e campos especificados modificados. Você usa a sintaxe do [inicializador de objeto](#) para especificar quais membros modificar e seus novos valores.

C#

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

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

public static void Main()
{
    var p1 = new Coords(0, 0);
```

```
Console.WriteLine(p1); // output: (0, 0)

var p2 = p1 with { X = 3 };
Console.WriteLine(p2); // output: (3, 0)

var p3 = p1 with { X = 1, Y = 4 };
Console.WriteLine(p3); // output: (1, 4)
}
```

## Struct record

A partir do C# 10, você pode definir tipos de estrutura de registro. Os tipos de registro fornecem funcionalidade interna para encapsular dados. Você pode definir os tipos `record struct` e `readonly record struct`. Um struct de registro não pode ser um `ref struct`. Saiba mais e obtenha exemplos em [Registros](#).

## Matrizes embutidas

Começando com C# 12, você pode declarar as *matrizes embutidas* como um tipo `struct`:

```
C#

[System.Runtime.CompilerServices.InlineArray(10)]
public struct CharBuffer
{
    private char _firstElement;
}
```

Uma matriz embutida é uma estrutura que contém um bloco contíguo de N elementos do mesmo tipo. É um equivalente de código seguro da declaração `buffer fixo` disponível apenas em código inseguro. Uma matriz embutida é um `struct` com as seguintes características:

- Ele contém um único campo.
- O struct não especifica um layout explícito.

Além disso, o compilador valida o atributo [System.Runtime.CompilerServices.InlineArrayAttribute](#):

- O comprimento deve ser maior que zero (`> 0`).
- O tipo de destino deve ser um struct.

Na maioria dos casos, uma matriz embutida pode ser acessada como uma matriz, tanto para leitura quanto para gravação de valores. Além disso, você pode usar os operadores `range` e `index`.

Existem restrições mínimas quanto ao tipo do campo único. Ele não pode ser um tipo de ponteiro, mas pode ser qualquer tipo de referência ou qualquer tipo de valor. Você pode usar matrizes embutidas com quase qualquer estrutura de dados C#.

As matrizes embutida são um recurso avançado da linguagem. Destinam-se a cenários de alto desempenho em que um bloco de elementos embutidos e contíguo é mais rápido do que outras estruturas de dados alternativas. Você pode aprender mais sobre matrizes embutidas a partir da [funcionalidade speclet](#)

## Inicialização de struct e valores padrão

Uma variável de um tipo `struct` contém diretamente os dados para `struct`. Isso cria uma distinção entre um valor não inicializado `struct`, que tem seu valor padrão e um inicializado `struct`, que armazena valores definidos pela construção dele. Por exemplo, considere o código a seguir:

C#

```
public readonly struct Measurement
{
    public Measurement()
    {
        Value = double.NaN;
        Description = "Undefined";
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; }

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement();
    Console.WriteLine(m1); // output: NaN (Undefined)

    var m2 = default(Measurement);
```

```
Console.WriteLine(m2); // output: 0 ()  
  
var ms = new Measurement[2];  
Console.WriteLine(string.Join(", ", ms)); // output: 0 (), 0 ()  
}
```

Como mostra o exemplo anterior, a [expressão de valor padrão](#) ignora um construtor sem parâmetros e produz o [valor padrão](#) do tipo de estrutura. A instanciação de matriz do tipo estrutura também ignora um construtor sem parâmetros e produz uma matriz preenchida com os valores padrão de um tipo de estrutura.

A situação mais comum em que você verá valores padrão está em matrizes ou em outras coleções em que o armazenamento interno inclui blocos de variáveis. O exemplo a seguir cria uma matriz de 30 estruturas `TemperatureRange`, cada uma com o valor padrão:

C#

```
// All elements have default values of 0:  
TemperatureRange[] lastMonth = new TemperatureRange[30];
```

Todos os campos membros de um struct precisam ser *atribuídos definitivamente* quando ele é criado porque os tipos `struct` armazenam dados diretamente. O valor `default` de um struct *definitivamente atribuiu* todos os campos a 0. Todos os campos devem ser atribuídos definitivamente quando um construtor é invocado. Você inicializa campos usando os seguintes mecanismos:

- Você pode adicionar *inicializadores de campo* a qualquer campo ou propriedade implementada automaticamente.
- Você pode inicializar quaisquer campos ou propriedades automáticas no corpo do construtor.

A partir do C# 11, se você não inicializar todos os campos em um struct, o compilador adicionará código ao construtor que inicializa esses campos ao valor padrão. O compilador executa sua análise de atribuição definida habitual. Todos os campos acessados antes de serem atribuídos ou não atribuídos definitivamente quando o construtor terminar de executar receberão seus valores padrão antes da execução do corpo do construtor. Se `this` for acessado antes de todos os campos serem atribuídos, o struct será inicializado para o valor padrão antes que o corpo do construtor seja executado.

C#

```

public readonly struct Measurement
{
    public Measurement(double value)
    {
        Value = value;
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public Measurement(string description)
    {
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; } = "Ordinary measurement";

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement(5);
    Console.WriteLine(m1); // output: 5 (Ordinary measurement)

    var m2 = new Measurement();
    Console.WriteLine(m2); // output: 0 ()

    var m3 = default(Measurement);
    Console.WriteLine(m3); // output: 0 ()
}

```

Cada `struct` tem um construtor sem parâmetro `public`. Se você escrever um construtor sem parâmetros, ele deverá ser público. Se um `struct` declarar qualquer inicializador de campo, ele deverá declarar explicitamente um `constructo`. Esse `constructo` não precisa ser sem parâmetros. Se um `struct` declarar um inicializador de campo, mas nenhum `constructo`, o compilador relatará um erro. Qualquer `constructo` declarado explicitamente (com parâmetros ou sem parâmetros) executa todos os inicializadores de campo para esse `struct`. Todos os campos sem um inicializador de campo ou uma atribuição em um `constructo` são definidos como `valor padrão`. Para obter mais informações, consulte nota de proposta de recurso de [construtores de `struct` sem parâmetros](#).

Começando no C# 12, tipos `struct` podem definir um `construtor primário` como parte de sua declaração. Os construtores primários fornecem uma sintaxe concisa para os

parâmetros do construtor que podem ser usados em todo o corpo `struct`, em qualquer declaração de membro para esse `struct`.

Se todos os campos de instância de um tipo de estrutura estiverem acessíveis, você também poderá instanciá-lo sem o operador `new`. Nesse caso, você deve inicializar todos os campos de instância antes do primeiro uso da instância. O seguinte exemplo mostra como fazer isso:

C#

```
public static class StructWithoutNew
{
    public struct Coords
    {
        public double x;
        public double y;
    }

    public static void Main()
    {
        Coords p;
        p.x = 3;
        p.y = 4;
        Console.WriteLine($"{p.x}, {p.y}"); // output: (3, 4)
    }
}
```

No caso dos [tipos de valor internos](#), use os literais correspondentes para especificar um valor do tipo.

## Limitações com o design de um tipo de estrutura

Os structs têm a maioria dos recursos de um tipo de [classe](#). Há algumas exceções e algumas exceções que foram removidas em versões mais recentes:

- Um tipo de estrutura não pode herdar de outro tipo de classe ou estrutura e não pode ser a base de uma classe. No entanto, um tipo de estrutura pode implementar [interfaces](#).
- Você não pode declarar um [finalizador](#) dentro de um tipo de estrutura.
- Antes do C# 11, um construtor de um tipo de estrutura deve inicializar todos os campos de instância do tipo.
- Antes do C# 10, você não pode declarar um construtor sem parâmetros.
- Antes do C# 10, você não pode inicializar um campo ou uma propriedade de instância em sua declaração.

# Passando variáveis de tipo de estrutura por referência

Quando você passa uma variável de tipo de estrutura para um método como um argumento ou retorna um valor de tipo de estrutura de um método, toda a instância de um tipo de estrutura é copiada. Passar por valor pode afetar o desempenho do código em cenários de alto desempenho que envolvem tipos de estrutura grandes. Você pode evitar a cópia de valor passando uma variável de tipo de estrutura por referência. Use os modificadores de parâmetro de método [ref](#), [out](#) ou [in](#) para indicar que um argumento deve ser passado por referência. Use [ref returns](#) para retornar um resultado de método por referência. Para obter mais informações, confira [Evitar alocações](#).

## restrição de struct

Você também usa a palavra-chave `struct` na restrição [struct](#) para especificar que um parâmetro de tipo é um tipo de valor não anulável. Os tipos de estrutura e [enumeração](#) atendem à restrição `struct`.

## Conversões

Para qualquer tipo de estrutura (exceto tipos [ref struct](#)), existem conversões de [boxing](#) e [unboxing](#) de e para os tipos [System.ValueType](#) e [System.Object](#). Também existem conversões de boxing e unboxing entre um tipo de estrutura e qualquer interface que ele implemente.

## Especificação da linguagem C#

Para saber mais, confira a seção [Structs](#) da [Especificação da linguagem C#](#).

Para mais informações sobre `struct` recursos, consulte as seguintes notas sobre a proposta de recurso:

- [C# 7.2 - Readonly structs](#)
- [C# 8 - Membros da instância ReadOnly](#)
- [C# 10 - Constructos de struct sem parâmetros](#)
- [C# 10 - Permitir with expressão em structs](#)
- [C# 10 - Registrar structs](#)
- [C# 11 - Structs de padrão automático](#)

## Confira também

- [Referência de C#](#)
- [Sistema de tipos do C#](#)
- [Diretrizes de design – Escolher entre classe e struct](#)
- [Diretrizes de design – Design de struct](#)

# Tipos de tupla (Referência do C#)

Artigo • 05/06/2023

O recurso de *tuplas* fornece sintaxe concisa para agrupar vários elementos de dados em uma estrutura de dados leve. O exemplo a seguir mostra como você pode declarar uma variável de tupla, inicializá-la e acessar os membros de dados:

C#

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

Conforme mostrado no exemplo anterior, para definir um tipo de tupla, especifique os tipos de todos os membros de dados e, opcionalmente, os [nomes de campo](#). Você não pode definir métodos em um tipo de tupla, mas pode usar os métodos fornecidos pelo .NET, conforme mostrado no exemplo a seguir:

C#

```
(double, int) t = (4.5, 3);
Console.WriteLine(t.ToString());
Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}.");
// Output:
// (4.5, 3)
// Hash code of (4.5, 3) is 718460086.
```

Os tipos de tupla dão suporte a [operadores de igualdade](#) `==` e `!=`. Para obter mais informações, confira a seção [Igualdade de tupla](#).

Os tipos de tupla são [tipos de valor](#). Os elementos de tupla são campos públicos. Isso torna as tuplas tipos de valor *mutáveis*.

Você pode definir as tuplas com um grande número arbitrário de elementos:

C#

```
var t =
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18,
```

```
19, 20, 21, 22, 23, 24, 25, 26);
Console.WriteLine(t.Item26); // output: 26
```

## Casos de uso das tuplas

Um dos casos de uso mais comuns das tuplas é como tipo de retorno do método. Ou seja, em vez de definir os parâmetros do método `out`, você pode agrupar os resultados do método em um tipo de retorno de tupla, conforme mostrado no exemplo a seguir:

C#

```
var xs = new[] { 4, 7, 9 };
var limits = FindMinMax(xs);
Console.WriteLine($"Limits of [{string.Join(" ", xs)}] are {limits.min} and
{limits.max}");
// Output:
// Limits of [4 7 9] are 4 and 9

var ys = new[] { -9, 0, 67, 100 };
var (minimum, maximum) = FindMinMax(ys);
Console.WriteLine($"Limits of [{string.Join(" ", ys)}] are {minimum} and
{maximum}");
// Output:
// Limits of [-9 0 67 100] are -9 and 100

(int min, int max) FindMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a
null or empty array.");
    }

    // Initialize min to MaxValue so every value in the input
    // is less than this initial value.
    var min = int.MaxValue;
    // Initialize max to MinValue so every value in the input
    // is greater than this initial value.
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
}
```

```
    return (min, max);  
}
```

Conforme mostrado no exemplo anterior, você pode trabalhar diretamente com a instância de tupla retornada ou [desconstruí-la](#) em variáveis diferentes.

Você também pode usar tipos de tupla, em vez de [tipos anônimos](#), por exemplo, em consultas LINQ. Para obter mais informações, confira [Como escolher entre tipos anônimos e tipos de tupla](#).

Normalmente, você usa as tuplas para agrupar elementos de dados ligeiramente relacionados. Em APIs públicas, defina uma [classe](#) ou um tipo de [estrutura](#).

## Nomes de campo de tupla

Você especifica explicitamente os nomes de campos de tupla em uma expressão de inicialização de tupla ou na definição de um tipo de tupla, conforme mostrado no exemplo a seguir:

C#

```
var t = (Sum: 4.5, Count: 3);  
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");  
  
(double Sum, int Count) d = (4.5, 3);  
Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");
```

Se você não especificar um nome de campo, ele poderá ser inferido pelo nome da variável correspondente em uma expressão de inicialização de tupla, conforme mostrado no exemplo a seguir:

C#

```
var sum = 4.5;  
var count = 3;  
var t = (sum, count);  
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

Isso é chamado de inicializadores de projeção de tupla. O nome de uma variável não é projetado em um nome de campo de tupla nos seguintes casos:

- O nome candidato é um nome membro de um tipo de tupla, por exemplo, `Item3`, `ToString` ou `Rest`.

- O nome candidato é uma duplicata de outro nome de campo de tupla, explícito ou implícito.

Nos casos anteriores, especifique explicitamente o nome de um campo ou acesse um campo pelo nome padrão.

Os nomes padrão de campos de tupla são `Item1`, `Item2`, `Item3` e assim por diante. Você sempre pode usar o nome padrão de um campo, mesmo quando um nome de campo for especificado explicitamente ou inferido, conforme mostrado no exemplo a seguir:

C#

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
Console.WriteLine($"The 3rd element is {t.Item3}.");
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

As comparações de [atribuição de tupla](#) e [igualdade de tupla](#) não levam em conta os nomes de campo.

No tempo de compilação, o compilador substitui os nomes de campo não padrão pelos nomes padrão correspondentes. Como resultado, os nomes de campo especificados explicitamente ou inferidos não ficam disponíveis no tempo de execução.

### 💡 Dica

Habilite a regra de estilo de código do .NET [IDE0037](#) para definir uma preferência por nomes de campo de tupla inferidos ou explícitos.

Começando no C# 12, você pode especificar um alias para um tipo de tupla com uma [diretiva using](#). O seguinte exemplo adiciona um alias `global using` para um tipo de tupla com dois valores inteiros para um valor permitido `Min` e `Max`:

C#

```
global using BandPass = (int Min, int Max);
```

Depois de declarar o alias, você pode usar o nome `BandPass` como um alias para esse tipo de tupla:

C#

```
BandPass bracket = (40, 100);
Console.WriteLine($"The bandpass filter is {bracket.Min} to {bracket.Max}");
```

Um alias não introduz um novo *tipo*, mas cria apenas um sinônimo para um tipo existente. Você pode desconstruir uma tupla declarada com o alias `BandPass` da mesma forma que pode fazer com o tipo de tupla subjacente:

C#

```
(int a, int b) = bracket;
Console.WriteLine($"The bracket is {a} to {b}");
```

Assim como na atribuição ou desconstrução de tupla, os nomes de membro de tupla não precisam corresponder; os tipos precisam.

Da mesma forma, um segundo alias com os mesmos tipos de paridade e membro pode ser usado de modo intercambiável com o alias original. Você pode declarar um segundo alias:

C#

```
using Range = (int Minimum, int Maximum);
```

Você pode atribuir uma tupla `Range` a uma tupla `BandPass`. Assim como acontece com toda a atribuição de tupla, os nomes de campo não precisam corresponder, apenas os tipos e a paridade.

C#

```
Range r = bracket;
Console.WriteLine($"The range is {r.Minimum} to {r.Maximum}");
```

Um alias para um tipo de tupla fornece mais informações semânticas quando você usa tuplas. Ele não introduz um novo tipo. Para fornecer segurança de tipo, você deve declarar um `record` posicional.

## Atribuição e desconstrução de tupla

O C# dá suporte à atribuição entre tipos de tupla que atendem às duas seguintes condições a seguir:

- ambos os tipos de tupla têm o mesmo número de elementos
- para cada posição de tupla, o tipo do elemento de tupla à direita é o mesmo ou pode ser convertido implicitamente no tipo do elemento de tupla à esquerda correspondente

Os valores do elemento de tupla são atribuídos seguindo a ordem dos elementos de tupla. Os nomes de campos de tupla são ignorados e não atribuídos, conforme mostrado no exemplo a seguir:

C#

```
(int, double) t1 = (17, 3.14);
(double First, double Second) t2 = (0.0, 1.0);
t2 = t1;
Console.WriteLine(${nameof(t2)}: {t2.First} and {t2.Second}");
// Output:
// t2: 17 and 3.14

(double A, double B) t3 = (2.0, 3.0);
t3 = t2;
Console.WriteLine(${nameof(t3)}: {t3.A} and {t3.B}");
// Output:
// t3: 17 and 3.14
```

Você também pode usar o operador de atribuição `=` para *desconstruir* uma instância de tupla em variáveis diferentes. Você pode fazer isso de várias maneiras:

- Use a palavra-chave `var` fora dos parênteses para declarar variáveis digitadas implicitamente e permita que o compilador infira os tipos:

C#

```
var t = ("post office", 3.6);
var (destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance}
kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- Declare explicitamente o tipo de cada campo entre parênteses:

C#

```
var t = ("post office", 3.6);
(string destination, double distance) = t;
Console.WriteLine($"Distance to {destination} is {distance}
kilometers.");
```

```
// Output:  
// Distance to post office is 3.6 kilometers.
```

- Declare alguns tipos explicitamente e outros tipos implicitamente (com `var`) dentro dos parênteses:

C#

```
var t = ("post office", 3.6);  
(var destination, double distance) = t;  
Console.WriteLine($"Distance to {destination} is {distance}  
kilometers.");  
// Output:  
// Distance to post office is 3.6 kilometers.
```

- Use as variáveis existentes:

C#

```
var destination = string.Empty;  
var distance = 0.0;  
  
var t = ("post office", 3.6);  
(destination, distance) = t;  
Console.WriteLine($"Distance to {destination} is {distance}  
kilometers.");  
// Output:  
// Distance to post office is 3.6 kilometers.
```

O destino de uma expressão de desconstrução pode incluir variáveis existentes e variáveis declaradas na declaração de desconstrução.

Você também pode combinar a desconstrução com [padrões correspondentes](#) para inspecionar as características dos campos em uma tupla. O exemplo a seguir realiza loops em vários inteiros e imprime aqueles que são divisíveis por 3. Ele desconstrói o resultado da tupla de `Int32.DivRem` e faz a correspondência com um `Remainder` de 0:

C#

```
for (int i = 4; i < 20; i++)  
{  
    if (Math.DivRem(i, 3) is ( Quotient: var q, Remainder: 0 ))  
    {  
        Console.WriteLine($"{i} is divisible by 3, with quotient {q}");  
    }  
}
```

Para obter mais informações sobre a desconstrução de tuplas e outros tipos, confira [Desconstrução de tuplas e outros tipos](#).

## Igualdade de tupla

Os tipos de tupla dão suporte aos `==` operadores e `!=`. Esses operadores comparam membros do operando à esquerda com os membros correspondentes do operando à direita, seguindo a ordem dos elementos de tupla.

C#

```
(int a, byte b) left = (5, 10);
(long a, int b) right = (5, 10);
Console.WriteLine(left == right); // output: True
Console.WriteLine(left != right); // output: False

var t1 = (A: 5, B: 10);
var t2 = (B: 5, A: 10);
Console.WriteLine(t1 == t2); // output: True
Console.WriteLine(t1 != t2); // output: False
```

Conforme mostrado no exemplo anterior, as operações `==` e `!=` não levam em conta os nomes de campo de tupla.

Duas tuplas são comparáveis quando ambas as seguintes condições são atendidas:

- Ambas as tuplas têm o mesmo número de elementos. Por exemplo, `t1 != t2` não é compilado, se `t1` e `t2` tiverem números diferentes de elementos.
- Para cada posição de tupla, os elementos correspondentes dos operandos de tupla à esquerda e à direita são comparáveis aos operadores `==` e `!=`. Por exemplo, `(1, (2, 3)) == ((1, 2), 3)` não é compilado porque `1` não é comparável a `(1, 2)`.

Os operadores `==` e `!=` comparam as tuplas em curto-circuito. Ou seja, uma operação é interrompida assim que atende a um par de elementos diferentes ou que atinge as extremidades das tuplas. No entanto, antes de qualquer comparação, *todos* os elementos de tupla são avaliados, conforme mostrado no exemplo a seguir:

C#

```
Console.WriteLine((Display(1), Display(2)) == (Display(3), Display(4)));

int Display(int s)
{
    Console.WriteLine(s);
    return s;
```

```
}
```

// Output:

```
// 1
// 2
// 3
// 4
// False
```

## Tuplas como parâmetros de saída

Normalmente, você refatora um método que tem parâmetros `out` em um método que retorna uma tupla. No entanto, há casos em que um parâmetro `out` pode ser de um tipo de tupla. O exemplo a seguir mostra como trabalhar com tuplas como parâmetros `out`:

C#

```
var limitsLookup = new Dictionary<int, (int Min, int Max)>()
{
    [2] = (4, 10),
    [4] = (10, 20),
    [6] = (0, 23)
};

if (limitsLookup.TryGetValue(4, out (int Min, int Max) limits))
{
    Console.WriteLine($"Found limits: min is {limits.Min}, max is
{limits.Max}");
}
// Output:
// Found limits: min is 10, max is 20
```

## Tuplas versus `System.Tuple`

As tuplas do C#, que são respaldadas por tipos `System.ValueTuple`, são diferentes das tuplas representadas por tipos `System.Tuple`. As principais diferenças são as seguintes:

- Os tipos `System.ValueTuple` são **tipos de valor**. Os `System.Tuple` tipos são **tipos de referência**.
- Os tipos `System.ValueTuple` são mutáveis. Os tipos `System.Tuple` são imutáveis.
- Os membros de dados dos tipos `System.ValueTuple` são campos. Os membros de dados dos tipos `System.Tuple` são propriedades.

# Especificação da linguagem C#

Para obter mais informações, consulte:

- [Tipos de tupla](#)
- [Operadores de igualdade de tupla](#)

## Confira também

- [Referência de C#](#)
- [Tipos de valor](#)
- [Como escolher entre tipos anônimos e tipos de tupla](#)
- [System.ValueTuple](#)

# Tipos que permitem valor nulo (referência do C#)

Artigo • 07/04/2023

Um *tipo de valor anulável* `T?` representa todos os valores do `tipo de valor` `T` subjacente e um valor `null` adicional. Por exemplo, você pode atribuir qualquer um dos seguintes três valores a uma variável `bool? : true, false` ou `null`. Um tipo de valor `T` subjacente não pode ser um tipo de valor anulável em si.

Qualquer tipo de valor anulável é uma instância da estrutura genérica `System.Nullable<T>`. Você pode se referir a um tipo de valor anulável com um tipo subjacente `T` em qualquer um dos seguintes formulários intercambiáveis: `Nullable<T>` ou `T?`.

Normalmente, você usa um tipo de valor anulável quando precisa representar o valor indefinido de um tipo de valor subjacente. Por exemplo, uma variável booleana ou `bool`, só pode ser `true` ou `false`. No entanto, em alguns aplicativos, um valor variável pode estar indefinido ou ausente. Por exemplo, um campo de dados pode conter `true` ou `false`, ou pode não conter nenhum valor, ou seja, `NULL`. Você pode usar o tipo `bool?` nesse cenário.

## Declaração e atribuição

Como um tipo de valor é implicitamente conversível para o tipo de valor anulável correspondente, você pode atribuir um valor a uma variável de um tipo de valor anulável, como faria com o tipo de valor subjacente. Você também pode atribuir o valor `null`. Por exemplo:

C#

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

O valor padrão de um tipo de valor anulável representa `null`, ou seja, é uma instância cuja propriedade `Nullable<T>.HasValue` retorna `false`.

## Exame de uma instância de um tipo de valor anulável

Você pode usar o [operador `is` com um padrão de tipo](#) para examinar uma instância de um tipo de valor anulável para `null` e recuperar um valor de um tipo subjacente:

C#

```
int? a = 42;
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Output:
// a is 42
```

Você sempre pode usar as seguintes propriedades somente leitura para examinar e obter um valor de uma variável de tipo de valor anulável:

- `Nullable<T>.HasValue` indica se uma instância de um tipo de valor anulável tem um valor do tipo subjacente dela.
- `Nullable<T>.Value` obtém o valor de um tipo subjacente quando `HasValue` é `true`. Quando `HasValue` é `false`, a propriedade `Value` gera uma `InvalidOperationException`.

O seguinte exemplo usa a propriedade `HasValue` para testar se a variável contém um valor antes de exibi-lo:

C#

```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
```

```
}
```

// Output:  
// b is 10

Você também pode comparar uma variável de um tipo de valor anulável com `null` em vez de usar a propriedade `HasValue`, como mostra o seguinte exemplo:

C#

```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Output:
// c is 7
```

## Conversão de um tipo de valor anulável para um tipo subjacente

Se você quiser atribuir um valor de um tipo de valor anulável a uma variável de tipo de valor não anulável, talvez seja necessário especificar o valor a ser atribuído no lugar de `null`. Use o [operador de avaliação de nulo ??](#) para fazer isso (você também pode usar o método `Nullable<T>.GetValueOrDefault(T)` para a mesma finalidade):

C#

```
int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}"); // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}"); // output: d is -1
```

Se quiser usar o valor [padrão](#) do tipo de valor subjacente no lugar de `null`, use o método `Nullable<T>.GetValueOrDefault()`.

Você também pode converter explicitamente um tipo de valor anulável em um tipo não anulável, como mostra o seguinte exemplo:

C#

```
int? n = null;

//int m1 = n;      // Doesn't compile
int n2 = (int)n; // Compiles, but throws an exception if n is null
```

Em tempo de execução, se o valor de um tipo de valor anulável for `null`, a conversão explícita vai gerar uma [InvalidOperationException](#).

Um tipo de valor não anulável `T` é implicitamente conversível para o tipo de valor anulável correspondente `T?`.

## Operadores suspensos

Os [operadores](#) unários e binários predefinidos ou quaisquer operadores sobrecarregados com suporte por um tipo de valor `T` também têm suporte pelo tipo de valor anulável correspondente `T?`. Esses operadores, também conhecidos como *operadores suspensos*, vão gerar `null` se um ou ambos os operadores forem `null`; caso contrário, o operador usará os valores contidos dos operadores dele para calcular o resultado. Por exemplo:

C#

```
int? a = 10;
int? b = null;
int? c = 10;

a++;          // a is 11
a = a * c;   // a is 110
a = a + b;   // a is null
```

### Observação

Para o tipo `bool?`, os operadores predefinidos `&` e `|` não seguirão as regras descritas nessa seção: o resultado de uma avaliação do operador poderá ser não nulo, mesmo quando um dos operandos for `null`. Para obter mais informações, confira a seção [Operadores lógicos booleanos anuláveis](#) do artigo [Operadores lógicos booleanos](#).

Para os [operadores de comparação](#) `<`, `>`, `<=` e `>=`, se um ou ambos os operandos forem `null`, o resultado será `false`; caso contrário, os valores contidos de operandos serão

comparados. Não presuma que como uma comparação (por exemplo, `<=`) retorna `false`, a comparação oposta (`>`) retorna `true`. O exemplo a seguir mostra que 10

- nem maior ou igual a `null`
- nem menor que `null`

C#

```
int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True
```

Para o [operador de igualdade](#) `==`, se ambos os operandos forem `null`, o resultado será `true`, se apenas um dos operandos for `null`, o resultado será `false`; caso contrário, os valores contidos dos operandos serão comparados.

Para o [operador de desigualdade](#) `!=`, se ambos os operandos forem `null`, o resultado será `false`, se apenas um dos operandos for `null`, o resultado será `true`; caso contrário, os valores contidos dos operandos serão comparados.

Se houver uma [conversão definida pelo usuário](#) entre dois tipos de valor, a mesma conversão também poderá ser usada entre os tipos de valor anuláveis correspondentes.

## Conversão boxing e unboxing

Uma instância de um tipo de valor anulável `T?` é [demarcada](#) da seguinte maneira:

- Se [HasValue](#) retorna `false`, a referência nula é produzida.
- Se [HasValue](#) retornar `true`, o valor correspondente do tipo de valor subjacente `T` será demarcado, não a instância de [Nullable<T>](#).

Você pode desfazer um valor demarcado de um tipo de valor `T` para o tipo de valor anulável correspondente `T?`, como mostra o seguinte exemplo:

```
C#  
  
int a = 41;  
object aBoxed = a;  
int? aNullable = (int?)aBoxed;  
Console.WriteLine($"Value of aNullable: {aNullable}");  
  
object aNullableBoxed = aNullable;  
if (aNullableBoxed is int valueOfA)  
{  
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");  
}  
// Output:  
// Value of aNullable: 41  
// aNullableBoxed is boxed int: 41
```

## Como identificar um tipo de valor anulável

O seguinte exemplo mostra como determinar se uma instância `System.Type` representa um tipo de valor anulável construído, ou seja, o tipo `System.Nullable<T>` com um parâmetro de tipo especificado `T`:

```
C#  
  
Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non-  
nullable")} value type");  
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non-  
nullable")} value type");  
  
bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;  
  
// Output:  
// int? is nullable value type  
// int is non-nullable value type
```

Como mostra o exemplo, você usa o operador `typeof` para criar uma instância `System.Type`.

Se você quiser determinar se uma instância é de um tipo de valor anulado, não use o método `Object.GetType` para fazer com que uma instância `Type` seja testada com o código anterior. Quando você chama o método `Object.GetType` em uma instância de um tipo de valor anulável, a instância é [demarcada para Object](#). Como a conversão boxing de uma instância não nula de um tipo de valor anulável é equivalente à

conversão boxing de um valor do tipo subjacente, `GetType` retorna uma instância `Type` que representa o tipo subjacente de um tipo de valor anulável:

```
C#
```

```
int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32
```

Além disso, não use o operador `is` para determinar se uma instância é de um tipo de valor anulável. Como mostra o seguinte exemplo, não é possível distinguir tipos de uma instância de tipo de valor anulável e a instância de tipo subjacente dela com o operador `is`:

```
C#
```

```
int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?
```

Em vez disso, use o operador `Nullable.GetUnderlyingType` do primeiro exemplo e o operador `typeof` para verificar se uma instância é de um tipo de valor anulável.

#### ⓘ Observação

Os métodos descritos nesta seção não são aplicáveis no caso de **tipos de referência anuláveis**.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- Tipos anuláveis
- Operadores suspensos
- Conversões anuláveis implícitas
- Conversões anuláveis explícitas
- Operadores de conversão suspensos

## Confira também

- Referência de C#
- O que exatamente "levantado" significa?
- System.Nullable<T>
- System.Nullable
- Nullable.GetUnderlyingType
- Tipos de referência anuláveis

# Tipos de referência (referência de C#)

Artigo • 07/04/2023

Há dois tipos em C#: tipos de referência e valor. Variáveis de tipos de referência armazenam referências em seus dados (objetos) enquanto que variáveis de tipos de valor contém diretamente seus dados. Com tipos de referência, duas variáveis podem fazer referência ao mesmo objeto; portanto, operações em uma variável podem afetar o objeto referenciado pela outra variável. Com tipos de valor, cada variável tem a própria cópia dos dados e as operações em uma variável não podem afetar a outra (exceto no caso das variáveis de parâmetros `in`, `ref` e `out`; confira o modificador de parâmetro [in](#), [ref](#) e [out](#)).

As seguintes palavras-chaves são usadas para declarar tipos de referência:

- [class](#)
- [interface](#)
- [delegate](#)
- [record](#)

O C# também oferece os seguintes tipos de referência internos:

- [dinâmico](#)
- [object](#)
- [cadeia de caracteres](#)

## Confira também

- [Referência de C#](#)
- [Palavras-chave de C#](#)
- [Tipos de ponteiro](#)
- [Tipos de valor](#)

# Tipos de referência internos (Referência de C#)

Artigo • 10/05/2023

C# tem muitos tipos de referência internas. Eles têm palavras-chave ou operadores que são sinônimos de um tipo na biblioteca do .NET.

## O tipo de objeto

O tipo `object` é um alias de `System.Object` no .NET. No sistema de tipos unificado do C#, todos os tipos, predefinidos e definidos pelo usuário, tipos de referência e tipos de valor, herdam direta ou indiretamente de `System.Object`. Você pode atribuir valores de qualquer tipo a variáveis do tipo `object`. Qualquer variável `object` pode ser atribuída ao seu valor padrão usando o literal `null`. Quando uma variável de um tipo de valor é convertida para um objeto, ela é chamada de *boxed*. Quando uma variável do objeto do tipo `object` é convertida para um tipo de valor, ela é chamada de *unboxed*. Para obter mais informações, consulte [Conversões boxing e unboxing](#).

## O tipo de cadeia de caracteres

O tipo `string` representa uma sequência de zero ou mais caracteres Unicode. `string` é um alias de `System.String` no .NET.

Embora `string` seja um tipo de referência, os [operadores de igualdade == e !=](#) são definidos para comparar os valores de objetos `string`, não referências. A igualdade baseada em valor torna o teste de igualdade de cadeia de caracteres mais intuitivo. Por exemplo:

C#

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

O exemplo anterior mostra "True" e, em seguida, "False" porque os conteúdos das cadeias de caracteres são equivalentes, mas `a` e `b` não fazem referência à mesma instância da cadeia de caracteres.

O [operador +](#) concatena as cadeias de caracteres:

C#

```
string a = "good " + "morning";
```

O código anterior cria um objeto de cadeia de caracteres que contém “good morning”.

Cadeias de caracteres são *imutável* – o conteúdo de um objeto de cadeia de caracteres não pode ser alterado depois que o objeto é criado. Por exemplo, quando você escreve esse código, o compilador, na verdade, cria um objeto de cadeia de caracteres para manter a nova sequência de caracteres e esse novo objeto é atribuído a `b`. A memória que tiver sido alocada para `b` (quando ela continha a cadeia de caracteres “h”) será elegível para a coleta de lixo.

C#

```
string b = "h";
b += "ello";
```

O [operador \[\]](#) pode ser usado para acesso somente leitura a caracteres individuais de uma cadeia de caracteres. Os valores de índice válidos começam em `0` e devem ser menores do que o comprimento da cadeia de caracteres:

C#

```
string str = "test";
char x = str[2]; // x = 's';
```

De maneira semelhante, o operador `[]` também pode ser usado para iterar em cada caractere em uma cadeia de caracteres:

C#

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
    Console.Write(str[i] + " ");
}
// Output: t e s t
```

## Literais de cadeia de caracteres

Literais de cadeia de caracteres são do tipo `string` e podem ser escritos de duas formas, entre aspas e literalmente.

*Literais de cadeia de caracteres brutos* estão disponíveis a partir do C# 11. literais de cadeias de caracteres brutos podem conter texto arbitrário sem precisar de sequências de escape. Literais de cadeia de caracteres bruta podem incluir espaço em branco e novas linhas, aspas embutidas e outros caracteres especiais. Literais de cadeia de caracteres brutos são colocados em um mínimo de três aspas duplas ("""""):

C#

```
"""
This is a multi-line
    string literal with the second line indented.
"""
```

Você pode até incluir uma sequência de três (ou mais) caracteres de aspas duplas. Se o texto exigir uma sequência inserida de aspas, você iniciará e terminará o literal de cadeia de caracteres bruto com mais aspas, conforme necessário:

C#

```
"""
This raw string literal has four """", count them: """ four!
embedded quote characters in a sequence. That's why it starts and ends
with five double quotes.

You could extend this example with as many embedded quotes as needed for
your text.
"""
```

Literais de cadeia de caracteres brutos normalmente têm as sequências de aspas iniciais e finais em linhas separadas do texto inserido. Literais de cadeia de caracteres brutos de várias linhas dão suporte a cadeias de caracteres que são as próprias cadeias de caracteres entre aspas:

C#

```
var message = """
"This is a very important message."
""";
Console.WriteLine(message);
// output: "This is a very important message."
```

Quando as aspas inicial e final estão em linhas separadas, as linhas novas que seguem a cotação de abertura e a citação final não são incluídas no conteúdo final. A sequência

de aspas de fechamento dita a coluna mais à esquerda para o literal da cadeia de caracteres. Você pode recuar um literal de cadeia de caracteres bruto para corresponder ao formato de código geral:

```
C#  
  
var message = """  
    "This is a very important message."  
    """;  
Console.WriteLine(message);  
// output: "This is a very important message."  
// The leftmost whitespace is not part of the raw string literal
```

As colunas à direita da sequência de aspas final são preservadas. Esse comportamento permite cadeias de caracteres brutas para formatos de dados como JSON, YAML ou XML, conforme mostrado no exemplo a seguir:

```
C#  
  
var json= """  
{  
    "prop": 0  
}  
""";
```

O compilador emitirá um erro se qualquer uma das linhas de texto se estender à esquerda da sequência de aspas de fechamento. As sequências de aspas de abertura e fechamento podem estar na mesma linha, desde que o literal da cadeia de caracteres não comece nem termine com um caractere de aspas:

```
C#  
  
var shortText = """He said "hello!" this morning."""";
```

Você pode combinar literais de cadeia de caracteres brutos com [interpolação de cadeia de caracteres](#) para incluir caracteres de citação e chaves na cadeia de caracteres de saída.

Os literais de cadeia de caracteres entre aspas são colocados entre aspas duplas (""):

```
C#  
  
"good morning" // a string literal
```

Os literais de cadeia de caracteres podem conter qualquer literal de caractere. Sequências de escape são incluídas. O exemplo a seguir usa a sequência de escape \\ de barra invertida, \u0066 para a letra f e \n para a nova linha.

C#

```
string a = "\\\u0066\n F";
Console.WriteLine(a);
// Output:
// \f
//  F
```

### ⓘ Observação

O código de escape \udddd (em que dddd é um número de quatro dígitos) representa o caractere Unicode U+ dddd. Os códigos de escape Unicode de oito dígitos também são reconhecidos: \Uddddddddd.

Os [literais de cadeia de caracteres textuais](#) começam com @ e também são colocados entre aspas duplas. Por exemplo:

C#

```
@"good morning" // a string literal
```

A vantagem das cadeias de caracteres textuais é que as sequências de escape *não* são processadas, o que facilita a escrita: Por exemplo, o texto a seguir corresponde a um nome de arquivo do Windows totalmente qualificado:

C#

```
@"c:\Docs\Source\a.txt" // rather than "c:\\Docs\\Source\\a.txt"
```

Para incluir aspas duplas em uma cadeia de caracteres com @, dobre-as:

C#

```
"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

## Cadeia de caracteres UTF-8 literais

As cadeias de caracteres no .NET são armazenadas usando a codificação UTF-16. UTF-8 é o padrão para protocolos Web e outras bibliotecas importantes. A partir do C# 11, você pode adicionar o sufixo `u8` a um literal de string para especificar a codificação UTF-8. Os literais UTF-8 são armazenados como objetos `ReadOnlySpan<byte>`. O tipo natural de um literal de cadeia de caracteres UTF-8 é `ReadOnlySpan<byte>`. Usar um literal de cadeia de caracteres UTF-8 cria uma declaração mais clara do que declarar o `System.ReadOnlySpan<T>` equivalente, conforme mostrado no código a seguir:

C#

```
ReadOnlySpan<byte> AuthWithTrailingSpace = new byte[] { 0x41, 0x55, 0x54,
0x48, 0x20 };
ReadOnlySpan<byte> AuthStringLiteral = "AUTH "u8;
```

Para armazenar um literal de cadeia de caracteres UTF-8 como uma matriz, é necessário o uso de `ReadOnlySpan<T>.ToArray()` para copiar os bytes que contêm o literal para a matriz mutável:

C#

```
byte[] AuthStringLiteral = "AUTH "u8.ToArray();
```

Literais de cadeia de caracteres UTF-8 não são constantes de tempo de compilação; são constantes de runtime. Portanto, eles não podem ser usados como o valor padrão para um parâmetro opcional. Literais de cadeia de caracteres UTF-8 não podem ser combinados com interpolação de cadeia de caracteres. Você não pode usar o token `$` e o sufixo `u8` na mesma expressão de cadeia de caracteres.

## O tipo de delegado

A declaração de um tipo de delegado é semelhante a uma assinatura de método. Ela tem um valor retornado e parâmetros de qualquer tipo:

C#

```
public delegate void MessageDelegate(string message);
public delegate int AnotherDelegate(MyType m, long num);
```

No .NET, os tipos `System.Action` e `System.Func` fornecem definições genéricas para muitos delegados comuns. Você provavelmente não precisa definir novos tipos de delegado personalizado. Em vez disso, é possível criar instâncias dos tipos genéricos fornecidos.

Um `delegate` é um tipo de referência que pode ser usado para encapsular um método nomeado ou anônimo. Representantes são semelhantes a ponteiros de função em C++. No entanto, os representantes são fortemente tipados e seguros. Para aplicativos de representantes, consulte [Representantes](#) e [Representantes genéricos](#). Os representantes são a base dos [Eventos](#). Um delegado pode ser instanciado associando-o a um método nomeado ou anônimo.

O delegado deve ser instanciado com um método ou expressão lambda que tenha um tipo de retorno compatível e parâmetros de entrada. Para obter mais informações sobre o grau de variação permitido na assinatura do método, consulte [Variação em representantes](#). Para uso com métodos anônimos, o delegado e o código a ser associado a ele são declarados juntos.

A combinação de representantes ou a remoção falham com uma exceção de runtime quando os tipos de delegado envolvidos em tempo de execução são diferentes devido à conversão de variante. O seguinte exemplo demonstra uma situação de falha:

```
C#  
  
Action<string> stringAction = str => {};  
Action<object> objectAction = obj => {};  
  
// Valid due to implicit reference conversion of  
// objectAction to Action<string>, but may fail  
// at run time.  
Action<string> combination = stringAction + objectAction;
```

Você pode criar um delegado com o tipo de runtime correto criando um novo objeto delegado. O exemplo a seguir demonstra como essa solução alternativa pode ser aplicada ao exemplo anterior.

```
C#  
  
Action<string> stringAction = str => {};  
Action<object> objectAction = obj => {};  
  
// Creates a new delegate instance with a runtime type of Action<string>.  
Action<string> wrappedObjectAction = new Action<string>(objectAction);  
  
// The two Action<string> delegate instances can now be combined.  
Action<string> combination = stringAction + wrappedObjectAction;
```

A partir do C# 9, você pode declarar [ponteiros de função](#), que usam sintaxe semelhante. Um ponteiro de função usa a instrução `calli` em vez de instanciar um tipo de delegado e chamar o método virtual `Invoke`.

# O tipo dinâmico

O tipo `dynamic` indica que o uso de variável e as referências aos seus membros ignoram a verificação de tipo de tempo de compilação. Em vez disso, essas operações são resolvidas em tempo de execução. O tipo `dynamic` simplifica o acesso a APIs COM, como as APIs de Automação do Office e também às APIs dinâmicas, como bibliotecas do IronPython e ao DOM (Modelo de Objeto do Documento) HTML.

O tipo `dynamic` se comporta como o tipo `object` na maioria das circunstâncias. Em particular, qualquer expressão não nula pode ser convertida no tipo `dynamic`. O tipo `dynamic` é diferente de `object` nas operações que contêm expressões do tipo `dynamic` não resolvidas ou com tipo verificado pelo compilador. O compilador junta as informações sobre a operação em pacotes e, posteriormente, essas informações são usadas para avaliar a operação em tempo de execução. Como parte do processo, as variáveis do tipo `dynamic` são compiladas em variáveis do tipo `object`. Portanto, o tipo `dynamic` existe somente em tempo de compilação e não em tempo de execução.

O exemplo a seguir compara uma variável do tipo `dynamic` a uma variável do tipo `object`. Para verificar o tipo de cada variável no tempo de compilação, coloque o ponteiro do mouse sobre `dyn` ou `obj` nas instruções `WriteLine`. Copie o seguinte código para um editor em que o IntelliSense está disponível. O IntelliSense mostra **dinâmico** para `dyn` e **objeto** para `obj`.

C#

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

As instruções `WriteLine` exibem os tipos de tempo de execução de `dyn` e `obj`. Nesse ponto, ambos têm o mesmo tipo, inteiro. A seguinte saída é produzida:

Console

```
System.Int32
```

```
System.Int32
```

Para ver a diferença entre `dyn` e `obj` em tempo de compilação, adicione as duas linhas a seguir entre as declarações e as instruções `WriteLine` no exemplo anterior.

```
C#
```

```
dyn = dyn + 3;  
obj = obj + 3;
```

Um erro de compilador será relatado em virtude da tentativa de adição de um inteiro e um objeto à expressão `obj + 3`. No entanto, nenhum erro será relatado para `dyn + 3`. A expressão contém `dyn` não é verificada em tempo de compilação, pois o tipo de `dyn` é `dynamic`.

O exemplo a seguir usa `dynamic` em várias declarações. O método `Main` também compara a verificação de tipo em tempo de compilação com a verificação de tipo em tempo de execução.

```
C#
```

```
using System;  
  
namespace DynamicExamples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            ExampleClass ec = new ExampleClass();  
            Console.WriteLine(ec.ExampleMethod(10));  
            Console.WriteLine(ec.ExampleMethod("value"));  
  
            // The following line causes a compiler error because  
ExampleMethod  
            // takes only one argument.  
            //Console.WriteLine(ec.ExampleMethod(10, 4));  
  
            dynamic dynamic_ec = new ExampleClass();  
            Console.WriteLine(dynamic_ec.ExampleMethod(10));  
  
            // Because dynamic_ec is dynamic, the following call to  
ExampleMethod  
            // with two arguments does not produce an error at compile time.  
            // However, it does cause a run-time error.  
            //Console.WriteLine(dynamic_ec.ExampleMethod(10, 4));  
        }  
    }
```

```

    }

    class ExampleClass
    {
        static dynamic _field;
        dynamic Prop { get; set; }

        public dynamic ExampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}

// Results:
// Local variable
// 2
// Local variable

```

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- §8.2.3 O tipo de objeto
- §8.2.4 O tipo de dinâmica
- §8.2.5 O tipo de cadeia de caracteres
- §8.2.8 Tipos delegados
- C# 11 – Literais brutos de cadeia de caracteres
- C# 11 – Literais brutos de cadeia de caracteres

## Confira também

- [Referência de C#](#)
- [Palavras-chave do C#](#)
- [Eventos](#)
- [Usando o tipo dynamic](#)
- [Melhores práticas para o uso de cadeias de caracteres](#)

- Operações básicas de cadeias de caracteres
- Criar novas cadeias de caracteres
- Operadores cast e teste de tipo
- Como converter com segurança usando a correspondência de padrões e os operadores is e as
- Passo a passo: Criando e usando objetos dinâmicos
- System.Object
- System.String
- System.Dynamic.DynamicObject

# Registros (referência em C#)

Artigo • 02/06/2023

A partir do C# 9, você usa o modificador `record` para definir um [tipo de referência](#) que fornece funcionalidade interna para encapsular dados. O C# 10 permite que a sintaxe `record class` como um sinônimo esclareça um tipo de referência e `record struct` defina um [tipo de valor](#) com funcionalidade semelhante.

Quando você declara um [construtor primário](#) em um registro, o compilador gera propriedades públicas para os parâmetros do construtor primário. Os parâmetros do construtor primário para um registro são chamados de *parâmetros posicionais*. O compilador cria *propriedades posicionais* que espelham o construtor primário ou os parâmetros posicionais. O compilador não sintetiza propriedades para parâmetros de construtor primário em tipos que não têm o modificador `record`.

Os dois exemplos a seguir demonstram os tipos de referência `record` (ou `record class`):

C#

```
public record Person(string FirstName, string LastName);
```

C#

```
public record Person
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
};
```

Os dois exemplos a seguir demonstram tipos de valor `record struct`:

C#

```
public readonly record struct Point(double X, double Y, double Z);
```

C#

```
public record struct Point
{
    public double X { get; init; }
    public double Y { get; init; }
```

```
    public double Z { get; init; }  
}
```

Você também pode criar registros com propriedades e campos mutáveis:

C#

```
public record Person  
{  
    public required string FirstName { get; set; }  
    public required string LastName { get; set; }  
};
```

Os structs de registro também podem ser mutáveis, tanto structs de registro posicional quanto structs de registro sem parâmetros posicionais:

C#

```
public record struct DataMeasurement(DateTime TakenAt, double Measurement);
```

C#

```
public record struct Point  
{  
    public double X { get; set; }  
    public double Y { get; set; }  
    public double Z { get; set; }  
}
```

Embora os registros possam ser mutáveis, eles se destinam principalmente a dar suporte a modelos de dados imutáveis. O tipo de registro oferece os seguintes recursos:

- Sintaxe concisa para criar um tipo de referência com propriedades imutáveis
- Comportamento interno útil para um tipo de referência centrado em dados:
  - Igualdade de valor
  - Sintaxe concisa para mutação não destrutiva
  - Formatação interna para exibição
- Suporte para hierarquias de herança

Os exemplos anteriores mostram algumas distinções entre registros que são tipos de referência e registros que são tipos de valor:

- Um `record` ou um `record class` declara um tipo de referência. A palavra-chave `class` é opcional, mas pode adicionar clareza aos leitores. Um `record struct` declara um tipo de valor.

- As propriedades posicionais são *imutáveis* em um `record class` e um `readonly record struct`. Eles são *mutáveis* em um `record struct`.

O restante deste artigo aborda os tipos `record class` e `record struct`. As diferenças são detalhadas em cada seção. Você deve decidir entre um `record class` e um `record struct` semelhante para decidir entre um `class` e um `struct`. O termo *registro* é usado para descrever o comportamento que se aplica a todos os tipos de registro. Tanto `record struct` quanto `record class` são usados para descrever o comportamento que se aplica a apenas struct ou tipos de classe, respectivamente. O tipo `record` foi introduzido no C# 9; os tipos `record struct` foram introduzidos no C# 10.

## Sintaxe posicional para definição de propriedade

Você pode usar parâmetros posicionais para declarar propriedades de um registro e inicializar os valores de propriedade ao criar uma instância:

C#

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

Quando você usa a sintaxe posicional para definição de propriedade, o compilador cria:

- Uma propriedade pública implementada automaticamente para cada parâmetro posicional fornecido na declaração de registro.
  - Para tipos `record` e `readonly record struct`: uma propriedade `init-only`.
  - Para tipos `record struct`: uma propriedade `read-write`.
- Um construtor primário cujos parâmetros correspondem aos parâmetros posicionais na declaração de registro.
- Para tipos de struct de registro, um construtor sem parâmetros que define cada campo como seu valor padrão.
- Um método `Deconstruct` com um parâmetro `out` para cada parâmetro posicional fornecido na declaração de registro. O método desconstrói propriedades definidas usando sintaxe posicional; ele ignora as propriedades definidas usando a sintaxe de propriedade padrão.

Talvez você queira adicionar atributos a qualquer um desses elementos que o compilador cria a partir da definição de registro. Você pode adicionar um *destino* a qualquer atributo aplicado às propriedades do registro posicional. O exemplo a seguir aplica-se a [System.Text.Json.Serialization.JsonPropertyAttribute](#) para cada propriedade do registro `Person`. O destino `property:` indica que o atributo é aplicado à propriedade gerada pelo compilador. Outros valores são `field:` para aplicar o atributo ao campo e `param:` para aplicar o atributo ao parâmetro.

C#

```
/// <summary>
/// Person record type
/// </summary>
/// <param name="FirstName">First Name</param>
/// <param name="LastName">Last Name</param>
/// <remarks>
/// The person type is a positional record containing the
/// properties for the first and last name. Those properties
/// map to the JSON elements "firstName" and "lastName" when
/// serialized or deserialized.
/// </remarks>
public record Person([property: JsonPropertyName("firstName")] string FirstName,
    [property: JsonPropertyName("lastName")] string LastName);
```

O exemplo anterior também mostra como criar comentários de documentação XML para o registro. Você pode adicionar a marca `<param>` para adicionar documentação para os parâmetros do construtor primário.

Se a definição de propriedade gerada automaticamente não for o que você deseja, você poderá definir sua própria propriedade com o mesmo nome. Por exemplo, talvez você queira alterar a acessibilidade ou a mutabilidade ou fornecer uma implementação para o acessador `get` ou `set`. Se você declarar a propriedade em sua origem, deverá inicializá-la do parâmetro posicional do registro. Se sua propriedade for uma propriedade implementada automaticamente, você deverá inicializar a propriedade. Se você adicionar um campo de backup em sua origem, deverá inicializar o campo de suporte. O desconstrutor gerado usa sua definição de propriedade. Por exemplo, o exemplo a seguir declara as propriedades `FirstName` e `LastName` de um registro posicional `public`, mas restringe o parâmetro posicional `Id` a `internal`. Você pode usar essa sintaxe para registros e tipos de struct de registro.

C#

```
public record Person(string FirstName, string LastName, string Id)
{
```

```
internal string Id { get; init; } = Id;  
}  
  
public static void Main()  
{  
    Person person = new("Nancy", "Davolio", "12345");  
    Console.WriteLine(person.FirstName); //output: Nancy  
}
```

Um tipo de registro não precisa declarar nenhuma propriedade posicional. Você pode declarar um registro sem nenhuma propriedade posicional e pode declarar outros campos e propriedades, como no exemplo a seguir:

C#

```
public record Person(string FirstName, string LastName)  
{  
    public string[] PhoneNumbers { get; init; } = Array.Empty<string>();  
};
```

Se você definir propriedades usando a sintaxe de propriedade padrão, mas omitir o modificador de acesso, as propriedades serão implicitamente `private`.

## Imutabilidade

Um *registro posicional* e um *struct de registro de leitura posicional* declaram propriedades `init-only`. Um *struct de registro posicional* declara propriedades `read-write`. Você pode substituir qualquer um desses padrões, conforme mostrado na seção anterior.

A imutabilidade pode ser útil quando você precisa de um tipo centrado em dados para ser seguro para threads ou se você depender de que um código hash permaneça o mesmo em uma tabela de hash. No entanto, a imutabilidade não é apropriada para todos os cenários de dados. O [Entity Framework Core](#), por exemplo, não dá suporte à atualização com tipos de entidade imutáveis.

As propriedades `init-only`, independentemente de serem criadas a partir de parâmetros posicionais (`record class` e `readonly record struct`) ou especificando acessadores `init`, têm *imutabilidade superficial*. Após a inicialização, você não pode alterar o valor das propriedades do tipo valor ou a referência de propriedades de tipo de referência. No entanto, os dados aos quais uma propriedade de tipo de referência se refere podem ser alterados. O exemplo a seguir mostra que o conteúdo de uma propriedade imutável de tipo de referência (uma matriz nesse caso) é mutável:

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    Person person = new("Nancy", "Davolio", new string[1] { "555-1234" });
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-1234

    person.PhoneNumbers[0] = "555-6789";
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-6789
}
```

Os recursos exclusivos para tipos de registro são implementados por métodos sintetizados pelo compilador e nenhum desses métodos compromete a imutabilidade por meio da modificação do estado do objeto. A menos que especificado, os métodos sintetizados são gerados para declarações `record`, `record struct` e `readonly record struct`.

## Igualdade de valor

Se você não substituir nem sobrecarregar métodos de maneira igual, o tipo que você declarar definirá como a igualdade é definida:

- Para tipos `class`, dois objetos serão iguais quando se referirem ao mesmo objeto na memória.
- Para tipos `struct`, dois objetos são iguais se forem do mesmo tipo e armazenarem os mesmos valores.
- Para tipos com o modificador `record` (`record class`, `record struct` e `readonly record struct`), dois objetos são iguais se forem do mesmo tipo e armazenarem os mesmos valores.

A definição de igualdade para um `record struct` é a mesma de um `struct`. A diferença é que, para um `struct`, a implementação está em `ValueType.Equals(Object)` e depende da reflexão. Para registros, a implementação é sintetizada pelo compilador e usa os membros de dados declarados.

A igualdade de referência é necessária para alguns modelos de dados. Por exemplo, o [Entity Framework Core](#) depende da igualdade de referência para garantir que ele use apenas uma instância de um tipo de entidade para o que é conceitualmente uma entidade. Por esse motivo, os registros e os structs de registro não são apropriados para uso como tipos de entidade no Entity Framework Core.

O exemplo a seguir ilustra a igualdade de valores dos tipos de registro:

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

Para implementar a igualdade de valor, o compilador sintetiza os vários métodos, incluindo:

- Uma substituição de [Object.Equals\(Object\)](#). Será um erro se a substituição for declarada explicitamente.  
Esse método é usado como base para o método estático [Object.Equals\(Object, Object\)](#) quando ambos os parâmetros são não nulos.
- Um `virtual`, ou `sealed`, `Equals(R? other)` em que `R` é o tipo de registro. Esse método implementa [IEquatable<T>](#). Esse método pode ser declarado explicitamente.
- Se o tipo de registro for derivado de um tipo de registro base `Base`, `Equals(Base? other)`. Será um erro se a substituição for declarada explicitamente. Se você fornecer sua própria implementação de `Equals(R? other)`, forneça também uma implementação de `GetHashCode`.
- Uma substituição de [Object.GetHashCode\(\)](#). Esse método pode ser declarado explicitamente.
- Substituições de operadores `==` e `!=`. Será um erro se os operadores forem declarados explicitamente.
- Se o tipo de registro for derivado de um tipo de registro base `protected override Type EqualityContract { get; }`. Essa propriedade pode ser declarada

explicitamente. Para obter mais informações, consulte [Igualdade nas hierarquias de herança](#).

O compilador não sintetiza um método quando um tipo de registro tem um método que corresponde à assinatura de um método sintetizado que pode ser declarado explicitamente.

## Mutação não destrutiva

Se você precisar copiar uma instância com algumas modificações, poderá usar uma expressão `with` para obter uma *mutação não destrutiva*. Uma expressão `with` faz uma nova instância de registro que é uma cópia de uma instância de registro existente, com propriedades e campos especificados modificados. Use a sintaxe do [inicializador de objetos](#) para especificar os valores a serem alterados, conforme mostrado no exemplo a seguir:

C#

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers =
    // System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers =
    // System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers =
    // System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

A expressão `with` pode definir propriedades posicionais ou propriedades criadas usando a sintaxe de propriedade padrão. As propriedades explicitamente declaradas devem ter um acessador `init` ou `set` para serem alteradas em uma expressão `with`.

O resultado de uma expressão `with` é uma *cópia superficial*, o que significa que, para uma propriedade de referência, somente a referência a uma instância é copiada. O registro original e a cópia acabam com uma referência à mesma instância.

Para implementar esse recurso para tipos `record class`, o compilador sintetiza um método `clone` e um construtor de cópia. O método `clone` virtual retorna um novo registro inicializado pelo construtor de cópia. Quando você usa uma expressão `with`, o compilador cria um código que chama o método `clone` e define as propriedades especificadas na expressão `with`.

Se você precisar de um comportamento de cópia diferente, poderá escrever seu próprio construtor de cópia em um `record class`. Se você fizer isso, o compilador não sintetizará. Faça o construtor `private` se o registro for `sealed`, caso contrário, faça `protected`. O compilador não sintetiza um construtor de cópia para tipos `record struct`. Você pode escrever, mas o compilador não gerará chamadas para expressões `with`. Os valores do `record struct` são copiados na atribuição.

Você não pode substituir o método `clone` e não pode criar um membro nomeado `Clone` em nenhum tipo de registro. O nome real do método `clone` é gerado pelo compilador.

## Formatação interna para exibição

Os tipos de registro têm um método `ToString` gerado pelo compilador que exibe os nomes e valores de propriedades e campos públicos. O método `ToString` retorna uma cadeia de caracteres do seguinte formato:

```
<nome do tipo de registro> { <nome da propriedade> = <valor>, <nome da propriedade> = <valor>, ...}
```

A cadeia de caracteres impressa para `<value>` é a cadeia de caracteres retornada pelo `ToString()` para o tipo da propriedade. No exemplo a seguir, `ChildNames` é um `System.Array`, em que `ToString` retorna `System.String[]`:

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[] }
```

Para implementar esse recurso em tipos `record class`, o compilador sintetiza um método `PrintMembers` virtual e uma substituição `ToString`. Em tipos `record struct`, esse membro é `private`. A substituição `ToString` cria um objeto `StringBuilder` com o nome do tipo seguido por um colchete de abertura. Ele chama `PrintMembers` para adicionar nomes e valores de propriedade e, em seguida, adiciona o colchete de fechamento. O exemplo a seguir mostra código semelhante ao que a substituição sintetizada contém:

C#

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("Teacher"); // type name
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

Você pode fornecer sua própria implementação de `PrintMembers` ou substituição `ToString`. Exemplos são fornecidos na seção [PrintMembersFormatação em registros derivados](#) posteriormente neste artigo. No C# 10 e posterior, sua implementação `ToString` pode incluir o modificador, o `sealed` que impede que o compilador sintetize uma implementação `ToString` para quaisquer registros derivados. Você pode criar uma representação de cadeia de caracteres consistente em uma hierarquia de tipos `record`. (Os registros derivados ainda terão um método `PrintMembers` gerado para todas as propriedades derivadas.)

## Herança

Esta seção se aplica somente a tipos `record class`.

Um registro pode herdar de outro registro. No entanto, um registro não pode herdar de uma classe, e uma classe não pode herdar de um registro.

## Parâmetros posicionais em tipos de registro derivados

O registro derivado declara parâmetros posicionais para todos os parâmetros no construtor primário do registro base. O registro base declara e inicializa essas

propriedades. O registro derivado não os oculta, mas apenas cria e inicializa propriedades para parâmetros que não são declarados em seu registro base.

O exemplo a seguir ilustra a herança com sintaxe de propriedade posicional:

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

## Igualdade nas hierarquias de herança

Esta seção se aplica a tipos `record class`, mas não a tipos `record struct`. Para que duas variáveis de registro sejam iguais, o tipo de tempo de execução deve ser igual. Os tipos das variáveis de conteúdo podem ser diferentes. A comparação de igualdade herdada é ilustrada no exemplo de código a seguir:

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}
```

No exemplo, todas as variáveis são declaradas como `Person`, mesmo quando a instância é um tipo derivado de `Student` ou `Teacher`. As instâncias têm as mesmas propriedades e os mesmos valores de propriedade. Mas `student == teacher` retorna `False`, embora ambas sejam variáveis do tipo `Person`, e `student == student2` retorna `True`, embora

uma seja uma variável `Person` e outra seja uma variável `Student`. O teste de igualdade depende do tipo de runtime do objeto real, não do tipo declarado da variável.

Para implementar esse comportamento, o compilador sintetiza uma propriedade `EqualityContract` que retorna um objeto `Type` que corresponde ao tipo do registro. O `EqualityContract` permite que os métodos de igualdade comparem o tipo de runtime de objetos quando eles estão verificando a igualdade. Se o tipo base de um registro for `object`, essa propriedade será `virtual`. Se o tipo base for outro tipo de registro, essa propriedade será uma substituição. Se o tipo de registro for `sealed`, essa propriedade será efetivamente `sealed` porque o tipo é `sealed`.

Quando o código compara duas instâncias de um tipo derivado, os métodos de igualdade sintetizados verificam a igualdade de todos os membros de dados da base e dos tipos derivados. O método `GetHashCode` sintetizado usa o método `GetHashCode` de todos os membros de dados declarados no tipo base e no tipo de registro derivado. Os membros de dados de um `record` incluem todos os campos declarados e o campo de backup sintetizado pelo compilador para quaisquer propriedades autoimplementadas.

## Expressões `with` em registros derivados

O resultado de uma expressão `with` tem o mesmo tipo em tempo de execução que o operando da expressão, como mostra o exemplo a seguir: Todas as propriedades do tipo de tempo de execução são copiadas, mas você só pode definir propriedades do tipo de tempo de compilação, como mostra o exemplo a seguir:

```
C#  
  
public record Point(int X, int Y)  
{  
    public int Zbase { get; set; }  
};  
public record NamedPoint(string Name, int X, int Y) : Point(X, Y)  
{  
    public int Zderived { get; set; }  
};  
  
public static void Main()  
{  
    Point p1 = new NamedPoint("A", 1, 2) { Zbase = 3, Zderived = 4 };  
  
    Point p2 = p1 with { X = 5, Y = 6, Zbase = 7 }; // Can't set Name or  
    // Zderived  
    Console.WriteLine(p2 is NamedPoint); // output: True  
    Console.WriteLine(p2);  
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = A, Zderived = 4  
}
```

```

    Point p3 = (NamedPoint)p1 with { Name = "B", X = 5, Y = 6, Zbase = 7,
    Zderived = 8 };
    Console.WriteLine(p3);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = B, Zderived = 8
}
}

```

## Formação PrintMembers em registros derivados

O método sintetizado `PrintMembers` de um tipo de registro derivado chama a implementação base. O resultado é que todas as propriedades públicas e campos de tipos derivados e base estão incluídos na saída `ToString`, conforme mostrado no exemplo a seguir:

C#

```

public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}

```

Você pode fornecer sua própria implementação do método `PrintMembers`. Se fizer isso, use a seguinte assinatura:

- Para um registro `sealed` que deriva de `object` (não declara um registro base):
`private bool PrintMembers(StringBuilder builder);`
- Para um registro `sealed` que deriva de outro registro (observe que o tipo delimitador é `sealed`, portanto, o método é efetivamente `sealed`): `protected override bool PrintMembers(StringBuilder builder);`
- Para um registro que não é `sealed` e que não deriva de objeto: `protected virtual bool PrintMembers(StringBuilder builder);`
- Para um registro que não é `sealed` e que não deriva de outro registro: `protected override bool PrintMembers(StringBuilder builder);`

Aqui está um exemplo de código que substitui os métodos sintetizados `PrintMembers`, um para um tipo de registro que deriva de objeto e outro para um tipo de registro que deriva de outro registro:

C#

```
public abstract record Person(string FirstName, string LastName, string[] PhoneNumbers)
{
    protected virtual bool PrintMembers(StringBuilder stringBuilder)
    {
        stringBuilder.Append($"FirstName = {FirstName}, LastName = {LastName}, ");
        stringBuilder.Append($"PhoneNumber1 = {PhoneNumbers[0]}, PhoneNumber2 = {PhoneNumbers[1]}");
        return true;
    }
}

public record Teacher(string FirstName, string LastName, string[] PhoneNumbers, int Grade)
    : Person(FirstName, LastName, PhoneNumbers)
{
    protected override bool PrintMembers(StringBuilder stringBuilder)
    {
        if (base.PrintMembers(stringBuilder))
        {
            stringBuilder.Append(", ");
        };
        stringBuilder.Append($"Grade = {Grade}");
        return true;
    }
};

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", new string[2] { "555-1234", "555-6789" }, 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, PhoneNumber1 = 555-1234, PhoneNumber2 = 555-6789, Grade = 3 }
}
```

### ⓘ Observação

No C# 10 e posterior, o compilador sintetizará `PrintMembers` em registros derivados mesmo quando um registro base tiver selado o método `ToString`. Você também pode criar sua própria implementação de `PrintMembers`.

## Comportamento do desconstrutor em registros derivados

O método `Deconstruct` de um registro derivado retorna os valores de todas as propriedades posicionais do tipo de tempo de compilação. Se o tipo de variável for um registro base, somente as propriedades do registro base serão desconstruídas, a menos que o objeto seja convertido no tipo derivado. O exemplo a seguir demonstra a chamada de um desconstrutor em um registro derivado.

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    var (firstName, lastName) = teacher; // Doesn't deconstruct Grade
    Console.WriteLine($"{firstName}, {lastName}");// output: Nancy, Davolio

    var (fName, lName, grade) = (Teacher)teacher;
    Console.WriteLine($"{fName}, {lName}, {grade}");// output: Nancy,
    Davolio, 3
}
```

## Restrições genéricas

A palavra-chave `record` é um modificador para um tipo `class` ou `struct`. Adicionar o modificador `record` inclui o comportamento descrito anteriormente neste artigo. Não há nenhuma restrição genérica que exija que um tipo seja um registro. Um `record class` satisfaz a restrição `class`. Um `record struct` satisfaz a restrição `struct`. Para obter mais informações, consulte [Restrições em parâmetros de tipo](#).

## Especificação da linguagem C#

Para obter mais informações, confira a seção [Classes](#) da [Especificação da linguagem C#](#).

Para obter mais informações sobre os recursos adicionados ao C# 9 e posteriores, confira as notas sobre a proposta do recurso a seguir:

- [Registros](#)
- [Setters init-only](#)

- Retornos de covariante

## Confira também

- [Referência de C#](#)
- [Diretrizes de design – Escolher entre classe e struct](#)
- [Diretrizes de design – Design de struct](#)
- [Sistema de tipos do C#](#)
- [Expressão with](#)

# class (Referência de C#)

Artigo • 07/04/2023

Classes são declaradas usando a palavra-chave `class`, conforme mostrado no exemplo a seguir:

C#

```
class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

## Comentários

Somente a herança única é permitida em C#. Em outras palavras, uma classe pode herdar a implementação de apenas uma classe base. No entanto, uma classe pode implementar mais de uma interface. A tabela a seguir mostra exemplos de implementação de interface e herança de classe:

Herança	Exemplo
Nenhum	<code>class ClassA { }</code>
Single	<code>class DerivedClass : BaseClass { }</code>
Nenhuma, implementa duas interfaces	<code>class ImplClass : IFace1, IFace2 { }</code>
Única, implementa uma interface	<code>class ImplDerivedClass : BaseClass, IFace1 { }</code>

Classes que você declara diretamente dentro de um namespace, não aninhadas em outras classes, podem ser [públicas](#) ou [internas](#). As classes são `internal` por padrão.

Os membros da classe, incluindo classes aninhadas, podem ser [públicos](#), [internos protegidos](#), [protegidos](#), [internos](#), [privados](#) ou [protegidos privados](#). Os membros são `private` por padrão.

Para obter mais informações, consulte [Modificadores de Acesso](#).

É possível declarar classes genéricas que têm parâmetros de tipo. Para obter mais informações, consulte [Classes genéricas](#).

Uma classe pode conter declarações dos seguintes membros:

- Construtores
- Constantes
- Fields
- Finalizadores
- Métodos
- Propriedades
- Indexadores
- Operadores
- Eventos
- Representantes
- Classes
- Interfaces
- Tipos de estrutura
- Tipos de enumeração

## Exemplo

O exemplo a seguir demonstra a declaração de métodos, construtores e campos de classe. Ele também demonstra a instanciação de objetos e a impressão de dados de instância. Neste exemplo, duas classes são declaradas. A primeira classe, `Child`, contém dois campos particulares (`name` e `age`), dois construtores públicos e um método público. A segunda classe, `StringTest`, é usada para conter `Main`.

C#

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }
}
```

```

// Constructor:
public Child(string name, int age)
{
    this.name = name;
    this.age = age;
}

// Printing method:
public void PrintChild()
{
    Console.WriteLine("{0}, {1} years old.", name, age);
}
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
   Child #1: Craig, 11 years old.
   Child #2: Sally, 10 years old.
   Child #3: N/A, 0 years old.
*/

```

## Comentários

Observe que, no exemplo anterior, os campos particulares (`name` e `age`) só podem ser acessados por meio dos métodos públicos da classe `Child`. Por exemplo, você não pode imprimir o nome do filho, do método `Main`, usando uma instrução como esta:

C#

```
Console.WriteLine(child1.name); // Error
```

Acessar membros particulares de `child` de `Main` seria possível apenas se `Main` fosse um membro da classe.

Tipos declarados dentro de uma classe sem um modificador de acesso têm o valor padrão de `private`, portanto, os membros de dados neste exemplo ainda seriam `private` se a palavra-chave fosse removida.

Por fim, observe que, para o objeto criado usando o construtor sem parâmetro (`child3`), o campo `age` foi inicializado como zero por padrão.

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Tipos de referência](#)

# interface (Referência de C#)

Artigo • 05/06/2023

Uma interface define um contrato. Qualquer `class` um ou `struct` que implemente esse contrato deve fornecer uma implementação dos membros definidos na interface. Uma interface pode definir uma implementação padrão para membros. Ela também pode definir membros `static` para fornecer uma única implementação para a funcionalidade comum. A partir do C# 11, uma interface pode definir membros `static abstract` ou `static virtual` para declarar que um tipo de implementação deve fornecer os membros declarados. Normalmente, os métodos `static virtual` declaram que uma implementação deve definir um conjunto de [operadores sobrecarregados](#).

No exemplo a seguir, a classe `ImplementationClass` deve implementar um método chamado `SampleMethod` que não tem parâmetros e retorna `void`.

Para obter mais informações e exemplos, consulte [Interfaces](#).

## Exemplo de interface

C#

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

Uma interface pode ser membro de um namespace ou de uma classe. Uma declaração de interface pode conter declarações (assinaturas sem nenhuma implementação) dos seguintes membros:

- Métodos
- Propriedades
- Indexadores
- Eventos

## Membros da interface padrão

Normalmente, essas declarações de membro anteriores não contêm um corpo. Um membro da interface pode declarar um corpo. Os corpos de membro em uma interface representam a *implementação padrão*. Membros com corpos permitem que a interface forneça uma implementação "padrão" para classes e structs que não fornecem uma implementação de substituição. Uma interface pode incluir:

- Constantes
- Operadores
- Construtor estático.
- Tipos aninhados
- Campos estáticos, métodos, propriedades, indexadores e eventos
- Declarações de membro usando a sintaxe de implementação explícita da interface.
- Modificadores de acesso explícitos (o acesso padrão é `public`).

## Membros estáticos abstratos e virtuais

A partir do C# 11, uma interface pode declarar membros `static abstract` e `static virtual` para todos os tipos de membro, exceto campos. As interfaces podem declarar que a implementação de tipos deve definir operadores ou outros membros estáticos. Esse recurso permite que algoritmos genéricos especifiquem comportamento semelhante a número. Você pode ver exemplos nos tipos numéricos no runtime do .NET, como `System.Numerics.INumber<TSelf>`. Essas interfaces definem operadores matemáticos comuns implementados por muitos tipos numéricos. O compilador deve resolver chamadas para métodos `static virtual` e `static abstract` no tempo de compilação. Os métodos `static virtual` e `static abstract` declarados em interfaces não têm um mecanismo de expedição de runtime análogo a métodos `virtual` ou `abstract` declarados em classes. Em vez disso, o compilador usa informações de tipo disponíveis no tempo de compilação. Portanto, os métodos `static virtual` são declarados quase exclusivamente `em interfaces genéricas`. Além disso, a maioria das

interfaces que declaram os métodos `static virtual` ou `static abstract` declaram que um dos parâmetros de tipo deve implementar a interface declarada. Por exemplo, a interface `INumber<T>` declara que `T` deve implementar `INumber<T>`. O compilador usa o argumento de tipo para resolver chamadas aos métodos e operadores declarados na declaração de interface. Por exemplo, o tipo `int` implementa `INumber<int>`. Quando o parâmetro de tipo `T` denota o argumento de tipo `int`, os membros `static` declarados em `int` são invocados. Como alternativa, quando `double` é o argumento de tipo, são invocados os membros `static` declarados no tipo `double`.

### ⓘ Importante

A expedição de método para métodos `static abstract` e `static virtual` declarados em interfaces é resolvida usando o tipo de tempo de compilação de uma expressão. Se o tipo de runtime de uma expressão for derivado de um tipo de tempo de compilação diferente, os métodos estáticos no tipo base (tempo de compilação) serão chamados.

É possível experimentar esse recurso trabalhando com o tutorial sobre [membros abstratos estáticos em interfaces](#).

## Herança de interface

As interfaces podem não conter o estado da instância. Embora os campos estáticos agora sejam permitidos, os campos de instância não são permitidos em interfaces. Não há suporte para [propriedades automáticas de instância](#) em interfaces, pois declarariam implicitamente um campo oculto. Essa regra tem um efeito útil nas declarações de propriedade. Em uma declaração de interface, o código a seguir não declara uma propriedade implementada automaticamente como o faria em um `class` ou `struct`. Em vez disso, ele declara uma propriedade que não tem uma implementação padrão, mas que deve ser implementada em qualquer tipo que implemente a interface:

C#

```
public interface INamed
{
    public string Name {get; set;}
}
```

Uma interface pode herdar de uma ou mais interfaces base. Quando uma interface [substitui um método](#) implementado em uma interface base, ela deve usar a sintaxe de

implementação de interface explícita.

Quando uma lista de tipos base contém uma classe base e interfaces, a classe base deve vir em primeiro na lista.

Uma classe que implementa uma interface pode implementar membros dessa interface explicitamente. Um membro implementado explicitamente não pode ser acessado por meio de uma instância da classe, mas apenas por meio de uma instância da interface. Além disso, os membros da interface padrão só podem ser acessados por meio de uma instância da interface.

Para obter mais informações sobre a implementação explícita, consulte [Implementação de interface explícita](#).

## Exemplo de implementação de interface

O exemplo a seguir demonstra a implementação da interface. Neste exemplo, a interface contém a declaração de propriedade e a classe contém a implementação. Qualquer instância de uma classe que implementa `IPoint` tem propriedades de inteiro `x` e `y`.

```
C#  
  
interface IPoint  
{  
    // Property signatures:  
    int X { get; set; }  
  
    int Y { get; set; }  
  
    double Distance { get; }  
}  
  
class Point : IPoint  
{  
    // Constructor:  
    public Point(int x, int y)  
    {  
        X = x;  
        Y = y;  
    }  
  
    // Property implementation:  
    public int X { get; set; }  
  
    public int Y { get; set; }  
  
    // Property implementation
```

```
    public double Distance =>
        Math.Sqrt(X * X + Y * Y);
}

class MainClass
{
    static void PrintPoint(IPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.X, p.Y);
    }

    static void Main()
    {
        IPoint p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Output: My Point: x=2, y=3
```

## Especificação da linguagem C#

Para obter mais informações, confira a seção [Interfaces](#) da Especificação da linguagem C#, a especificação do recurso para [Membros da interface C# 8 – padrão](#) e a especificação de recursos para [C# 11 – membros abstratos estáticos em interfaces](#)

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Tipos de referência](#)
- [Interfaces](#)
- [Usando propriedades](#)
- [Usando indexadores](#)

# Tipos de referência anulável (referência do C#)

Artigo • 10/05/2023

## ⓘ Observação

Este artigo aborda os tipos de referência anulável. Você também pode declarar os [tipos de valor anulável](#).

Os tipos de referência anulável estão disponíveis em código que aceitou um *contexto com reconhecimento anulável*. Os tipos de referência anulável, os avisos de análise estática nula e o [operador tolerante a nulo](#) são recursos de linguagem opcionais. Todos eles estão desativados por padrão. Um *contexto anulável* é controlado no nível do projeto, usando as configurações de compilação ou em código usando pragmas.

## ⓘ Importante

Todos os modelos de projeto que começam com o .NET 6 (C# 10) habilitam o *contexto anulável* para o projeto. Os projetos criados com modelos anteriores não incluem esse elemento, e esses recursos ficam desativados até que você os habilite no arquivo de projeto ou use pragmas.

Em um contexto com reconhecimento anulável:

- Uma variável de um tipo de referência `T` deve ser inicializada com não nulo e pode nunca ser atribuída a um valor que possa ser `null`.
- Uma variável de um tipo de referência `T?` pode ser inicializada com `null` ou atribuída a `null`, mas é necessário que seja verificada em relação a `null`, antes de desreferenciar.
- Uma variável `m` do tipo `T?` é considerada não nula, quando você aplica o operador tolerante a nulo, como em `m!`.

As distinções entre um tipo de referência não anulável `T` e um tipo de referência anulável `T?` são impostas pela interpretação do compilador das regras anteriores. Uma variável do tipo `T` e uma variável do tipo `T?` são representadas pelo mesmo tipo do .NET. O exemplo a seguir declara uma cadeia de caracteres não anulável e uma cadeia de caracteres anulável e, em seguida, usa o operador tolerante a nulo para atribuir um valor a uma cadeia de caracteres não anulável:

C#

```
string notNull = "Hello";
string? nullable = default;
notNull = nullable!; // null forgiveness
```

As variáveis `notNull` e `nullable` são representadas pelo tipo `String`. Como os tipos não anulável e anulável são armazenados como o mesmo tipo, há vários locais em que o uso de um tipo de referência anulável não é permitido. Em geral, um tipo de referência anulável não pode ser usado como classe base ou interface implementada. Um tipo de referência anulável não pode ser usado nas criações de objeto nem nas expressões de teste de tipo. Um tipo de referência anulável não pode ser o tipo de uma expressão de acesso a membro. Os exemplos a seguir mostram esses constructos:

C#

```
public MyClass : System.Object? // not allowed
{
}

var nullEmpty = System.String?.Empty; // Not allowed
var maybeObject = new object?(); // Not allowed
try
{
    if (thing is string? nullableString) // not allowed
        Console.WriteLine(nullableString);
} catch (Exception? e) // Not Allowed
{
    Console.WriteLine("error");
}
```

## Referências anuláveis e análise estática

Os exemplos na seção anterior ilustram a natureza dos tipos de referência anulável. Os tipos de referência anulável não são novos tipos de classe, mas sim anotações sobre os tipos de referência existentes. O compilador usa essas anotações para ajudar a encontrar possíveis erros de referência nula no código. Não há diferença de runtime entre um tipo de referência não anulável e um tipo de referência anulável. O compilador não adiciona verificações de runtime para tipos de referência não anulável. Os benefícios estão na análise do tempo de compilação. O compilador gera avisos que ajudam a localizar e corrigir possíveis erros nulos no código. Você declara a intenção e o compilador avisa quando o código violar essa intenção.

Em um contexto habilitado para anulável, o compilador executa uma análise estática nas variáveis de qualquer tipo de referência, anulável e não anulável. O compilador rastreia o *estado nulo* de cada variável de referência como *não nulo* ou *talvez nulo*. O estado padrão de uma referência não anulável é *não anulável*. O estado padrão de uma variável de referência anulável é *talvez nulo*.

Os tipos de referência não anulável devem estar sempre seguros para desreferenciar, pois o *estado nulo* é *não nulo*. Para impor essa regra, o compilador emitirá avisos se um tipo de referência não anulável não for inicializado para um valor não nulo. Variáveis locais devem ser atribuídas onde forem declaradas. Cada campo deve receber um valor *não nulo*, em um inicializador de campo ou em cada construtor. O compilador emite avisos quando uma referência não anulável é atribuída a uma referência cujo estado é *talvez nulo*. Em geral, uma referência não anulável é *não nula* e nenhum aviso é emitido quando essas variáveis são desreferenciadas.

### ⓘ Observação

Se você atribuir uma expressão *talvez nula* a um tipo de referência não anulável, o compilador gerará um aviso. O compilador gera avisos para essa variável até que seja atribuída a uma expressão *não nula*.

Os tipos de referência anulável podem ser inicializados ou atribuídos a `null`. Portanto, a análise estática deve determinar se uma variável é *não nula*, antes que seja desreferenciada. Se uma referência anulável for determinada como *talvez nula*, a atribuição a uma variável de referência não anulável gerará um aviso do compilador. A classe a seguir mostra exemplos desses avisos:

C#

```
public class ProductDescription
{
    private string shortDescription;
    private string? detailedDescription;

    public ProductDescription() // Warning! shortDescription not
initialized.
    {

    }

    public ProductDescription(string productDescription) =>
        this.shortDescription = productDescription;

    public void SetDescriptions(string productDescription, string?
details=null)
    {
```

```

        shortDescription = productDescription;
        detailedDescription = details;
    }

    public string GetDescription()
    {
        if (detailedDescription.Length == 0) // Warning! dereference
possible null
        {
            return shortDescription;
        }
        else
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
    }

    public string FullDescription()
    {
        if (detailedDescription == null)
        {
            return shortDescription;
        }
        else if (detailedDescription.Length > 0) // OK, detailedDescription
can't be null.
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
        return shortDescription;
    }
}

```

O snippet a seguir mostra onde o compilador emite avisos ao usar essa classe:

C#

```

string shortDescription = default; // Warning! non-nullable set to null;
var product = new ProductDescription(shortDescription); // Warning! static
analysis knows shortDescription maybe null.

string description = "widget";
var item = new ProductDescription(description);

item.SetDescriptions(description, "These widgets will do everything.");

```

Os exemplos anteriores demonstram como a análise estática do compilador determina o *estado nulo* das variáveis de referência. O compilador aplica regras de linguagem para verificações e atribuições nulas para informar a análise. O compilador não pode fazer suposições sobre a semântica de métodos ou propriedades. Se você chamar métodos que executam verificações nulas, o compilador não poderá saber se esses métodos

afetam o *estado nulo* de uma variável. Existem atributos que você pode adicionar às APIs, para informar o compilador sobre a semântica dos argumentos e os valores retornados. Esses atributos foram aplicados a muitas APIs comuns nas bibliotecas do .NET Core. Por exemplo, [IsNullOrEmpty](#) foi atualizado e o compilador interpreta esse método corretamente como verificação nula. Para obter mais informações sobre os atributos que se aplicam à análise estática de *estado nulo*, confira o artigo sobre [Atributos anuláveis](#).

## Definição de contexto anulável

Existem duas maneiras de controlar o contexto anulável. No nível do projeto, você pode adicionar a configuração do projeto `<Nullable>enable</Nullable>`. Em um único arquivo de origem do C#, você pode adicionar o pragma `#nullable enable` para habilitar o contexto anulável. Confira o artigo sobre [definição de uma estratégia anulável](#). Antes do .NET 6, os novos projetos usam o padrão, `<Nullable>disable</Nullable>`. A partir do .NET 6, os novos projetos incluem o elemento `<Nullable>enable</Nullable>` no arquivo de projeto.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes propostas para a [especificação da linguagem C#](#):

- [Tipos de referência anuláveis](#)
- [Especificação de tipos de referência anuláveis de rascunho](#)

## Confira também

- [Referência de C#](#)
- [Tipos de valor anuláveis](#)

# void (referência de C#)

Artigo • 07/04/2023

Você usa `void` como o tipo de retorno de um [método](#) (ou uma [função local](#)) para especificar que tal método não retorna um valor.

C#

```
public static void Display(IEnumerable<int> numbers)
{
    if (numbers is null)
    {
        return;
    }

    Console.WriteLine(string.Join(" ", numbers));
}
```

Você também pode usar `void` como um tipo referencial para declarar um ponteiro para um tipo desconhecido. Para obter mais informações, consulte [Tipos de ponteiros](#).

Você não pode usar `void` como tipo de variável.

## Confira também

- [Referência de C#](#)
- [System.Void](#)

# Instruções de declaração

Artigo • 23/06/2023

Uma instrução de declaração declara uma nova variável local, constante local ou [variável de referência local](#). Para declarar uma variável local, especifique seu tipo e forneça seu nome. Você pode declarar várias variáveis do mesmo tipo em uma instrução, como mostra o seguinte exemplo:

C#

```
string greeting;
int a, b, c;
List<double> xs;
```

Em uma instrução de declaração, você também pode inicializar uma variável com seu valor inicial:

C#

```
string greeting = "Hello";
int a = 3, b = 2, c = a + b;
List<double> xs = new();
```

Os exemplos anteriores especificam explicitamente o tipo de uma variável. Você também pode deixar o compilador inferir o tipo de uma variável de sua expressão de inicialização. Para fazer isso, use a palavra-chave `var` em vez do nome de um tipo. Para saber mais, confira a seção [Variáveis locais de tipo implícito](#).

Para declarar uma constante local, use a [palavra-chave const](#), como mostra o seguinte exemplo:

C#

```
const string Greeting = "Hello";
const double MinLimit = -10.0, MaxLimit = -MinLimit;
```

Ao declarar uma constante local, você também deve inicializá-la.

Para obter informações sobre variáveis de referência local, confira a seção [Variáveis de referência](#).

## Variáveis locais tipadas implicitamente

Ao declarar uma variável local, você pode deixar o compilador inferir o tipo da variável da expressão de inicialização. Para fazer isso, use a palavra-chave `var` em vez do nome de um tipo:

C#

```
var greeting = "Hello";
Console.WriteLine(greeting.GetType()); // output: System.String

var a = 32;
Console.WriteLine(a.GetType()); // output: System.Int32

var xs = new List<double>();
Console.WriteLine(xs.GetType()); // output:
System.Collections.Generic.List`1[System.Double]
```

Como mostra o exemplo anterior, variáveis locais de tipo implícito são fortemente tipadas.

### ① Observação

Quando você usa `var` no contexto com reconhecimento anulável habilitado e o tipo de uma expressão de inicialização é um tipo de referência, o compilador sempre infere um tipo de referência anulável mesmo que o tipo de uma expressão de inicialização não seja anulável.

Um uso comum de `var` é com uma expressão de invocação de construtor. O uso de `var` permite não repetir um nome de tipo em uma declaração de variável e instanciação de objeto, como mostra o exemplo a seguir:

C#

```
var xs = new List<int>();
```

A partir do C# 9.0, você pode usar uma expressão new do tipo destino como alternativa:

C#

```
List<int> xs = new();
List<int>? ys = new();
```

Ao trabalhar com tipos anônimos, você deve usar variáveis locais tipadas implicitamente. O exemplo a seguir mostra uma expressão de consulta que usa um tipo

anônimo para conter o nome e o número de telefone de um cliente:

C#

```
var fromPhoenix = from cust in customers
                  where cust.City == "Phoenix"
                  select new { cust.Name, cust.Phone };

foreach (var customer in fromPhoenix)
{
    Console.WriteLine($"Name={customer.Name}, Phone={customer.Phone}");
}
```

No exemplo anterior, você não pode especificar explicitamente o tipo da variável `fromPhoenix`. O tipo é `IEnumerable<T>`, mas nesse caso `T` é um tipo anônimo e você não pode fornecer seu nome. É por isso que você precisa usar `var`. Pelo mesmo motivo, você deve usar `var` ao declarar a variável de iteração `customer` na instrução `foreach`.

Para obter mais informações sobre variáveis locais tipadas implicitamente, confira [Variáveis locais tipadas implicitamente](#).

Na correspondência de padrões, a palavra-chave `var` é usada em um [var padrão](#).

## Variáveis de referência

Ao declarar uma variável local e adicionar a palavra-chave `ref` antes do tipo da variável, você declara uma *variável de referência* ou um local `ref`:

C#

```
ref int alias = ref variable;
```

Uma variável de referência é uma variável que se refere a outra variável, que é chamada de *referenciante*. Ou seja, uma variável de referência é um *alias* para seu referenciante. Quando você atribui um valor a uma variável de referência, esse valor é atribuído ao referenciador. Quando você lê o valor de uma variável de referência, o valor do referenciante é retornado. O exemplo a seguir demonstra esse comportamento:

C#

```
int a = 1;
ref int alias = ref a;
Console.WriteLine($"{(a, alias) is ({a}, {alias})}"); // output: (a, alias)
is (1, 1)
```

```
a = 2;
Console.WriteLine($"(a, alias) is ({a}, {alias})"); // output: (a, alias)
is (2, 2)

alias = 3;
Console.WriteLine($"(a, alias) is ({a}, {alias})"); // output: (a, alias)
is (3, 3)
```

Use o `ref` operador de atribuição `= ref` para alterar o referenciante de uma variável de referência, como mostra o exemplo a seguir:

C#

```
void Display(int[] s) => Console.WriteLine(string.Join(" ", s));

int[] xs = { 0, 0, 0 };
Display(xs);

ref int element = ref xs[0];
element = 1;
Display(xs);

element = ref xs[^1];
element = 3;
Display(xs);
// Output:
// 0 0 0
// 1 0 0
// 1 0 3
```

No exemplo anterior, a variável de referência `element` é inicializada como um alias para o primeiro elemento de matriz. Ela é reatribuída `ref` para fazer referência ao último elemento de matriz.

Você pode definir uma variável local `ref readonly`. Você não pode atribuir um valor a uma variável `ref readonly`. No entanto, você pode reatribuir `ref` essa variável de referência, como mostra o exemplo a seguir:

C#

```
int[] xs = { 1, 2, 3 };

ref readonly int element = ref xs[0];
// element = 100; error CS0131: The left-hand side of an assignment must be
// a variable, property or indexer
Console.WriteLine(element); // output: 1
```

```
element = ref xs[^1];
Console.WriteLine(element); // output: 3
```

Você pode atribuir um [retorno de referência](#) a uma variável de referência, como mostra o exemplo a seguir:

C#

```
using System;

public class NumberStore
{
    private readonly int[] numbers = { 1, 30, 7, 1557, 381, 63, 1027, 2550,
511, 1023 };

    public ref int GetReferenceToMax()
    {
        ref int max = ref numbers[0];
        for (int i = 1; i < numbers.Length; i++)
        {
            if (numbers[i] > max)
            {
                max = ref numbers[i];
            }
        }
        return ref max;
    }

    public override string ToString() => string.Join(" ", numbers);
}

public static class ReferenceReturnExample
{
    public static void Run()
    {
        var store = new NumberStore();
        Console.WriteLine($"Original sequence: {store.ToString()}");

        ref int max = ref store.GetReferenceToMax();
        max = 0;
        Console.WriteLine($"Updated sequence: {store.ToString()}");
        // Output:
        // Original sequence: 1 30 7 1557 381 63 1027 2550 511 1023
        // Updated sequence: 1 30 7 1557 381 63 1027 0 511 1023
    }
}
```

No exemplo anterior, o método `GetReferenceToMax` é um método *returns-by-ref*. Ele não retorna o valor máximo em si, mas um retorno de referência que é um alias para o elemento de matriz que contém o valor máximo. O método `Run` atribui um retorno de

referência à variável de referência `max`. Em seguida, atribuindo a `max`, ele atualiza o armazenamento interno da instância `store`. Você também pode definir um método `ref readonly`. Os chamadores de um método `ref readonly` não podem atribuir um valor ao seu retorno de referência.

A variável de iteração da instrução `foreach` pode ser uma variável de referência. Para obter mais informações, consulte a seção sobre a [instrução foreach](#) do artigo [Instruções de iteração](#).

Em cenários críticos de desempenho, o uso de variáveis de referência e retornos pode aumentar o desempenho evitando operações de cópia potencialmente caras.

O compilador garante que uma variável de referência não sobreviva ao seu referenciador e permaneça válida durante todo o tempo de vida. Para saber mais, confira a seção [Contextos seguros de referência](#) da [Especificação da linguagem C#](#).

Para obter informações sobre os campos `ref`, confira a seção [refcampos](#) do artigo [reftipos de estrutura](#).

## ref com escopo

A palavra-chave contextual `scoped` restringe o tempo de vida de um valor. O modificador `scoped` restringe o tempo de vida *ref-safe-to-escape* ou *safe-to-escape*, respectivamente, ao método atual. Efetivamente, adicionar o modificador `scoped` declara que seu código não estenderá o tempo de vida da variável.

Você pode aplicar `scoped` a um parâmetro ou variável local. O modificador `scoped` pode ser aplicado a parâmetros e locais quando o tipo for um `ref struct`. Caso contrário, o modificador `scoped` poderá ser aplicado somente a [variáveis de referência](#) locais. Isso inclui variáveis locais declaradas com o modificador `ref` e os parâmetros declarados com os modificadores `in`, `ref` ou `out`.

O modificador `scoped` é adicionado implicitamente a `this` em métodos declarados em parâmetros `struct`, `out` e `ref` quando o tipo for um `ref struct`.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Instruções de declaração](#)

- [Variáveis de referência e retornos](#)

Para obter mais informações sobre o modificador `scoped`, consulte a nota de proposta [Aprimoramentos de struct de baixo nível](#).

## Confira também

- [Referência de C#](#)
- [Inicializadores de objeto e de coleção](#)
- [ref keyword](#)
- [Reducir alocações de memória usando novos recursos do C#](#)
- [preferências 'var' \(regras de estilo IDE0007 e IDE0008\)](#)

# Tipos internos (referência C#)

Artigo • 10/05/2023

A tabela a seguir lista os tipos internos de [valor](#) do C#:

Palavra-chave do tipo C#	Tipo .NET
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
nint	System.IntPtr
nuint	System.UIntPtr
long	System.Int64
ulong	System.UInt64
short	System.Int16
ushort	System.UInt16

A tabela a seguir lista os tipos internos de [referência](#) do C#:

Palavra-chave do tipo C#	Tipo .NET
object	System.Object
string	System.String
dynamic	System.Object

Na tabela anterior, cada palavra-chave do tipo C# da coluna à esquerda (exceto `dynamic`) é um alias do tipo .NET correspondente. Eles são intercambiáveis. Por

exemplo, as declarações a seguir declaram variáveis do mesmo tipo:

C#

```
int a = 123;  
System.Int32 b = 123;
```

A palavra-chave `void` representa a ausência de um tipo. Use-a como o tipo de retorno de um método que não retorne um valor.

## Confira também

- Usar palavras-chave de linguagem em vez de nomes de tipo de estrutura (regra de estilo IDE0049)
- Referência de C#
- Valores padrão de tipos C#

# Tipos não gerenciados (referência em C#)

Artigo • 16/06/2023

Um tipo é um **tipo não gerenciado** se for um dos seguintes:

- `sbyte, byte, short, ushort, int, uint, long, ulong, nint, nuint, char, float, double, decimal ou bool`
- Todo tipo **enumerado**
- Todo tipo **ponteiro**
- Qualquer tipo **struct** definido pelo usuário que contenha somente campos de tipos não gerenciados.

Você pode usar a **restrição unmanaged** para especificar que um parâmetro de tipo é um tipo não gerenciado, não ponteiro e não anulável.

Um tipo de struct *construído* que contém campos somente de tipos não gerenciados também é do tipo não gerenciado, como mostra o seguinte exemplo:

C#

```
using System;

public struct Coords<T>
{
    public T X;
    public T Y;
}

public class UnmanagedTypes
{
    public static void Main()
    {
        DisplaySize<Coords<int>>();
        DisplaySize<Coords<double>>();
    }

    private unsafe static void DisplaySize<T>() where T : unmanaged
    {
        Console.WriteLine($"{typeof(T)} is unmanaged and its size is {sizeof(T)} bytes");
    }
}

// Output:
// Coords`1[System.Int32] is unmanaged and its size is 8 bytes
// Coords`1[System.Double] is unmanaged and its size is 16 bytes
```

Um struct genérico pode ser a fonte de tipos construídos gerenciados e não gerenciados. O exemplo anterior define um struct `Coords<T>` genérico e apresenta os exemplos de tipos construídos não gerenciados. O exemplo de um tipo gerenciado é `Coords<object>`. Ele é gerenciado porque tem os campos do tipo `object`, não gerenciados. Se você quiser que *todos* os tipos construídos sejam do tipo não gerenciado, use a restrição `unmanaged` na definição de um struct genérico:

C#

```
public struct Coords<T> where T : unmanaged
{
    public T X;
    public T Y;
}
```

## Especificação da linguagem C#

Para saber mais, confira a seção [Tipos de ponteiro](#) na [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Tipos de ponteiro](#)
- [Tipos relacionados a memória e extensão](#)
- [Operador sizeof](#)
- [stackalloc](#)

# Valores padrão de tipos C# (referência de C#)

Artigo • 07/04/2023

A seguinte tabela mostra os valores padrão de tipos C#:

Type	Valor padrão
Qualquer tipo de referência	<code>null</code>
Qualquer tipo numérico integral interno	0 (zero)
Qualquer tipo numérico de ponto flutuante interno	0 (zero)
<code>bool</code>	<code>false</code>
<code>char</code>	<code>'\0'</code> (U+0000)
enumeração	O valor é produzido pela expressão <code>(E)0</code> , em que <code>E</code> é o identificador de enumeração.
<code>struct</code>	O valor produzido pela configuração de todos os campos tipo-valor para seus valores padrão e todos os campos tipo-referência para <code>null</code> .
Qualquer tipo de valor que permite valor nulo	Uma instância para a qual a propriedade <code>HasValue</code> é <code>false</code> e a propriedade <code>Value</code> não está definida. Esse valor padrão também é conhecido como o valor <i>null</i> do tipo de valor que permite valor nulo.

## Expressões de valor padrão

Use o [operador default](#) para produzir o valor padrão de um tipo, como mostra o seguinte exemplo:

C#

```
int a = default(int);
```

Você pode usar o `default literal` para inicializar uma variável com o valor padrão de seu tipo:

```
C#
```

```
int a = default;
```

## Construtor sem parâmetros de um tipo de valor

Para um tipo de valor, o construtor *implícito* sem parâmetros também produz o valor padrão do tipo, como mostra o seguinte exemplo:

```
C#
```

```
var n = new System.Numerics.Complex();
Console.WriteLine(n); // output: (0, 0)
```

Em tempo de execução, se a instância de `System.Type` representar um tipo de valor, você poderá usar o método `Activator.CreateInstance(Type)` para invocar o construtor sem parâmetros a fim de obter o valor padrão do tipo.

### ⓘ Observação

Em C# 10 e versões posteriores, um **tipo de estrutura** (que é um tipo de valor) pode ter um **construtor explícito sem parâmetros** que pode produzir um valor não padrão do tipo. Portanto, recomendamos usar o operador `default` ou o literal `default` para produzir o valor padrão de um tipo.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Valores padrão](#)
- [Construtores padrão](#)
- [C# 10 - Constructos de struct sem parâmetros](#)
- [C# 11 - Structs de padrão automático](#)

## Confira também

- Referência de C#
- Construtores

# Palavras-chave C#

Artigo • 22/09/2022

As palavras-chave são identificadores reservados predefinidos com significados especiais para o compilador. Elas não podem ser usadas como identificadores em seu programa, a não ser que incluem @ como prefixo. Por exemplo, @if é um identificador válido, mas if não é porque if é uma palavra-chave.

A primeira tabela neste tópico lista as palavras-chave que são identificadores reservados em qualquer parte de um programa C#. A segunda tabela neste tópico lista as palavras-chave contextuais em C#. As palavras-chave contextuais têm significado especial somente em um contexto limitado de programa e podem ser usadas como identificadores fora de contexto. Em geral, à medida que novas palavras-chave são adicionadas na linguagem C#, elas são adicionadas como palavras-chave contextuais para evitar a interrupção de programas escritos em versões anteriores.

abstract

as

base

bool

break

byte

case

catch

char

checked

class

const

continue

decimal

default

delegate

do

double

senão

enumeração

event

explicit

extern

false  
finally  
fixed  
float  
for  
foreach  
goto  
if  
implicit  
Em  
int  
interface  
interno  
is  
lock  
longo

namespace  
novo  
null  
object  
operator  
out  
override  
params  
private  
protected  
público  
readonly  
ref  
return  
sbyte  
sealed  
short  
sizeof  
stackalloc

static  
cadeia de caracteres  
struct  
switch

this  
throw  
true  
try  
typeof  
uint  
ulong  
unchecked  
unsafe  
ushort  
using  
virtual  
void  
volatile  
while

## Palavras-chave contextuais

Uma palavra-chave contextual é usada para fornecer um significado específico no código, mas não é uma palavra reservada no C#. Algumas palavras-chave contextuais, como `partial` e `where`, têm significados especiais em dois ou mais contextos.

add  
and  
alias  
ascending  
args  
async  
await  
by  
descending  
dinâmico  
equals  
from  
  
get  
global  
grupo  
init  
into

join  
let  
managed (convenção de chamada de ponteiro de função)  
nameof  
nint  
not  
  
notnull  
nuint  
on  
or  
orderby  
parcial (tipo)  
partial (método)  
record  
remove  
select  
  
set  
unmanaged (convenção de chamada de ponteiro de função)  
unmanaged (restrição de tipo genérico)  
value  
var  
when (condição de filtro)  
where (restrição de tipo genérico)  
where (cláusula de consulta)  
with  
yield

## Confira também

- Referência de C#

# Modificadores de acesso (Referência de C#)

Artigo • 07/04/2023

Os modificadores de acesso são palavras-chave usadas para especificar a acessibilidade declarada de um membro ou de um tipo. Esta seção apresenta os cinco modificadores de acesso:

- `public`
- `protected`
- `internal`
- `private`
- `file`

Os sete níveis de acessibilidade a seguir podem ser especificados usando os modificadores de acesso:

- `public`: o acesso não é restrito.
- `protected`: o acesso é limitado à classe que os contém ou aos tipos derivados da classe que os contém.
- `internal`: o acesso é limitado ao assembly atual.
- `protected internal`: o acesso é limitado ao assembly atual ou aos tipos derivados da classe que os contém.
- `private`: o acesso é limitado ao tipo recipiente.
- `private protected`: o acesso é limitado à classe que o contém ou a tipos derivados da classe que o contém no assembly atual.
- `file`: o tipo declarado só está visível no arquivo de origem atual. Os tipos com escopo de arquivo geralmente são usados para geradores de origem.

Esta seção também apresenta os seguintes conceitos:

- **Níveis de acessibilidade**: utilização dos quatro modificadores de acesso para declarar seis níveis de acessibilidade.
- **Domínio de acessibilidade**: especifica em que lugar, nas seções do programa, um membro pode ser referenciado.
- **Restrições no uso de níveis de acessibilidade**: um resumo das restrições sobre o uso de níveis de acessibilidade declarados.

## Confira também

- Adicionar modificadores de acessibilidade (regra de estilo IDE0040)
- Referência de C#
- Guia de Programação em C#
- Palavras-chave do C#
- Modificadores de acesso
- Palavras-chave de acesso
- Modificadores

# Níveis de acessibilidade (Referência de C#)

Artigo • 17/08/2023

Use os modificadores de acesso, `public`, `protected`, `internal` ou `private`, para especificar um dos níveis de acessibilidade declarada a seguir para membros.

Acessibilidade declarada	Significado
<code>public</code>	O acesso não é restrito.
<code>protected</code>	O acesso é limitado à classe que os contém ou aos tipos derivados da classe que os contém.
<code>internal</code>	O acesso é limitado ao assembly atual.
<code>protected internal</code>	O acesso é limitado ao assembly atual ou aos tipos derivados da classe que os contém.
<code>private</code>	O acesso é limitado ao tipo recipiente.
<code>private protected</code>	O acesso é limitado à classe que o contém ou a tipos derivados da classe que o contém no assembly atual.

Apenas um modificador de acesso é permitido para um membro ou tipo, exceto quando você usa as combinações `protected internal` e `private protected`.

Os modificadores de acesso não são permitidos em namespaces. Namespaces não têm nenhuma restrição de acesso.

Dependendo do contexto no qual ocorre uma declaração de membro, apenas algumas acessibilidades declaradas são permitidas. Se não for especificado nenhum modificador de acesso em uma declaração de membro, uma acessibilidade padrão será usada.

Os tipos de nível superior, que não estão aninhados em outros tipos, podem ter apenas a acessibilidade `internal` ou `public`. A acessibilidade de padrão para esses tipos é `internal`.

Tipos aninhados, que são membros de outros tipos, podem ter acessibilidades declaradas conforme indicado na tabela a seguir.

Membros de	Acessibilidade de membro padrão	Acessibilidade declarada permitida do membro
enum	public	Nenhum
class	private	public protected internal private protected internal private protected
interface	public	public protected internal private * protected internal private protected
struct	private	public internal private

\* Um membro `interface` com acessibilidade `private` deve ter uma implementação padrão.

### ⓘ Observação

Se uma classe ou struct for modificada com o modificador de palavra-chave `record`, os mesmos modificadores de acesso serão permitidos.

Além disso, com o modificador `record`, a acessibilidade padrão do membro ainda é `private`, tanto para a classe quanto para a estrutura.

A acessibilidade de um tipo aninhado depende do [domínio de acessibilidade](#), que é determinado pela acessibilidade declarada do membro e pelo domínio da acessibilidade do tipo imediatamente contido. Entretanto, o domínio de acessibilidade de um tipo aninhado não pode exceder o do tipo contido.

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Domínio de acessibilidade](#)
- [Restrições ao uso de níveis de acessibilidade](#)
- [Modificadores de acesso](#)
- [público](#)
- [private](#)
- [protected](#)
- [interno](#)

# Domínio de acessibilidade (Referência de C#)

Artigo • 07/04/2023

O domínio de acessibilidade de um membro especifica em que seções do programa um membro pode ser referenciado. Se o membro estiver aninhado dentro de outro tipo, seu domínio de acessibilidade será determinado pelo [nível de acessibilidade](#) do membro e pelo domínio de acessibilidade do tipo imediatamente contido.

O domínio de acessibilidade de um tipo de nível superior é, pelo menos, o texto do programa do projeto no qual ele é declarado. Isto é, o domínio inclui todos os arquivos de origem deste projeto. O domínio de acessibilidade de um tipo aninhado é, pelo menos, o texto do programa do tipo no qual ele é declarado. Isto é, o domínio é o corpo do tipo, que inclui todos os tipos aninhados. O domínio de acessibilidade de um tipo aninhado nunca excede o do tipo contido. Esses conceitos são demonstrados no exemplo a seguir.

## Exemplo

Este exemplo contém um tipo de nível superior, `T1` e duas classes aninhadas, `M1` e `M2`. As classes contêm campos que têm diferentes acessibilidades declaradas. No método `Main`, um comentário segue cada instrução para indicar o domínio de acessibilidade de cada membro. Observe que as instruções que tentam referenciar os membros inacessíveis são comentadas. Se você quiser ver os erros do compilador causados referenciando um membro inacessível, remova os comentários um de cada vez.

C#

```
public class T1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;

    static T1()
    {
        // T1 can access public or internal members
        // in a public or private (or internal) nested class.
        M1.publicInt = 1;
        M1.internalInt = 2;
        M2.publicInt = 3;
        M2.internalInt = 4;
    }
}
```

```
// Cannot access the private member privateInt
// in either class:
// M1.privateInt = 2; //CS0122
}

public class M1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;
}

private class M2
{
    public static int publicInt = 0;
    internal static int internalInt = 0;
    private static int privateInt = 0;
}

class MainClass
{
    static void Main()
    {
        // Access is unlimited.
        T1.publicInt = 1;

        // Accessible only in current assembly.
        T1.internalInt = 2;

        // Error CS0122: inaccessible outside T1.
        // T1.privateInt = 3;

        // Access is unlimited.
        T1.M1.publicInt = 1;

        // Accessible only in current assembly.
        T1.M1.internalInt = 2;

        // Error CS0122: inaccessible outside M1.
        //     T1.M1.privateInt = 3;

        // Error CS0122: inaccessible outside T1.
        //     T1.M2.publicInt = 1;

        // Error CS0122: inaccessible outside T1.
        //     T1.M2.internalInt = 2;

        // Error CS0122: inaccessible outside M2.
        //     T1.M2.privateInt = 3;

        // Keep the console open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

```
    }  
}
```

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Níveis de acessibilidade](#)
- [Restrições ao uso de níveis de acessibilidade](#)
- [Modificadores de acesso](#)
- [público](#)
- [private](#)
- [protected](#)
- [interno](#)

# Restrições ao uso de níveis de acessibilidade (Referência em C#)

Artigo • 07/04/2023

Quando você especifica um tipo em uma declaração, verifique se o nível de acessibilidade do tipo é dependente do nível de acessibilidade de um membro ou de outro tipo. Por exemplo, a classe base direta deve ser, pelo menos, tão acessível quanto a classe derivada. As seguintes declarações causam um erro do compilador porque a classe base `BaseClass` é menos acessível que a `MyClass`:

C#

```
class BaseClass {...}  
public class MyClass: BaseClass {...} // Error
```

A tabela a seguir resume as restrições nos níveis de acessibilidade declarada.

Contexto	Comentários
Classes	A classe base direta de um tipo de classe deve ser, pelo menos, tão acessível quanto o próprio tipo de classe.
Interfaces	As interfaces base explícitas de um tipo de interface devem ser, pelo menos, tão acessíveis quanto o próprio tipo de interface.
Representantes	O tipo de retorno e os tipos de parâmetro de um tipo delegado devem ser, pelo menos, tão acessíveis quanto o próprio tipo delegado.
Constantes	O tipo de uma constante deve ser, pelo menos, tão acessível quanto a própria constante.
Fields	O tipo de um campo deve ser, pelo menos, tão acessível quanto o próprio campo.
Métodos	O tipo de retorno e os tipos de parâmetro de um método devem ser, pelo menos, tão acessíveis quanto o próprio método.
Propriedades	O tipo de uma propriedade deve ser, pelo menos, tão acessível quanto a propriedade em si.
Eventos	O tipo de um evento deve ser, pelo menos, tão acessível quanto o próprio evento.
Indexadores	O tipo e os tipos de parâmetro de um indexador devem ser, pelo menos, tão acessíveis quanto o próprio indexador.

Contexto	Comentários
Operadores	O tipo de retorno e os tipos de parâmetro de um operador devem ser, pelo menos, tão acessíveis quanto o próprio operador.
Construtores	Os tipos de parâmetro de um construtor devem ser, pelo menos, tão acessíveis quanto o próprio construtor.

## Exemplo

O exemplo a seguir contém declarações incorretas de tipos diferentes. O comentário que segue cada declaração indica o erro do compilador esperado.

C#

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
}
```

```
public B MyProp
{
    set
    {
    }
}

MyDelegate d = new MyDelegate(B.MyPrivateMethod);
// Even when B is declared public, you still get the error:
// "The parameter B.MyPrivateMethod is not accessible due to
// protection level."

public static B operator +(A m1, B m2)
{
    // Error: The type B is less accessible
    // than the operator A.operator +(A,B)
    return new B();
}

static void Main()
{
    Console.WriteLine("Compiled successfully");
}
}
```

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Domínio de acessibilidade](#)
- [Níveis de acessibilidade](#)
- [Modificadores de acesso](#)
- [público](#)
- [private](#)
- [protected](#)
- [interno](#)

# internal (Referência de C#)

Artigo • 09/05/2023

A palavra-chave `internal` é um [modificador de acesso](#) para tipos e membros de tipo.

Esta página aborda o acesso `internal`. A palavra-chave `internal` também faz parte do modificador de acesso [protected internal](#).

Tipos ou membros internos são acessíveis somente em arquivos no mesmo assembly, como neste exemplo:

C#

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

Para obter uma comparação de `internal` com os outros modificadores de acesso, consulte [Níveis de acessibilidade](#) e [Modificadores de acesso](#).

Para saber mais sobre assemblies, confira [Assembly no .NET](#).

Um uso comum do acesso interno é no desenvolvimento baseado em componente, porque ele permite que um grupo de componentes colabore de maneira particular, sem serem expostos para o restante do código do aplicativo. Por exemplo, uma estrutura para a criação de interfaces gráficas do usuário pode fornecer as classes `Control` e `Form`, que cooperam através do uso de membros com acesso interno. Uma vez que esses membros são internos, eles não são expostos ao código que está usando a estrutura.

É um erro fazer referência a um tipo ou um membro com acesso interno fora do assembly no qual ele foi definido.

## Exemplo 1

Este exemplo contém dois arquivos, `Assembly1.cs` e `Assembly1_a.cs`. O primeiro arquivo contém uma classe base interna `BaseClass`. No segundo arquivo, uma tentativa de instanciar a `BaseClass` produzirá um erro.

```
C#
```

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
    public static int intM = 0;
}
```

```
C#
```

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
    static void Main()
    {
        var myBase = new BaseClass(); // CS0122
    }
}
```

## Exemplo 2

Neste exemplo, use os mesmos arquivos que você usou no exemplo 1 e altere o nível de acessibilidade da `BaseClass` para `public`. Também altere o nível de acessibilidade do membro `intM` para `internal`. Nesse caso, você pode instanciar a classe, mas não pode acessar o membro interno.

```
C#
```

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
    internal static int intM = 0;
}
```

```
C#
```

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
    static void Main()
    {
        var myBase = new BaseClass(); // Ok.
        BaseClass.intM = 444; // CS0117
    }
}
```

```
}
```

## Especificação da Linguagem C#

Para obter mais informações, veja [Acessibilidade declarada na Especificação da Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Níveis de acessibilidade](#)
- [Modificadores](#)
- [público](#)
- [private](#)
- [protected](#)

# private (Referência de C#)

Artigo • 08/06/2023

A palavra-chave `private` é um modificador de acesso de membro.

Esta página aborda o acesso `private`. A palavra-chave `private` também faz parte do modificador de acesso [private protected](#).

Acesso particular é o nível de acesso menos permissivo. Membros particulares são acessíveis somente dentro do corpo da classe ou do struct em que são declarados, como neste exemplo:

C#

```
class Employee
{
    private int _i;
    double _d;    // private access by default
}
```

Tipos aninhados no mesmo corpo também podem acessar os membros particulares.

É um erro em tempo de compilação fazer referência a um membro particular fora da classe ou do struct em que ele é declarado.

Para obter uma comparação de `private` com os outros modificadores de acesso, consulte [Níveis de acessibilidade](#) e [Modificadores de acesso](#).

## Exemplo

Neste exemplo, a classe `Employee` contém dois membros de dados particulares, `_name` e `_salary`. Como membros particulares, eles não podem ser acessados, exceto por métodos de membro. Métodos públicos chamados `GetName` e `Salary` foram adicionados para permitir acesso controlado aos membros particulares. O membro `_name` é acessado por meio de um método público e o membro `_salary` é acessado por meio de uma propriedade somente leitura pública. Para obter mais informações, consulte [Propriedades](#).

C#

```
class Employee2
{
```

```
private readonly string _name = "FirstName, LastName";
private readonly double _salary = 100.0;

public string GetName()
{
    return _name;
}

public double Salary
{
    get { return _salary; }
}
}

class PrivateTest
{
    static void Main()
    {
        var e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //     string n = e._name;
        //     double s = e._salary;

        // '_name' is indirectly accessed via method:
        string n = e.GetName();

        // '_salary' is indirectly accessed via property
        double s = e.Salary;
    }
}
```

## Especificação da linguagem C#

Para obter mais informações, veja [Acessibilidade declarada na Especificação da Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Níveis de acessibilidade](#)
- [Modificadores](#)

- público
- protected
- interno

# protected (Referência de C#)

Artigo • 07/04/2023

A palavra-chave `protected` é um modificador de acesso de membro.

## ⓘ Observação

Esta página aborda o acesso `protected`. A palavra-chave `protected` também faz parte dos modificadores de acesso `protected internal` e `private protected`.

Um membro protegido é acessível dentro de sua classe e por instâncias da classe derivada.

Para obter uma comparação de `protected` com os outros modificadores de acesso, consulte [Níveis de acessibilidade](#).

## Exemplo 1

Um membro protegido de uma classe base será acessível em uma classe derivada somente se o acesso ocorrer por meio do tipo da classe derivada. Por exemplo, considere o seguinte segmento de código:

C#

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        var a = new A();
        var b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

A instrução `a.x = 10` gera um erro porque ela é feita dentro do método estático Main e não em uma instância da classe B.

Membros de struct não podem ser protegidos porque o struct não pode ser herdado.

## Exemplo 2

Nesse exemplo, a classe `DerivedPoint` é derivada de `Point`. Portanto, você pode acessar os membros protegidos da classe base diretamente da classe derivada.

C#

```
class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        var dpoint = new DerivedPoint();

        // Direct access to protected members.
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");
    }
}
// Output: x = 10, y = 15
```

Se você alterar os níveis de acesso de `x` e `y` para `private`, o compilador emitirá as mensagens de erro:

`'Point.y' is inaccessible due to its protection level.`

`'Point.x' is inaccessible due to its protection level.`

## Especificação da linguagem C#

Para obter mais informações, veja [Acessibilidade declarada na Especificação da Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- Referência de C#
- Guia de Programação em C#
- Palavras-chave do C#
- Modificadores de acesso
- Níveis de acessibilidade
- Modificadores
- público
- private
- interno
- Questões de segurança de palavras-chave virtuais internas

# public (Referência de C#)

Artigo • 07/04/2023

A palavra-chave `public` é um modificador de acesso para tipos e membros de tipo. Acesso público é o nível de acesso mais permissivo. Não há nenhuma restrição quanto ao acesso a membros públicos, como neste exemplo:

```
C#  
  
class SampleClass  
{  
    public int x; // No access restrictions.  
}
```

Consulte [Modificadores de acesso](#) e [Níveis de acessibilidade](#) para obter mais informações.

## Exemplo

No exemplo a seguir, duas classes são declaradas, `PointTest` e `Program`. Os membros públicos `x` e `y` de `PointTest` são acessados diretamente de `Program`.

```
C#  
  
class PointTest  
{  
    public int x;  
    public int y;  
}  
  
class Program  
{  
    static void Main()  
    {  
        var p = new PointTest();  
        // Direct access to public members.  
        p.x = 10;  
        p.y = 15;  
        Console.WriteLine($"x = {p.x}, y = {p.y}");  
    }  
}  
// Output: x = 10, y = 15
```

Se alterar o nível de acesso de `public` para `particular` ou `protetido`, você receberá a mensagem de erro:

'PointTest.y' é inacessível devido ao seu nível de proteção.

## Especificação da linguagem C#

Para obter mais informações, veja [Acessibilidade declarada na Especificação da Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Modificadores de acesso](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Níveis de acessibilidade](#)
- [Modificadores](#)
- [private](#)
- [protected](#)
- [interno](#)

# protected internal (referência do C#)

Artigo • 07/04/2023

A combinação de palavras-chave `protected internal` é um modificador de acesso de membro. Um membro protegido interno é acessível no assembly atual ou nos tipos que derivam da classe recipiente. Para obter uma comparação de `protected internal` com os outros modificadores de acesso, consulte [Níveis de acessibilidade](#).

## Exemplo

Um membro protegido interno de uma classe base é acessível em qualquer tipo em seu assembly recipiente. Ele também é acessível em uma classe derivada localizada em outro assembly somente se o acesso ocorre por meio de uma variável do tipo de classe derivada. Por exemplo, considere o seguinte segmento de código:

```
C#  
  
// Assembly1.cs  
// Compile with: /target:library  
public class BaseClass  
{  
    protected internal int myValue = 0;  
}  
  
class TestAccess  
{  
    void Access()  
    {  
        var baseObject = new BaseClass();  
        baseObject.myValue = 5;  
    }  
}
```

```
C#  
  
// Assembly2.cs  
// Compile with: /reference:Assembly1.dll  
class DerivedClass : BaseClass  
{  
    static void Main()  
    {  
        var baseObject = new BaseClass();  
        var derivedObject = new DerivedClass();  
  
        // Error CS1540, because myValue can only be accessed by  
        // classes derived from BaseClass.
```

```
// baseObject.myValue = 10;

// OK, because this class derives from BaseClass.
derivedObject.myValue = 10;
}

}
```

Este exemplo contém dois arquivos, `Assembly1.cs` e `Assembly2.cs`. O primeiro arquivo contém uma classe base pública, `BaseClass`, e outra classe, `TestAccess`. `BaseClass` tem um membro interno protegido, `myValue`, que é acessado pelo tipo `TestAccess`. No segundo arquivo, uma tentativa de acessar `myValue` por meio de uma instância de `BaseClass` produzirá um erro, enquanto um acesso a esse membro por meio de uma instância de uma classe derivada, `DerivedClass`, terá êxito.

Membros de struct não podem ser `protected internal` porque o struct não pode ser herdado.

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)
- [Níveis de acessibilidade](#)
- [Modificadores](#)
- [público](#)
- [private](#)
- [interno](#)
- [Questões de segurança de palavras-chave virtuais internas](#)

# private protected (referência do C#)

Artigo • 07/04/2023

A combinação de palavras-chave `private protected` é um modificador de acesso de membro. Um membro particular protegido é acessível por tipos derivados da classe recipiente, mas apenas dentro de seu assembly recipiente. Para obter uma comparação de `private protected` com os outros modificadores de acesso, consulte [Níveis de acessibilidade](#).

## ⓘ Observação

O modificador de acesso `private protected` é válido no C# versão 7.2 e posterior.

## Exemplo

Um membro particular protegido de uma classe base é acessível de tipos derivados em seu assembly recipiente apenas se o tipo estático da variável é o tipo da classe derivada. Por exemplo, considere o seguinte segmento de código:

C#

```
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        var baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

C#

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

Este exemplo contém dois arquivos, `Assembly1.cs` e `Assembly2.cs`. O primeiro arquivo contém uma classe base pública, `BaseClass`, e um tipo derivado dela, `DerivedClass1`. `BaseClass` tem um membro particular protegido, `myValue`, que `DerivedClass1` tenta acessar de duas maneiras. A primeira tentativa de acessar `myValue` por meio de uma instância de `BaseClass` produzirá um erro. No entanto, a tentativa de usá-lo como um membro herdado em `DerivedClass1` terá êxito.

No segundo arquivo, uma tentativa de acessar `myValue` como um membro herdado de `DerivedClass2` produzirá um erro, pois ele é acessível apenas por tipos derivados em `Assembly1`.

Se `Assembly1.cs` contiver um `InternalsVisibleToAttribute` que nomeia `Assembly2`, a classe derivada `DerivedClass2` terá acesso aos `private protected` membros declarados em `BaseClass`. `InternalsVisibleTo` torna os membros `private protected` visíveis para classes derivadas em outros assemblies.

Membros de struct não podem ser `private protected` porque o struct não pode ser herdado.

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores de acesso](#)

- Níveis de acessibilidade
- Modificadores
- público
- private
- interno
- Questões de segurança de palavras-chave virtuais internas

# abstract (Referência de C#)

Artigo • 07/04/2023

O modificador `abstract` indica que o item que está sendo modificado tem uma implementação ausente ou incompleta. O modificador abstrato pode ser usado com classes, métodos, propriedades, indexadores e eventos. Use o modificador `abstract` em uma declaração de classe para indicar que uma classe se destina somente a ser uma classe base de outras classes, não instanciada por conta própria. Membros marcados como abstratos precisam ser implementados por classes não abstratas que derivam da classe abstrata.

## Exemplo 1

Neste exemplo, a classe `Square` deve fornecer uma implementação de `GetArea` porque deriva de `Shape`:

```
C#  
  
abstract class Shape  
{  
    public abstract int GetArea();  
}  
  
class Square : Shape  
{  
    private int _side;  
  
    public Square(int n) => _side = n;  
  
    // GetArea method is required to avoid a compile-time error.  
    public override int GetArea() => _side * _side;  
  
    static void Main()  
    {  
        var sq = new Square(12);  
        Console.WriteLine($"Area of the square = {sq.GetArea()}");  
    }  
}  
// Output: Area of the square = 144
```

As classes abstratas têm os seguintes recursos:

- Uma classe abstrata não pode ser instanciada.
- Uma classe abstrata pode conter acessadores e métodos abstratos.

- Não é possível modificar uma classe abstrata com o modificador `sealed` porque os dois modificadores têm significados opostos. O modificador `sealed` impede que uma classe seja herdada e o modificador `abstract` requer uma classe a ser herdada.
- Uma classe não abstrata derivada de uma classe abstrata deve incluir implementações reais de todos os acessadores e métodos abstratos herdados.

Use o modificador `abstract` em uma declaração de método ou propriedade para indicar que o método ou propriedade não contém a implementação.

Os métodos abstratos têm os seguintes recursos:

- Um método abstrato é implicitamente um método virtual.
- Declarações de método abstrato são permitidas apenas em classes abstratas.
- Como uma declaração de método abstrato não fornece nenhuma implementação real, não há nenhum corpo de método, a declaração do método simplesmente termina com um ponto e vírgula e não há chaves (`{ }`) após a assinatura. Por exemplo:

C#

```
public abstract void MyMethod();
```

A implementação é fornecida por uma [substituição](#) de método, que é um membro de uma classe não abstrata.

- É um erro usar os modificadores `static` ou `virtual` em uma declaração de método abstrato.

Propriedades abstratas se comportam como métodos abstratos, exceto pelas diferenças na sintaxe de declaração e chamada.

- É um erro usar o modificador `abstract` em uma propriedade estática.
- Uma propriedade herdada abstrata pode ser substituída em uma classe derivada incluindo uma declaração de propriedade que usa o modificador `override`.

Para obter mais informações sobre classes abstratas, consulte [Classes e membros de classes abstratas e lacrados](#).

Uma classe abstrata deve fornecer uma implementação para todos os membros de interface.

Uma classe abstrata que implementa uma interface pode mapear os métodos de interface em métodos abstratos. Por exemplo:

```
C#  
  
interface I  
{  
    void M();  
}  
  
abstract class C : I  
{  
    public abstract void M();  
}
```

## Exemplo 2

Nesse exemplo, a classe `DerivedClass` é derivada de uma classe abstrata `BaseClass`. A classe abstrata contém um método abstrato, `AbstractMethod` e duas propriedades abstratas, `X` e `Y`.

```
C#  
  
// Abstract class  
abstract class BaseClass  
{  
    protected int _x = 100;  
    protected int _y = 150;  
  
    // Abstract method  
    public abstract void AbstractMethod();  
  
    // Abstract properties  
    public abstract int X { get; }  
    public abstract int Y { get; }  
}  
  
class DerivedClass : BaseClass  
{  
    public override void AbstractMethod()  
    {  
        _x++;  
        _y++;  
    }  
  
    public override int X // overriding property  
    {  
        get  
    }
```

```
        return _x + 10;
    }
}

public override int Y // overriding property
{
    get
    {
        return _y + 10;
    }
}

static void Main()
{
    var o = new DerivedClass();
    o.AbstractMethod();
    Console.WriteLine($"x = {o.X}, y = {o.Y}");
}
// Output: x = 111, y = 161
```

No exemplo anterior, se você tentar instanciar a classe abstrata usando uma instrução como esta:

C#

```
BaseClass bc = new BaseClass(); // Error
```

Você receberá uma mensagem de erro informando que o compilador não pode criar uma instância da classe abstrata "BaseClass".

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Modificadores](#)
- [virtual](#)
- [override](#)
- [Palavras-chave do C#](#)

# async (Referência de C#)

Artigo • 27/03/2023

Use o modificador `async` para especificar que um método, uma [expressão lambda](#) ou um [método anônimo](#) é assíncrono. Se você usar esse modificador em um método ou expressão, ele será referido como um *método assíncrono*. O exemplo a seguir define um método assíncrono chamado `ExampleMethodAsync`:

C#

```
public async Task<int> ExampleMethodAsync()
{
    //...
}
```

Se você é iniciante em programação assíncrona ou não entende como um método assíncrono usa o [operador await](#) para fazer trabalhos potencialmente longos sem bloquear o thread do chamador, leia a introdução em [Programação assíncrona com async e await](#). O código a seguir é encontrado dentro de um método assíncrono e chama o método [HttpClient.GetStringAsync](#):

C#

```
string contents = await httpClient.GetStringAsync(requestUrl);
```

Um método assíncrono será executado de forma síncrona até atingir sua primeira expressão `await` e, nesse ponto, ele será suspenso até que a tarefa aguardada seja concluída. Enquanto isso, o controle será retornado ao chamador do método, como exibido no exemplo da próxima seção.

Se o método que a palavra-chave `async` modifica não contiver uma expressão ou instrução `await`, ele será executado de forma síncrona. Um aviso do compilador o alertará sobre quaisquer métodos assíncronos que não contenham instruções `await`, pois essa situação poderá indicar um erro. Consulte [Aviso do compilador \(nível 1\) CS4014](#).

A palavra-chave `async` é contextual, pois ela será uma palavra-chave somente quando modificar um método, uma expressão lambda ou um método anônimo. Em todos os outros contextos, ela será interpretada como um identificador.

## Exemplo

O exemplo a seguir mostra a estrutura e o fluxo de controle entre um manipulador de eventos assíncronos, `StartButton_Click` e um método assíncrono, `ExampleMethodAsync`. O resultado do método assíncrono é o número de caracteres de uma página da Web. O código será adequado para um aplicativo do WPF (Windows Presentation Foundation) ou da Windows Store que você criar no Visual Studio, consulte os comentários do código para configurar o aplicativo.

Você pode executar esse código no Visual Studio como um aplicativo do WPF (Windows Presentation Foundation) ou um aplicativo da Windows Store. Você precisa de um controle de botão chamado `StartButton` e de um controle de caixa de texto chamado `ResultsTextBox`. Lembre-se de definir os nomes e o manipulador para que você tenha algo assim:

XAML

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0"
VerticalAlignment="Top" Width="75"
    Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0"
TextWrapping="Wrap"
    Text="&lt;Enter a URL&gt;" VerticalAlignment="Top" Width="310"
Name="ResultsTextBox"/>
```

Para executar o código como um aplicativo WPF:

- Cole este código na classe `MainWindow` em `MainWindow.xaml.cs`.
- Adicione uma referência a `System.Net.Http`.
- Adicione uma diretiva `using` a `System.Net.Http`.

Para executar o código como um aplicativo da Windows Store:

- Cole este código na classe `MainPage` em `MainPage.xaml.cs`.
- Adicione usando diretivas para `System.Net.Http` e `System.Threading.Tasks`.

C#

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync
    returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
```

```

        // Note that you could put "await ExampleMethodAsync()" in the next
        line where
            // "length" is, but due to when '+=' fetches the value of
        ResultsTextBox, you
            // would not see the global side effect of ExampleMethodAsync
        setting the text.
            ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await
    httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}
// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292

```

### Importante

Para obter mais informações sobre tarefas e o código que é executado enquanto aguarda uma tarefa, consulte [Programação assíncrona com `async` e `await`](#). Para obter um exemplo de console completo que usa elementos semelhantes, consulte [Processar tarefas assíncronas à medida que elas são concluídas \(C#\)](#).

## Tipos de retorno

Um método assíncrono pode conter os seguintes tipos de retorno:

- `Task`
- `Task<TResult>`
- `void`. Os métodos `async void` geralmente são desencorajados para código que não são manipuladores de eventos, porque os chamadores não podem `await` esses métodos e devem implementar um mecanismo diferente para relatar a conclusão bem-sucedida ou condições de erro.

- Qualquer tipo que tenha um método `GetAwaiter` acessível. O tipo `System.Threading.Tasks.ValueTask<TResult>` é um exemplo de uma implementação assim. Ele está disponível ao adicionar o pacote NuGet `System.Threading.Tasks.Extensions`.

O método assíncrono não pode declarar os parâmetros `in`, `ref` nem `out` e também não pode ter um [valor retornado por referência](#), mas pode chamar métodos que tenham esses parâmetros.

Você especificará `Task<TResult>` como o tipo de retorno de um método assíncrono se a instrução `return` do método especificar um operando do tipo `TResult`. Você usará `Task` se nenhum valor significativo for retornado quando o método for concluído. Isto é, uma chamada ao método retorna uma `Task`, mas quando a `Task` for concluída, qualquer expressão `await` que esteja aguardando a `Task` será avaliada como `void`.

O tipo de retorno `void` é usado principalmente para definir manipuladores de eventos que exigem esse tipo de retorno. O chamador de um método assíncrono de retorno `void` não pode aguardá-lo e capturar exceções acionadas pelo método.

Você retorna outro tipo, geralmente um tipo de valor, que tenha um método `GetAwaiter` para minimizar as alocações de memória nas seções do código críticas ao desempenho.

Para obter mais informações e exemplos, consulte [Tipos de retorno assíncronos](#).

## Confira também

- [AsyncStateMachineAttribute](#)
- [await](#)
- [Programação assíncrona com async e await](#)
- [Processar tarefas assíncronas conforme elas são concluídas](#)
- [Blog do .NET: como assíncrono/espera realmente funciona em C#](#) ↗

# const (Referência de C#)

Artigo • 05/06/2023

Use a palavra-chave `const` para declarar um campo constante ou uma constante local. Campos e locais constantes não são variáveis e não podem ser modificados. As constantes podem ser números, valores booleanos, cadeias de caracteres ou uma referência nula. Não crie uma constante para representar informações que você espera mudar a qualquer momento. Por exemplo, não use um campo constante para armazenar o preço de um serviço, um número de versão de produto ou a marca de uma empresa. Esses valores podem mudar ao longo do tempo e como os compiladores propagam constantes, outro código compilado com as bibliotecas terá que ser recompilado para ver as alterações. Veja também a palavra-chave [readonly](#). Por exemplo:

C#

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

A partir do C# 10, as [cadeias de caracteres interpoladas](#) podem ser constantes, se todas as expressões usadas também forem cadeias de caracteres constantes. Esse recurso pode melhorar o código que cria cadeias de caracteres constantes:

C#

```
const string Language = "C#";
const string Platform = ".NET";
const string Version = "10.0";
const string FullProductName = $"{Platform} - Language: {Language} Version: {Version}";
```

## Comentários

O tipo de uma declaração constante especifica o tipo dos membros que a declaração apresenta. O inicializador de uma constante local ou de um campo constante deve ser uma expressão constante que pode ser implicitamente convertida para o tipo de destino.

Uma expressão constante é uma expressão que pode ser completamente avaliada em tempo de compilação. Portanto, os únicos valores possíveis para constantes de tipos de

referência são cadeias de caracteres e uma referência nula.

A declaração constante pode declarar constantes múltiplas como:

C#

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

O modificador `static` não é permitido em uma declaração constante.

A constante pode fazer parte de uma expressão constante, como a seguir:

C#

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

### ① Observação

A palavra-chave `readonly` é diferente da palavra-chave `const`. O campo `const` pode ser inicializado apenas na declaração do campo. Um campo `readonly` pode ser inicializado na declaração ou em um construtor. Portanto, campos `readonly` podem ter valores diferentes dependendo do construtor usado. Além disso, embora um campo `const` seja uma constante em tempo de compilação, o campo `readonly` pode ser usado para constantes de tempo de execução, como nesta linha: `public static readonly uint l1 = (uint)DateTime.Now.Ticks;`

## Exemplos

C#

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int C1 = 5;
        public const int C2 = C1 + 5;

        public SampleClass(int p1, int p2)
        {
            x = p1;
            y = p2;
        }
    }
}
```

```
        }
    }

    static void Main()
    {
        var mC = new SampleClass(11, 22);
        Console.WriteLine($"x = {mC.x}, y = {mC.y}");
        Console.WriteLine($"C1 = {SampleClass.C1}, C2 = {SampleClass.C2}");
    }
}
/* Output
   x = 11, y = 22
   C1 = 5, C2 = 10
*/
```

O seguinte exemplo demonstra como declarar uma constante local:

C#

```
public class SealedTest
{
    static void Main()
    {
        const int C = 707;
        Console.WriteLine($"My local constant = {C}");
    }
}
// Output: My local constant = 707
```

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Constantes](#)
- [Expressões constantes](#)

## Confira também

- [Referência de C#](#)
- [Palavras-chave de C#](#)
- [readonly](#)

# event (Referência de C#)

Artigo • 10/05/2023

A palavra-chave `event` é usada para declarar um evento em uma classe publicadora.

## Exemplo

O exemplo a seguir mostra como declarar e acionar um evento que usa o `EventHandler` como o tipo delegado subjacente. Para obter o exemplo de código completo, que também mostra como usar o tipo delegado `EventHandler<TEventArgs>` genérico e como assinar um evento e criar um método de manipulador de eventos, consulte [Como publicar eventos em conformidade com as diretrizes do .NET](#).

C#

```
public class SampleEventArgs
{
    public SampleEventArgs(string text) { Text = text; }
    public string Text { get; } // readonly
}

public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event in a thread-safe manner using the ?. operator.
        SampleEvent?.Invoke(this, new SampleEventArgs("Hello"));
    }
}
```

Os eventos são um tipo especial de delegado multicast que só podem ser invocados de dentro da classe (ou classes derivadas) ou struct em que eles são declarados (a classe publicadora). Se outras classes ou structs assinarem o evento, seus respectivos métodos de manipulador de eventos serão chamados quando a classe publicadora acionar o evento. Para obter mais informações e exemplos de código, consulte [Eventos e Delegados](#).

Os eventos podem ser marcados como [public](#), [private](#), [protected](#), [internal](#), [protected internal](#) ou [private protected](#). Esses modificadores de acesso definem como os usuários da classe podem acessar o evento. Para obter mais informações, consulte [Modificadores de Acesso](#).

## Palavras-chave e eventos

As palavras-chave a seguir aplicam-se a eventos.

Palavra-chave	Descrição	Para obter mais informações
<a href="#">static</a>	Torna o evento disponível para chamadores a qualquer momento, mesmo se não existir nenhuma instância da classe.	<a href="#">Classes static e membros de classes static</a>
<a href="#">virtual</a>	Permite que classes derivadas substituam o comportamento do evento, usando a palavra-chave <a href="#">override</a> .	<a href="#">Herança</a>
<a href="#">sealed</a>	Especifica que, para classes derivadas, o evento não é mais virtual.	
<a href="#">abstract</a>	O compilador não gerará mais os blocos de acessador de evento <code>add</code> e <code>remove</code> , portanto, as classes derivadas devem fornecer sua própria implementação.	

Um evento pode ser declarado como um evento estático, usando a palavra-chave [static](#). Isso torna o evento disponível para chamadores a qualquer momento, mesmo se não existir nenhuma instância da classe. Para obter mais informações, consulte [Classes estáticas e membros de classes estáticas](#).

Um evento pode ser marcado como um evento virtual, usando a palavra-chave [virtual](#). Isso habilita as classes derivadas a substituírem o comportamento do evento, usando a palavra-chave [override](#). Para obter mais informações, consulte [Herança](#). Um evento que substitui um evento virtual também pode ser [sealed](#), o que especifica que ele não é mais virtual para classes derivadas. Por fim, um evento pode ser declarado [abstract](#), o que significa que o compilador não gerará os blocos de acessador de evento `add` e `remove`. Portanto, classes derivadas devem fornecer sua própria implementação.

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- Referência de C#
- Guia de Programação em C#
- Palavras-chave do C#
- add
- remove
- Modificadores
- Como combinar delegados (delegados multicast)

# extern (Referência de C#)

Artigo • 07/04/2023

O modificador `extern` é usado para declarar um método implementado externamente. Um uso comum do modificador `extern` é com o atributo `DllImport` quando você está usando serviços Interop para chamar código não gerenciado. Nesse caso, o método também deve ser declarado como `static` conforme mostrado no seguinte exemplo:

C#

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

A palavra-chave `extern` também pode definir um alias de assembly externo que possibilita referenciar diferentes versões do mesmo componente de dentro de um único assembly. Para obter mais informações, consulte [alias externo](#).

É um erro usar os modificadores `abstract` e `extern` juntos para modificar o mesmo membro. Usar o modificador `extern` significa que esse método é implementado fora do código C#, enquanto que usar o modificador `abstract` significa que a implementação do método não é fornecida na classe.

A palavra-chave `extern` possui utilizações mais limitadas em C# do que em C++. Para comparar a palavra-chave de C# com a palavra-chave de C++, consulte [Usando extern para especificar vínculos na referência da linguagem C++](#).

## Exemplo 1

Neste exemplo, o programa recebe uma cadeia de caracteres do usuário e a exibe dentro de uma caixa de mensagem. O programa usa o método `MessageBox` importado da biblioteca User32.dll.

C#

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);

    static int Main()
    {
```

```

        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}

```

## Exemplo 2

Este exemplo ilustra um programa C# que chama uma biblioteca em C (uma DLL nativa).

1. Crie o seguinte arquivo em C e atribua o nome `cmdll.c`:

```

C

// cmdll.c
// Compile with: -LD
int __declspec(dllexport) SampleMethod(int i)
{
    return i*10;
}

```

2. Abra uma janela do Prompt de Comando de Ferramentas Nativas do Visual Studio x64 (ou x32) do diretório de instalação do Visual Studio e compile o arquivo `cmdll.c` digitando `cl -LD cmdll.c` no prompt de comando.

3. No mesmo diretório, crie o seguinte arquivo em C# e atribua o nome `cm.cs`:

```

C#

// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cmdll.dll")]
    public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.",
        SampleMethod(5));
    }
}

```

4. Abra uma janela do Prompt de Comando de Ferramentas Nativas do Visual Studio x64 (ou x32) do diretório de instalação do Visual Studio e compile o arquivo `cm.cs`

ao digitar:

`csc cm.cs` (para o prompt de comando do x64) – ou – `csc -platform:x86 cm.cs` (para o prompt de comando do x32)

Isso criará o arquivo executável `cm.exe`.

5. Execute `cm.exe`. O método `SampleMethod` passa o valor 5 ao arquivo de DLL que retorna o valor multiplicado por 10. O programa produz a seguinte saída:

Saída

```
SampleMethod() returns 50.
```

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [System.Runtime.InteropServices.DllImportAttribute](#)
- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores](#)

# in (modificador genérico) (Referência de C#)

Artigo • 07/04/2023

Para parâmetros de tipo genérico, a palavra-chave `in` especifica que o parâmetro de tipo é contravariante. Você pode usar a palavra-chave `in` em delegados e interfaces genéricas.

A contravariância permite que você use um tipo menos derivado do que aquele especificado pelo parâmetro genérico. Isso permite a conversão implícita de classes que implementam interfaces contravariantes e a conversão implícita de tipos delegados. A covariância e a contravariância em parâmetros de tipo genérico têm suporte para tipos de referência, mas não para tipos de valor.

Um tipo pode ser declarado como contravariante em uma interface genérica ou como delegado somente se ele define o tipo de parâmetros de um método e não o tipo de retorno de um método. Os parâmetros `In`, `ref` e `out` precisam ser invariáveis, ou seja, nem covariantes nem contravariantes.

Uma interface que tem um parâmetro de tipo contravariante permite que os seus métodos aceitem argumentos de tipos menos derivados que aqueles especificados pelo parâmetro de tipo de interface. Por exemplo, na interface `IComparer<T>`, o tipo `T` é contravariante e você pode atribuir um objeto do tipo `IComparer<Person>` a um objeto do tipo `IComparer<Employee>`, sem usar nenhum método de conversão especial caso `Employee` herde `Person`.

Um delegado contravariante pode ser atribuído a outro delegado do mesmo tipo, mas com um parâmetro de tipo genérico menos derivado.

Para obter mais informações, consulte [Covariância e contravariância](#).

## Interface genérica contravariante

O exemplo a seguir mostra como declarar, estender e implementar uma interface genérica contravariante. Ele também mostra como você pode usar a conversão implícita para classes que implementam essa interface.

C#

```
// Contravariant interface.  
interface IContravariant<in A> { }
```

```

// Extending contravariant interface.
interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.
class Sample<A> : IContravariant<A> { }

class Program
{
    static void Test()
    {
        IContravariant<Object> iobj = new Sample<Object>();
        IContravariant<String> istr = new Sample<String>();

        // You can assign iobj to istr because
        // the IContravariant interface is contravariant.
        istr = iobj;
    }
}

```

## Delegado genérico contravariante

O exemplo a seguir mostra como declarar, instanciar e invocar um delegado genérico contravariante. Ele também mostra como você pode converter implicitamente um tipo delegado.

C#

```

// Contravariant delegate.
public delegate void DContravariant<in A>(A argument);

// Methods that match the delegate signature.
public static void SampleControl(Control control)
{ }
public static void SampleButton(Button button)
{ }

public void Test()
{

    // Instantiating the delegates with the methods.
    DContravariant<Control> dControl = SampleControl;
    DContravariant<Button> dButton = SampleButton;

    // You can assign dControl to dButton
    // because the DContravariant delegate is contravariant.
    dButton = dControl;

    // Invoke the delegate.
    dButton(new Button());
}

```

# Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [out](#)
- [Covariância e Contravariância](#)
- [Modificadores](#)

# Modificador new (referência em C#)

Artigo • 07/04/2023

Quando usada como um modificador de declaração, a palavra-chave `new` oculta explicitamente um membro herdado de uma classe base. Quando você oculta um membro herdado, a versão derivada do membro substitui a versão da classe base. Isso pressupõe que a versão da classe base do membro esteja visível, pois ela já estaria oculta se fosse marcada como `private` ou, em alguns casos, `internal`. Embora seja possível ocultar membros `public` ou `protected` sem usar o modificador `new`, você recebe um aviso do compilador. Se você usar `new` para ocultar explicitamente um membro, ele suprime o aviso.

Você também pode usar a palavra-chave `new` para [criar uma instância de um tipo](#) ou como uma [restrição de tipo genérico](#).

Para ocultar um membro herdado, declare-o na classe derivada usando o mesmo nome de membro e modifique-o com a palavra-chave `new`. Por exemplo:

```
C#  
  
public class BaseC  
{  
    public int x;  
    public void Invoke() { }  
}  
public class DerivedC : BaseC  
{  
    new public void Invoke() { }  
}
```

Neste exemplo, `BaseC.Invoke` é ocultado por `DerivedC.Invoke`. O campo `x` não é afetado porque não é ocultado por um nome semelhante.

A ocultação de nome por meio da herança assume uma das seguintes formas:

- Normalmente, uma constante, campo, propriedade ou tipo que é apresentado em uma classe ou struct oculta todos os membros da classe base que compartilham seu nome. Há casos especiais. Por exemplo, se você declarar que um novo campo com o nome `N` tem um tipo que não pode ser invocado e um tipo base declarar que `N` é um método, o novo campo não ocultará a declaração base na sintaxe de invocação. Para obter mais informações, consulte a seção [Pesquisa de membro](#) na [especificação da linguagem C#](#).

- Um método introduzido em uma classe ou struct oculta propriedades, campos e tipos que compartilham esse nome na classe base. Ele também oculta todos os métodos da classe base que têm a mesma assinatura.
- Um indexador introduzido em uma classe ou struct oculta todos os indexadores de classe base que têm a mesma assinatura.

É um erro usar `new` e `override` no mesmo membro, porque os dois modificadores têm significados mutuamente exclusivos. O modificador `new` cria um novo membro com o mesmo nome e faz com que o membro original seja ocultado. O modificador `override` estende a implementação de um membro herdado.

Usar o modificador `new` em uma declaração que não oculta um membro herdado gera um aviso.

## Exemplos

Neste exemplo, uma classe base, `BaseC` e uma classe derivada, `DerivedC`, usam o mesmo nome do campo `x`, que oculta o valor do campo herdado. O exemplo demonstra o uso do modificador `new`. Ele também demonstra como acessar os membros ocultos da classe base usando seus nomes totalmente qualificados.

C#

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
```

```
/*
Output:
100
55
22
*/
```

Neste exemplo, uma classe aninhada oculta uma classe que tem o mesmo nome na classe base. O exemplo demonstra como usar o modificador `new` para eliminar a mensagem de aviso e como acessar os membros da classe oculta usando seus nomes totalmente qualificados.

C#

```
public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1 = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/
```

Se você remover o modificador `new`, o programa ainda será compilado e executado, mas você receberá o seguinte aviso:

text

The keyword new is required on 'MyDerivedC.x' because it hides inherited member ' MyBaseC.x '.

## Especificação da linguagem C#

Para obter mais informações, consulte a seção [O modificador new na especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores](#)
- [Controle de versão com as palavras-chave override e new](#)
- [Quando usar as palavras-chave override e new](#)

# out (modificador genérico) (Referência de C#)

Artigo • 09/05/2023

Para parâmetros de tipo genérico, a palavra-chave `out` especifica que o parâmetro de tipo é covariante. Você pode usar a palavra-chave `out` em delegados e interfaces genéricas.

A covariância permite que você use um tipo mais derivado do que aquele especificado pelo parâmetro genérico. Isso permite a conversão implícita de classes que implementam interfaces covariantes e a conversão implícita de tipos delegados. A covariância e a contravariância têm suporte para tipos de referência, mas não para tipos de valor.

Uma interface que tem um parâmetro de tipo covariante permite que seus métodos retornem tipos mais derivados do que aqueles especificados pelo parâmetro de tipo. Por exemplo, já que no .NET Framework 4, em `IEnumerable<T>`, o tipo T é covariante, você pode atribuir um objeto do tipo `IEnumerable(Of String)` a um objeto do tipo `IEnumerable(Of Object)` sem usar nenhum método de conversão especial.

Pode ser atribuído a um delegado covariante outro delegado do mesmo tipo, mas com um parâmetro de tipo genérico mais derivado.

Para obter mais informações, consulte [Covariância e contravariância](#).

## Exemplo – interface genérica covariante

O exemplo a seguir mostra como declarar, estender e implementar uma interface genérica covariante. Ele também mostra como usar a conversão implícita para classes que implementam uma interface covariante.

C#

```
// Covariant interface.  
interface ICovariant<out R> { }  
  
// Extending covariant interface.  
interface IExtCovariant<out R> : ICovariant<R> { }  
  
// Implementing covariant interface.  
class Sample<R> : ICovariant<R> { }  
  
class Program
```

```

{
    static void Test()
    {
        ICovariant<Object> iobj = new Sample<Object>();
        ICovariant<String> istr = new Sample<String>();

        // You can assign istr to iobj because
        // the ICovariant interface is covariant.
        iobj = istr;
    }
}

```

Em uma interface genérica, um parâmetro de tipo pode ser declarado covariante se ele satisfizer as condições a seguir:

- O parâmetro de tipo é usado apenas como um tipo de retorno dos métodos de interface e não é usado como um tipo de argumentos de método.

#### ➊ Observação

Há uma exceção a essa regra. Se, em uma interface covariante, você tiver um delegado genérico contravariante como um parâmetro de método, você poderá usar o tipo covariante como um parâmetro de tipo genérico para o delegado. Para obter mais informações sobre delegados genéricos covariantes e contravariantes, consulte [Variância em delegados e Usando variância para delegados genéricos Func e Action](#).

- O parâmetro de tipo não é usado como uma restrição genérica para os métodos de interface.

## Exemplo – delegado genérico covariante

O exemplo a seguir mostra como declarar, instanciar e invocar um delegado genérico covariante. Ele também mostra como converter implicitamente os tipos delegados.

C#

```

// Covariant delegate.
public delegate R DCovariant<out R>();

// Methods that match the delegate signature.
public static Control SampleControl()
{ return new Control(); }

public static Button SampleButton()
{ return new Button(); }

```

```
public void Test()
{
    // Instantiate the delegates with the methods.
    DCovariant<Control> dControl = SampleControl;
    DCovariant<Button> dButton = SampleButton;

    // You can assign dButton to dControl
    // because the DCovariant delegate is covariant.
    dControl = dButton;

    // Invoke the delegate.
    dControl();
}
```

Um delegado genérico, um tipo pode ser declarado covariante se for usado apenas como um tipo de retorno do método e não usado para argumentos de método.

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Variação em interfaces genéricas](#)
- [Em](#)
- [Modificadores](#)

# Substituição (referência de C#)

Artigo • 07/04/2023

O modificador `override` é necessário para estender ou modificar a implementação abstrata ou virtual de um método, propriedade, indexador ou evento herdado.

No seguinte exemplo, a classe `Square` deve fornecer uma implementação substituída de `GetArea` porque `GetArea` é herdado da classe abstrata `Shape`:

C#

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Um método `override` fornece uma nova implementação de um membro herdado de uma classe base. O método que é substituído por uma declaração `override` é conhecido como o método base substituído. O método `override` deve ter a mesma assinatura que o método base substituído. Os métodos `override` dão suporte a tipos de retorno covariantes. Em particular, o tipo de retorno de um método `override` pode derivar do tipo de retorno do método base correspondente.

Você não pode substituir um método não virtual ou estático. O método base substituído deve ser `virtual`, `abstract` ou `override`.

Uma declaração `override` não pode alterar a acessibilidade do método `virtual`. O método `override` e o método `virtual` devem ter o mesmo modificador de nível de

acesso.

Não é possível usar os modificadores `new`, `static` ou `virtual` para modificar um método `override`.

Uma declaração de propriedade de substituição deve especificar exatamente o mesmo modificador de acesso, tipo e nome que a propriedade herdada. Do C# 9.0 em diante, as propriedades de substituição somente leitura dão suporte a tipos de retorno covariantes. A propriedade substituída deve ser `virtual`, `abstract` ou `override`.

Para obter mais informações sobre como usar a palavra-chave `override`, consulte [Controle de versão com as palavras-chave override e new](#) e [Quando usar as palavras-chave override e new](#). Para obter informações sobre herança, consulte [Herança](#).

## Exemplo

Este exemplo define uma classe base chamada `Employee` e uma classe derivada chamada `SalesEmployee`. A classe `SalesEmployee` inclui um campo extra, `salesbonus`, e substitui o método `CalculatePay` para levar isso em consideração.

C#

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

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal _basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            Name = name;
            _basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return _basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
```

```

{
    // New field that will affect the base pay.
    private decimal _salesbonus;

    // The constructor calls the base-class version, and
    // initializes the salesbonus field.
    public SalesEmployee(string name, decimal basepay, decimal
salesbonus)
        : base(name, basepay)
    {
        _salesbonus = salesbonus;
    }

    // Override the CalculatePay method
    // to take bonus into account.
    public override decimal CalculatePay()
    {
        return _basepay + _salesbonus;
    }
}

static void Main()
{
    // Create some new employees.
    var employee1 = new SalesEmployee("Alice", 1000, 500);
    var employee2 = new Employee("Bob", 1200);

    Console.WriteLine($"Employee1 {employee1.Name} earned:
{employee1.CalculatePay()}");
    Console.WriteLine($"Employee2 {employee2.Name} earned:
{employee2.CalculatePay()}");
}
/*
Output:
Employee1 Alice earned: 1500
Employee2 Bob earned: 1200
*/

```

## Especificação da linguagem C#

Para obter mais informações, confira a seção [Substituir métodos](#) da [Especificação da linguagem C#](#).

Para obter mais informações sobre tipos de retorno covariantes, confira a [nota de proposta de recursos](#).

## Confira também

- Referência de C#
- Herança
- Palavras-chave de C#
- Modificadores
- `abstract`
- `virtual`
- `new` (modificador)
- Polimorfismo

# readonly (Referência de C#)

Artigo • 07/04/2023

A palavra-chave `readonly` é um modificador que pode ser usado em quatro contextos:

- Em uma [declaração de campo](#), `readonly` indica que a atribuição ao campo só pode ocorrer como parte da declaração ou em um construtor na mesma classe. Um campo `readonly` pode ser atribuído e reatribuído várias vezes na declaração de campo e no construtor.

Um campo `readonly` não pode ser atribuído depois da saída do construtor. Essa regra tem implicações diferentes para tipos de valor e tipos de referência:

- Como os tipos de valor contêm diretamente seus dados, um campo que é um tipo de valor `readonly` é imutável.
- Como os tipos de referência contêm uma referência a seus dados, um campo que é um tipo de referência `readonly` deve sempre se referir ao mesmo objeto. Esse objeto não é imutável. O modificador `readonly` impede que o campo seja substituído por uma instância diferente do tipo de referência. No entanto, o modificador não impede que dados da instância do campo sejam modificados por meio do campo somente leitura.

## ⚠️ Aviso

Um tipo visível externamente que contém um campo somente leitura visível externamente que é um tipo de referência mutável pode ser uma vulnerabilidade de segurança e pode disparar um aviso [CA2104](#) : “Não declare tipos de referência mutáveis somente leitura”.

- Em uma definição de tipo [`readonly struct`](#), `readonly` indica que o tipo de estrutura é imutável. Para saber mais, confira a seção [Struct readonly](#) do artigo [Tipos de struct](#).
- Em uma declaração de membro de instância dentro de um tipo de estrutura, `readonly` indica que um membro de instância não modifica o estado da estrutura. Para saber mais, confira a seção [Membros de instância readonly](#) do artigo [Tipos de struct](#).
- Em um [retorno de método ref readonly](#), o modificador `readonly` indica que o método retorna uma referência e que as gravações não são permitidas para essa referência.

# Exemplo de campo readonly

Neste exemplo, o valor do campo `year` não pode ser alterado no método `ChangeYear`, ainda que seja atribuído a ele um valor no construtor da classe:

C#

```
class Age
{
    private readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if uncommented.
    }
}
```

Você pode atribuir um valor a um campo `readonly` apenas nos seguintes contextos:

- Quando a variável é inicializada na declaração, por exemplo:

C#

```
public readonly int y = 5;
```

- Em um construtor de instância da classe que contém a declaração de campo de instância.
- No construtor estático da classe que contém a declaração do campo estático.

Esses contextos de construtor também são os únicos em que é válido passar um campo `readonly` como um parâmetro `out` ou `ref`.

## ⓘ Observação

A palavra-chave `readonly` é diferente da palavra-chave `const`. O campo `const` pode ser inicializado apenas na declaração do campo. Um campo `readonly` pode ser atribuído várias vezes na declaração do campo e em qualquer construtor. Portanto, campos `readonly` podem ter valores diferentes dependendo do construtor usado. Além disso, enquanto um campo `const` é uma constante em tempo de compilação, o campo `readonly` pode ser usado para constantes de runtime, como no exemplo a seguir:

C#

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

C#

```
public class SamplePoint
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SamplePoint()
    {
        // Initialize a readonly instance field
        z = 24;
    }

    public SamplePoint(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SamplePoint p1 = new SamplePoint(11, 21, 32);    // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        p2.x = 55;    // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
    /*
     * Output:
     *      p1: x=11, y=21, z=32
     *      p2: x=55, y=25, z=24
     */
}
```

No exemplo anterior, se você usar uma instrução semelhante ao seguinte exemplo:

C#

```
p2.y = 66;           // Error
```

você receberá a mensagem de erro do compilador:

Um campo somente leitura não pode ser atribuído (exceto em um construtor ou inicializador de variável)

## Exemplo de retorno de `readonly` de `ref`

O modificador `readonly` em um `ref return` indica que a referência retornada não pode ser modificada. O exemplo a seguir retorna uma referência para a origem. Ele usa o modificador `readonly` para indicar que os chamadores não podem modificar a origem:

C#

```
private static readonly SamplePoint s_origin = new SamplePoint(0, 0, 0);
public static ref readonly SamplePoint Origin => ref s_origin;
```

O tipo retornado não precisa ser um `readonly struct`. Qualquer tipo que possa ser retornado por `ref` pode ser retornado por `ref readonly`.

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

Você também pode ver as propostas de especificação de linguagem:

- [structs readonly ref e readonly](#)
- [membros do struct readonly](#)

## Confira também

- [Adicionar modificador `readonly` \(regra de estilo IDE0044\)](#)
- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores](#)
- [const](#)
- [Fields](#)

# sealed (Referência de C#)

Artigo • 07/04/2023

Quando aplicado a uma classe, o modificador `sealed` impede que outras classes herdem dela. No exemplo a seguir, a classe `B` herda da classe `A`, mas nenhuma classe pode herdar da classe `B`.

C#

```
class A {}
sealed class B : A {}
```

Você também pode usar o modificador `sealed` em um método ou propriedade que substitui um método ou propriedade virtual em uma classe base. Com isso, você pode permitir que classes sejam derivadas de sua classe e impedir que substituam métodos ou propriedades virtuais.

## Exemplo

No exemplo a seguir, `Z` herda de `Y`, mas `Z` não pode substituir a função virtual `F` declarada em `X` e lacrada em `Y`.

C#

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

Quando define novos métodos ou propriedades em uma classe, você pode impedir que classes derivadas as substituam ao não declará-las como [virtuais](#).

É um erro usar o modificador [abstract](#) com uma classe selada, porque uma classe abstrata deve ser herdada por uma classe que fornece uma implementação dos métodos ou propriedades abstratas.

Quando aplicado a um método ou propriedade, o modificador [sealed](#) sempre deve ser usado com [override](#).

Como structs são lacrados implicitamente, eles não podem ser herdados.

Para obter mais informações, consulte [Herança](#).

Para obter mais exemplos, consulte [Classes e membros de classes abstratas e lacradas](#).

C#

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150
```

No exemplo anterior, você pode tentar herdar da classe lacrada usando a instrução a seguir:

```
class MyDerivedC: SealedClass {} // Error
```

O resultado é uma mensagem de erro:

```
'MyDerivedC': cannot derive from sealed type 'SealedClass'
```

## Comentários

Para determinar se deve lacrar uma classe, método ou propriedade, geralmente você deve considerar os dois pontos a seguir:

- Os possíveis benefícios que as classes derivadas podem obter por meio da capacidade de personalizar a sua classe.
- O potencial de as classes derivadas modificarem suas classes de forma que elas deixem de funcionar corretamente ou como esperado.

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Classes static e membros de classes static](#)
- [Classes e membros de classes abstract e sealed](#)
- [Modificadores de acesso](#)
- [Modificadores](#)
- [override](#)
- [virtual](#)

# static (Referência de C#)

Artigo • 07/04/2023

Esta página aborda a palavra-chave do modificador `static`. A palavra-chave `static` também faz parte da diretiva [using static](#).

Use o modificador `static` para declarar um membro estático que pertença ao próprio tipo, em vez de um objeto específico. O modificador `static` pode ser usado para declarar classes `static`. Em classes, interfaces e structs, você pode adicionar o modificador `static` a campos, métodos, propriedades, operadores, eventos e construtores. O modificador `static` não pode ser usado com indexadores ou finalizadores. Para obter mais informações, consulte [Classes estáticas e membros de classes estáticas](#).

Você pode adicionar o `static` modificador a uma [função local](#). Uma função local estática não pode capturar variáveis locais ou o estado da instância.

A partir do C# 9.0, você pode adicionar o modificador `static` a uma [expressão lambda](#) ou [método anônimo](#). Um método anônimo estático não pode capturar variáveis locais ou o estado da instância entre escopos.

## Exemplo: classe estática

A seguinte classe é declarada como `static` e contém apenas métodos `static`:

```
C#  
  
static class CompanyEmployee  
{  
    public static void DoSomething() { /*...*/ }  
    public static void DoSomethingElse() { /*...*/ }  
}
```

Uma constante ou declaração de tipo é, implicitamente, um membro `static`. Um membro `static` não pode ser referenciado por meio de uma instância. Em vez disso, ele é referenciado pelo nome do tipo. Por exemplo, considere a seguinte classe:

```
C#  
  
public class MyBaseC  
{  
    public struct MyStruct
```

```
{  
    public static int x = 100;  
}
```

Para fazer referência ao `x` do membro `static`, use o nome totalmente qualificado `MyBaseC.MyStruct.x`, a menos que o membro esteja acessível a partir do mesmo escopo:

C#

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

Embora uma instância de uma classe contenha uma cópia separada de todos os campos de instância da classe, há apenas uma cópia de cada campo `static`.

Não é possível usar `this` para referenciar métodos `static` ou acessadores de propriedade.

Se a palavra-chave `static` for aplicada a uma classe, todos os membros da classe deverão ser `static`.

Classes, interfaces e classes `static` podem ter construtores `static`. Um construtor `static` é chamado em algum ponto entre o momento em que o programa é iniciado e a classe é instanciada.

#### ⓘ Observação

A palavra-chave `static` tem utilizações mais limitadas do que no C++. Para comparar com a palavra-chave do C++, consulte [Storage classes \(C++\)](#) (Classes de armazenamento (C++)).

Para demonstrar os membros `static`, considere uma classe que representa um funcionário da empresa. Suponha que a classe contém um método para contar funcionários e um campo para armazenar o número de funcionários. O método e o campo não pertencem a nenhuma instância de funcionário. Em vez disso, eles pertencem à classe de funcionários como um todo. Portanto, eles devem ser declarados como membros `static` da classe.

## Exemplo: campo e método estático

Este exemplo lê o nome e a ID de um novo funcionário, incrementa o contador de funcionário em um e exibe as informações do novo funcionário e do novo número de funcionários. Esse programa lê, do teclado, o número atual de funcionários.

C#

```
public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object.
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID: {e.id}");
        Console.WriteLine($"New Number of Employees:
{Employee4.employeeCounter}");
    }
}/*

```

```
Input:  
Matthias Berndt  
AF643G  
15  
*  
Sample Output:  
Enter the employee's name: Matthias Berndt  
Enter the employee's ID: AF643G  
Enter the current number of employees: 15  
Name: Matthias Berndt  
ID: AF643G  
New Number of Employees: 16  
*/
```

## Exemplo: inicialização estática

Este exemplo mostra que você pode inicializar um campo `static` usando outro campo `static` que ainda não foi declarado. Os resultados serão indefinidos até que você atribua explicitamente um valor ao `static` campo.

C#

```
class Test  
{  
    static int x = y;  
    static int y = 5;  
  
    static void Main()  
    {  
        Console.WriteLine(Test.x);  
        Console.WriteLine(Test.y);  
  
        Test.x = 99;  
        Console.WriteLine(Test.x);  
    }  
}  
/*  
Output:  
0  
5  
99  
*/
```

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- Referência de C#
- Guia de Programação em C#
- Palavras-chave do C#
- Modificadores
- using static directive
- Classes static e membros de classes static

# unsafe (Referência de C#)

Artigo • 08/06/2023

A palavra-chave `unsafe` denota um contexto inseguro, que é necessário para qualquer operação que envolva ponteiros. Para obter mais informações, consulte [Código não seguro e ponteiros](#).

Você pode usar o modificador `unsafe` na declaração de um tipo ou membro. Toda a extensão textual do tipo ou membro é, portanto, considerada um contexto inseguro. Por exemplo, a seguir está um método declarado com o modificador `unsafe`:

C#

```
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

O escopo do contexto inseguro se estende da lista de parâmetros até o final do método, portanto, os ponteiros também podem ser usados na lista de parâmetros:

C#

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

Você também pode usar um bloco não seguro para habilitar o uso de um código não seguro dentro desse bloco. Por exemplo:

C#

```
unsafe
{
    // Unsafe context: can use pointers here.
}
```

Para compilar o código não seguro, você deve especificar a opção do compilador [AllowUnsafeBlocks](#). O código não seguro não é verificável pelo Common Language Runtime.

## Exemplo

C#

```
// compile with: -unsafe
class UnsafeTest
{
    // Unsafe method: takes pointer to int.
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```

## Especificação da linguagem C#

Para saber mais, confira [código unsafe](#) na [Especificação da Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Palavras-chave de C#](#)
- [fixedinstrução](#)
- [Código não seguro, tipos de ponteiro e ponteiros de função](#)
- [Operadores relacionados a ponteiro](#)

# virtual (Referência de C#)

Artigo • 07/04/2023

A palavra-chave `virtual` é usada para modificar uma declaração de método, propriedade, indexador ou evento e permitir que ela seja substituída em uma classe derivada. Por exemplo, esse método pode ser substituído por qualquer classe que o herde:

C#

```
public virtual double Area()
{
    return x * y;
}
```

A implementação de um membro virtual pode ser alterada por um [membro de substituição](#) em uma classe derivada. Para obter mais informações sobre como usar a palavra-chave `virtual`, consulte [Controle de versão com as palavras-chave override e new](#) e [Quando usar as palavras-chave override e new](#).

## Comentários

Quando um método virtual é invocado, o tipo de tempo de execução do objeto é verificado para um membro de substituição. O membro de substituição na classe mais derivada é chamado, que pode ser o membro original, se nenhuma classe derivada tiver substituído o membro.

Por padrão, os métodos não são virtuais. Você não pode substituir um método não virtual.

Não é possível usar o modificador `virtual` com os modificadores `static`, `abstract`, `private` ou `override`. O exemplo a seguir mostra uma propriedade virtual:

C#

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int _num;
    public virtual int Number
```

```

    {
        get { return _num; }
        set { _num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string _name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
            }
            else
            {
                _name = "Unknown";
            }
        }
    }
}

```

Propriedades virtuais se comportam como métodos virtuais, exceto pelas diferenças na sintaxe de declaração e chamada.

- É um erro usar o modificador `virtual` em uma propriedade estática.
- Uma propriedade herdada virtual pode ser substituída em uma classe derivada incluindo uma declaração de propriedade que usa o modificador `override`.

## Exemplo

Neste exemplo, a classe `Shape` contém as duas coordenadas `x`, `y` e o método virtual `Area()`. Classes de forma diferentes como `Circle`, `Cylinder` e `Sphere` herdam a classe `Shape` e a área de superfície é calculada para cada figura. Cada classe derivada tem a própria implementação de substituição do `Area()`.

Observe que as classes herdadas `Circle`, `Sphere` e `Cylinder` usam construtores que inicializam a classe base, conforme mostrado na declaração a seguir.

C#

```
public Cylinder(double r, double h): base(r, h) {}
```

O programa a seguir calcula e exibe a área apropriada para cada figura invocando a implementação apropriada do método `Area()`, de acordo com o objeto que está associado ao método.

C#

```
class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double _x, _y;

        public Shape()
        {
        }

        public Shape(double x, double y)
        {
            _x = x;
            _y = y;
        }

        public virtual double Area()
        {
            return _x * _y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * _x * _x;
        }
    }

    public class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
```

```

    }

    public override double Area()
    {
        return 4 * PI * _x * _x;
    }
}

public class Cylinder : Shape
{
    public Cylinder(double r, double h) : base(r, h)
    {

    }

    public override double Area()
    {
        return 2 * PI * _x * _x + 2 * PI * _x * _y;
    }
}

static void Main()
{
    double r = 3.0, h = 5.0;
    Shape c = new Circle(r);
    Shape s = new Sphere(r);
    Shape l = new Cylinder(r, h);
    // Display results.
    Console.WriteLine("Area of Circle    = {0:F2}", c.Area());
    Console.WriteLine("Area of Sphere    = {0:F2}", s.Area());
    Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
}
/*
Output:
Area of Circle    = 28.27
Area of Sphere    = 113.10
Area of Cylinder = 150.80
*/

```

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- Polimorfismo
- abstract

- [override](#)
- [new \(modificador\)](#)

# volatile (Referência de C#)

Artigo • 07/04/2023

A palavra-chave `volatile` indica que um campo pode ser modificado por vários threads que estão em execução ao mesmo tempo. O compilador, o sistema do runtime e até mesmo o hardware podem reorganizar as leituras e gravações para locais de memória por motivos de desempenho. Os campos declarados `volatile` são excluídos de determinados tipos de otimizações. Não há nenhuma garantia de uma única ordenação total de gravações voláteis como visto em todos os threads de execução. Para obter mais informações, consulte a classe [Volatile](#).

## ⓘ Observação

Em um sistema multiprocessador, uma operação de leitura volátil não garante obter o valor mais recente gravado nesse local de memória por nenhum processador. Da mesma forma, uma operação de gravação volátil não garante que o valor gravado seja imediatamente visível para outros processadores.

A palavra-chave `volatile` pode ser aplicada a campos desses tipos:

- Tipos de referência.
- Tipos de ponteiro (em um contexto sem segurança). Observe que embora o ponteiro em si possa ser volátil, o objeto para o qual ele aponta não pode. Em outras palavras, você não pode declarar um "ponteiro como volátil".
- Tipos simples, como `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float` e `bool`.
- Um tipo `enum` com um dos seguintes tipos base: `byte`, `sbyte`, `short`, `ushort`, `int` ou `uint`.
- Parâmetros de tipo genérico conhecidos por serem tipos de referência.
- `IntPtr` e `UIntPtr`.

Outros tipos, inclusive `double` e `long`, não podem ser marcados como `volatile`, pois as leituras e gravações nos campos desses tipos não podem ser garantidas como atômicas. Para proteger o acesso multithreaded a esses tipos de campo, use os membros da classe [Interlocked](#) ou proteja o acesso usando a instrução `lock`.

A palavra-chave `volatile` pode ser aplicada somente aos campos de uma `class` ou `struct`. As variáveis locais não podem ser declaradas como `volatile`.

## Exemplo

O exemplo a seguir mostra como declarar uma variável de campo público como `volatile`.

C#

```
class VolatileTest
{
    public volatile int sharedStorage;

    public void Test(int i)
    {
        sharedStorage = i;
    }
}
```

O exemplo a seguir demonstra como um thread de trabalho ou auxiliar pode ser criado e usado para executar o processamento em paralelo com o do thread primário. Para saber mais sobre multithreading, confira [Threading gerenciado](#).

C#

```
public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        bool work = false;
        while (!_shouldStop)
        {
            work = !work; // simulate some work
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    public static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
    }
}
```

```

    workerThread.Start();
    Console.WriteLine("Main thread: starting worker thread...");

    // Loop until the worker thread activates.
    while (!workerThread.IsAlive)
        ;

    // Put the main thread to sleep for 500 milliseconds to
    // allow the worker thread to do some work.
    Thread.Sleep(500);

    // Request that the worker thread stop itself.
    workerObject.RequestStop();

    // Use the Thread.Join method to block the current thread
    // until the object's thread terminates.
    workerThread.Join();
    Console.WriteLine("Main thread: worker thread has terminated.");
}

// Sample output:
// Main thread: starting worker thread...
// Worker thread: terminating gracefully.
// Main thread: worker thread has terminated.
}

```

Com o modificador `volatile` adicionado à declaração de `_shouldStop` definida, você sempre obterá os mesmos resultados (semelhante ao trecho mostrado no código anterior). No entanto, sem esse modificador no membro `_shouldStop`, o comportamento é imprevisível. O método `DoWork` pode otimizar o acesso do membro, resultando na leitura de dados obsoletos. Devido à natureza da programação multithreaded, o número de leituras obsoletas é imprevisível. Diferentes execuções do programa produzirão resultados um pouco diferentes.

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Especificação da linguagem C#: palavra-chave volatile](#)
- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Modificadores](#)

- Instrução lock
- Interlocked

# Palavras-chave de instrução (Referência de C#)

Artigo • 07/04/2023

As instruções são instruções do programa. Exceto conforme descrito nos tópicos referenciados na tabela a seguir, as instruções são executadas em sequência. A tabela a seguir lista as palavras-chave da instrução C#. Para obter mais informações sobre instruções que não são expressos com qualquer palavra-chave, consulte [Instruções](#).

Categoria	Palavras-chave de C#
Instruções de seleção	<code>if</code> , <code>switch</code>
Instruções de iteração	<code>do</code> , <code>for</code> , <code>foreach</code> , <code>while</code>
Instruções de hiperlink	<code>break</code> , <code>continue</code> , <code>goto</code> , <code>return</code>
Instruções para tratamento de exceções	<code>throw</code> , <code>try-catch</code> , <code>try-finally</code> , <code>try-catch-finally</code>
Instruções checked e unchecked	<code>checked</code> , <code>unchecked</code>
Instrução fixed	<code>fixed</code>
Instrução lock	<code>lock</code>
Instrução yield	<code>yield</code>

## Confira também

- [Referência de C#](#)
- [Instruções](#)
- [Palavras-chave do C#](#)

# Instruções para manipulação de exceções – `throw`, `try-catch`, `try-finally` e `try-catch-finally`

Artigo • 05/06/2023

Use as instruções `throw` e `try` para trabalhar com exceções. Use a instrução `throw` para gerar uma exceção. Use a instrução `try` para capturar e tratar exceções que podem ocorrer durante a execução de um bloco de código.

## A instrução `throw`

A instrução `throw` lança uma exceção:

```
C#  
  
if (shapeAmount <= 0)  
{  
    throw new ArgumentException(nameof(shapeAmount), "Amount of  
    shapes must be positive.");  
}
```

Em uma instrução `throw e;`, o resultado da expressão `e` deve ser implicitamente conversível para [System.Exception](#).

Você pode usar as classes de exceção internas, por exemplo, [ArgumentOutOfRangeException](#) ou [InvalidOperationException](#). O .NET também fornece os métodos auxiliares para gerar exceções em determinadas condições: [ArgumentNullException.ThrowIfNull](#) e [ArgumentException.ThrowIfNullOrEmpty](#). Você também pode definir suas classes de exceção derivadas de [System.Exception](#). Para obter mais informações, consulte [Criando e lançando exceções](#).

Dentro de um [bloco catch](#), você pode usar uma instrução `throw;` para gerar novamente a exceção que é tratada pelo bloco `catch`:

```
C#  
  
try  
{  
    ProcessShapes(shapeAmount);  
}  
catch (Exception e)
```

```
{  
    LogError(e, "Shape processing failed.");  
    throw;  
}
```

## ① Observação

`throw;` preserva o rastreamento de pilha original da exceção, que é armazenado na propriedade `Exception.StackTrace`. Ao contrário disso, `throw e;` atualiza a propriedade `StackTrace` de `e`.

Quando uma exceção é lançada, o CLR (Common Language Runtime) procura o bloco `catch` que trata essa exceção. Se o método executado no momento não contiver um bloco `catch`, o CLR procurará no método que chamou o método atual e assim por diante para cima na pilha de chamadas. Se nenhum bloco `catch` for encontrado, o CLR encerrará o thread em execução. Para obter mais informações, consulte a seção [Como exceções são tratadas](#) da [Especificação da linguagem C#](#).

## A expressão `throw`

Você também pode usar `throw` como expressão. Isso pode ser conveniente em vários casos, que incluem:

- o operador condicional. O seguinte exemplo usa uma expressão `throw` para gerar um `ArgumentException` quando a matriz passada `args` está vazia:

C#

```
string first = args.Length >= 1  
    ? args[0]  
    : throw new ArgumentException("Please supply at least one  
argument.");
```

- o operador de união nula. O seguinte exemplo usa uma expressão `throw` para gerar um `ArgumentNullException` quando a cadeia de caracteres a ser atribuída a uma propriedade é `null`:

C#

```
public string Name  
{  
    get => name;  
    set => name = value ??
```

```
        throw new ArgumentNullException(paramName: nameof(value),  
    message: "Name cannot be null");  
}
```

- um método ou [lambda](#) com corpo de expressão. O seguinte exemplo usa uma expressão `throw` para gerar um [InvalidCastException](#) para indicar que não há suporte para uma conversão em um valor [DateTime](#):

C#

```
DateTime ToDateTime(IFormatProvider provider) =>  
    throw new InvalidCastException("Conversion to a DateTime is  
not supported.");
```

## A instrução try

Você pode usar a instrução `try` em qualquer uma das seguintes formas: [try-catch](#) - para lidar com exceções que podem ocorrer durante a execução do código dentro de um bloco `try`, [try-finally](#) - para especificar o código que é executado quando o controle sai do bloco `try` e [try-catch-finally](#) - como uma combinação dos dois formatos anteriores.

## A instrução try-catch

Use a instrução `try-catch` para tratar exceções que podem ocorrer durante a execução de um bloco de código. Coloque o código em que uma exceção pode ocorrer dentro de um bloco `try`. Use uma *cláusula catch* para especificar o tipo base das exceções que você deseja manipular no bloco `catch` correspondente:

C#

```
try  
{  
    var result = Process(-3, 4);  
    Console.WriteLine($"Processing succeeded: {result}");  
}  
catch (ArgumentException e)  
{  
    Console.WriteLine($"Processing failed: {e.Message}");  
}
```

Você pode fornecer várias cláusulas `catch`:

C#

```
try
{
    var result = await ProcessAsync(-3, 4, cancellationToken);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
catch (OperationCanceledException)
{
    Console.WriteLine("Processing is cancelled.");
}
```

Quando ocorre uma exceção, as cláusulas catch são examinadas na ordem especificada, de cima para baixo. No máximo, apenas um bloco `catch` é executado para qualquer exceção gerada. Como o exemplo anterior também mostra, você pode omitir a declaração de uma variável de exceção e especificar apenas o tipo de exceção em uma cláusula catch. Uma cláusula catch sem qualquer tipo de exceção especificado corresponde a qualquer exceção e, se presente, deve ser a última cláusula catch.

Se você quiser lançar novamente uma exceção capturada, use a [instrução throw](#), como mostra o seguinte exemplo:

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e)
{
    LogError(e, "Processing failed.");
    throw;
}
```

### ⚠ Observação

`throw;` preserva o rastreamento de pilha original da exceção, que é armazenado na propriedade `Exception.StackTrace`. Ao contrário disso, `throw e;` atualiza a propriedade `StackTrace` de `e`.

## Um filtro de exceção `when`

Junto com um tipo de exceção, você também pode especificar um filtro de exceção que examina ainda mais uma exceção e decide se o bloco `catch` correspondente manipula essa exceção. Um filtro de exceção é uma expressão booliana que segue a palavra-chave `when`, como mostra o seguinte exemplo:

```
C#  
  
try  
{  
    var result = Process(-3, 4);  
    Console.WriteLine($"Processing succeeded: {result}");  
}  
catch (Exception e) when (e is ArgumentException || e is  
DivideByZeroException)  
{  
    Console.WriteLine($"Processing failed: {e.Message}");  
}
```

O exemplo anterior usa um filtro de exceção para fornecer um bloco `catch` para lidar com exceções de dois tipos especificados.

Você poderá fornecer várias cláusulas `catch` para o mesmo tipo de exceção se elas distinguirem por filtros de exceção. Uma dessas cláusulas pode não ter filtro de exceção. Se essa cláusula existir, ela deverá ser a última das cláusulas que especificam esse tipo de exceção.

Se uma cláusula `catch` tiver um filtro de exceção, ela poderá especificar o tipo de exceção que é igual ou menos derivado do que um tipo de exceção de uma cláusula `catch` que aparece depois dela. Por exemplo, se um filtro de exceção estiver presente, uma cláusula `catch (Exception e)` não precisará ser a última cláusula.

## Exceções nos métodos assíncrono e iterador

Se ocorrer uma exceção em uma função assíncrona, ela será propagada para o chamador da função quando você `aguardar` o resultado da função, como mostra o seguinte exemplo:

```
C#  
  
public static async Task Run()  
{  
    try  
    {  
        Task<int> processing = ProcessAsync(-1);  
        Console.WriteLine("Launched processing.");  
    }
```

```

        int result = await processing;
        Console.WriteLine($"Result: {result}");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine($"Processing failed: {e.Message}");
    }
    // Output:
    // Launched processing.
    // Processing failed: Input must be non-negative. (Parameter 'input')
}

private static async Task<int> ProcessAsync(int input)
{
    if (input < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(input), "Input must be
non-negative.");
    }

    await Task.Delay(500);
    return input;
}

```

Se ocorrer uma exceção em um [método iterador](#), ela se propagará para o chamador somente quando o iterador avançar para o próximo elemento.

## A instrução try-finally

Em uma instrução `try-finally`, o bloco `finally` é executado quando o controle sai do bloco `try`. O controle pode deixar o bloco `try` como resultado de

- execução normal,
- execução de uma [instrução de salto](#) (ou seja, `return`, `break`, `continue` ou `goto`), ou
- propagação de uma exceção fora do bloco `try`.

O exemplo a seguir usa o bloco `finally` para redefinir o estado de um objeto antes que o controle deixe o método:

C#

```

public async Task HandleRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    finally
    {
        Busy = false;
    }
}

```

```
    }
    finally
    {
        Busy = false;
    }
}
```

Você também pode usar o bloco `finally` para limpar recursos alocados usados no bloco `try`.

### ⚠ Observação

Quando o tipo de um recurso implementa a interface `IDisposable` ou `IAsyncDisposable`, considere a instrução `using`. A instrução `using` garante que os recursos adquiridos sejam descartados quando o controle sair da instrução `using`. O compilador transforma uma instrução `using` em uma instrução `try-finally`.

Em quase todos os casos, os blocos `finally` são executados. Os únicos casos em que os blocos `finally` não são executados envolvem o encerramento imediato de um programa. Por exemplo, esse encerramento pode ocorrer devido à chamada `Environment.FailFast` ou a uma exceção `OverflowException` ou `InvalidOperationException`. A maioria dos sistemas operacionais realiza uma limpeza razoável de recursos como parte da interrupção e do descarregamento do processo.

## A instrução `try-catch-finally`

Use uma instrução `try-catch-finally` para lidar com exceções que podem ocorrer durante a execução do bloco `try` e especifique o código que deve ser executado quando o controle sair da instrução `try`:

C#

```
public async Task ProcessRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    catch (Exception e) when (e is not OperationCanceledException)
    {
        LogError(e, $"Failed to process request for item ID {itemId}.");
        throw;
    }
}
```

```
    }
    finally
    {
        Busy = false;
    }

}
```

Quando uma exceção é tratada por um bloco `catch`, o bloco `finally` é executado após a execução desse bloco `catch` (mesmo que outra exceção ocorra durante a execução do bloco `catch`). Para obter informações sobre blocos `catch` e `finally`, consulte as seções [A instrução try-catch](#) e [A instrução try-finally](#), respectivamente.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Instrução throw](#)
- [Instrução try](#)
- [Exceções](#)

## Confira também

- [Referência de C#](#)
- [Exceções e manipulação de exceções](#)
- [Tratando e gerando exceções no .NET](#)
- [gerar preferências \(regra de estilo IDE0016\)](#)
- [AppDomain.FirstChanceException](#)
- [AppDomain.UnhandledException](#)
- [TaskScheduler.UnobservedTaskException](#)

# Instruções para manipulação de exceções – `throw`, `try-catch`, `try-finally` e `try-catch-finally`

Artigo • 05/06/2023

Use as instruções `throw` e `try` para trabalhar com exceções. Use a instrução `throw` para gerar uma exceção. Use a instrução `try` para capturar e tratar exceções que podem ocorrer durante a execução de um bloco de código.

## A instrução `throw`

A instrução `throw` lança uma exceção:

```
C#  
  
if (shapeAmount <= 0)  
{  
    throw new ArgumentException(nameof(shapeAmount), "Amount of  
    shapes must be positive.");  
}
```

Em uma instrução `throw e;`, o resultado da expressão `e` deve ser implicitamente conversível para [System.Exception](#).

Você pode usar as classes de exceção internas, por exemplo, [ArgumentOutOfRangeException](#) ou [InvalidOperationException](#). O .NET também fornece os métodos auxiliares para gerar exceções em determinadas condições: [ArgumentNullException.ThrowIfNull](#) e [ArgumentException.ThrowIfNullOrEmpty](#). Você também pode definir suas classes de exceção derivadas de [System.Exception](#). Para obter mais informações, consulte [Criando e lançando exceções](#).

Dentro de um [bloco catch](#), você pode usar uma instrução `throw;` para gerar novamente a exceção que é tratada pelo bloco `catch`:

```
C#  
  
try  
{  
    ProcessShapes(shapeAmount);  
}  
catch (Exception e)
```

```
{  
    LogError(e, "Shape processing failed.");  
    throw;  
}
```

## ① Observação

`throw;` preserva o rastreamento de pilha original da exceção, que é armazenado na propriedade `Exception.StackTrace`. Ao contrário disso, `throw e;` atualiza a propriedade `StackTrace` de `e`.

Quando uma exceção é lançada, o CLR (Common Language Runtime) procura o bloco `catch` que trata essa exceção. Se o método executado no momento não contiver um bloco `catch`, o CLR procurará no método que chamou o método atual e assim por diante para cima na pilha de chamadas. Se nenhum bloco `catch` for encontrado, o CLR encerrará o thread em execução. Para obter mais informações, consulte a seção [Como exceções são tratadas](#) da [Especificação da linguagem C#](#).

## A expressão `throw`

Você também pode usar `throw` como expressão. Isso pode ser conveniente em vários casos, que incluem:

- o operador condicional. O seguinte exemplo usa uma expressão `throw` para gerar um `ArgumentException` quando a matriz passada `args` está vazia:

C#

```
string first = args.Length >= 1  
    ? args[0]  
    : throw new ArgumentException("Please supply at least one  
argument.");
```

- o operador de união nula. O seguinte exemplo usa uma expressão `throw` para gerar um `ArgumentNullException` quando a cadeia de caracteres a ser atribuída a uma propriedade é `null`:

C#

```
public string Name  
{  
    get => name;  
    set => name = value ??
```

```
        throw new ArgumentNullException(paramName: nameof(value),  
    message: "Name cannot be null");  
}
```

- um método ou [lambda](#) com corpo de expressão. O seguinte exemplo usa uma expressão `throw` para gerar um [InvalidCastException](#) para indicar que não há suporte para uma conversão em um valor [DateTime](#):

C#

```
DateTime ToDateTime(IFormatProvider provider) =>  
    throw new InvalidCastException("Conversion to a DateTime is  
not supported.");
```

## A instrução `try`

Você pode usar a instrução `try` em qualquer uma das seguintes formas: [try-catch](#) - para lidar com exceções que podem ocorrer durante a execução do código dentro de um bloco `try`, [try-finally](#) - para especificar o código que é executado quando o controle sai do bloco `try` e [try-catch-finally](#) - como uma combinação dos dois formatos anteriores.

## A instrução `try-catch`

Use a instrução `try-catch` para tratar exceções que podem ocorrer durante a execução de um bloco de código. Coloque o código em que uma exceção pode ocorrer dentro de um bloco `try`. Use uma *cláusula catch* para especificar o tipo base das exceções que você deseja manipular no bloco `catch` correspondente:

C#

```
try  
{  
    var result = Process(-3, 4);  
    Console.WriteLine($"Processing succeeded: {result}");  
}  
catch (ArgumentException e)  
{  
    Console.WriteLine($"Processing failed: {e.Message}");  
}
```

Você pode fornecer várias cláusulas `catch`:

C#

```
try
{
    var result = await ProcessAsync(-3, 4, cancellationToken);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
catch (OperationCanceledException)
{
    Console.WriteLine("Processing is cancelled.");
}
```

Quando ocorre uma exceção, as cláusulas catch são examinadas na ordem especificada, de cima para baixo. No máximo, apenas um bloco `catch` é executado para qualquer exceção gerada. Como o exemplo anterior também mostra, você pode omitir a declaração de uma variável de exceção e especificar apenas o tipo de exceção em uma cláusula catch. Uma cláusula catch sem qualquer tipo de exceção especificado corresponde a qualquer exceção e, se presente, deve ser a última cláusula catch.

Se você quiser lançar novamente uma exceção capturada, use a [instrução throw](#), como mostra o seguinte exemplo:

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e)
{
    LogError(e, "Processing failed.");
    throw;
}
```

### ⚠ Observação

`throw;` preserva o rastreamento de pilha original da exceção, que é armazenado na propriedade `Exception.StackTrace`. Ao contrário disso, `throw e;` atualiza a propriedade `StackTrace` de `e`.

Um filtro de exceção `when`

Junto com um tipo de exceção, você também pode especificar um filtro de exceção que examina ainda mais uma exceção e decide se o bloco `catch` correspondente manipula essa exceção. Um filtro de exceção é uma expressão booliana que segue a palavra-chave `when`, como mostra o seguinte exemplo:

```
C#  
  
try  
{  
    var result = Process(-3, 4);  
    Console.WriteLine($"Processing succeeded: {result}");  
}  
catch (Exception e) when (e is ArgumentException || e is  
DivideByZeroException)  
{  
    Console.WriteLine($"Processing failed: {e.Message}");  
}
```

O exemplo anterior usa um filtro de exceção para fornecer um bloco `catch` para lidar com exceções de dois tipos especificados.

Você poderá fornecer várias cláusulas `catch` para o mesmo tipo de exceção se elas distinguirem por filtros de exceção. Uma dessas cláusulas pode não ter filtro de exceção. Se essa cláusula existir, ela deverá ser a última das cláusulas que especificam esse tipo de exceção.

Se uma cláusula `catch` tiver um filtro de exceção, ela poderá especificar o tipo de exceção que é igual ou menos derivado do que um tipo de exceção de uma cláusula `catch` que aparece depois dela. Por exemplo, se um filtro de exceção estiver presente, uma cláusula `catch (Exception e)` não precisará ser a última cláusula.

## Exceções nos métodos assíncrono e iterador

Se ocorrer uma exceção em uma função assíncrona, ela será propagada para o chamador da função quando você `aguardar` o resultado da função, como mostra o seguinte exemplo:

```
C#  
  
public static async Task Run()  
{  
    try  
    {  
        Task<int> processing = ProcessAsync(-1);  
        Console.WriteLine("Launched processing.");  
    }
```

```

        int result = await processing;
        Console.WriteLine($"Result: {result}");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine($"Processing failed: {e.Message}");
    }
    // Output:
    // Launched processing.
    // Processing failed: Input must be non-negative. (Parameter 'input')
}

private static async Task<int> ProcessAsync(int input)
{
    if (input < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(input), "Input must be
non-negative.");
    }

    await Task.Delay(500);
    return input;
}

```

Se ocorrer uma exceção em um [método iterador](#), ela se propagará para o chamador somente quando o iterador avançar para o próximo elemento.

## A instrução `try-finally`

Em uma instrução `try-finally`, o bloco `finally` é executado quando o controle sai do bloco `try`. O controle pode deixar o bloco `try` como resultado de

- execução normal,
- execução de uma [instrução de salto](#) (ou seja, `return`, `break`, `continue` ou `goto`), ou
- propagação de uma exceção fora do bloco `try`.

O exemplo a seguir usa o bloco `finally` para redefinir o estado de um objeto antes que o controle deixe o método:

C#

```

public async Task HandleRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    finally
    {
        Busy = false;
    }
}

```

```
    }
    finally
    {
        Busy = false;
    }
}
```

Você também pode usar o bloco `finally` para limpar recursos alocados usados no bloco `try`.

### ⚠ Observação

Quando o tipo de um recurso implementa a interface `IDisposable` ou `IAsyncDisposable`, considere a instrução `using`. A instrução `using` garante que os recursos adquiridos sejam descartados quando o controle sair da instrução `using`. O compilador transforma uma instrução `using` em uma instrução `try-finally`.

Em quase todos os casos, os blocos `finally` são executados. Os únicos casos em que os blocos `finally` não são executados envolvem o encerramento imediato de um programa. Por exemplo, esse encerramento pode ocorrer devido à chamada `Environment.FailFast` ou a uma exceção `OverflowException` ou `InvalidOperationException`. A maioria dos sistemas operacionais realiza uma limpeza razoável de recursos como parte da interrupção e do descarregamento do processo.

## A instrução `try-catch-finally`

Use uma instrução `try-catch-finally` para lidar com exceções que podem ocorrer durante a execução do bloco `try` e especifique o código que deve ser executado quando o controle sair da instrução `try`:

C#

```
public async Task ProcessRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    catch (Exception e) when (e is not OperationCanceledException)
    {
        LogError(e, $"Failed to process request for item ID {itemId}.");
        throw;
    }
}
```

```
    }
    finally
    {
        Busy = false;
    }

}
```

Quando uma exceção é tratada por um bloco `catch`, o bloco `finally` é executado após a execução desse bloco `catch` (mesmo que outra exceção ocorra durante a execução do bloco `catch`). Para obter informações sobre blocos `catch` e `finally`, consulte as seções [A instrução try-catch](#) e [A instrução try-finally](#), respectivamente.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Instrução throw](#)
- [Instrução try](#)
- [Exceções](#)

## Confira também

- [Referência de C#](#)
- [Exceções e manipulação de exceções](#)
- [Tratando e gerando exceções no .NET](#)
- [gerar preferências \(regra de estilo IDE0016\)](#)
- [AppDomain.FirstChanceException](#)
- [AppDomain.UnhandledException](#)
- [TaskScheduler.UnobservedTaskException](#)

# Instruções para manipulação de exceções – `throw`, `try-catch`, `try-finally` e `try-catch-finally`

Artigo • 05/06/2023

Use as instruções `throw` e `try` para trabalhar com exceções. Use a instrução `throw` para gerar uma exceção. Use a instrução `try` para capturar e tratar exceções que podem ocorrer durante a execução de um bloco de código.

## A instrução `throw`

A instrução `throw` lança uma exceção:

```
C#  
  
if (shapeAmount <= 0)  
{  
    throw new ArgumentException(nameof(shapeAmount), "Amount of  
    shapes must be positive.");  
}
```

Em uma instrução `throw e;`, o resultado da expressão `e` deve ser implicitamente conversível para [System.Exception](#).

Você pode usar as classes de exceção internas, por exemplo, [ArgumentOutOfRangeException](#) ou [InvalidOperationException](#). O .NET também fornece os métodos auxiliares para gerar exceções em determinadas condições: [ArgumentNullException.ThrowIfNull](#) e [ArgumentException.ThrowIfNullOrEmpty](#). Você também pode definir suas classes de exceção derivadas de [System.Exception](#). Para obter mais informações, consulte [Criando e lançando exceções](#).

Dentro de um [bloco catch](#), você pode usar uma instrução `throw;` para gerar novamente a exceção que é tratada pelo bloco `catch`:

```
C#  
  
try  
{  
    ProcessShapes(shapeAmount);  
}  
catch (Exception e)
```

```
{  
    LogError(e, "Shape processing failed.");  
    throw;  
}
```

## ① Observação

`throw;` preserva o rastreamento de pilha original da exceção, que é armazenado na propriedade `Exception.StackTrace`. Ao contrário disso, `throw e;` atualiza a propriedade `StackTrace` de `e`.

Quando uma exceção é lançada, o CLR (Common Language Runtime) procura o bloco `catch` que trata essa exceção. Se o método executado no momento não contiver um bloco `catch`, o CLR procurará no método que chamou o método atual e assim por diante para cima na pilha de chamadas. Se nenhum bloco `catch` for encontrado, o CLR encerrará o thread em execução. Para obter mais informações, consulte a seção [Como exceções são tratadas](#) da [Especificação da linguagem C#](#).

## A expressão `throw`

Você também pode usar `throw` como expressão. Isso pode ser conveniente em vários casos, que incluem:

- o operador condicional. O seguinte exemplo usa uma expressão `throw` para gerar um `ArgumentException` quando a matriz passada `args` está vazia:

C#

```
string first = args.Length >= 1  
    ? args[0]  
    : throw new ArgumentException("Please supply at least one  
argument.");
```

- o operador de união nula. O seguinte exemplo usa uma expressão `throw` para gerar um `ArgumentNullException` quando a cadeia de caracteres a ser atribuída a uma propriedade é `null`:

C#

```
public string Name  
{  
    get => name;  
    set => name = value ??
```

```
        throw new ArgumentNullException(paramName: nameof(value),  
    message: "Name cannot be null");  
}
```

- um método ou [lambda](#) com corpo de expressão. O seguinte exemplo usa uma expressão `throw` para gerar um [InvalidCastException](#) para indicar que não há suporte para uma conversão em um valor [DateTime](#):

C#

```
DateTime ToDateTime(IFormatProvider provider) =>  
    throw new InvalidCastException("Conversion to a DateTime is  
not supported.");
```

## A instrução try

Você pode usar a instrução `try` em qualquer uma das seguintes formas: [try-catch](#) - para lidar com exceções que podem ocorrer durante a execução do código dentro de um bloco `try`, [try-finally](#) - para especificar o código que é executado quando o controle sai do bloco `try` e [try-catch-finally](#) - como uma combinação dos dois formatos anteriores.

## A instrução try-catch

Use a instrução `try-catch` para tratar exceções que podem ocorrer durante a execução de um bloco de código. Coloque o código em que uma exceção pode ocorrer dentro de um bloco `try`. Use uma *cláusula catch* para especificar o tipo base das exceções que você deseja manipular no bloco `catch` correspondente:

C#

```
try  
{  
    var result = Process(-3, 4);  
    Console.WriteLine($"Processing succeeded: {result}");  
}  
catch (ArgumentException e)  
{  
    Console.WriteLine($"Processing failed: {e.Message}");  
}
```

Você pode fornecer várias cláusulas `catch`:

C#

```
try
{
    var result = await ProcessAsync(-3, 4, cancellationToken);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
catch (OperationCanceledException)
{
    Console.WriteLine("Processing is cancelled.");
}
```

Quando ocorre uma exceção, as cláusulas catch são examinadas na ordem especificada, de cima para baixo. No máximo, apenas um bloco `catch` é executado para qualquer exceção gerada. Como o exemplo anterior também mostra, você pode omitir a declaração de uma variável de exceção e especificar apenas o tipo de exceção em uma cláusula catch. Uma cláusula catch sem qualquer tipo de exceção especificado corresponde a qualquer exceção e, se presente, deve ser a última cláusula catch.

Se você quiser lançar novamente uma exceção capturada, use a [instrução throw](#), como mostra o seguinte exemplo:

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e)
{
    LogError(e, "Processing failed.");
    throw;
}
```

### ⚠ Observação

`throw;` preserva o rastreamento de pilha original da exceção, que é armazenado na propriedade `Exception.StackTrace`. Ao contrário disso, `throw e;` atualiza a propriedade `StackTrace` de `e`.

Um filtro de exceção `when`

Junto com um tipo de exceção, você também pode especificar um filtro de exceção que examina ainda mais uma exceção e decide se o bloco `catch` correspondente manipula essa exceção. Um filtro de exceção é uma expressão booliana que segue a palavra-chave `when`, como mostra o seguinte exemplo:

```
C#  
  
try  
{  
    var result = Process(-3, 4);  
    Console.WriteLine($"Processing succeeded: {result}");  
}  
catch (Exception e) when (e is ArgumentException || e is  
DivideByZeroException)  
{  
    Console.WriteLine($"Processing failed: {e.Message}");  
}
```

O exemplo anterior usa um filtro de exceção para fornecer um bloco `catch` para lidar com exceções de dois tipos especificados.

Você poderá fornecer várias cláusulas `catch` para o mesmo tipo de exceção se elas distinguirem por filtros de exceção. Uma dessas cláusulas pode não ter filtro de exceção. Se essa cláusula existir, ela deverá ser a última das cláusulas que especificam esse tipo de exceção.

Se uma cláusula `catch` tiver um filtro de exceção, ela poderá especificar o tipo de exceção que é igual ou menos derivado do que um tipo de exceção de uma cláusula `catch` que aparece depois dela. Por exemplo, se um filtro de exceção estiver presente, uma cláusula `catch (Exception e)` não precisará ser a última cláusula.

## Exceções nos métodos assíncrono e iterador

Se ocorrer uma exceção em uma função assíncrona, ela será propagada para o chamador da função quando você `aguardar` o resultado da função, como mostra o seguinte exemplo:

```
C#  
  
public static async Task Run()  
{  
    try  
    {  
        Task<int> processing = ProcessAsync(-1);  
        Console.WriteLine("Launched processing.");  
    }
```

```

        int result = await processing;
        Console.WriteLine($"Result: {result}");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine($"Processing failed: {e.Message}");
    }
    // Output:
    // Launched processing.
    // Processing failed: Input must be non-negative. (Parameter 'input')
}

private static async Task<int> ProcessAsync(int input)
{
    if (input < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(input), "Input must be
non-negative.");
    }

    await Task.Delay(500);
    return input;
}

```

Se ocorrer uma exceção em um [método iterador](#), ela se propagará para o chamador somente quando o iterador avançar para o próximo elemento.

## A instrução `try-finally`

Em uma instrução `try-finally`, o bloco `finally` é executado quando o controle sai do bloco `try`. O controle pode deixar o bloco `try` como resultado de

- execução normal,
- execução de uma [instrução de salto](#) (ou seja, `return`, `break`, `continue` ou `goto`), ou
- propagação de uma exceção fora do bloco `try`.

O exemplo a seguir usa o bloco `finally` para redefinir o estado de um objeto antes que o controle deixe o método:

C#

```

public async Task HandleRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    finally
    {
        Busy = false;
    }
}

```

```
    }
    finally
    {
        Busy = false;
    }
}
```

Você também pode usar o bloco `finally` para limpar recursos alocados usados no bloco `try`.

### ⚠ Observação

Quando o tipo de um recurso implementa a interface `IDisposable` ou `IAsyncDisposable`, considere a instrução `using`. A instrução `using` garante que os recursos adquiridos sejam descartados quando o controle sair da instrução `using`. O compilador transforma uma instrução `using` em uma instrução `try-finally`.

Em quase todos os casos, os blocos `finally` são executados. Os únicos casos em que os blocos `finally` não são executados envolvem o encerramento imediato de um programa. Por exemplo, esse encerramento pode ocorrer devido à chamada `Environment.FailFast` ou a uma exceção `OverflowException` ou `InvalidOperationException`. A maioria dos sistemas operacionais realiza uma limpeza razoável de recursos como parte da interrupção e do descarregamento do processo.

## A instrução `try-catch-finally`

Use uma instrução `try-catch-finally` para lidar com exceções que podem ocorrer durante a execução do bloco `try` e especifique o código que deve ser executado quando o controle sair da instrução `try`:

C#

```
public async Task ProcessRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    catch (Exception e) when (e is not OperationCanceledException)
    {
        LogError(e, $"Failed to process request for item ID {itemId}.");
        throw;
    }
}
```

```
    }
    finally
    {
        Busy = false;
    }

}
```

Quando uma exceção é tratada por um bloco `catch`, o bloco `finally` é executado após a execução desse bloco `catch` (mesmo que outra exceção ocorra durante a execução do bloco `catch`). Para obter informações sobre blocos `catch` e `finally`, consulte as seções [A instrução try-catch](#) e [A instrução try-finally](#), respectivamente.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Instrução throw](#)
- [Instrução try](#)
- [Exceções](#)

## Confira também

- [Referência de C#](#)
- [Exceções e manipulação de exceções](#)
- [Tratando e gerando exceções no .NET](#)
- [gerar preferências \(regra de estilo IDE0016\)](#)
- [AppDomain.FirstChanceException](#)
- [AppDomain.UnhandledException](#)
- [TaskScheduler.UnobservedTaskException](#)

# Instruções para manipulação de exceções – `throw`, `try-catch`, `try-finally` e `try-catch-finally`

Artigo • 05/06/2023

Use as instruções `throw` e `try` para trabalhar com exceções. Use a instrução `throw` para gerar uma exceção. Use a instrução `try` para capturar e tratar exceções que podem ocorrer durante a execução de um bloco de código.

## A instrução `throw`

A instrução `throw` lança uma exceção:

```
C#  
  
if (shapeAmount <= 0)  
{  
    throw new ArgumentException(nameof(shapeAmount), "Amount of  
    shapes must be positive.");  
}
```

Em uma instrução `throw e;`, o resultado da expressão `e` deve ser implicitamente conversível para [System.Exception](#).

Você pode usar as classes de exceção internas, por exemplo, [ArgumentOutOfRangeException](#) ou [InvalidOperationException](#). O .NET também fornece os métodos auxiliares para gerar exceções em determinadas condições: [ArgumentNullException.ThrowIfNull](#) e [ArgumentException.ThrowIfNullOrEmpty](#). Você também pode definir suas classes de exceção derivadas de [System.Exception](#). Para obter mais informações, consulte [Criando e lançando exceções](#).

Dentro de um [bloco catch](#), você pode usar uma instrução `throw;` para gerar novamente a exceção que é tratada pelo bloco `catch`:

```
C#  
  
try  
{  
    ProcessShapes(shapeAmount);  
}  
catch (Exception e)
```

```
{  
    LogError(e, "Shape processing failed.");  
    throw;  
}
```

## ① Observação

`throw;` preserva o rastreamento de pilha original da exceção, que é armazenado na propriedade `Exception.StackTrace`. Ao contrário disso, `throw e;` atualiza a propriedade `StackTrace` de `e`.

Quando uma exceção é lançada, o CLR (Common Language Runtime) procura o bloco `catch` que trata essa exceção. Se o método executado no momento não contiver um bloco `catch`, o CLR procurará no método que chamou o método atual e assim por diante para cima na pilha de chamadas. Se nenhum bloco `catch` for encontrado, o CLR encerrará o thread em execução. Para obter mais informações, consulte a seção [Como exceções são tratadas](#) da [Especificação da linguagem C#](#).

## A expressão `throw`

Você também pode usar `throw` como expressão. Isso pode ser conveniente em vários casos, que incluem:

- o operador condicional. O seguinte exemplo usa uma expressão `throw` para gerar um `ArgumentException` quando a matriz passada `args` está vazia:

C#

```
string first = args.Length >= 1  
    ? args[0]  
    : throw new ArgumentException("Please supply at least one  
argument.");
```

- o operador de união nula. O seguinte exemplo usa uma expressão `throw` para gerar um `ArgumentNullException` quando a cadeia de caracteres a ser atribuída a uma propriedade é `null`:

C#

```
public string Name  
{  
    get => name;  
    set => name = value ??
```

```
        throw new ArgumentNullException(paramName: nameof(value),  
    message: "Name cannot be null");  
}
```

- um método ou [lambda](#) com corpo de expressão. O seguinte exemplo usa uma expressão `throw` para gerar um [InvalidCastException](#) para indicar que não há suporte para uma conversão em um valor [DateTime](#):

C#

```
DateTime ToDateTime(IFormatProvider provider) =>  
    throw new InvalidCastException("Conversion to a DateTime is  
not supported.");
```

## A instrução `try`

Você pode usar a instrução `try` em qualquer uma das seguintes formas: [try-catch](#) - para lidar com exceções que podem ocorrer durante a execução do código dentro de um bloco `try`, [try-finally](#) - para especificar o código que é executado quando o controle sai do bloco `try` e [try-catch-finally](#) - como uma combinação dos dois formatos anteriores.

## A instrução `try-catch`

Use a instrução `try-catch` para tratar exceções que podem ocorrer durante a execução de um bloco de código. Coloque o código em que uma exceção pode ocorrer dentro de um bloco `try`. Use uma *cláusula catch* para especificar o tipo base das exceções que você deseja manipular no bloco `catch` correspondente:

C#

```
try  
{  
    var result = Process(-3, 4);  
    Console.WriteLine($"Processing succeeded: {result}");  
}  
catch (ArgumentException e)  
{  
    Console.WriteLine($"Processing failed: {e.Message}");  
}
```

Você pode fornecer várias cláusulas `catch`:

C#

```
try
{
    var result = await ProcessAsync(-3, 4, cancellationToken);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
catch (OperationCanceledException)
{
    Console.WriteLine("Processing is cancelled.");
}
```

Quando ocorre uma exceção, as cláusulas catch são examinadas na ordem especificada, de cima para baixo. No máximo, apenas um bloco `catch` é executado para qualquer exceção gerada. Como o exemplo anterior também mostra, você pode omitir a declaração de uma variável de exceção e especificar apenas o tipo de exceção em uma cláusula catch. Uma cláusula catch sem qualquer tipo de exceção especificado corresponde a qualquer exceção e, se presente, deve ser a última cláusula catch.

Se você quiser lançar novamente uma exceção capturada, use a [instrução throw](#), como mostra o seguinte exemplo:

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e)
{
    LogError(e, "Processing failed.");
    throw;
}
```

### ⚠ Observação

`throw;` preserva o rastreamento de pilha original da exceção, que é armazenado na propriedade `Exception.StackTrace`. Ao contrário disso, `throw e;` atualiza a propriedade `StackTrace` de `e`.

## Um filtro de exceção `when`

Junto com um tipo de exceção, você também pode especificar um filtro de exceção que examina ainda mais uma exceção e decide se o bloco `catch` correspondente manipula essa exceção. Um filtro de exceção é uma expressão booliana que segue a palavra-chave `when`, como mostra o seguinte exemplo:

```
C#  
  
try  
{  
    var result = Process(-3, 4);  
    Console.WriteLine($"Processing succeeded: {result}");  
}  
catch (Exception e) when (e is ArgumentException || e is  
DivideByZeroException)  
{  
    Console.WriteLine($"Processing failed: {e.Message}");  
}
```

O exemplo anterior usa um filtro de exceção para fornecer um bloco `catch` para lidar com exceções de dois tipos especificados.

Você poderá fornecer várias cláusulas `catch` para o mesmo tipo de exceção se elas distinguirem por filtros de exceção. Uma dessas cláusulas pode não ter filtro de exceção. Se essa cláusula existir, ela deverá ser a última das cláusulas que especificam esse tipo de exceção.

Se uma cláusula `catch` tiver um filtro de exceção, ela poderá especificar o tipo de exceção que é igual ou menos derivado do que um tipo de exceção de uma cláusula `catch` que aparece depois dela. Por exemplo, se um filtro de exceção estiver presente, uma cláusula `catch (Exception e)` não precisará ser a última cláusula.

## Exceções nos métodos assíncrono e iterador

Se ocorrer uma exceção em uma função assíncrona, ela será propagada para o chamador da função quando você `aguardar` o resultado da função, como mostra o seguinte exemplo:

```
C#  
  
public static async Task Run()  
{  
    try  
    {  
        Task<int> processing = ProcessAsync(-1);  
        Console.WriteLine("Launched processing.");  
    }
```

```

        int result = await processing;
        Console.WriteLine($"Result: {result}");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine($"Processing failed: {e.Message}");
    }
    // Output:
    // Launched processing.
    // Processing failed: Input must be non-negative. (Parameter 'input')
}

private static async Task<int> ProcessAsync(int input)
{
    if (input < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(input), "Input must be
non-negative.");
    }

    await Task.Delay(500);
    return input;
}

```

Se ocorrer uma exceção em um [método iterador](#), ela se propagará para o chamador somente quando o iterador avançar para o próximo elemento.

## A instrução `try-finally`

Em uma instrução `try-finally`, o bloco `finally` é executado quando o controle sai do bloco `try`. O controle pode deixar o bloco `try` como resultado de

- execução normal,
- execução de uma [instrução de salto](#) (ou seja, `return`, `break`, `continue` ou `goto`), ou
- propagação de uma exceção fora do bloco `try`.

O exemplo a seguir usa o bloco `finally` para redefinir o estado de um objeto antes que o controle deixe o método:

C#

```

public async Task HandleRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    finally
    {
        Busy = false;
    }
}

```

```
    }
    finally
    {
        Busy = false;
    }
}
```

Você também pode usar o bloco `finally` para limpar recursos alocados usados no bloco `try`.

### ⚠ Observação

Quando o tipo de um recurso implementa a interface `IDisposable` ou `IAsyncDisposable`, considere a instrução `using`. A instrução `using` garante que os recursos adquiridos sejam descartados quando o controle sair da instrução `using`. O compilador transforma uma instrução `using` em uma instrução `try-finally`.

Em quase todos os casos, os blocos `finally` são executados. Os únicos casos em que os blocos `finally` não são executados envolvem o encerramento imediato de um programa. Por exemplo, esse encerramento pode ocorrer devido à chamada `Environment.FailFast` ou a uma exceção `OverflowException` ou `InvalidOperationException`. A maioria dos sistemas operacionais realiza uma limpeza razoável de recursos como parte da interrupção e do descarregamento do processo.

## A instrução `try-catch-finally`

Use uma instrução `try-catch-finally` para lidar com exceções que podem ocorrer durante a execução do bloco `try` e especifique o código que deve ser executado quando o controle sair da instrução `try`:

C#

```
public async Task ProcessRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    catch (Exception e) when (e is not OperationCanceledException)
    {
        LogError(e, $"Failed to process request for item ID {itemId}.");
        throw;
    }
}
```

```
    }
    finally
    {
        Busy = false;
    }

}
```

Quando uma exceção é tratada por um bloco `catch`, o bloco `finally` é executado após a execução desse bloco `catch` (mesmo que outra exceção ocorra durante a execução do bloco `catch`). Para obter informações sobre blocos `catch` e `finally`, consulte as seções [A instrução try-catch](#) e [A instrução try-finally](#), respectivamente.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Instrução throw](#)
- [Instrução try](#)
- [Exceções](#)

## Confira também

- [Referência de C#](#)
- [Exceções e manipulação de exceções](#)
- [Tratando e gerando exceções no .NET](#)
- [gerar preferências \(regra de estilo IDE0016\)](#)
- [AppDomain.FirstChanceException](#)
- [AppDomain.UnhandledException](#)
- [TaskScheduler.UnobservedTaskException](#)

# Instruções checked e unchecked (referência de C#)

Artigo • 05/06/2023

As instruções `checked` e `unchecked` especificam o contexto de verificação de estouro para operações e conversões aritméticas do tipo integral. Quando ocorre um estouro aritmético inteiro, o contexto de verificação de estouro define o que acontece. Em um contexto verificado, um `System.OverflowException` é lançado; se o estouro acontece em uma expressão de constante, ocorre um erro em tempo de compilação. Em um contexto não verificado, o resultado da operação é truncado pelo descarte dos bits de ordem superior que não se ajustam ao tipo de destino. Por exemplo, no caso de adição, ele encapsula o valor máximo para o valor mínimo. O seguinte exemplo mostra a mesma operação em um contexto verificado e não verificado:

```
C#  
  
uint a = uint.MaxValue;  
  
unchecked  
{  
    Console.WriteLine(a + 3); // output: 2  
}  
  
try  
{  
    checked  
    {  
        Console.WriteLine(a + 3);  
    }  
}  
catch (OverflowException e)  
{  
    Console.WriteLine(e.Message); // output: Arithmetic operation resulted  
    in an overflow.  
}
```

## ⓘ Observação

O comportamento de operadores e conversões *definidos pelo usuário* no caso do estouro pode ser diferente do descrito no parágrafo anterior. Em particular, os **operadores verificados definidos pelo usuário** podem não gerar uma exceção em um contexto verificado.

Para obter mais informações, consulte as seções [Estouro aritmético e divisão por zero](#) e [Operadores verificados definidos pelo usuário](#) do artigo [Operadores aritméticos](#).

Para especificar o contexto de verificação de estouro para uma expressão, você também pode usar os operadores `checked` e `unchecked`, como mostra o exemplo a seguir:

```
C#  
  
double a = double.MaxValue;  
  
int b = unchecked((int)a);  
Console.WriteLine(b); // output: -2147483648  
  
try  
{  
    b = checked((int)a);  
}  
catch (OverflowException e)  
{  
    Console.WriteLine(e.Message); // output: Arithmetic operation resulted  
    in an overflow.  
}
```

Os operadores e as instruções `checked` e `unchecked` afetam apenas o contexto de verificação de estouro para as operações que estão *textualmente* dentro dos parênteses do operador ou do bloco de instrução, como mostra o exemplo a seguir:

```
C#  
  
int Multiply(int a, int b) => a * b;  
  
int factor = 2;  
  
try  
{  
    checked  
    {  
        Console.WriteLine(Multiply(factor, int.MaxValue)); // output: -2  
    }  
}  
catch (OverflowException e)  
{  
    Console.WriteLine(e.Message);  
}  
  
try  
{  
    checked  
    {  
        Console.WriteLine(Multiply(factor, factor * int.MaxValue));  
    }  
}
```

```
    }
    catch (OverflowException e)
    {
        Console.WriteLine(e.Message); // output: Arithmetic operation resulted
        in an overflow.
    }
```

No exemplo anterior, a primeira invocação da função local `Multiply` mostra que a instrução `checked` não afeta o contexto de verificação de estouro dentro da função `Multiply`, pois nenhuma exceção é gerada. Na segunda invocação da função `Multiply`, a expressão que calcula o segundo argumento da função é avaliada em um contexto verificado e resulta em uma exceção, pois está textualmente dentro do bloco da instrução `checked`.

## Operações afetadas pelo contexto de verificação de estouro

O contexto de verificação de estouro afeta as seguintes operações:

- Os seguintes [operadores aritméticos](#) internos: operadores `++`, `--`, `-` unários e `+`, `-`, `*` e `/` binários, quando seus operandos são do tipo integral (ou seja, tipo [numérico integral](#) ou `char`) ou um tipo [enumerado](#).
- [Conversões numéricas explícitas](#) entre tipos integrais ou de `float` ou `double` para um tipo integral.

### ⓘ Observação

Quando você converte um valor `decimal` em um tipo integral e o resultado está fora do intervalo do tipo de destino, um `OverflowException` é sempre gerado, independentemente do contexto de verificação de estouro.

- A partir do C# 11, operadores e conversões verificados definidos pelo usuário. Saiba mais na seção [Operadores verificados definidos pelo usuário](#) do artigo [Operadores aritméticos](#).

## Contexto de verificação de estouro padrão

Se você não especificar o contexto de verificação de estouro, o valor da opção do compilador `CheckForOverflowUnderflow` definirá o contexto padrão para expressões

não constantes. Por padrão, o valor dessa opção é removido e as conversões e operações aritméticas de tipo integral são executadas em um contexto **não verificado**.

Expressões constantes são avaliadas por padrão em um contexto verificado e ocorre um erro em tempo de compilação em caso de estouro. Você pode especificar explicitamente um contexto não verificado para uma expressão constante com o operador ou a instrução `unchecked`.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [As instruções marcadas e desmarcadas](#)
- [Os operadores verificados e não verificados](#)
- [Operadores verificados e não verificados definidos pelo usuário – C# 11](#)

## Confira também

- [Referência de C#](#)
- [Opção do compilador `CheckForOverflowUnderflow`](#)

# Instruções checked e unchecked (referência de C#)

Artigo • 05/06/2023

As instruções `checked` e `unchecked` especificam o contexto de verificação de estouro para operações e conversões aritméticas do tipo integral. Quando ocorre um estouro aritmético inteiro, o contexto de verificação de estouro define o que acontece. Em um contexto verificado, um `System.OverflowException` é lançado; se o estouro acontece em uma expressão de constante, ocorre um erro em tempo de compilação. Em um contexto não verificado, o resultado da operação é truncado pelo descarte dos bits de ordem superior que não se ajustam ao tipo de destino. Por exemplo, no caso de adição, ele encapsula o valor máximo para o valor mínimo. O seguinte exemplo mostra a mesma operação em um contexto verificado e não verificado:

```
C#  
  
uint a = uint.MaxValue;  
  
unchecked  
{  
    Console.WriteLine(a + 3); // output: 2  
}  
  
try  
{  
    checked  
    {  
        Console.WriteLine(a + 3);  
    }  
}  
catch (OverflowException e)  
{  
    Console.WriteLine(e.Message); // output: Arithmetic operation resulted  
    in an overflow.  
}
```

## ⓘ Observação

O comportamento de operadores e conversões *definidos pelo usuário* no caso do estouro pode ser diferente do descrito no parágrafo anterior. Em particular, os **operadores verificados definidos pelo usuário** podem não gerar uma exceção em um contexto verificado.

Para obter mais informações, consulte as seções [Estouro aritmético e divisão por zero](#) e [Operadores verificados definidos pelo usuário](#) do artigo [Operadores aritméticos](#).

Para especificar o contexto de verificação de estouro para uma expressão, você também pode usar os operadores `checked` e `unchecked`, como mostra o exemplo a seguir:

```
C#  
  
double a = double.MaxValue;  
  
int b = unchecked((int)a);  
Console.WriteLine(b); // output: -2147483648  
  
try  
{  
    b = checked((int)a);  
}  
catch (OverflowException e)  
{  
    Console.WriteLine(e.Message); // output: Arithmetic operation resulted  
    in an overflow.  
}
```

Os operadores e as instruções `checked` e `unchecked` afetam apenas o contexto de verificação de estouro para as operações que estão *textualmente* dentro dos parênteses do operador ou do bloco de instrução, como mostra o exemplo a seguir:

```
C#  
  
int Multiply(int a, int b) => a * b;  
  
int factor = 2;  
  
try  
{  
    checked  
    {  
        Console.WriteLine(Multiply(factor, int.MaxValue)); // output: -2  
    }  
}  
catch (OverflowException e)  
{  
    Console.WriteLine(e.Message);  
}  
  
try  
{  
    checked  
    {  
        Console.WriteLine(Multiply(factor, factor * int.MaxValue));  
    }  
}
```

```
    }
    catch (OverflowException e)
    {
        Console.WriteLine(e.Message); // output: Arithmetic operation resulted
        in an overflow.
    }
```

No exemplo anterior, a primeira invocação da função local `Multiply` mostra que a instrução `checked` não afeta o contexto de verificação de estouro dentro da função `Multiply`, pois nenhuma exceção é gerada. Na segunda invocação da função `Multiply`, a expressão que calcula o segundo argumento da função é avaliada em um contexto verificado e resulta em uma exceção, pois está textualmente dentro do bloco da instrução `checked`.

## Operações afetadas pelo contexto de verificação de estouro

O contexto de verificação de estouro afeta as seguintes operações:

- Os seguintes [operadores aritméticos](#) internos: operadores `++`, `--`, `-` unários e `+`, `-`, `*` e `/` binários, quando seus operandos são do tipo integral (ou seja, tipo [numérico integral](#) ou `char`) ou um tipo [enumerado](#).
- [Conversões numéricas explícitas](#) entre tipos integrais ou de `float` ou `double` para um tipo integral.

### ⓘ Observação

Quando você converte um valor `decimal` em um tipo integral e o resultado está fora do intervalo do tipo de destino, um `OverflowException` é sempre gerado, independentemente do contexto de verificação de estouro.

- A partir do C# 11, operadores e conversões verificados definidos pelo usuário. Saiba mais na seção [Operadores verificados definidos pelo usuário](#) do artigo [Operadores aritméticos](#).

## Contexto de verificação de estouro padrão

Se você não especificar o contexto de verificação de estouro, o valor da opção do compilador `CheckForOverflowUnderflow` definirá o contexto padrão para expressões

não constantes. Por padrão, o valor dessa opção é removido e as conversões e operações aritméticas de tipo integral são executadas em um contexto **não verificado**.

Expressões constantes são avaliadas por padrão em um contexto verificado e ocorre um erro em tempo de compilação em caso de estouro. Você pode especificar explicitamente um contexto não verificado para uma expressão constante com o operador ou a instrução `unchecked`.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [As instruções marcadas e desmarcadas](#)
- [Os operadores verificados e não verificados](#)
- [Operadores verificados e não verificados definidos pelo usuário – C# 11](#)

## Confira também

- [Referência de C#](#)
- [Opção do compilador `CheckForOverflowUnderflow`](#)

# Instruções checked e unchecked (referência de C#)

Artigo • 05/06/2023

As instruções `checked` e `unchecked` especificam o contexto de verificação de estouro para operações e conversões aritméticas do tipo integral. Quando ocorre um estouro aritmético inteiro, o contexto de verificação de estouro define o que acontece. Em um contexto verificado, um `System.OverflowException` é lançado; se o estouro acontece em uma expressão de constante, ocorre um erro em tempo de compilação. Em um contexto não verificado, o resultado da operação é truncado pelo descarte dos bits de ordem superior que não se ajustam ao tipo de destino. Por exemplo, no caso de adição, ele encapsula o valor máximo para o valor mínimo. O seguinte exemplo mostra a mesma operação em um contexto verificado e não verificado:

```
C#  
  
uint a = uint.MaxValue;  
  
unchecked  
{  
    Console.WriteLine(a + 3); // output: 2  
}  
  
try  
{  
    checked  
    {  
        Console.WriteLine(a + 3);  
    }  
}  
catch (OverflowException e)  
{  
    Console.WriteLine(e.Message); // output: Arithmetic operation resulted  
    in an overflow.  
}
```

## ⓘ Observação

O comportamento de operadores e conversões *definidos pelo usuário* no caso do estouro pode ser diferente do descrito no parágrafo anterior. Em particular, os **operadores verificados definidos pelo usuário** podem não gerar uma exceção em um contexto verificado.

Para obter mais informações, consulte as seções [Estouro aritmético e divisão por zero](#) e [Operadores verificados definidos pelo usuário](#) do artigo [Operadores aritméticos](#).

Para especificar o contexto de verificação de estouro para uma expressão, você também pode usar os operadores `checked` e `unchecked`, como mostra o exemplo a seguir:

```
C#  
  
double a = double.MaxValue;  
  
int b = unchecked((int)a);  
Console.WriteLine(b); // output: -2147483648  
  
try  
{  
    b = checked((int)a);  
}  
catch (OverflowException e)  
{  
    Console.WriteLine(e.Message); // output: Arithmetic operation resulted  
    in an overflow.  
}
```

Os operadores e as instruções `checked` e `unchecked` afetam apenas o contexto de verificação de estouro para as operações que estão *textualmente* dentro dos parênteses do operador ou do bloco de instrução, como mostra o exemplo a seguir:

```
C#  
  
int Multiply(int a, int b) => a * b;  
  
int factor = 2;  
  
try  
{  
    checked  
    {  
        Console.WriteLine(Multiply(factor, int.MaxValue)); // output: -2  
    }  
}  
catch (OverflowException e)  
{  
    Console.WriteLine(e.Message);  
}  
  
try  
{  
    checked  
    {  
        Console.WriteLine(Multiply(factor, factor * int.MaxValue));  
    }  
}
```

```
    }
    catch (OverflowException e)
    {
        Console.WriteLine(e.Message); // output: Arithmetic operation resulted
        in an overflow.
    }
```

No exemplo anterior, a primeira invocação da função local `Multiply` mostra que a instrução `checked` não afeta o contexto de verificação de estouro dentro da função `Multiply`, pois nenhuma exceção é gerada. Na segunda invocação da função `Multiply`, a expressão que calcula o segundo argumento da função é avaliada em um contexto verificado e resulta em uma exceção, pois está textualmente dentro do bloco da instrução `checked`.

## Operações afetadas pelo contexto de verificação de estouro

O contexto de verificação de estouro afeta as seguintes operações:

- Os seguintes [operadores aritméticos](#) internos: operadores `++`, `--`, `-` unários e `+`, `-`, `*` e `/` binários, quando seus operandos são do tipo integral (ou seja, tipo [numérico integral](#) ou `char`) ou um tipo [enumerado](#).
- [Conversões numéricas explícitas](#) entre tipos integrais ou de `float` ou `double` para um tipo integral.

### ⓘ Observação

Quando você converte um valor `decimal` em um tipo integral e o resultado está fora do intervalo do tipo de destino, um `OverflowException` é sempre gerado, independentemente do contexto de verificação de estouro.

- A partir do C# 11, operadores e conversões verificados definidos pelo usuário. Saiba mais na seção [Operadores verificados definidos pelo usuário](#) do artigo [Operadores aritméticos](#).

## Contexto de verificação de estouro padrão

Se você não especificar o contexto de verificação de estouro, o valor da opção do compilador `CheckForOverflowUnderflow` definirá o contexto padrão para expressões

não constantes. Por padrão, o valor dessa opção é removido e as conversões e operações aritméticas de tipo integral são executadas em um contexto **não verificado**.

Expressões constantes são avaliadas por padrão em um contexto verificado e ocorre um erro em tempo de compilação em caso de estouro. Você pode especificar explicitamente um contexto não verificado para uma expressão constante com o operador ou a instrução `unchecked`.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [As instruções marcadas e desmarcadas](#)
- [Os operadores verificados e não verificados](#)
- [Operadores verificados e não verificados definidos pelo usuário – C# 11](#)

## Confira também

- [Referência de C#](#)
- [Opção do compilador `CheckForOverflowUnderflow`](#)

# instrução fixed – fixar uma variável para operações de ponteiro

Artigo • 05/06/2023

A instrução `fixed` impede que o [coletor de lixo](#) realoque uma variável móvel e declara um ponteiro para essa variável. O endereço de uma variável fixa, ou fixada, não é alterado durante a execução da instrução. Você pode usar o ponteiro declarado somente dentro da instrução correspondente `fixed`. O ponteiro declarado é somente leitura e não pode ser modificado:

C#

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = bytes)
    {
        Console.WriteLine($"The address of the first array element:
{((long)pointerToFirst:X}.");
        Console.WriteLine($"The value of the first array element:
{*pointerToFirst.}");
    }
}
// Output is similar to:
// The address of the first array element: 2173F80B5C8.
// The value of the first array element: 1.
```

## ① Observação

Você pode usar a instrução `fixed` somente em um contexto **não seguro**. O código que contém blocos não seguros deve ser compilado com a opção do compilador `AllowUnsafeBlocks`.

Você pode inicializar o ponteiro declarado da seguinte maneira:

- Com uma matriz, como mostra o exemplo no início deste artigo. O ponteiro inicializado contém o endereço do primeiro elemento de matriz.
- Com um endereço de uma variável. Use o operador `address-of&`, como mostra o seguinte exemplo:

C#

```

unsafe
{
    int[] numbers = { 10, 20, 30 };
    fixed (int* toFirst = &numbers[0], toLast = &numbers[^1])
    {
        Console.WriteLine(toLast - toFirst); // output: 2
    }
}

```

Os campos de objeto são outro exemplo de variáveis móveis que podem ser fixadas.

Quando o ponteiro inicializado contém o endereço de um campo de objeto ou um elemento de matriz, a instrução `fixed` garante que o coletor de lixo não realoque ou descarte a instância de objeto que o contém durante a execução do corpo da instrução.

- Com a instância do tipo que implementa um método chamado `GetPinnableReference`. Esse método deve retornar uma variável `ref` para um tipo não gerenciado. Os tipos do .NET `System.Span<T>` e `System.ReadOnlySpan<T>` fazem uso desse padrão. Você pode fixar instâncias span, como mostra o exemplo a seguir:

```

C#

unsafe
{
    int[] numbers = { 10, 20, 30, 40, 50 };
    Span<int> interior = numbers.AsSpan()[1..^1];
    fixed (int* p = interior)
    {
        for (int i = 0; i < interior.Length; i++)
        {
            Console.Write(p[i]);
        }
        // output: 203040
    }
}

```

Para obter mais informações, veja a referência de API [Span<T>.GetPinnableReference\(\)](#).

- Com uma cadeia de caracteres, como mostra o exemplo a seguir:

```
C#
```

```
unsafe
{
    var message = "Hello!";
    fixed (char* p = message)
    {
        Console.WriteLine(*p); // output: H
    }
}
```

- Com um [buffer de tamanho fixo](#).

É possível alocar memória na pilha, onde ela não está sujeita à coleta de lixo e, portanto, não precisa ser fixada. Para fazer isso, use uma [expressão stackalloc](#).

Você também pode usar a palavra-chave `fixed` para declarar um [buffer de tamanho fixo](#).

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- A instrução `fixed`
- Variáveis fixas e móveis

## Confira também

- [Referência de C#](#)
- [Código não seguro, tipos de ponteiro e ponteiros de função](#)
- [Operadores relacionados ao ponteiro](#)
- [unsafe](#)

# Parâmetros de método (Referência de C#)

Artigo • 07/04/2023

No C#, argumentos podem ser passados para parâmetros por valor ou por referência. Lembre-se de que os tipos C# podem ser tipos de referência (`class`) ou tipos de valor (`struct`):

- *Aprovação por valor* significa aprovar uma cópia da variável para o método.
- *Aprovação por referência* significa aprovar o acesso à variável para o método.
- Uma variável de um *tipo de referência* contém uma referência aos seus dados.
- Uma variável de um *tipo de valor* contém seu valor diretamente.

Como um struct é um *tipo de valor*, ao [passar um struct por valor](#) a um método, esse método receberá e operará em uma cópia do argumento do struct. O método não tem acesso ao struct original no método de chamada e, portanto, não é possível alterá-lo de forma alguma. O método pode alterar somente a cópia.

Uma instância de classe é um *tipo de referência* e não é um tipo de valor. Quando [um tipo de referência é passado por valor](#) a um método, esse método receberá uma cópia da referência para a instância da classe. Ou seja, o método chamado recebe uma cópia do endereço da instância e o método de chamada retém o endereço original da instância. A instância de classe no método de chamada tem um endereço, o parâmetro do método chamado tem uma cópia do endereço e os dois endereços se referem ao mesmo objeto. Como o parâmetro contém apenas uma cópia do endereço, o método chamado não pode alterar o endereço da instância de classe no método de chamada. No entanto, o método chamado pode usar a cópia do endereço para acessar os membros de classe que o endereço original e a cópia do endereço referenciam. Se o método chamado alterar um membro de classe, a instância da classe original no método de chamada também será alterada.

O resultado do exemplo a seguir ilustra a diferença. O valor do campo `willIChange` da instância da classe foi alterado pela chamada ao método `ClassTaker`, pois o método usa o endereço no parâmetro para localizar o campo especificado da instância da classe. O campo `willIChange` do struct no método de chamada não foi alterado pela chamada ao método `StructTaker`, pois o valor do argumento é uma cópia do próprio struct e não uma cópia de seu endereço. `StructTaker` altera a cópia e a cópia será perdida quando a chamada para `StructTaker` for concluída.

```

class TheClass
{
    public string? willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    public static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Class field = Changed
   Struct field = Not Changed
*/

```

A forma como um argumento é aprovado e se é um tipo de referência ou tipo de valor controla quais modificações feitas no argumento são visíveis pelo chamador.

## Aprovar um tipo de valor por valor

Quando se aprova um tipo de valor por valor:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **não serão** visíveis pelo chamador.

O exemplo a seguir demonstra a passagem de parâmetros de tipo de valor por valor. A variável `n` é passada por valor para o método `SquareIt`. Quaisquer alterações que ocorrem dentro do método não têm efeito sobre o valor original da variável.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(n); // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(int x)
// The parameter x is passed by value.
// Changes to x will not affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 5
*/
```

A variável `n` é um tipo de valor. Ela contém seus dados, o valor `5`. Quando `SquareIt` é invocado, o conteúdo de `n` é copiado para o parâmetro `x`, que é elevado ao quadrado dentro do método. No entanto, em `Main`, o valor de `n` é o mesmo depois de chamar o método `SquareIt` como era antes. A alteração que ocorre dentro do método afeta apenas a variável local `x`.

## Aprovar um tipo de valor por referência

Quando se aprova um tipo de valor por referência:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir é o mesmo que o exemplo anterior, exceto que o argumento é passado como um parâmetro `ref`. O valor do argumento subjacente, `n`, é alterado quando `x` é alterado no método.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(ref n); // Passing the variable by reference.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(ref int x)
// The parameter x is passed by reference.
// Changes to x will affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 25
*/
```

Neste exemplo, não é o valor de `n` que é passado; em vez disso, é passada uma referência a `n`. O parâmetro `x` não é um `int`; é uma referência a um `int`, nesse caso, uma referência a `n`. Portanto, quando `x` é elevado ao quadrado dentro do método, o que é realmente elevado ao quadrado é aquilo a que `x` se refere, `n`.

## Aprovar um tipo de referência por valor

Quando se aprova um tipo de *referênciapor valor*:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **não serão** visíveis pelo chamador.

- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir demonstra a passagem de um parâmetro de tipo de referência, `arr`, por valor, para um método, `Change`. Como o parâmetro é uma referência a `arr`, é possível alterar os valores dos elementos da matriz. No entanto, a tentativa de reatribuir o parâmetro para um local diferente de memória só funciona dentro do método e não afeta a variável original, `arr`.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(int[] pArray)
{
    pArray[0] = 888; // This change affects the original element.
    pArray = new int[5] { -3, -1, -2, -3, -4 }; // This change is local.
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: 888
*/
```

No exemplo anterior, a matriz, `arr`, que é um tipo de referência, é passada para o método sem o parâmetro `ref`. Nesse caso, uma cópia da referência, que aponta para `arr`, é passada para o método. A saída mostra que é possível para o método alterar o conteúdo de um elemento de matriz, nesse caso de `1` para `888`. No entanto, alocar uma nova parte da memória usando o operador `new` dentro do método `Change` faz a variável `pArray` referenciar uma nova matriz. Portanto, quaisquer alterações realizadas depois disso não afetarão a matriz original, `arr`, criada dentro de `Main`. Na verdade, duas matrizes são criadas neste exemplo, uma dentro de `Main` e outra dentro do método `Change`.

## Aprovar um tipo de referência por referência

Quando se aprova um tipo de *referênciapor referência*:

- Se o método atribuir o parâmetro para se referir a um objeto diferente, essas alterações **ficarão** visíveis pelo chamador.
- Se o método modificar o estado do objeto referenciado pelo parâmetro, essas alterações **ficarão** visíveis pelo chamador.

O exemplo a seguir é o mesmo que o exemplo anterior, exceto que a palavra-chave `ref` é adicionada ao cabeçalho e à chamada do método. Quaisquer alterações que ocorrem no método afetam a variável original no programa de chamada.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(ref int[] pArray)
{
    // Both of the following changes will affect the original variables:
    pArray[0] = 888;
    pArray = new int[5] { -3, -1, -2, -3, -4 };
    System.Console.WriteLine("Inside the method, the first element is: {0}",
    pArray[0]);
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: -3
*/
```

Todas as alterações que ocorrem dentro do método afetam a matriz original em `Main`. Na verdade, a matriz original é realocada usando o operador `new`. Portanto, depois de chamar o método `Change`, qualquer referência a `arr` aponta para a matriz de cinco elementos, criada no método `Change`.

## Escopo de referências e valores

Os métodos podem armazenar os valores de parâmetros em campos. Quando os parâmetros são aprovados por valor, o método é sempre seguro. Os valores são copiados e os tipos de referência podem ser acessados quando armazenados em um campo. Para aprovar parâmetros por referência com segurança, é necessário que o compilador defina quando é seguro atribuir uma referência a uma nova variável. Para

cada expressão, o compilador define um *escopo* que vincula o acesso a uma expressão ou variável. O compilador usa dois escopos: *safe\_to\_escape* e *ref\_safe\_to\_escape*.

- *safe\_to\_escape* define o escopo em que qualquer expressão pode ser acessada com segurança.
- *ref\_safe\_to\_escape* define o escopo em que uma *referência* a qualquer expressão pode ser acessada ou modificada com segurança.

Informalmente, você pode considerar esses escopos como o mecanismo para garantir que seu código nunca acesse ou modifique uma referência que não seja mais válida. Uma referência é válida desde que se refira a um objeto ou struct válido. O escopo *safe\_to\_escape* define quando uma variável pode ser atribuída ou reatribuída. O escopo *ref\_safe\_to\_escape* define quando uma variável *pode ser atribuída* ou reatribuída por *ref*. A atribuição concede a uma variável um novo valor. A *atribuição de referência* indica para a variável que ela deve *se referir a* um local de armazenamento diferente.

## Modificadores

Os parâmetros declarados para um método sem *in*, *ref* nem *out* são passados para o método chamado pelo valor. Os modificadores *ref*, *in* e *out* diferem nas regras de atribuição:

- O argumento para um parâmetro *ref* deve ser atribuído de maneira definitiva. O método chamado pode reatribuir esse parâmetro.
- O argumento para um parâmetro *in* deve ser atribuído de maneira definitiva. O método chamado não pode reatribuir esse parâmetro.
- O argumento para um parâmetro *out* não precisa ser atribuído de maneira definitiva. O método chamado deve atribuir o parâmetro.

Esta seção descreve as palavras-chave que podem ser usadas ao declarar parâmetros de método:

- *params* especifica que esse parâmetro pode receber um número variável de argumentos.
- *in* especifica que esse parâmetro é passado por referência, mas é lido apenas pelo método chamado.
- *ref* especifica que esse parâmetro é passado por referência e pode ser lido ou gravado pelo método chamado.
- *out* especifica que esse parâmetro é passado por referência e é gravado pelo método chamado.

## Confira também

- [Referência de C#](#)
- [Palavras-chave do C#](#)
- Listas de argumentos na [Especificação de Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# params (Referência de C#)

Artigo • 07/04/2023

Usando a palavra-chave `params`, você pode especificar um [parâmetro do método](#) que aceita um número variável de argumentos. O tipo de parâmetro deve ser uma matriz unidimensional.

Nenhum parâmetro adicional é permitido após a palavra-chave `params` em uma declaração de método e apenas uma palavra-chave `params` é permitida em uma declaração de método.

Se o tipo declarado do parâmetro `params` não for uma matriz unidimensional, ocorrerá o erro do compilador [CS0225](#).

Ao chamar um método com um parâmetro `params`, você pode passar:

- Uma lista separada por vírgulas de argumentos do tipo dos elementos da matriz.
- Uma matriz de argumentos do tipo especificado.
- Sem argumentos. Se você não enviar nenhum argumento, o comprimento da lista `params` será zero.

## Exemplo

O exemplo a seguir demonstra várias maneiras em que os argumentos podem ser enviados para um parâmetro `params`.

C#

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
    }
}
```

```

        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}

/*
Output:
  1 2 3 4
  1 a test

  5 6 7 8 9
  2 b test again
  System.Int32[]
*/

```

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)

- Guia de Programação em C#
- Palavras-chave do C#
- Parâmetros de método

# Modificador de parâmetro in (referência do C#)

Artigo • 07/04/2023

A palavra-chave `in` faz com que os argumentos sejam passados por referência, mas garante que o argumento não seja modificado. Ela torna o parâmetro formal um alias para o argumento, que deve ser uma variável. Em outras palavras, qualquer operação no parâmetro é feita no argumento. É como as palavras-chave `ref` ou `out`, exceto que os argumentos `in` não podem ser modificados pelo método chamado. Enquanto os argumentos `ref` podem ser modificados, os argumentos `out` devem ser modificados pelo método chamado, e essas modificações podem ser observadas no contexto da chamada.

```
C#  
  
int readonlyArgument = 44;  
InArgExample(readonlyArgument);  
Console.WriteLine(readonlyArgument);      // value is still 44  
  
void InArgExample(in int number)  
{  
    // Uncomment the following line to see error CS8331  
    //number = 19;  
}
```

O exemplo anterior demonstra que o modificador `in` é geralmente desnecessário no site de chamada. Ele apenas é necessário na declaração do método.

## ⓘ Observação

A palavra-chave `in` também pode ser usada com um parâmetro de tipo genérico para especificar que o parâmetro de tipo é contravariante, como parte de uma instrução `foreach` ou como parte de uma cláusula `join` em uma consulta LINQ. Para obter mais informações sobre o uso da palavra-chave `in` nesses contextos, confira [in](#) que fornece links para todos esses usos.

As variáveis passadas como argumentos `in` precisam ser inicializadas antes de serem passadas em uma chamada de método. No entanto, o método chamado não pode atribuir um valor nem modificar o argumento.

Embora os modificadores de parâmetro `in`, `out` e `ref` sejam considerados parte de uma assinatura, os membros declarados em um único tipo não podem diferir apenas na assinatura por `in`, `ref` e `out`. Portanto, os métodos não poderão ser sobre carregados se a única diferença for que um método usa um argumento `ref` ou `out` e o outro usa um argumento `in`. Por exemplo, o código a seguir, não será compilado:

C#

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on in, ref and out".
    public void SampleMethod(in int i) { }
    public void SampleMethod(ref int i) { }
}
```

A sobre carga com base na presença de `in` é permitida:

C#

```
class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}
```

## Regras de resolução de sobre carga

Você pode entender as regras de resolução de sobre carga para métodos por valor versus argumentos `in`, compreendendo a motivação dos argumentos `in`. A definição de métodos usando parâmetros `in` é uma possível otimização de desempenho. Alguns argumentos de tipo `struct` podem ser grandes em tamanho e, quando os métodos são chamados em loops rígidos ou caminhos de código críticos, o custo da cópia dessas estruturas é crítico. Os métodos declaram parâmetros `in` para especificar que argumentos podem ser passados por referência com segurança porque o método chamado não modifica o estado desse argumento. A passagem desses argumentos por referência evita a cópia (possivelmente) dispendiosa.

A especificação de `in` em argumentos no site de chamada é normalmente opcional.

Não há diferença semântica entre passar argumentos por valor e transmiti-los por meio de referência usando o modificador `in`. O modificador `in` no site de chamada é opcional, pois não é necessário indicar que o valor do argumento pode ser alterado. Você adiciona explicitamente o modificador `in` no site de chamada para garantir que o

argumento seja passado por referência, e não por valor. O uso explícito de `in` tem dois efeitos:

Primeiro: especificar `in` no site de chamada força o compilador a selecionar um método definido com um parâmetro `in` correspondente. Caso contrário, quando dois métodos diferem apenas na presença de `in`, a sobrecarga por valor é uma correspondência melhor.

Segundo: especificar `in` declara sua intenção de passar um argumento por referência. O argumento usado com `in` deve representar um local ao qual se possa fazer referência diretamente. As mesmas regras gerais para argumentos `out` e `ref` se aplicam: você não pode usar constantes, propriedades comuns ou outras expressões que produzem valores. Caso contrário, a omissão de `in` no site de chamada informa ao compilador que você permitirá a criação de uma variável temporária para passar por referência de somente leitura para o método. O compilador cria uma variável temporária para superar várias restrições com argumentos `in`:

- Uma variável temporária permite constantes de tempo de compilação como parâmetros `in`.
- Uma variável temporária permite propriedades ou outras expressões para parâmetros `in`.
- Uma variável temporária permite argumentos em que há uma conversão implícita do tipo de argumento para o tipo de parâmetro.

Em todas as instâncias anteriores, o compilador cria uma variável temporária que armazena o valor da constante, da propriedade ou de outra expressão.

O código a seguir ilustra essas regras:

C#

```
static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`
```

Agora, vamos supor que outro método usando argumentos por valor estivesse disponível. Os resultados são alterados conforme mostrado no código a seguir:

```
C#  
  
static void Method(int argument)  
{  
    // implementation removed  
}  
  
static void Method(in int argument)  
{  
    // implementation removed  
}  
  
Method(5); // Calls overload passed by value  
Method(5L); // CS1503: no implicit conversion from long to int  
short s = 0;  
Method(s); // Calls overload passed by value.  
Method(in s); // CS1503: cannot convert from in short to in int  
int i = 42;  
Method(i); // Calls overload passed by value  
Method(in i); // passed by readonly reference, explicitly using `in`
```

A única chamada de método em que o argumento é passado por referência é a chamada final.

#### ① Observação

O código anterior usa `int` como o tipo de argumento por questão de simplicidade. Como `int` não é maior que uma referência na maioria dos computadores modernos, não há nenhum benefício em passar um único `int` como uma referência `readonly`.

## Limitações em parâmetros `in`

Não é possível usar as palavras-chave `in`, `ref` e `out` para os seguintes tipos de métodos:

- Métodos assíncronos, que você define usando o modificador `async`.
- Métodos de iterador, que incluem uma instrução `yield return` ou `yield break`.
- O primeiro argumento de um método de extensão não pode ter o modificador `in`, a menos que esse argumento seja um `struct`.

- O primeiro argumento de um método de extensão em que esse argumento é um tipo genérico (mesmo quando esse tipo é restrito a ser um struct).

Saiba mais sobre o modificador `in` e as diferenças entre ele e `ref` e `out` no artigo sobre [alocações](#).

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

# ref (referência de C#)

Artigo • 28/06/2023

Você usa a palavra-chave `ref` nos seguintes contextos:

- Em uma assinatura de método e em uma chamada de método, para passar um argumento a um método por referência. Para obter mais informações, veja [Passar um argumento por referência](#).
- Em uma assinatura de método para retornar um valor para o chamador por referência. Para obter mais informações, consulte [Reference return values](#) (Valores retornados de referência).
- Em uma declaração de uma variável local, para declarar uma variável de referência. Para obter mais informações, confira a seção [Variáveis de referência](#) do artigo [Instruções de declaração](#).
- Como parte de uma [expressão de referência condicional](#) ou um [operador de atribuição de referência](#).
- Em uma declaração `struct` para declarar um `ref struct`. Para obter mais informações, confira o artigo [reftipos de estrutura](#).
- Em uma definição `ref struct`, para declarar um campo `ref`. Para saber mais, confira a seção [refcampos](#) do artigo [reftipos de estrutura](#).

## Passando um argumento por referência

Quando usado na lista de parâmetros do método, a palavra-chave `ref` indica que um argumento é passado por referência, não por valor. A palavra-chave `ref` torna o parâmetro formal um alias para o argumento, que deve ser uma variável. Em outras palavras, qualquer operação no parâmetro é feita no argumento.

Por exemplo, suponha que o chamador passe uma expressão de variável local ou uma expressão de acesso de elemento de matriz. Em seguida, o método chamado pode substituir o objeto ao qual o parâmetro `ref` se refere. Nesse caso, a variável local do chamador ou do elemento da matriz referem-se ao novo objeto quando o método retorna.

### ⓘ Observação

Não confunda o conceito de passar por referência com o conceito de tipos de referência. Os dois conceitos não são iguais. Um parâmetro de método pode ser modificado por `ref`, independentemente de ele ser um tipo de valor ou um tipo de

referência. Não há nenhuma conversão boxing de um tipo de valor quando ele é passado por referência.

Para usar um parâmetro `ref`, a definição do método e o método de chamada devem usar explicitamente a palavra-chave `ref`, como mostrado no exemplo a seguir. (Exceto que o método de chamada pode omitir `ref` ao fazer uma chamada COM.)

C#

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

Um argumento passado para um parâmetro `ref` ou `in` precisa ser inicializado antes de ser passado. Esse requisito é diferente dos parâmetros `out`, cujos argumentos não precisam ser inicializados explicitamente antes de serem passados.

Os membros de uma classe não podem ter assinaturas que se diferem somente por `ref`, `in` ou `out`. Ocorrerá um erro de compilador se a única diferença entre os dois membros de um tipo for que um deles tem um parâmetro `ref` e o outro tem um parâmetro `out` ou `in`. O código a seguir, por exemplo, não é compilado.

C#

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

No entanto, os métodos podem ser sobre carregados quando um método tem um parâmetro `ref`, `in` ou `out` e o outro tem um parâmetro que é passado por valor, conforme mostrado no exemplo a seguir.

C#

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

Em outras situações que exigem correspondência de assinatura, como ocultar ou substituir, `in`, `ref` e `out` fazem parte da assinatura e não são correspondentes.

As propriedades não são variáveis. Elas são métodos e não podem ser passadas para parâmetros `ref`.

Não é possível usar as palavras-chave `ref`, `in` e `out` para os seguintes tipos de métodos:

- Métodos assíncronos, que você define usando o modificador `async`.
- Métodos de iterador, que incluem uma instrução `yield return` ou `yield break`.

Os [métodos de extensão](#) também têm restrições sobre o uso dessas palavras-chave:

- A palavra-chave `out` não pode ser usada no primeiro argumento de um método de extensão.
- A palavra-chave `ref` não pode ser usada no primeiro argumento de um método de extensão quando o argumento não é um struct ou um tipo genérico não restrito a ser um struct.
- A palavra-chave `in` não pode ser usada, a menos que o primeiro argumento seja um struct. A palavra-chave `in` não pode ser usada em nenhum tipo genérico, mesmo quando restrito a ser um struct.

## Passando um argumento por referência: um exemplo

Os exemplos anteriores passam tipos de valor por referência. Você também pode usar a palavra-chave `ref` para passar tipos de referência por referência. Passar um tipo de referência por referência permite que o método chamado substitua o objeto ao qual se refere o parâmetro de referência no chamador. O local de armazenamento do objeto é passado para o método como o valor do parâmetro de referência. Se você alterar o valor no local de armazenamento do parâmetro (para apontar para um novo objeto), irá alterar também o local de armazenamento ao qual se refere o chamador. O exemplo a seguir passa uma instância de um tipo de referência como um parâmetro `ref`.

C#

```
class Product
{
    public Product(string name, int newID)
    {
        ItemName = name;
        ItemID = newID;
    }

    public string ItemName { get; set; }
    public int ItemID { get; set; }
}

private static void ChangeByReference(ref Product itemRef)
{
    // Change the address that is stored in the itemRef parameter.
    itemRef = new Product("Stapler", 99999);

    // You can change the value of one of the properties of
    // itemRef. The change happens to item in Main as well.
    itemRef.ItemID = 12345;
}

private static void ModifyProductsByReference()
{
    // Declare an instance of Product and display its initial values.
    Product item = new Product("Fasteners", 54321);
    System.Console.WriteLine("Original values in Main. Name: {0}, ID:
{1}\n",
    item.ItemName, item.ItemID);

    // Pass the product instance to ChangeByReference.
    ChangeByReference(ref item);
    System.Console.WriteLine("Back in Main. Name: {0}, ID: {1}\n",
    item.ItemName, item.ItemID);
}

// This method displays the following output:
// Original values in Main. Name: Fasteners, ID: 54321
// Back in Main. Name: Stapler, ID: 12345
```

Para obter mais informações sobre como passar tipos de referência por valor e por referência, consulte [Passando parâmetros de tipo de referência](#).

## Valores retornados por referência

Valores retornados por referência (ou ref returns) são valores que um método retorna por referência para o chamador. Ou seja, o chamador pode modificar o valor retornado

por um método e essa alteração será refletida no estado do objeto no método chamado.

Um valor retornado por referência é definido usando a palavra-chave `ref`:

- Na assinatura do método. Por exemplo, a assinatura de método a seguir indica que o método `GetCurrentPrice` retorna um valor `Decimal` por referência.

```
C#
```

```
public ref decimal GetCurrentPrice()
```

- Entre o token `return` e a variável retornada em uma instrução `return` no método.

Por exemplo:

```
C#
```

```
return ref DecimalArray[0];
```

Para que o chamador modifique o estado do objeto, o valor retornado de referência deve ser armazenado em uma variável que é definida explicitamente como um [variável de referência](#).

Aqui está um exemplo de retorno de `ref` mais completo, mostrando a assinatura do método e o corpo do método.

```
C#
```

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

O método chamado também poderá declarar o valor retornado como `ref readonly` para retornar o valor por referência e, em seguida, impor que o código de chamada não possa modificar o valor retornado. O método de chamada pode evitar a cópia retornada com um valor ao armazenar o valor em uma variável de referência `ref readonly` local.

Para obter um exemplo, consulte [Um exemplo de `ref returns` e `ref locals`](#).

# Um exemplo de ref returns e ref locals

O exemplo a seguir define uma classe `Book` que tem dois campos `String`, `Title` e `Author`. Ele também define uma classe `BookCollection` que inclui uma matriz privada de objetos `Book`. Objetos de catálogo individuais são retornados por referência chamando o respectivo método `GetBookByTitle`.

C#

```
public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The",
Author = "Jack London" },
                           new Book { Title = "Tale of Two Cities, A", Author =
"Charles Dickens" }
                         };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
    }
}
```

Quando o chamador armazena o valor retornado pelo método `GetBookByTitle` como um ref local, as alterações que o chamador faz ao valor retornado são refletidas no objeto `BookCollection`, conforme mostra o exemplo a seguir.

C#

```
var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//      Call of the Wild, The, by Jack London
//      Tale of Two Cities, A, by Charles Dickens
//
//      Republic, The, by Plato
//      Tale of Two Cities, A, by Charles Dickens
```

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Palavras-chave de C#](#)
- [Parâmetros de método](#)

# Modificador de parâmetro `out` (Referência de C#)

Artigo • 07/04/2023

A palavra-chave `out` faz com que os argumentos sejam passados por referência. Ela torna o parâmetro formal um alias para o argumento, que deve ser uma variável. Em outras palavras, qualquer operação no parâmetro é feita no argumento. É como a palavra-chave `ref`, exceto pelo fato de que `ref` requer que a variável seja inicializada antes de ser passada. Também é como a palavra-chave `in`, exceto que `in` não permite que o método chamado modifique o valor do argumento. Para usar um parâmetro `out`, a definição do método e o método de chamada devem usar explicitamente a palavra-chave `out`. Por exemplo:

C#

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);      // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

## ⓘ Observação

A palavra-chave `out` também pode ser usada com um parâmetro de tipo genérico para especificar que o parâmetro de tipo é covariante. Para obter mais informações sobre o uso da palavra-chave `out` nesse contexto, consulte [out \(modificador genérico\)](#).

Variáveis passadas como argumentos `out` não precisam ser inicializadas antes de serem passadas em uma chamada de método. No entanto, o método chamado é necessário para atribuir um valor antes que o método seja retornado.

As palavras-chave `in`, `ref` e `out` não são consideradas parte da assinatura do método para fins de resolução de sobrecarga. Portanto, os métodos não poderão ser sobrecarregados se a única diferença for que um método usa um argumento `ref` ou `in` e o outro usa um argumento `out`. Por exemplo, o código a seguir, não será compilado:

C#

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

A sobrecarga será válida, no entanto, se um método usar um argumento `ref`, `in` ou `out` e o outro não tiver nenhum desses modificadores, desta forma:

C#

```
class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) => i = 5;
}
```

O compilador escolherá a melhor sobrecarga correspondendo os modificadores de parâmetro no site de chamada aos modificadores de parâmetro usados na chamada do método.

Propriedades não são variáveis e portanto não podem ser passadas como parâmetros `out`.

Não é possível usar as palavras-chave `in`, `ref` e `out` para os seguintes tipos de métodos:

- Métodos assíncronos, que você define usando o modificador `async`.
- Métodos de iterador, que incluem uma instrução `yield return` ou `yield break`.

Além disso, os [métodos de extensão](#) têm as seguintes restrições:

- A palavra-chave `out` não pode ser usada no primeiro argumento de um método de extensão.
- A palavra-chave `ref` não pode ser usada no primeiro argumento de um método de extensão quando o argumento não é um struct ou um tipo genérico não restrito a ser um struct.
- A palavra-chave `in` não pode ser usada, a menos que o primeiro argumento seja um struct. A palavra-chave `in` não pode ser usada em nenhum tipo genérico, mesmo quando restrito a ser um struct.

# Declarando parâmetros `out`

Declarar um método com argumentos `out` é uma solução clássica para retornar vários valores. Considere [tuplas de valor](#) para cenários semelhantes. O exemplo a seguir usa `out` para retornar três variáveis com uma única chamada de método. Observe que o terceiro argumento é atribuído a `null`. Isso permite que os métodos retornem valores opcionalmente.

C#

```
void Method(out int answer, out string message, out string? stillNull)
{
    answer = 44;
    message = "I've been returned";
    stillNull = null;
}

int argNumber;
string argMessage;
string? argDefault;
Method(out argNumber, out argMessage, out argDefault);
Console.WriteLine(argNumber);
Console.WriteLine(argMessage);
Console.WriteLine(argDefault == null);

// The example displays the following output:
//      44
//      I've been returned
//      True
```

# Chamando um método com um argumento `out`

Você pode declarar uma variável em uma instrução separada antes de passá-la como um `out` argumento. O exemplo a seguir declara uma variável chamada `number` antes de passá-la para o método [Int32.TryParse](#), que tenta converter uma cadeia de caracteres em um número.

C#

```
string numberAsString = "1640";

int number;
if (Int32.TryParse(numberAsString, out number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
```

```
// The example displays the following output:  
//     Converted '1640' to 1640
```

Você também pode declarar a `out` variável na lista de argumentos da chamada de método, em vez de em uma declaração de variável separada. Isso produz um código mais compacto e legível, além de impedir que você atribua accidentalmente um valor à variável antes da chamada de método. O exemplo a seguir é semelhante ao exemplo anterior, exceto por definir a variável `number` na chamada para o método [Int32.TryParse](#).

C#

```
string numberAsString = "1640";  
  
if (Int32.TryParse(numberAsString, out int number))  
    Console.WriteLine($"Converted '{numberAsString}' to {number}");  
else  
    Console.WriteLine($"Unable to convert '{numberAsString}'");  
// The example displays the following output:  
//     Converted '1640' to 1640
```

No exemplo anterior, a variável `number` é fortemente tipada como um `int`. Você também pode declarar uma variável local de tipo implícito, como no exemplo a seguir.

C#

```
string numberAsString = "1640";  
  
if (Int32.TryParse(numberAsString, out var number))  
    Console.WriteLine($"Converted '{numberAsString}' to {number}");  
else  
    Console.WriteLine($"Unable to convert '{numberAsString}'");  
// The example displays the following output:  
//     Converted '1640' to 1640
```

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- Declaração de variável embutida (regra de estilo [IDE0018](#))
- Referência de C#
- Guia de Programação em C#

- Palavras-chave do C#
- Parâmetros de método

# namespace

Artigo • 07/04/2023

A palavra-chave `namespace` é usada para declarar um escopo que contém um conjunto de objetos relacionados. Você pode usar um namespace para organizar elementos de código e criar tipos globalmente exclusivos.

C#

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

As declarações de `namespace` em escopo de arquivo permitem declarar que todos os tipos em um arquivo estão em um único namespace. As declarações de namespace com escopo de arquivo estão disponíveis com o C# 10. O exemplo a seguir é semelhante ao exemplo anterior, mas usa uma declaração de namespace em escopo de arquivo:

C#

```
using System;

namespace SampleFileScopedNamespace;

class SampleClass { }

interface ISampleInterface { }

struct SampleStruct { }

enum SampleEnum { a, b }

delegate void SampleDelegate(int i);
```

O exemplo anterior não inclui um namespace aninhado. Namespaces com escopo de arquivo não podem incluir declarações de namespace adicionais. Você não pode declarar um namespace aninhado ou um segundo namespace com escopo de arquivo:

```
C#  
  
namespace SampleNamespace;  
  
class AnotherSampleClass  
{  
    public void AnotherSampleMethod()  
    {  
        System.Console.WriteLine(  
            "SampleMethod inside SampleNamespace");  
    }  
}  
  
namespace AnotherNamespace; // Not allowed!  
  
namespace ANestedNamespace // Not allowed!  
{  
    // declarations...  
}
```

Dentro de um namespace, é possível declarar zero ou mais dos seguintes tipos:

- [class](#)
- [interface](#)
- [struct](#)
- [enumeração](#)
- [delegate](#)
- namespaces aninhados podem ser declarados, exceto em declarações de namespace com escopo de arquivo

O compilador adiciona um namespace padrão. Este namespace sem nome, às vezes chamado de namespace global, está presente em todos os arquivos. Ele contém declarações não incluídas em um namespace declarado. Qualquer identificador no namespace global está disponível para uso em um namespace nomeado.

Os namespaces têm acesso público implicitamente. Para uma discussão sobre os modificadores de acesso que você pode atribuir a elementos em um namespace, consulte [Modificadores de acesso](#).

É possível definir um namespace em duas ou mais declarações. Por exemplo, o exemplo a seguir define duas classes como parte do namespace `MyCompany`:

```
C#
```

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

O exemplo a seguir mostra como chamar um método estático em um namespace aninhado.

C#

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}

// Output: Hello
```

## Especificação da linguagem C#

Para saber mais, confira a seção [Namespaces](#) da [Especificação da linguagem C#](#). Para obter mais informações sobre declarações de namespace com escopo de arquivo, consulte a [especificação do recurso](#).

## Confira também

- Preferências de declaração de namespace (IDE0160 e IDE0161)
- Referência de C#
- Palavras-chave de C#
- using
- using static
- Qualificador de alias de namespace ::
- Namespaces

# using (referência de C#)

Artigo • 20/03/2023

A palavra-chave `using` tem dois usos principais:

- A [instrução `using`](#) define um escopo no final do qual um objeto é descartado.
- A [diretiva `using`](#) cria um alias para um namespace ou importa tipos definidos em outros namespaces.

## Confira também

- [Referência de C#](#)
- [Palavras-chave de C#](#)

# usando diretiva

Artigo • 20/03/2023

A diretiva `using` permite que você use os tipos definidos em um namespace, sem especificar o namespace totalmente qualificado desse tipo. Na forma básica, a diretiva `using` importa todos os tipos de um único namespace, conforme mostrado no exemplo a seguir:

C#

```
using System.Text;
```

Você pode aplicar dois modificadores a uma diretiva `using`:

- O modificador `global` tem o mesmo efeito que adicionar a mesma diretiva `using` a cada arquivo de origem do projeto. Esse modificador foi introduzido no C# 10.
- O modificador `static` importa os membros `static` e tipos aninhados de um único tipo, em vez de importar todos os tipos em um namespace.

Você pode combinar ambos os modificadores, para importar os membros estáticos de um tipo em todos os arquivos de origem do projeto.

Você também pode criar um alias para um namespace ou um tipo com uma *diretiva using alias*.

C#

```
using Project = PC.MyCompany.Project;
```

Você pode usar o modificador `global` em uma *diretiva using alias*.

## ⓘ Observação

A palavra-chave `using` também é usada para criar *instruções using*, o que ajuda a garantir que objetos `IDisposable`, tais como arquivos e fontes, sejam tratados corretamente. Para obter mais informações sobre a *Instrução using*, confira [Instrução using](#).

O escopo de uma diretiva `using`, sem o modificador `global`, é limitado ao arquivo em que ele aparece.

A diretiva `using` pode aparecer:

- No início de um arquivo de código-fonte, antes de quaisquer declarações de namespace ou de tipo.
- Em qualquer namespace, mas antes dos namespaces ou tipos declarados nesse namespace, a menos que o modificador `global` seja usado. Nesse caso, a diretiva deve aparecer antes de todas as declarações de namespace e de tipo.

Caso contrário, serão gerados erros do compilador [CS1529](#).

Crie uma diretiva `using` para usar os tipos em um namespace sem precisar especificar o namespace. Uma diretiva `using` não fornece acesso a namespaces aninhados no namespace especificado. Os namespaces vêm em duas categorias: definidos pelo usuário e definidos pelo sistema. Os namespaces definidos pelo usuário são namespaces definidos em seu código. Para obter uma lista dos namespaces definidos pelo sistema, consulte [Navegador de API do .NET](#).

## modificador global

Adicionar o modificador `global` a uma diretiva `using` significa que o uso é aplicável a todos os arquivos na compilação (normalmente um projeto). A diretiva `global using` foi adicionada no C# 10. Sua sintaxe é:

```
C#  
  
global using <fully-qualified-namespace>;
```

onde o *namespace totalmente qualificado* é o nome totalmente qualificado do namespace, cujos tipos podem ser referenciados sem especificar o namespace.

Uma diretiva *global using* pode aparecer no início de qualquer arquivo de código-fonte. Todas as diretivas `global using` em um único arquivo devem aparecer antes:

- Todas as diretivas `using` sem o modificador `global`.
- Todas as declarações de namespace e de tipo no arquivo.

Você pode adicionar diretivas `global using` a qualquer arquivo de origem.

Normalmente, você vai querer mantê-las em um único local. A ordem das diretivas `global using` não importa, seja em um único arquivo ou entre arquivos.

O modificador `global` pode ser combinado com o modificador `static`. O modificador `global` pode ser aplicado a uma *diretiva using alias*. Em ambos os casos, o escopo da

diretiva inclui todos os arquivos na compilação atual. O exemplo a seguir permite o uso de todos os métodos declarados no [System.Math](#) em todos os arquivos do projeto:

```
C#
```

```
global using static System.Math;
```

Você também pode incluir globalmente um namespace, adicionando um item `<Using>` ao arquivo de projeto, por exemplo, `<Using Include="My.Awesome.Namespace" />`. Para obter mais informações, confira [Item <Using>](#).

### ⓘ Importante

Os modelos C# para o .NET 6 usam *instruções de nível superior*. Se você já tiver atualizado para o .NET 6, talvez seu aplicativo não corresponda ao código descrito neste artigo. Para obter mais informações, consulte o artigo sobre [Novos modelos C# geram instruções de nível superior](#)

O SDK do .NET 6 também adiciona um conjunto de diretivas *implícitas global using* para projetos que usam os seguintes SDKs:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Essas diretivas implícitas `global using` incluem os namespaces mais comuns para o tipo de projeto.

## modificador estático

A diretiva `using static` nomeia um tipo cujos membros estáticos e tipos aninhados podem ser acessados sem especificar um nome de tipo. Sua sintaxe é:

```
C#
```

```
using static <fully-qualified-type-name>;
```

O `<fully-qualified-type-name>` é o nome do tipo cujos membros estáticos e tipos aninhados podem ser referenciados sem especificar um nome de tipo. Se você não fornecer um nome de tipo totalmente qualificado (o nome do namespace completo juntamente com o nome do tipo), o C# gerará o erro de compilador [CS0246](#): "O tipo ou

o nome do namespace 'type/namespace' não pôde ser encontrado (uma diretiva using ou uma referência de assembly está ausente?)".

A diretiva `using static` aplica-se a qualquer tipo que tenha membros estático (ou tipos aninhados), mesmo que ele também tenha membros de instância. No entanto, os membros da instância podem ser invocados apenas por meio de instância de tipo.

Você pode acessar os membros estáticos de um tipo sem precisar qualificar o acesso com o nome do tipo:

C#

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

Normalmente, quando você chamar um membro estático, fornece o nome do tipo juntamente com o nome do membro. Inserir repetidamente o mesmo nome de tipo para invocar os membros do tipo pode resultar em código obscuro detalhado. Por exemplo, a seguinte definição de uma classe `Circle` referencia vários membros da classe `Math`.

C#

```
using System;

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * Math.PI; }
    }
}
```

```
}

    public double Area
    {
        get { return Math.PI * Math.Pow(Radius, 2); }
    }
}
```

Ao eliminar a necessidade de referenciar explicitamente a classe `Math`, sempre que um membro é referenciado, a diretiva `using static` produz um código mais limpo:

C#

```
using System;
using static System.Math;

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }
}
```

`using static` importa somente os membros estáticos acessíveis e os tipos aninhados declarados no tipo especificado. Os membros herdados não são importados. Você pode importar de qualquer tipo nomeado com uma diretiva `using static`, incluindo módulos do Visual Basic. Se funções de nível superior do F# aparecerem nos metadados como membros estáticos de um tipo nomeado cujo nome é um identificador válido do C#, as funções do F# poderão ser importadas.

`using static` torna os métodos de extensão declarados no tipo especificado disponível para pesquisa de método de extensão. No entanto, os nomes dos métodos de extensão não são importados no escopo para a referência não qualificada no código.

Métodos com o mesmo nome importados de diferentes tipos por diferentes diretivas `using static` na mesma unidade de compilação ou namespace formam um grupo de métodos. A resolução de sobrecarga nesses grupos de métodos segue as regras normais de C#.

O exemplo a seguir usa a diretiva `using static` para tornar os membros estáticos das classes `Console`, `Math` e `String` disponíveis sem a necessidade de especificar seu nome de tipo.

C#

```
using System;
using static System.Console;
using static System.Math;
using static System.String;

class Program
{
    static void Main()
    {
        Write("Enter a circle's radius: ");
        var input = ReadLine();
        if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
            var c = new Circle(radius);

            string s = "\nInformation about the circle:\n";
            s = s + Format("    Radius: {0:N2}\n", c.Radius);
            s = s + Format("    Diameter: {0:N2}\n", c.Diameter);
            s = s + Format("    Circumference: {0:N2}\n", c.Circumference);
            s = s + Format("    Area: {0:N2}\n", c.Area);
            WriteLine(s);
        }
        else {
            WriteLine("Invalid input...");
        }
    }

    public class Circle
    {
        public Circle(double radius)
        {
            Radius = radius;
        }

        public double Radius { get; set; }
```

```

public double Diameter
{
    get { return 2 * Radius; }
}

public double Circumference
{
    get { return 2 * Radius * PI; }
}

public double Area
{
    get { return PI * Pow(Radius, 2); }
}

// The example displays the following output:
//      Enter a circle's radius: 12.45
//
//      Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95

```

No exemplo, a diretiva `using static` também poderia ter sido aplicada ao tipo `Double`. Adicionar essa diretiva tornaria possível chamar o método `TryParse(String, Double)` sem especificar um nome de tipo. No entanto, usar `TryParse` sem um nome de tipo cria um código menos legível, pois torna-se necessário verificar as diretivas `using static` para determinar qual método `TryParse` do tipo numérico é chamado.

`using static` também se aplica aos tipos `enum`. Ao adicionar `using static` com a enumeração, o tipo não é mais necessário para usar os membros de enumeração.

C#

```

using static Color;

enum Color
{
    Red,
    Green,
    Blue
}

class Program
{
    public static void Main()
    {
        Color color = Green;
    }
}

```

# using alias

Crie uma diretiva de alias `using` para tornar mais fácil a qualificação de um identificador para um namespace ou tipo. Em qualquer diretiva `using`, o namespace totalmente qualificado ou o tipo deve ser usado independentemente das diretivas `using` que vêm antes. Nenhum alias `using` pode ser usado na declaração de uma diretiva `using`. Por exemplo, o exemplo a seguir gera um erro do compilador:

C#

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

O exemplo a seguir mostra como definir e usar um alias de `using` para um namespace:

C#

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            var mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

Uma diretiva `using alias` não pode ter um tipo genérico aberto no lado direito. Por exemplo, você não pode criar um `using alias` para um `List<T>`, mas pode criar um para um `List<int>`.

O exemplo a seguir mostra como definir uma diretiva `using` e um alias `using` para uma classe:

C#

```
using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    class MainClass
    {
        static void Main()
        {
            var instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            var instance2 = new UsingAlias();
            Console.WriteLine(instance2);
        }
    }
}
// Output:
//    You are in NameSpace1.MyClass.
//    You are in NameSpace2.MyClass.
```

## Como usar o namespace My do Visual Basic

O namespace `Microsoft.VisualBasic.MyServices` (`My` no Visual Basic) oferece acesso rápido e intuitivo a inúmeras classes do .NET, permitindo que você grave um código que interaja com o computador, com o aplicativo, com as configurações, com os recursos e assim por diante. Embora tenha sido projetado originalmente para ser usado com o Visual Basic, o namespace `MyServices` pode ser usado em aplicativos C#.

Para obter mais informações sobre como usar o namespace `MyServices` no Visual Basic, consulte [Desenvolvimento com My](#).

Você precisa adicionar uma referência ao `assemblyMicrosoft.VisualBasic.dll` no projeto. Nem todas as classes no namespace `MyServices` podem ser chamadas em um aplicativo C#: por exemplo, a classe `FileSystemProxy` não é compatível. Em vez disso, nesse caso específico, podem ser usados os métodos estáticos que fazem parte da `FileSystem`, que também estão contidos na `VisualBasic.dll`. Por exemplo, veja como usar um desses métodos para duplicar um diretório:

C#

```
// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");
```

## Especificação da linguagem C#

Para obter mais informações, consulte [Diretivas using](#) na [Especificação da Linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

Para obter mais informações sobre o modificador *global using*, confira a [especificação de recurso global using – C# 10](#).

## Confira também

- [Referência de C#](#)
- [Palavras-chave de C#](#)
- [Namespaces](#)
- [Regra de estilo IDE0005 – Remover diretivas 'using' desnecessárias](#)
- [Regra de estilo IDE0065 – posicionamento da diretiva 'using'](#)
- [usinginstrução](#)

# Instrução using – garantir o uso correto de objetos descartáveis

Artigo • 20/03/2023

A instrução `using` garante o uso correto de uma instância [IDisposable](#):

C#

```
var numbers = new List<int>();
using (StreamReader reader = File.OpenText("numbers.txt"))
{
    string line;
    while ((line = reader.ReadLine()) is not null)
    {
        if (int.TryParse(line, out int number))
        {
            numbers.Add(number);
        }
    }
}
```

Quando o controle sai do bloco da instrução `using`, uma instância adquirida [IDisposable](#) é descartada. Em particular, a instrução `using` garante que uma instância descartável seja descartada mesmo que ocorra uma exceção dentro do bloco da instrução `using`. No exemplo anterior, um arquivo aberto é fechado depois que todas as linhas são processadas.

Use a instrução `await using` para usar corretamente uma instância [IAsyncDisposable](#):

C#

```
await using (var resource = new AsyncDisposableExample())
{
    // Use the resource
}
```

Para obter mais informações sobre o uso de instâncias [IAsyncDisposable](#), confira a seção [Usando descartáveis assíncronos](#) do artigo [Implementar um método DisposeAsync](#).

Você também pode usar uma `using` declaração que não exige chaves:

C#

```
static IEnumerable<int> LoadNumbers(string filePath)
{
    using StreamReader reader = File.OpenText(filePath);

    var numbers = new List<int>();
    string line;
    while ((line = reader.ReadLine()) is not null)
    {
        if (int.TryParse(line, out int number))
        {
            numbers.Add(number);
        }
    }
    return numbers;
}
```

Quando declarada em uma declaração `using`, uma variável local é descartada no final do escopo em que é declarada. No exemplo anterior, o descarte ocorre no final de um método.

Uma variável declarada pela instrução ou declaração `using` é somente leitura. Não é possível reatribuí-la nem passá-la como um parâmetro `ref` ou `out`.

Você pode declarar várias instâncias do mesmo tipo em uma instrução `using`, como mostra o seguinte exemplo:

C#

```
using (StreamReader numbersFile = File.OpenText("numbers.txt"), wordsFile =
File.OpenText("words.txt"))
{
    // Process both files
}
```

Quando você declara várias instâncias em uma instrução `using`, elas são descartadas na ordem inversa da declaração.

Você também pode usar a instrução e a declaração `using` com uma instância de um `struct` `ref` que se ajusta ao padrão descartável. Ou seja, ele tem um método de instância `Dispose`, que é acessível, sem parâmetros e tem um tipo `void` de retorno.

A instrução `using` também pode ser da seguinte forma:

C#

```
using (expression)
{
```

```
// ...  
}
```

em que `expression` produz uma instância descartável. O exemplo a seguir demonstra que:

C#

```
StreamReader reader = File.OpenText(filePath);  
  
using (reader)  
{  
    // Process file content  
}
```

### ⚠️ Aviso

No exemplo anterior, depois que o controle deixa a instrução `using`, uma instância descartável permanece no escopo enquanto ela já está descartada. Se você usar essa instância mais adiante, poderá encontrar uma exceção, por exemplo, `ObjectDisposedException`. É por isso que recomendamos declarar uma variável descartável dentro da instrução `using` ou com a declaração `using`.

## Especificação da linguagem C#

Para obter mais informações, confira a seção [A instrução using](#) da especificação da linguagem C# e a nota de proposta sobre o "uso baseado em padrões" e "declarações using".

## Confira também

- [Referência de C#](#)
- [System.IDisposable](#)
- [System.IAsyncDisposable](#)
- [Usando objetos que implementam IDisposable](#)
- [Implementar um método Dispose](#)
- [Implementar um método DisposeAsync](#)
- [Como usar uma simples instrução 'using' \(regra de estilo IDE0063\)](#)
- [diretiva using](#)

# extern alias (Referência de C#)

Artigo • 07/04/2023

Talvez seja necessário referenciar duas versões de assemblies que têm os mesmos nomes de tipo totalmente qualificado. Por exemplo, você pode ter que usar duas ou mais versões de um assembly no mesmo aplicativo. Ao usar um alias de assembly externo, os namespaces de cada assembly podem ser encapsulados dentro de namespaces de nível raiz nomeados pelo alias, permitindo que eles sejam utilizados no mesmo arquivo.

## ① Observação

A palavra-chave `extern` também é usada como um modificador de método, declarando um método escrito em código não gerenciado.

Para referenciar dois assemblies com os mesmos nomes de tipo totalmente qualificado, um alias deve ser especificado em um prompt de comando, da seguinte maneira:

```
/r:GridV1=grid.dll
```

```
/r:GridV2=grid20.dll
```

Isso cria os alias externos `GridV1` e `GridV2`. Para usar esses aliases de dentro de um programa, referencie-os usando a palavra-chave `extern`. Por exemplo:

```
extern alias GridV1;
```

```
extern alias GridV2;
```

Cada declaração de alias externo apresenta um namespace de nível raiz adicional que funciona de forma paralela (mas não dentro) com o namespace global. Portanto, os tipos de cada assembly podem ser referenciados sem ambiguidade, usando seus nomes totalmente qualificados, enraizados no alias de namespace apropriado.

No exemplo anterior, `GridV1::Grid` seria o controle de grade da `grid.dll` e `GridV2::Grid` seria o controle de grade da `grid20.dll`.

## Como usar o Visual Studio

Se você estiver usando o Visual Studio, os alias poderão ser fornecidos de maneira semelhante.

Adicione referência de *grid.dll* e *grid20.dll* ao seu projeto no Visual Studio. Abra uma guia de propriedade e altere os alias de global para GridV1 e GridV2, respectivamente.

Use esses alias da mesma forma acima

C#

```
extern alias GridV1;  
  
extern alias GridV2;
```

Agora você pode criar alias para um namespace ou para um tipo *usando a diretiva de alias*. Para obter mais informações, confira [Diretiva using](#).

C#

```
using Class1V1 = GridV1::Namespace.Class1;  
  
using Class1V2 = GridV2::Namespace.Class1;
```

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Operador ::](#)
- [Referências \(opções do compilador C#\)](#)

# Restrição new (Referência em C#)

Artigo • 07/04/2023

A restrição `new` especifica que um argumento de tipo em uma declaração de classe ou método genérica deve ter um construtor público sem parâmetros. Para usar a restrição `new`, o tipo não pode ser abstrato.

Aplique a restrição `new` a um parâmetro de tipo quando uma classe genérica criar novas instâncias do tipo, conforme mostrado no exemplo a seguir:

C#

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

Quando você usa a restrição `new()` com outras restrições, ela deve ser especificada por último:

C#

```
public class ItemFactory2<T>
    where T : IComparable, new()
{ }
```

Para obter mais informações, consulte [Restrições a parâmetros de tipo](#).

Você também pode usar a palavra-chave `new` para [criar uma instância de um tipo](#) ou como um [modificador de declaração de membro](#).

## Especificação da linguagem C#

Para obter mais informações, confira a seção [Restrições de parâmetro de tipo](#) na [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)

- Guia de Programação em C#
- Palavras-chave do C#
- Genéricos

# where (restrição de tipo genérico) (Referência de C#)

Artigo • 07/04/2023

A cláusula `where` em uma definição genérica especifica restrições sobre os tipos que são usados como argumentos para parâmetros de tipo em um tipo genérico, método, delegado ou função local. Restrições podem especificar interfaces, classes base ou exigir que um tipo genérico seja uma referência, valor ou tipo não gerenciado. Eles declaram recursos que o argumento de tipo deve ter e devem ser colocados após qualquer classe base declarada ou interfaces implementadas.

Por exemplo, você pode declarar uma classe genérica, `AGenericClass`, de modo que o parâmetro de tipo `T` implementa a interface `IComparable<T>`:

C#

```
public class AGenericClass<T> where T : IComparable<T> { }
```

## ⓘ Observação

Para obter mais informações sobre a cláusula `where` em uma expressão de consulta, consulte [Cláusula where](#).

A cláusula `where` também pode incluir uma restrição de classe base. A restrição de classe base declara que um tipo a ser usado como um argumento de tipo para aquele tipo genérico tem a classe especificada como uma classe base ou é essa classe base. Se a restrição de classe base for usada, ela deverá aparecer antes de qualquer outra restrição nesse parâmetro de tipo. Alguns tipos não têm permissão como uma restrição de classe base: `Object`, `Array` e `ValueType`. O exemplo a seguir mostra os tipos que agora podem ser especificados como classe base:

C#

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

Em um contexto anulável, a nulidade do tipo de classe base é imposta. Se a classe base não for anulável (por exemplo `Base`), o argumento de tipo deverá ser não anulável. Se a classe base for anulável (por exemplo, `Base?`), o argumento de tipo poderá ser um tipo de referência anulável ou não anulável. O compilador emitirá um aviso se o argumento de tipo for um tipo de referência anulável quando a classe base não for anulável.

A cláusula `where` pode especificar que o tipo é um `class` ou um `struct`. A restrição `struct` elimina a necessidade de especificar uma restrição de classe base de `System.ValueType`. O tipo `System.ValueType` não pode ser usado como uma restrição de classe base. O exemplo a seguir mostra as restrições `class` e `struct`:

```
C#  
  
class MyClass<T, U>  
    where T : class  
    where U : struct  
{ }
```

Em um contexto anulável, a restrição `class` requer que um tipo seja um tipo de referência não anulável. Para permitir tipos de referência anuláveis, use a restrição `class?`, que permite tipos de referência anuláveis e não anuláveis.

A cláusula `where` pode incluir a restrição `notnull`. A restrição `notnull` limita o parâmetro de tipo a tipos não anuláveis. O tipo pode ser um [tipo de valor](#) ou um tipo de referência não anulável. A `notnull` restrição está disponível para código compilado em um [nullable enable contexto](#). Ao contrário de outras restrições, se um argumento de tipo violar a restrição `notnull`, o compilador gerará um aviso em vez de um erro. Os avisos são gerados apenas em um contexto `nullable enable`.

A adição de tipos de referência anuláveis introduz uma ambiguidade potencial no significado de `T?` em métodos genéricos. Se `T` for um `struct`, `T?` é o mesmo que `System.Nullable<T>`. No entanto, se `T` for um tipo de referência, `T?` significa que `null` é um valor válido. A ambiguidade surge porque os métodos de substituição não podem incluir restrições. A nova restrição `default` resolve essa ambiguidade. Você a adicionará quando uma classe base ou interface declarar duas sobrecargas de um método, uma que especifica a restrição `struct` e outra que não tenha a restrição `struct` ou `class` aplicada:

```
C#  
  
public abstract class B  
{  
    public void M<T>(T? item) where T : struct { } }
```

```
    public abstract void M<T>(T? item);  
}
```

Você usa a restrição `default` para especificar que sua classe derivada substitui o método sem a restrição em sua classe derivada ou a implementação de interface explícita. Ele só é válido em métodos que substituem métodos base ou implementações de interface explícitas:

C#

```
public class D : B  
{  
    // Without the "default" constraint, the compiler tries to override the  
    first method in B  
    public override void M<T>(T? item) where T : default { }  
}
```

### ⓘ Importante

Declarações genéricas que incluem a restrição `notnull` podem ser usadas em um contexto alheio anulável, mas o compilador não impõe a restrição.

C#

```
#nullable enable  
class NotNullContainer<T>  
    where T : notnull  
{  
}  
#nullable restore
```

A cláusula `where` também pode incluir uma restrição `unmanaged`. A restrição `unmanaged` limita o parâmetro de tipo a tipos conhecidos como [tipos não gerenciados](#). Usando a restrição `unmanaged`, é mais fácil escrever o código de interoperabilidade de nível baixo em C#. Essa restrição habilita rotinas reutilizáveis em todos os tipos não gerenciados. A restrição `unmanaged` não pode ser combinada à restrição `class` ou `struct`. A restrição `unmanaged` impõe que o tipo deve ser um `struct`:

C#

```
class UnManagedWrapper<T>  
    where T : unmanaged  
{ }
```

A cláusula `where` também pode incluir uma restrição de construtor, `new()`. Essa restrição torna possível criar uma instância de um parâmetro de tipo usando o operador `new`. A restrição `new()` informa o compilador que qualquer argumento de tipo fornecido deve ter um construtor sem parâmetros acessível. Por exemplo:

```
C#  
  
public class MyGenericClass<T> where T : IComparable<T>, new()  
{  
    // The following line is not possible without new() constraint:  
    T item = new T();  
}
```

A restrição `new()` aparece por último na cláusula `where`. A restrição `new()` não pode ser combinada às restrições `struct` ou `unmanaged`. Todos os tipos que satisfazem as restrições devem ter um construtor sem parâmetros acessível, tornando a restrição `new()` redundante.

Com vários parâmetros de tipo, use uma cláusula `where` para cada parâmetro de tipo, por exemplo:

```
C#  
  
public interface IMyInterface { }  
  
namespace CodeExample  
{  
    class Dictionary<TKey, TValue>  
        where TKey : IComparable<TKey>  
        where TValue : IMyInterface  
    {  
        public void Add(TKey key, TValue val) { }  
    }  
}
```

Você também pode anexar restrições a parâmetros de tipo de métodos genéricos, como mostrado no exemplo a seguir:

```
C#  
  
public void MyMethod<T>(T t) where T : IMyInterface { }
```

Observe que a sintaxe para descrever as restrições de parâmetro de tipo em delegados é a mesma que a dos métodos:

C#

```
delegate T MyDelegate<T>() where T : new();
```

Para obter informações sobre delegados genéricos, consulte [Delegados genéricos](#).

Para obter detalhes sobre a sintaxe e o uso de restrições, consulte [Restrições a parâmetros de tipo](#).

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Introdução aos genéricos](#)
- [Restrição new](#)
- [Restrições a parâmetros de tipo](#)

# base (Referência de C#)

Artigo • 24/03/2023

A palavra-chave `base` é usada para acessar os membros da classe base por meio de uma classe derivada. Use-a para os seguintes casos:

- Chamar um método que foi substituído por outro método na classe base.
- Especificar qual construtor de classe base deve ser chamado ao criar instâncias da classe derivada.

O acesso à classe base é permitido somente em um construtor, em um método de instância ou em um acessador de propriedade de instância.

Usar a palavra-chave `base` em um método estático gera um erro.

A classe base que é acessada é a que é especificada na declaração de classe. Por exemplo, se você especificar `class ClassB : ClassA`, os membros da ClassA são acessados da ClassB, independentemente da classe base da ClassA.

## Exemplo 1

Neste exemplo, tanto a classe base `Person` quanto a classe derivada `Employee` têm um método chamado `GetInfo`. Usando a palavra-chave `base`, é possível chamar o método `GetInfo` da classe base por meio da classe derivada.

```
C#  
  
public class Person  
{  
    protected string ssn = "444-55-6666";  
    protected string name = "John L. Malgraine";  
  
    public virtual void GetInfo()  
    {  
        Console.WriteLine("Name: {0}", name);  
        Console.WriteLine("SSN: {0}", ssn);  
    }  
}  
class Employee : Person  
{  
    public string id = "ABC567EFG";  
    public override void GetInfo()  
    {  
        // Calling the base class GetInfo method:  
        base.GetInfo();  
    }  
}
```

```
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

Para obter exemplos adicionais, consulte [new](#), [virtual](#) e [override](#).

## Exemplo 2

Este exemplo mostra como especificar o construtor da classe base chamado ao criar instâncias de uma classe derivada.

C#

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
```

```
{  
    // This constructor will call BaseClass.BaseClass()  
    public DerivedClass() : base()  
    {  
    }  
  
    // This constructor will call BaseClass.BaseClass(int i)  
    public DerivedClass(int i) : base(i)  
    {  
    }  
  
    static void Main()  
    {  
        DerivedClass md = new DerivedClass();  
        DerivedClass md1 = new DerivedClass(1);  
    }  
}  
/*  
Output:  
in BaseClass()  
in BaseClass(int i)  
*/
```

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [this](#)

# this (Referência de C#)

Artigo • 07/04/2023

A palavra-chave `this` refere-se à instância atual da classe e também é usada como um modificador do primeiro parâmetro de um método de extensão.

## ⓘ Observação

Este artigo discute o uso de `this` com instâncias de classe. Para obter mais informações sobre seu uso em métodos de extensão, consulte [Métodos de extensão](#).

Veja a seguir usos comuns de `this`:

- Para qualificar membros ocultados por nomes semelhantes, por exemplo:

```
C#  
  
public class Employee  
{  
    private string alias;  
    private string name;  
  
    public Employee(string name, string alias)  
    {  
        // Use this to qualify the members of the class  
        // instead of the constructor parameters.  
        this.name = name;  
        this.alias = alias;  
    }  
}
```

- Para passar um objeto como parâmetro para outros métodos, por exemplo:

```
C#  
  
CalcTax(this);
```

- Para declarar [indexadores](#), por exemplo:

```
C#  
  
public int this[int param]  
{  
    get { return array[param]; }
```

```
    set { array[param] = value; }
```

Funções de membro estático, por existirem no nível da classe e não como parte de um objeto, não têm um ponteiro `this`. É um erro se referir a `this` em um método estático.

## Exemplo

Neste exemplo, `this` é usado para qualificar os membros de classe `Employee`, `name` e `alias`, que são ocultados por nomes semelhantes. Ele também é usado para passar um objeto para o método `CalcTax`, que pertence a outra classe.

C#

```
class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }

    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}
```

```
class MainClass
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
    Name: Mingda Pan
    Alias: mpan
    Taxes: $240.00
*/
```

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- Preferências de estilo de código this (IDE0003 e IDE0009)
- Referência de C#
- Guia de Programação em C#
- Palavras-chave do C#
- base
- Métodos

# null (Referência de C#)

Artigo • 07/04/2023

A palavra-chave `null` é um literal que representa uma referência nula, que não faz referência a qualquer objeto. `null` é o valor padrão de variáveis do tipo de referência. Os tipos de valor comuns não podem ser nulos, exceto para [tipos de valor anuláveis](#).

O seguinte exemplo demonstra alguns comportamentos da palavra-chave `null`:

C#

```
class Program
{
    class MyClass
    {
        public void MyMethod() { }

    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
```

```
    bool b = (t.Equals(s));
    Console.WriteLine(b);

    // Equality operator also returns false when one
    // operand is null.
    Console.WriteLine("Empty string {0} null string", s == t ? "equals":
"does not equal");

    // Returns true.
    Console.WriteLine("null == null is {0}", null == null);

    // A value type cannot be null
    // int i = null; // Compiler error!

    // Use a nullable value type instead:
    int? i = null;

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}

}
```

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Palavras-chave de C#](#)
- [Valores padrão de tipos C#](#)
- [Nada \(Visual Basic\)](#)

# bool (referência de C#)

Artigo • 10/05/2023

A palavra-chave de tipo `bool` é um alias para o tipo de estrutura `System.Boolean` do .NET que representa um valor booleano, que pode ser `true` ou `false`.

Para executar operações lógicas com valores do tipo `bool`, use operadores [lógicos booleanos](#). O tipo `bool` é o tipo de resultado de operadores de [comparação](#) e [igualdade](#). Uma expressão `bool` pode ser uma expressão condicional de controle nas instruções `if`, `do`, `while` e `for` e no [operador condicional ?:](#).

O valor padrão do tipo `bool` é `false`.

## Literais

Você pode usar os literais `true` e `false` para inicializar uma variável `bool` ou passar um valor `bool`:

C#

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked"); // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked"); // output: Not
checked
```

## Lógica booleana de três valores

Use o tipo `bool?` se você precisar oferecer suporte à lógica de três valores, por exemplo, ao trabalhar com bancos de dados que dão suporte a um tipo booleano de três valores. Para os operandos `bool?`, os operadores `&` e `|` predefinidos oferecem suporte à lógica de três valores. Para obter mais informações, confira a seção [Operadores lógicos booleanos anuláveis](#) do artigo [Operadores lógicos booleanos](#).

Para obter mais informações sobre tipos que permitem valor nulo, consulte [Tipos que permitem valor nulo](#).

## Conversões

O C# fornece apenas duas conversões que envolvem o tipo `bool`. Esse tipo é uma conversão implícita para o tipo que permite valor nulo `bool?` correspondente e uma conversão explícita do tipo `bool?`. No entanto, o .NET fornece métodos adicionais que você pode usar para converter de ou para o tipo `bool`. Para obter mais informações, consulte a seção [Convertendo de e para valores booleanos](#) da página de referência da API `System.Boolean`.

## Especificação da linguagem C#

Para obter mais informações, confira a seção [O tipo bool](#) da [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Tipos de valor](#)
- [operadores true e false](#)

# padrão – Referência de C#

Artigo • 07/04/2023

Você pode usar a palavra-chave `default` nos seguintes contextos:

- Para especificar o rótulo padrão na [instrução switch](#).
- Como o [operador padrão ou literal](#) para produzir o valor padrão de um tipo.
- Como a restrição de tipo `default` em uma substituição de método genérico ou implementação de interface explícita.

## Confira também

- [Referência de C#](#)
- [Palavras-chave de C#](#)

# add (Referência de C#)

Artigo • 07/04/2023

A palavra-chave contextual `add` é usada para definir um acessador de evento personalizado que é invocado quando o código cliente assina seu [evento](#). Se você fornecer um acessador `add` personalizado, também será fornecer um acessador [remove](#).

## Exemplo

O exemplo a seguir mostra um evento que tem acessadores `add` e `remove` personalizados. Para obter o exemplo completo, confira [Como implementar eventos de interface](#).

C#

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

Normalmente, não é necessário fornecer seus próprios acessadores de eventos personalizados. Os acessadores que são gerados automaticamente pelo compilador quando você declara um evento são suficientes para a maioria dos cenários.

## Confira também

- [Eventos](#)

# get (Referência de C#)

Artigo • 08/06/2023

A palavra-chave `get` define um método do *acessador* em uma propriedade ou um indexador que retorna o valor da propriedade ou o elemento do indexador. Para obter mais informações, consulte [Propriedades](#), [Propriedades autoimplementadas](#) e [Indexadores](#).

O exemplo a seguir define um acessador `get` e um acessador `set` para uma propriedade chamada `Seconds`. Ela usa um campo particular chamado `_seconds` para dar suporte ao valor da propriedade.

C#

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}
```

Geralmente, o acessador `get` consiste em uma única instrução que retorna um valor, como no exemplo anterior. Você pode implementar o acessador `get` como um membro apto para expressão. O exemplo a seguir implementa os acessadores `get` e `set` como membros aptos para expressão.

C#

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

Para casos simples em que os acessadores `get` e `set` de uma propriedade não realizam nenhuma outra operação, a não ser a configuração ou a recuperação de um valor em um campo de suporte particular, você pode tirar proveito do suporte do compilador do C# para propriedades autoimplementadas. O exemplo a seguir implementa `Hours` como uma propriedade autoimplementada.

C#

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

## Especificação da Linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Propriedades](#)

# init (Referência de C#)

Artigo • 29/06/2023

Em C# 9 e versões posteriores, a palavra-chave `init` define um método *de acessador* em uma propriedade ou um indexador. Um setter somente de inicialização atribui um valor à propriedade ou ao elemento indexador **somente** durante a construção do objeto. Isso impõe a imutabilidade para que, após o objeto ser inicializado, ele não possa ser alterado novamente.

Para obter mais informações e exemplos, consulte [Propriedades, Propriedades autoimplementadas e Indexadores](#).

O exemplo a seguir define um acessador `get` e um acessador `init` para uma propriedade chamada `YearOfBirth`. Ela usa um campo particular chamado `_yearOfBirth` para dar suporte ao valor da propriedade.

C#

```
class Person_InitExample
{
    private int _yearOfBirth;

    public int YearOfBirth
    {
        get { return _yearOfBirth; }
        init { _yearOfBirth = value; }
    }
}
```

Geralmente, o acessador `init` consiste em uma única instrução que retorna um valor, como no exemplo anterior. Observe que, devido a `init`, o seguinte **não** funcionará:

C#

```
var john = new Person_InitExample
{
    YearOfBirth = 1984
};

john.YearOfBirth = 1926; //Not allowed, as its value can only be set once in
the constructor
```

Um acessador `init` não força os chamadores a definir a propriedade. Em vez disso, permite que um inicializador de objeto defina o valor inicial, proibindo a modificação

posterior. Você pode adicionar o modificador `required` para forçar os chamadores a definir uma propriedade. O exemplo a seguir mostra o mesmo comportamento:

```
C#  
  
class Person_InitExampleNullability  
{  
    private int? _yearOfBirth;  
  
    public int? YearOfBirth  
    {  
        get => _yearOfBirth;  
        init => _yearOfBirth = value;  
    }  
}
```

O acessador `init` pode ser usado como membro apto para expressão. Exemplo:

```
C#  
  
class Person_InitExampleExpressionBodied  
{  
    private int _yearOfBirth;  
  
    public int YearOfBirth  
    {  
        get => _yearOfBirth;  
        init => _yearOfBirth = value;  
    }  
}
```

O acessador `init` também pode ser usado em propriedades autoimplementadas, como demonstra o seguinte código de exemplo:

```
C#  
  
class Person_InitExampleAutoProperty  
{  
    public int YearOfBirth { get; init; }  
}
```

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Propriedades](#)

# tipo parcial (Referência em C#)

Artigo • 07/04/2023

As definições de tipo parcial permitem que a definição de uma classe, estrutura, interface ou registro seja dividida em vários arquivos.

Em *File1.cs*:

```
C#  
  
namespace PC  
{  
    partial class A  
    {  
        int num = 0;  
        void MethodA() { }  
        partial void MethodC();  
    }  
}
```

Em *File2.cs*, a declaração:

```
C#  
  
namespace PC  
{  
    partial class A  
    {  
        void MethodB() { }  
        partial void MethodC() { }  
    }  
}
```

## Comentários

Dividir um tipo de classe, struct ou interface em vários arquivos pode ser útil quando você está trabalhando com projetos grandes ou com o código gerado automaticamente, como o código fornecido pelo [Designer de Formulários do Windows](#). Um tipo parcial pode conter um [método parcial](#). Para obter mais informações, consulte [Classes parciais e métodos](#).

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Modificadores](#)
- [Introdução aos genéricos](#)

# Método parcial (C# Reference)

Artigo • 10/03/2023

Um método parcial tem sua assinatura definida em uma parte de um tipo parcial e sua implementação definida em outra parte do tipo. Os métodos parciais permitem que os designers de classe forneçam ganchos de método, semelhantes a manipuladores de eventos, que os desenvolvedores podem decidir implementar ou não. Se o desenvolvedor não fornecer uma implementação, o compilador removerá a assinatura no tempo de compilação. As seguintes condições são aplicáveis a métodos parciais:

- As declarações devem começar com a palavra-chave contextual [partial](#).
- As assinaturas em ambas as partes do tipo parcial devem ser correspondentes.

A palavra-chave `partial` não é permitida em construtores, finalizadores, operadores sobrecarregados, declarações de propriedade ou declarações de evento.

Um método parcial não precisa ter uma implementação nos seguintes casos:

- Ele não tem modificadores de acessibilidade (incluindo o [private](#) padrão).
- Ele retorna [void](#).
- Ele não tem parâmetros [out](#).
- Ele não tem nenhum dos seguintes modificadores [virtual](#), [override](#), [sealed](#), [new](#) ou [extern](#).

Qualquer método que não esteja em conformidade com todas essas restrições (por exemplo, método `public virtual partial void`), deve fornecer uma implementação.

O exemplo a seguir mostra um método parcial definido em duas partes de uma classe parcial:

```
C#  
  
namespace PM  
{  
    partial class A  
    {  
        partial void OnSomethingHappened(string s);  
    }  
  
    // This part can be in a separate file.  
    partial class A  
    {  
    }
```

```
// Comment out this method and the program
// will still compile.
partial void OnSomethingHappened(String s)
{
    Console.WriteLine("Something happened: {0}", s);
}
}
```

Métodos parciais também podem ser úteis em combinação com geradores de origem. Por exemplo, uma expressão regular pode ser definida usando o seguinte padrão:

C#

```
[GeneratedRegex("(dog|cat|fish)")]
partial bool IsPetMatch(string input);
```

Para obter mais informações, consulte [Classes parciais e métodos](#).

## Confira também

- [Referência de C#](#)
- [Tipo parcial](#)

# remove (Referência de C#)

Artigo • 07/04/2023

A palavra-chave contextual `remove` é usada para definir um acessador de eventos personalizado invocado quando o código cliente cancela a assinatura do seu [evento](#). Se você fornecer um acessador `remove` personalizado, também será necessário fornecer um acessador [add](#).

## Exemplo

O exemplo a seguir mostra um evento com os acessadores `add` e `remove` personalizados. Para obter o exemplo completo, confira [Como implementar eventos de interface](#).

C#

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

Normalmente, não é necessário fornecer seus próprios acessadores de eventos personalizados. Os acessadores que são gerados automaticamente pelo compilador quando você declara um evento são suficientes para a maioria dos cenários.

## Confira também

- [Eventos](#)

# set (Referência de C#)

Artigo • 08/06/2023

A palavra-chave `set` define um método *acessador* em uma propriedade ou indexador que atribui um valor ao elemento da propriedade ou do elemento. Para obter mais informações e exemplos, consulte [Propriedades](#), [Propriedades autoimplementadas](#) e [Indexadores](#).

O exemplo a seguir define um acessador `get` e um acessador `set` para uma propriedade chamada `Seconds`. Ela usa um campo particular chamado `_seconds` para dar suporte ao valor da propriedade.

C#

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}
```

Geralmente, o acessador `set` consiste em uma única instrução que retorna um valor, como no exemplo anterior. Você pode implementar o acessador `set` como um membro apto para expressão. O exemplo a seguir implementa os acessadores `get` e `set` como membros com corpo de expressão.

C#

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

Para casos simples em que os acessadores `get` e `set` de uma propriedade não realizam nenhuma outra operação, a não ser a configuração ou a recuperação de um valor em um campo de suporte particular, você pode tirar proveito do suporte do compilador do C# para propriedades autoimplementadas. O exemplo a seguir implementa `Hours` como uma propriedade autoimplementada.

C#

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)
- [Propriedades](#)

# when (Referência de C#)

Artigo • 07/04/2023

Você usa a palavra-chave contextual `when` para especificar uma condição de filtro nos seguintes contextos:

- Na instrução `catch` de um bloco `try/catch` ou `try/catch/finally`.
- Como uma [proteção de maiúsculas e minúsculas](#) na instrução `switch`.
- Como uma [proteção de maiúsculas e minúsculas](#) na expressão `switch`.

## when em uma instrução catch

A `when` palavra-chave pode ser usada em uma `catch` instrução para especificar uma condição que deve ser verdadeira para o manipulador para que uma exceção específica seja executada. Sua sintaxe é:

C#

```
catch (ExceptionType [e]) when (expr)
```

em que `expr` é uma expressão que é avaliada como um valor booleano. Se ele retornar `true`, o manipulador de exceção será executado, se `false`, não executará.

O exemplo a seguir usa a palavra-chave `when` para executar os manipuladores condicionalmente para um `HttpRequestException` dependendo do texto da mensagem de exceção.

C#

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Console.WriteLine(MakeRequest().Result);
    }

    public static async Task<string> MakeRequest()
    {
        var client = new HttpClient();
        var streamTask = client.GetStringAsync("https://localhost:10000");
    }
}
```

```
try
{
    var responseText = await streamTask;
    return responseText;
}
catch (HttpRequestException e) when (e.Message.Contains("301"))
{
    return "Site Moved";
}
catch (HttpRequestException e) when (e.Message.Contains("404"))
{
    return "Page Not Found";
}
catch (HttpRequestException e)
{
    return e.Message;
}
}
```

## Confira também

- instruções try/catch
- instrução try/catch/finally

# value (Referência de C#)

Artigo • 07/04/2023

A palavra-chave contextual `value` é usada no acessador `set` em declarações de `property` e `indexer`. É semelhante a um parâmetro de entrada de um método. A palavra `value` faz referência ao valor que o código do cliente está tentando atribuir à propriedade ou ao indexador. No exemplo a seguir, `MyDerivedClass` tem uma propriedade chamada `Name` que usa o parâmetro `value` para atribuir uma nova cadeia de caracteres ao campo de suporte `_name`. Do ponto de vista do código cliente, a operação é gravada como uma atribuição simples.

C#

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int _num;
    public virtual int Number
    {
        get { return _num; }
        set { _num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string _name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
            }
            else
            {
                _name = "Unknown";
            }
        }
    }
}
```

```
        }  
    }  
}
```

Para obter mais informações, confira os artigos [Propriedades](#) e [Indexadores](#).

## Especificação da linguagem C#

Para obter mais informações, consulte a [Especificação da linguagem C#](#). A especificação da linguagem é a fonte definitiva para a sintaxe e o uso de C#.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)
- [Palavras-chave do C#](#)

# instrução yield – fornecer o próximo elemento

Artigo • 05/06/2023

Use a instrução `yield` em um [iterador](#) para fornecer o próximo valor ou sinalizar o fim de uma iteração. A instrução `yield` tem as duas seguintes formas:

- `yield return`: para fornecer o próximo valor na iteração, como mostra o seguinte exemplo:

```
C#  
  
foreach (int i in ProduceEvenNumbers(9))  
{  
    Console.Write(i);  
    Console.WriteLine(" ");  
}  
// Output: 0 2 4 6 8  
  
IEnumerable<int> ProduceEvenNumbers(int upto)  
{  
    for (int i = 0; i <= upto; i += 2)  
    {  
        yield return i;  
    }  
}
```

- `yield break`: para sinalizar explicitamente o fim da iteração, como mostra o seguinte exemplo:

```
C#  
  
Console.WriteLine(string.Join(" ", TakeWhilePositive(new[] { 2, 3, 4,  
5, -1, 3, 4})));  
// Output: 2 3 4 5  
  
Console.WriteLine(string.Join(" ", TakeWhilePositive(new[] { 9, 8, 7  
})));  
// Output: 9 8 7  
  
IEnumerable<int> TakeWhilePositive(IEnumerable<int> numbers)  
{  
    foreach (int n in numbers)  
    {  
        if (n > 0)  
        {  
            yield return n;  
        }  
    }  
}
```

```

        }
        else
        {
            yield break;
        }
    }
}

```

A iteração também é concluída quando o controle atinge o final de um iterador.

Nos exemplos anteriores, o tipo de retorno dos iteradores é `IEnumerable<T>` (em casos não genéricos, use `IEnumerable` como tipo de retorno de um iterador). Você também pode usar `IAsyncEnumerable<T>` como tipo de retorno de um iterador. Isso torna o iterador assíncrono. Use a instrução `await foreach` para iterar sobre o resultado do iterador, como mostra o seguinte exemplo:

```
C#
await foreach (int n in GenerateNumbersAsync(5))
{
    Console.Write(n);
    Console.Write(" ");
}
// Output: 0 2 4 6 8

async IAsyncEnumerable<int> GenerateNumbersAsync(int count)
{
    for (int i = 0; i < count; i++)
    {
        yield return await ProduceNumberAsync(i);
    }
}

async Task<int> ProduceNumberAsync(int seed)
{
    await Task.Delay(1000);
    return 2 * seed;
}
```

`IEnumerator<T>` ou `IEnumerator` também podem ser o tipo de retorno de um iterador. Isso é útil quando você implementa o método `GetEnumerator` nos seguintes cenários:

- Você projeta o tipo que implementa a interface `IEnumerable<T>` ou `IEnumerable`.
- Você adiciona uma instância ou método de extensão `GetEnumerator` para habilitar a iteração sobre a instância do tipo com a instrução `foreach`, como mostra o seguinte exemplo:

C#

```
public static void Example()
{
    var point = new Point(1, 2, 3);
    foreach (int coordinate in point)
    {
        Console.Write(coordinate);
        Console.Write(" ");
    }
    // Output: 1 2 3
}

public readonly record struct Point(int X, int Y, int Z)
{
    public IEnumerator<int> GetEnumerator()
    {
        yield return X;
        yield return Y;
        yield return Z;
    }
}
```

Você não pode usar instruções `yield` em:

- métodos com os parâmetros `in`, `ref` ou `out`
- expressões lambda e métodos anônimos
- métodos que contêm blocos inseguros

## Execução de um iterador

A chamada de um iterador não o executa imediatamente, como mostra o seguinte exemplo:

C#

```
var numbers = ProduceEvenNumbers(5);
Console.WriteLine("Caller: about to iterate.");
foreach (int i in numbers)
{
    Console.WriteLine($"Caller: {i}");
}

IEnumerable<int> ProduceEvenNumbers(int upto)
{
    Console.WriteLine("Iterator: start.");
    for (int i = 0; i <= upto; i += 2)
    {
        Console.WriteLine($"Iterator: about to yield {i}");
```

```
        yield return i;
        Console.WriteLine($"Iterator: yielded {i}");
    }
    Console.WriteLine("Iterator: end.");
}
// Output:
// Caller: about to iterate.
// Iterator: start.
// Iterator: about to yield 0
// Caller: 0
// Iterator: yielded 0
// Iterator: about to yield 2
// Caller: 2
// Iterator: yielded 2
// Iterator: about to yield 4
// Caller: 4
// Iterator: yielded 4
// Iterator: end.
```

Como mostra o exemplo anterior, quando você começa a iterar sobre o resultado de um iterador, o iterador é executado até que a primeira instrução `yield return` seja atingida. Em seguida, a execução de um iterador é suspensa e o chamador obtém o primeiro valor de iteração e o processa. Em cada iteração subsequente, a execução de um iterador é retomada após a instrução `yield return` que causou a suspensão anterior e continua até que a próxima instrução `yield return` seja atingida. A iteração é concluída quando o controle atinge o final de um iterador ou uma instrução `yield break`.

## Especificação da linguagem C#

Para saber mais, confira a seção [A instrução `yield`](#) na [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Iteradores](#)
- [Iterar em coleções em C#](#)
- [foreach](#)
- [await foreach](#)

# Palavras-chave de consulta (Referência de C#)

Artigo • 08/06/2023

Esta seção contém as palavras-chave contextuais usadas em expressões de consulta.

## Nesta seção

Cláusula	Descrição
from	Especifica uma fonte de dados e uma variável de intervalo (semelhante a uma variável de iteração).
where	Filtre elementos de origem baseados em uma ou mais expressões booleanas separadas por operadores AND e OR lógicos ( <code>&amp;&amp;</code> ou <code>  </code> ).
select	Especifica o tipo e a forma que os elementos na sequência retornada terão quando a consulta for executada.
grupo	Agrupa os resultados da consulta de acordo com um valor de chave especificado.
into	Fornece um identificador que pode funcionar como uma referência aos resultados de uma cláusula join, group ou select.
orderby	Classifica os resultados da consulta em ordem crescente ou decrescente com base no comparador padrão para o tipo de elemento.
join	Une duas fontes de dados com base em uma comparação de igualdade entre dois critérios de correspondência especificados.
let	Introduz uma variável de intervalo para armazenar os resultados de subexpressão em uma expressão de consulta.
Em	Palavra-chave contextual em uma cláusula join.
on	Palavra-chave contextual em uma cláusula join.
equals	Palavra-chave contextual em uma cláusula join.
by	Palavra-chave contextual em uma cláusula group.
ascending	Palavra-chave contextual em uma cláusula orderby.
descending	Palavra-chave contextual em uma cláusula orderby.

## Confira também

- Palavras-chave do C#
- LINQ (Consulta Integrada à Linguagem)
- LINQ em C#

# Cláusula from (Referência de C#)

Artigo • 07/04/2023

Uma expressão de consulta deve começar com uma cláusula `from`. Além disso, uma expressão de consulta pode conter subconsultas, que também começam com uma cláusula `from`. A cláusula `from` especifica o seguinte:

- A fonte de dados na qual a consulta ou subconsulta será executada.
- Uma *variável de intervalo* local que representa cada elemento na sequência de origem.

A variável de intervalo e a fonte de dados são fortemente tipadas. A fonte de dados referenciada na cláusula `from` deve ter um tipo de `IEnumerable`, `IEnumerable<T>` ou um tipo derivado, por exemplo, `IQueryable<T>`.

No exemplo a seguir, `numbers` é a fonte de dados e `num` é a variável de intervalo.

Observe que ambas as variáveis são fortemente tipadas, mesmo com o uso da palavra-chave `var`.

C#

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
                      where num < 5
                      select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
} // Output: 4 1 3 2 0
```

## A variável de intervalo

O compilador infere que o tipo da variável de intervalo quando a fonte de dados implementa `IEnumerable<T>`. Por exemplo, se a fonte tem um tipo de `IEnumerable<Customer>`, então, a variável de intervalo será inferida como `Customer`. O tipo deve ser especificado explicitamente somente quando a fonte for um tipo `IEnumerable` não genérico, como `ArrayList`. Para obter mais informações, consulte [Como consultar um ArrayList com LINQ](#).

No exemplo anterior, `num` é inferido como do tipo `int`. Como a variável de intervalo é fortemente tipada, é possível chamar métodos nela ou usá-la em outras operações. Por exemplo, em vez de gravar `select num`, grave `select num.ToString()` para fazer com que a expressão de consulta retorne uma sequência de cadeias de caracteres em vez de números inteiros. Também é possível gravar `select num + 10` para fazer com que a expressão retorne a sequência 14, 11, 13, 12, 10. Para obter mais informações, consulte [cláusula select](#).

A variável de intervalo é como uma variável de iteração em uma instrução `foreach`, com a exceção de uma diferença muito importante: na verdade, uma variável de intervalo nunca armazena dados da fonte. É apenas uma conveniência sintática que habilita a consulta a descrever o que ocorrerá quando ela for executada. Para obter mais informações, consulte [Introdução a Consultas de LINQ \(C#\)](#).

## Cláusulas from compostas

Em alguns casos, cada elemento na sequência de origem pode ser uma sequência ou conter uma sequência. Por exemplo, a fonte de dados pode ser um `IEnumerable<Student>` em que cada objeto do aluno na sequência contenha uma lista de resultados de avaliações. Para acessar a lista interna dentro de cada elemento `Student`, use cláusulas compostas `from`. A técnica é parecida com o uso de instruções `foreach` aninhadas. É possível adicionar cláusulas `where` ou `orderby` a qualquer cláusula `from` para filtrar os resultados. O exemplo a seguir mostra uma sequência de objetos `Student`, em cada um contém uma `List` interna de inteiros que representam de resultados de avaliações. Para acessar a lista interna, use uma cláusula composta `from`. É possível inserir cláusulas entre as duas cláusulas `from`, se necessário.

C#

```
class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public string LastName { get; set; }
```

```

    public List<int> Scores {get; set;}
}

static void Main()
{
    // Use a collection initializer to create the data source. Note that
    // each element in the list contains an inner sequence of scores.
    List<Student> students = new List<Student>
    {
        new Student {LastName="Omelchenko", Scores= new List<int> {97,
72, 81, 60}},
        new Student {LastName="O'Donnell", Scores= new List<int> {75, 84,
91, 39}},
        new Student {LastName="Mortensen", Scores= new List<int> {88, 94,
65, 85}},
        new Student {LastName="Garcia", Scores= new List<int> {97, 89,
85, 82}},
        new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91,
70}}
    };

    // Use a compound from to access the inner sequence within each
    // element.
    // Note the similarity to a nested foreach statement.
    var scoreQuery = from student in students
                      from score in student.Scores
                      where score > 90
                      select new { Last = student.LastName, score };

    // Execute the queries.
    Console.WriteLine("scoreQuery:");
    // Rest the mouse pointer on scoreQuery in the following line to
    // see its type. The type is IEnumerable<'a>, where 'a is an
    // anonymous type defined as new {string Last, int score}. That is,
    // each instance of this anonymous type has two members, a string
    // (Last) and an int (score).
    foreach (var student in scoreQuery)
    {
        Console.WriteLine("{0} Score: {1}", student.Last,
student.score);
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
*/

```

# Usando Várias Cláusulas from para Realizar Uniões

Uma cláusula composta `from` é usada para acessar coleções internas em uma fonte de dados única. No entanto, uma consulta também pode conter várias cláusulas `from` que geram consultas complementares de fontes de dados independentes. Essa técnica habilita a execução de determinados tipos de operações de união que não são possíveis por meio da cláusula `join`.

A exemplo a seguir mostra como duas cláusulas `from` podem ser usadas para formar uma união cruzada completa de duas fontes de dados.

C#

```
class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper,
pair.lower);
        }
    }
}
```

```

    }

    Console.WriteLine("Filtered non-equijoin:");
    // Rest the mouse pointer over joinQuery2 to verify its type.
    foreach (var pair in joinQuery2)
    {
        Console.WriteLine("{0} is matched to {1}", pair.lower,
pair.upper);
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
/* Output:
   Cross join:
   A is matched to x
   A is matched to y
   A is matched to z
   B is matched to x
   B is matched to y
   B is matched to z
   C is matched to x
   C is matched to y
   C is matched to z
   Filtered non-equijoin:
   y is matched to A
   y is matched to B
   y is matched to C
   z is matched to A
   z is matched to B
   z is matched to C
*/

```

Para obter mais informações sobre as operações de união que usam várias cláusulas `from`, consulte [Executar junções externas esquerdas](#).

## Confira também

- Palavras-chave de Consulta (LINQ)
- LINQ (Consulta Integrada à Linguagem)

# Cláusula where (Referência de C#)

Artigo • 20/07/2023

A cláusula `where` é usada em uma expressão de consulta para especificar quais elementos da fonte de dados serão retornados na expressão de consulta. Aplica-se uma condição booliana (*predicate*) para cada elemento de origem (referenciado pela variável de intervalo) e retorna aqueles para os quais a condição especificada for verdadeira. Uma única expressão de consulta pode conter várias cláusulas `where` e uma única cláusula pode conter várias subexpressões de predicado.

## Exemplo 1

No exemplo a seguir, a cláusula `where` filtra todos os números, exceto aqueles que são menores que cinco. Se você remover a cláusula `where`, todos os números da fonte de dados serão retornados. A expressão `num < 5` é o predicado aplicado a cada elemento.

C#

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

## Exemplo 2

Dentro de uma única cláusula `where`, você pode especificar quantos predicados forem necessários usando os operadores `&&` e `||`. No exemplo a seguir, a consulta especifica

dois predicados para selecionar apenas os números pares que são menores que cinco.

C#

```
class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.WriteLine(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }
    // Output:
    // 4 2 0
    // 4 2 0
```

## Exemplo 3

Uma cláusula `where` pode conter um ou mais métodos que retornam valores booleanos. No exemplo a seguir, a cláusula `where` usa um método para determinar se o valor atual da variável de intervalo é par ou ímpar.

C#

```

class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

        // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}

//Output: 4 8 6 2 0

```

## Comentários

A cláusula `where` é um mecanismo de filtragem. Ela pode ser posicionada em quase qualquer lugar em uma expressão de consulta, exceto que ela não pode ser a primeira ou a última cláusula. A cláusula `where` pode aparecer antes ou depois de uma cláusula `group` dependendo se você tiver que filtrar os elementos de origem antes ou depois de eles serem agrupados.

Se um predicado especificado não for válido para os elementos na fonte de dados, o resultado será um erro em tempo de compilação. Essa é uma vantagem da verificação de tipo forte fornecida pelo LINQ.

Em tempo de compilação, a palavra-chave `where` é convertida em uma chamada para o método de operador de consulta padrão `Where`.

## Confira também

- Palavras-chave de Consulta (LINQ)
- Cláusula From
- Cláusula select
- Filtrar dados
- LINQ em C#

# Cláusula select (Referência de C#)

Artigo • 20/07/2023

Em uma expressão de consulta, a cláusula `select` especifica o tipo de valores que serão produzidos quando a consulta é executada. O resultado é baseado na avaliação de todas as cláusulas anteriores e em quaisquer expressões na cláusula `select` em si. Uma expressão de consulta deve terminar com uma cláusula `select` ou uma cláusula `group`.

O exemplo a seguir mostra uma cláusula `select` simples em uma expressão de consulta.

C#

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

O tipo da sequência produzida pela cláusula `select` determina o tipo da variável de consulta `queryHighScores`. No caso mais simples, a cláusula `select` apenas especifica a variável de intervalo. Isso faz com que a sequência retornada contenha elementos do mesmo tipo que a fonte de dados. Para obter mais informações, consulte [Relacionamentos de tipo em operações de consulta LINQ](#). No entanto, a cláusula `select` também fornece um mecanismo poderoso para transformar (ou *projetar*) dados de origem em novos tipos. Para obter mais informações, consulte [Transformações de dados com LINQ \(C#\)](#).

## Exemplo

O exemplo a seguir mostra todas as diferentes formas que uma cláusula `select` pode tomar. Em cada consulta, observe a relação entre a cláusula `select` e o tipo da variável de consulta (`studentQuery1`, `studentQuery2` e assim por diante).

C#

```
class SelectSample2
{
    // Define some classes
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
        public ContactInfo GetContactInfo(SelectSample2 app, int id)
        {
            ContactInfo cInfo =
                (from ci in app.contactList
                 where ci.ID == id
                 select ci)
                .FirstOrDefault();

            return cInfo;
        }

        public override string ToString()
        {
            return First + " " + Last + ":" + ID;
        }
    }

    public class ContactInfo
    {
        public int ID { get; set; }
        public string Email { get; set; }
        public string Phone { get; set; }
        public override string ToString() { return Email + "," + Phone; }
    }
}

public class ScoreInfo
{
    public double Average { get; set; }
    public int ID { get; set; }
}

// The primary data source
List<Student> students = new List<Student>()
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111,
Scores= new List<int>() {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores=
```

```
new List<int>() {75, 84, 91, 39}],
    new Student {First="Sven", Last="Mortensen", ID=113, Scores=
new List<int>() {88, 94, 65, 91}],
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new
List<int>() {97, 89, 85, 82}},
};

// Separate data source for contact info.
List<ContactInfo> contactList = new List<ContactInfo>()
{
    new ContactInfo {ID=111, Email="SvetlanO@Contoso.com",
Phone="206-555-0108"},
    new ContactInfo {ID=112, Email="ClaireO@Contoso.com",
Phone="206-555-0298"},
    new ContactInfo {ID=113, Email="SvenMort@Contoso.com",
Phone="206-555-1130"},
    new ContactInfo {ID=114, Email="CesarGar@Contoso.com",
Phone="206-555-0521"}
};

static void Main(string[] args)
{
    SelectSample2 app = new SelectSample2();

    // Produce a filtered sequence of unmodified Students.
    IEnumerable<Student> studentQuery1 =
        from student in app.students
        where student.ID > 111
        select student;

    Console.WriteLine("Query1: select range_variable");
    foreach (Student s in studentQuery1)
    {
        Console.WriteLine(s.ToString());
    }

    // Produce a filtered sequence of elements that contain
    // only one property of each Student.
    IEnumerable<String> studentQuery2 =
        from student in app.students
        where student.ID > 111
        select student.Last;

    Console.WriteLine("\r\n studentQuery2: select
range_variable.Property");
    foreach (string s in studentQuery2)
    {
        Console.WriteLine(s);
    }

    // Produce a filtered sequence of objects created by
    // a method call on each Student.
    IEnumerable<ContactInfo> studentQuery3 =
        from student in app.students
        where student.ID > 111
```

```
    select student.GetContactInfo(app, student.ID);

    Console.WriteLine("\r\n studentQuery3: select
range_variable.Method");
    foreach (ContactInfo ci in studentQuery3)
    {
        Console.WriteLine(ci.ToString());
    }

    // Produce a filtered sequence of ints from
    // the internal array inside each Student.
    IEnumerable<int> studentQuery4 =
        from student in app.students
        where student.ID > 111
        select student.Scores[0];

    Console.WriteLine("\r\n studentQuery4: select
range_variable[index]");
    foreach (int i in studentQuery4)
    {
        Console.WriteLine("First score = {0}", i);
    }

    // Produce a filtered sequence of doubles
    // that are the result of an expression.
    IEnumerable<double> studentQuery5 =
        from student in app.students
        where student.ID > 111
        select student.Scores[0] * 1.1;

    Console.WriteLine("\r\n studentQuery5: select expression");
    foreach (double d in studentQuery5)
    {
        Console.WriteLine("Adjusted first score = {0}", d);
    }

    // Produce a filtered sequence of doubles that are
    // the result of a method call.
    IEnumerable<double> studentQuery6 =
        from student in app.students
        where student.ID > 111
        select student.Scores.Average();

    Console.WriteLine("\r\n studentQuery6: select expression2");
    foreach (double d in studentQuery6)
    {
        Console.WriteLine("Average = {0}", d);
    }

    // Produce a filtered sequence of anonymous types
    // that contain only two properties from each Student.
    var studentQuery7 =
        from student in app.students
        where student.ID > 111
        select new { student.First, student.Last };
```

```

Console.WriteLine("\r\n studentQuery7: select new anonymous
type");
foreach (var item in studentQuery7)
{
    Console.WriteLine("{0}, {1}", item.Last, item.First);
}

// Produce a filtered sequence of named objects that contain
// a method return value and a property from each Student.
// Use named types if you need to pass the query variable
// across a method boundary.
IEnumerable<ScoreInfo> studentQuery8 =
    from student in app.students
    where student.ID > 111
    select new ScoreInfo
    {
        Average = student.Scores.Average(),
        ID = student.ID
    };

Console.WriteLine("\r\n studentQuery8: select new named type");
foreach (ScoreInfo si in studentQuery8)
{
    Console.WriteLine("ID = {0}, Average = {1}", si.ID,
si.Average);
}

// Produce a filtered sequence of students who appear on a
contact list
// and whose average is greater than 85.
IEnumerable<ContactInfo> studentQuery9 =
    from student in app.students
    where student.Scores.Average() > 85
    join ci in app.contactList on student.ID equals ci.ID
    select ci;

Console.WriteLine("\r\n studentQuery9: select result of join
clause");
foreach (ContactInfo ci in studentQuery9)
{
    Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
}

// Keep the console window open in debug mode
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

/*
 * Output
Query1: select range_variable
Claire O'Donnell:112
Sven Mortensen:113
Cesar Garcia:114

```

```

studentQuery2: select range_variable.Property
O'Donnell
Mortensen
Garcia

studentQuery3: select range_variable.Method
Claire0@Contoso.com,206-555-0298
SvenMort@Contoso.com,206-555-1130
CesarGar@Contoso.com,206-555-0521

studentQuery4: select range_variable[index]
First score = 75
First score = 88
First score = 97

studentQuery5: select expression
Adjusted first score = 82.5
Adjusted first score = 96.8
Adjusted first score = 106.7

studentQuery6: select expression2
Average = 72.25
Average = 84.5
Average = 88.25

studentQuery7: select new anonymous type
O'Donnell, Claire
Mortensen, Sven
Garcia, Cesar

studentQuery8: select new named type
ID = 112, Average = 72.25
ID = 113, Average = 84.5
ID = 114, Average = 88.25

studentQuery9: select result of join clause
ID = 114, Email = CesarGar@Contoso.com
*/

```

Conforme mostrado em `studentQuery8` no exemplo anterior, às vezes, convém que os elementos da sequência retornada contenham apenas um subconjunto das propriedades dos elementos de origem. Mantendo a sequência retornada a menor possível, é possível reduzir os requisitos de memória e aumentar a velocidade da execução da consulta. É possível fazer isso criando um tipo anônimo na cláusula `select` e usando um inicializador de objeto para inicializá-lo com as propriedades adequadas do elemento de origem. Para obter um exemplo de como fazer isso, consulte [Inicializadores de objeto e coleção](#).

## Comentários

No tempo de compilação, a cláusula `select` é convertida em uma chamada de método para o operador de consulta padrão `Select`.

## Confira também

- [Referência de C#](#)
- [Palavras-chave de Consulta \(LINQ\)](#)
- [Cláusula From](#)
- [partial \(método\) \(Referência do C#\)](#)
- [Tipos anônimos](#)
- [LINQ em C#](#)

# Cláusula group (Referência de C#)

Artigo • 07/04/2023

A cláusula `group` retorna uma sequência de objetos `IGrouping< TKey, TElement >` que contém zero ou mais itens que correspondem ao valor de chave do grupo. Por exemplo, é possível agrupar uma sequência de cadeias de caracteres de acordo com a primeira letra de cada cadeia de caracteres. Nesse caso, a primeira letra é a chave, tem um tipo `char` e é armazenada na propriedade `Key` de cada objeto `IGrouping< TKey, TElement >`. O compilador infere o tipo da chave.

É possível finalizar uma expressão de consulta com uma cláusula `group`, conforme mostrado no exemplo a seguir:

C#

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

Caso deseje executar mais operações de consulta em cada grupo, é possível especificar um identificador temporário usando a palavra-chave contextual `into`. Ao usar `into`, é necessário continuar a consulta e, em algum momento, finalizá-la com uma instrução `select` ou outra cláusula `group`, conforme mostrado no trecho a seguir:

C#

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

Exemplos mais completos do uso de `group` com e sem `into` serão apresentados na seção Exemplo deste artigo.

## Enumerando os resultados de uma consulta de grupo

Como os objetos `IGrouping<TKey,TElement>` produzidos por uma consulta `group` são essencialmente uma lista de listas, você deve usar um loop aninhado `foreach` para acessar os itens em cada grupo. O loop externo itera nas chaves de grupo e o loop interno itera em cada item do grupo em si. Um grupo pode ter uma chave sem nenhum elemento. Este é o loop `foreach` que executa a consulta nos exemplos de código anteriores:

C#

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    // Explicit type for student could also be used here.
    foreach (var student in studentGroup)
    {
        Console.WriteLine(" {0}, {1}", student.Last, student.First);
    }
}
```

## Tipos de chave

As chaves de grupo podem ser de qualquer tipo, como uma cadeia de caracteres, um tipo numérico interno, um tipo nomeado definido pelo usuário ou um tipo anônimo.

### Agrupar por cadeia de caracteres

Os exemplos de código anteriores usaram um `char`. Em vez disso, uma chave de cadeia de caracteres pode facilmente ter sido especificada, por exemplo, o sobrenome completo:

C#

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

### Agrupar por bool

O exemplo a seguir mostra o uso de um valor booleano para uma chave dividir os resultados em dois grupos. Observe que o valor é produzido por uma subexpressão na cláusula `group`.

C#

```
class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
        // each element
        //     in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
new List<int> {97, 72, 81, 60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores=
new List<int> {75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores= new
List<int> {99, 89, 91, 95}},
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new
List<int> {72, 81, 65, 84}},
            new Student {First="Debra", Last="Garcia", ID=115, Scores= new
List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {

```

```

        Console.WriteLine(studentGroup.Key == true ? "High averages" :
    "Low averages");
        foreach (var student in studentGroup)
    {
        Console.WriteLine(" {0}, {1}:{2}", student.Last,
student.First, student.Scores.Average());
    }
}

// Keep the console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}
/* Output:
 Low averages
 Omelchenko, Svetlana:77.5
 O'Donnell, Claire:72.25
 Garcia, Cesar:75.5
 High averages
 Mortensen, Sven:93.5
 Garcia, Debra:88.25
*/

```

## Agrupar por alcance numérico

O próximo exemplo usa uma expressão para criar chaves de grupo numéricas que representam um intervalo de percentil. Observe o uso de `let` como um local conveniente para armazenar um resultado de chamada de método, para que não seja necessário chamar o método duas vezes na cláusula `group`. Para obter mais informações sobre como usar métodos com segurança em expressões de consulta, confira [Como tratar exceções nas expressões de consulta](#).

C#

```

class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
        each element
    }
}

```

```

        // in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
new List<int> {97, 72, 81, 60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores=
new List<int> {75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores= new
List<int> {99, 89, 91, 95}},
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new
List<int> {72, 81, 65, 84}},
            new Student {First="Debra", Last="Garcia", ID=115, Scores= new
List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    // This method groups students into percentile ranges based on their
    // grade average. The Average method returns a double, so to produce a
whole
    // number it is necessary to cast to int before dividing by 10.
    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg / 10) into g
            orderby g.Key
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine("Students with an average between {0} and
{1}", temp, temp + 10);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("  {0}, {1}:{2}", student.Last,
student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Students with an average between 70 and 80

```

```
Omelchenko, Svetlana:77.5
O'Donnell, Claire:72.25
Garcia, Cesar:75.5
Students with an average between 80 and 90
    Garcia, Debra:88.25
Students with an average between 90 and 100
    Mortensen, Sven:93.5
*/
```

## Agrupando por chaves compostas

Use uma chave composta para agrupar elementos de acordo com mais de uma chave. Uma chave composta é criada usando um tipo anônimo ou nomeado para armazenar o elemento-chave. No exemplo a seguir, suponha que uma classe `Person` foi declarada com membros nomeados `surname` e `city`. A cláusula `group` faz com que um grupo separado seja criado para cada conjunto de pessoas com o mesmo sobrenome e a mesma cidade.

C#

```
group person by new {name = person.surname, city = person.city};
```

Use um tipo nomeado se for necessário passar a variável de consulta para outro método. Crie uma classe especial usando as propriedades autoimplementadas das chaves e, em seguida, substitua os métodos `Equals` e `GetHashCode`. Também é possível usar um struct; nesse caso, não é exatamente necessário substituir esses métodos. Para obter mais informações, confira [Como implementar uma classe leve com propriedades autoimplementadas](#) e [Como consultar arquivos duplicados em uma árvore de diretório](#). O último artigo apresenta um exemplo de código que demonstra como usar uma chave composta com um tipo nomeado.

## Exemplo 1

O exemplo a seguir mostra a norma padrão para ordenar dados de origem em grupos quando nenhuma lógica de consulta adicional for aplicada aos grupos. Isso é chamado de “agrupamento sem uma continuação”. Os elementos em uma matriz de cadeias de caracteres são agrupados de acordo com a primeira letra. O resultado da consulta é um tipo `IGrouping< TKey, TElement >` que contém uma propriedade `Key` pública do tipo `char` e uma coleção `IEnumerable< T >` que contém cada item no agrupamento.

O resultado de uma cláusula `group` é uma sequência de sequências. Portanto, para acessar os elementos individuais dentro de cada grupo retornado, use um loop

aninhado `foreach` dentro do loop que itera as chaves de grupo, conforme mostrado no exemplo a seguir.

C#

```
class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "blueberry", "chimpanzee", "abacus", "banana",
"apple", "cheese" };

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0}':",
wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   Words that start with the letter 'b':
   blueberry
   banana
   Words that start with the letter 'c':
   chimpanzee
   cheese
   Words that start with the letter 'a':
   abacus
   apple
*/
```

## Exemplo 2

Este exemplo mostra como executar a lógica adicional nos grupos depois criá-los, usando uma *continuação* com `into`. Para obter mais informações, consulte [into](#). O

exemplo a seguir consulta cada grupo para selecionar apenas aqueles cujo valor da chave é uma vogal.

```
C#  
  
class GroupClauseExample2  
{  
    static void Main()  
    {  
        // Create the data source.  
        string[] words2 = { "blueberry", "chimpanzee", "abacus", "banana",  
"apple", "cheese", "elephant", "umbrella", "anteater" };  
  
        // Create the query.  
        var wordGroups2 =  
            from w in words2  
            group w by w[0] into grps  
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'  
                   || grps.Key == 'o' || grps.Key == 'u')  
            select grps;  
  
        // Execute the query.  
        foreach (var wordGroup in wordGroups2)  
        {  
            Console.WriteLine("Groups that start with a vowel: {0}",  
wordGroup.Key);  
            foreach (var word in wordGroup)  
            {  
                Console.WriteLine("    {0}", word);  
            }  
        }  
  
        // Keep the console window open in debug mode  
        Console.WriteLine("Press any key to exit.");  
        Console.ReadKey();  
    }  
}  
/* Output:  
 Groups that start with a vowel: a  
     abacus  
     apple  
     anteater  
 Groups that start with a vowel: e  
     elephant  
 Groups that start with a vowel: u  
     umbrella  
 */
```

## Comentários

No tempo de compilação, as cláusulas `group` são convertidas em chamadas para o método [GroupBy](#).

## Confira também

- [IGrouping< TKey, TElement >](#)
- [GroupBy](#)
- [ThenBy](#)
- [ThenByDescending](#)
- [Palavras-chave de consulta](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Criar um grupo aninhado](#)
- [Agrupar resultados de consultas](#)
- [Executar uma subconsulta em uma operação de agrupamento](#)

# into (Referência de C#)

Artigo • 07/04/2023

A palavra-chave contextual `into` pode ser usada para criar um identificador temporário para armazenar os resultados de uma cláusula `group`, `join` ou `select` em um novo identificador. Esse identificador por si só pode ser um gerador de comandos de consulta adicionais. Quando usado em uma cláusula `group` ou `select`, o uso do novo identificador é, às vezes, conhecido como uma *continuação*.

## Exemplo

O exemplo a seguir mostra o uso da palavra-chave `into` para habilitar um identificador temporário `fruitGroup` que tem um tipo inferido de `IGrouping`. Usando o identificador, é possível invocar o método `Count` em cada grupo e selecionar apenas os grupos que contêm duas ou mais palavras.

C#

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bananas",
"apricots" };

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words =
fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine(" {0} has {1} elements.", item.FirstLetter,
item.Words);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

```
}
```

```
/* Output:
```

```
 a has 2 elements.
```

```
 b has 2 elements.
```

```
 */
```

O uso de `into` em uma cláusula `group` só é necessário quando você deseja realizar operações de consulta adicionais em cada grupo. Para obter mais informações, consulte [Cláusula group](#).

Para obter um exemplo do uso de `into` em uma cláusula `join`, consulte [cláusula join](#).

## Confira também

- [Palavras-chave de Consulta \(LINQ\)](#)
- [LINQ em C#](#)
- [Cláusula group](#)

# Cláusula orderby (Referência de C#)

Artigo • 07/04/2023

Em uma expressão de consulta, a cláusula `orderby` faz com que a sequência ou subsequência (grupo) retornada seja classificada em ordem crescente ou decrescente. Várias chaves podem ser especificadas para executar uma ou mais operações de classificação secundárias. A classificação é executada pelo comparador padrão para o tipo do elemento. A ordem de classificação padrão é crescente. Também é possível especificar um comparador personalizado. No entanto, está disponível somente por meio da sintaxe baseada em método. Para obter mais informações, consulte [Classificando dados](#).

## Exemplo 1

No exemplo a seguir, a primeira consulta classifica as palavras em ordem alfabética começando em A e a segunda consulta classifica as mesmas palavras em ordem decrescente. (A palavra-chave `ascending` é o valor de classificação padrão e pode ser omitida.)

C#

```
class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit // "ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }
    }
}
```

```

// Execute the query.
Console.WriteLine(Environment.NewLine + "Descending:");
foreach (string s in sortDescendingQuery)
{
    Console.WriteLine(s);
}

// Keep the console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/

```

## Exemplo 2

O exemplo a seguir executa uma classificação primária pelos sobrenomes dos alunos e, em seguida, uma classificação secundária pelos seus nomes.

```

C#

class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
        // each element
        // in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111},
            new Student {First="Claire", Last="O'Donnell", ID=112},
        };
    }
}

```

```

        new Student {First="Sven", Last="Mortensen", ID=113},
        new Student {First="Cesar", Last="Garcia", ID=114},
        new Student {First="Debra", Last="Garcia", ID=115}
    };

    return students;
}
static void Main(string[] args)
{
    // Create the data source.
    List<Student> students = GetStudents();

    // Create the query.
    IEnumerable<Student> sortedStudents =
        from student in students
        orderby student.Last ascending, student.First ascending
        select student;

    // Execute the query.
    Console.WriteLine("sortedStudents:");
    foreach (Student student in sortedStudents)
        Console.WriteLine(student.Last + " " + student.First);

    // Now create groups and sort the groups. The query first sorts the
    names
    // of all students so that they will be in alphabetical order after
    they are
    // grouped. The second orderby sorts the group keys in alpha order.
    var sortedGroups =
        from student in students
        orderby student.Last, student.First
        group student by student.Last[0] into newGroup
        orderby newGroup.Key
        select newGroup;

    // Execute the query.
    Console.WriteLine(Environment.NewLine + "sortedGroups:");
    foreach (var studentGroup in sortedGroups)
    {
        Console.WriteLine(studentGroup.Key);
        foreach (var student in studentGroup)
        {
            Console.WriteLine("  {0}, {1}", student.Last,
student.First);
        }
    }

    // Keep the console window open in debug mode
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
/* Output:
sortedStudents:
Garcia Cesar

```

```
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
    Garcia, Cesar
    Garcia, Debra
M
    Mortensen, Sven
O
    O'Donnell, Claire
    Omelchenko, Svetlana
*/
```

## Comentários

Em tempo de compilação, a cláusula `orderby` é convertida em uma chamada para o método `OrderBy`. Várias chaves na cláusula `orderby` são traduzidas para chamadas de método `ThenBy`.

## Confira também

- [Referência de C#](#)
- [Palavras-chave de Consulta \(LINQ\)](#)
- [LINQ em C#](#)
- [Cláusula group](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)

# Cláusula join (Referência de C#)

Artigo • 07/04/2023

A cláusula `join` é útil para associar elementos de sequências de origem diferentes que não têm nenhuma relação direta no modelo de objeto. O único requisito é que os elementos em cada fonte compartilhem algum valor que possa ser comparado pela igualdade. Por exemplo, um distribuidor de alimentos pode ter uma lista de fornecedores de um determinado produto e uma lista de compradores. Uma cláusula `join` pode ser usada, por exemplo, para criar uma lista de fornecedores e compradores daquele produto, que estejam na mesma região especificada.

Uma cláusula `join` recebe duas sequências de origem como entrada. Os elementos em cada sequência devem ser ou conter uma propriedade que possa ser comparada com uma propriedade correspondente na outra sequência. A cláusula `join` compara a igualdade das chaves especificadas, usando a palavra-chave especial `equals`. Todas as junções realizadas pela cláusula `join` são junções por igualdade. A forma da saída de uma cláusula `join` depende do tipo específico de junção que você está realizando. A seguir estão os três tipos de junção mais comuns:

- Junção interna
- Junção de grupo
- Junção externa esquerda

## Junção interna

O exemplo a seguir mostra uma junção por igualdade interna simples. Essa consulta produz uma sequência simples de pares "nome de produto / categoria". A mesma cadeia de caracteres de categoria aparecerá em vários elementos. Se um elemento de `categories` não tiver `products` correspondente, essa categoria não aparecerá nos resultados.

C#

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
//produces flat sequence
```

Para obter mais informações, consulte [Executar junções internas](#).

## Junção de grupo

Uma cláusula `join` com um expressão `into` é chamada de junção de grupo.

C#

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

Uma junção de grupo produz uma sequência de resultados hierárquicos, que associa os elementos na sequência de origem à esquerda com um ou mais elementos correspondentes na sequência de origem do lado direito. Uma junção de grupo não tem nenhum equivalente em termos relacionais. Ela é, essencialmente, uma sequência de matrizes de objetos.

Se nenhum elemento da sequência de origem à direita que corresponda a um elemento na origem à esquerda for encontrado, a cláusula `join` produzirá uma matriz vazia para aquele item. Portanto, a junção de grupo é, basicamente, uma junção por igualdade interna, exceto pelo fato de que a sequência de resultado é organizada em grupos.

É só selecionar os resultados de uma junção de grupo e você poderá acessar os itens, mas você não poderá identificar a chave na qual eles correspondem. Portanto, geralmente há maior utilidade em selecionar os resultados da junção de grupo em um novo tipo que também tenha o nome da chave, conforme mostrado no exemplo anterior.

Além disso, é claro que você pode usar o resultado de uma junção de grupo como o gerador de outra subconsulta:

C#

```
var innerGroupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    from prod2 in prodGroup
    where prod2.UnitPrice > 2.50M
    select prod2;
```

Para obter mais informações, consulte [Executar junções agrupadas](#).

## Junção externa esquerda

Em uma junção externa esquerda, todos os elementos na sequência de origem à esquerda são retornados, mesmo que não haja elementos correspondentes na sequência à direita. Para executar uma junção externa esquerda no LINQ, use o método `DefaultIfEmpty` em combinação com uma junção de grupo para especificar um elemento padrão do lado direito, que será produzido caso um elemento do lado esquerdo não tenha correspondência. Você pode usar `null` como o valor padrão para qualquer tipo de referência ou pode especificar um tipo padrão definido pelo usuário. No exemplo a seguir, é mostrado um tipo padrão definido pelo usuário:

C#

```
var leftOuterJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
    prodGroup
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty,
    CategoryID = 0 })
    select new { CatName = category.Name, ProdName = item.Name };
```

Para obter mais informações, consulte [Executar junções externas esquerdas](#).

## O operador `equals`

Uma cláusula `join` realiza uma junção por igualdade. Em outras palavras, você só pode basear correspondências na igualdade de duas chaves. Não há suporte para outros tipos de comparações, como "maior que" ou "não é igual a". Para se certificar de que todas as junções são junções por igualdade, a cláusula `join` usa a palavra-chave `equals` em vez do operador `==`. A palavra-chave `equals` só pode ser usada em uma cláusula `join` e ela se difere do operador `==` em alguns aspectos importantes. Ao comparar cadeias de caracteres, `equals` tem uma sobrecarga para comparar por valor e o operador `==` usa a igualdade de referência. Quando ambos os lados da comparação tiverem variáveis de cadeia de caracteres idênticas, `equals` e `==` alcançarão o mesmo resultado: `true`. Isso ocorre porque, quando um programa declara duas ou mais variáveis de cadeia de caracteres equivalentes, o compilador armazena todas elas no mesmo local. Consulte [Igualdade de referência e a centralização de cadeia de caracteres](#) para obter mais informações. Outra diferença importante é a comparação nula: `null equals null` é avaliada como `false` com o operador `equals`, em vez do operador `==`, que a

avalia como true. Por último, o comportamento de escopo é diferente: com `equals`, a chave esquerda consome a sequência de origem externa e a chave direita consome a origem interna. A origem externa somente está no escopo no lado esquerdo de `equals` e a sequência de origem interna somente está no escopo no lado direito.

## Junções por não igualdade

Você pode realizar junções por não igualdade, uniões cruzadas e outras operações de junção personalizadas, usando várias cláusulas `from` para introduzir novas sequências de maneira independente em uma consulta. Para obter mais informações, consulte [Executar operações de junção personalizadas](#).

## Junções em coleções de objetos versus tabelas relacionais

Em uma expressão de consulta integrada à linguagem, as operações de junção são realizadas em coleções de objetos. As coleções de objetos não podem ser "unidas" exatamente da mesma forma que duas tabelas relacionais. Em LINQ as cláusulas `join` explícitas só são necessárias quando duas sequências de origem não estão ligadas por nenhuma relação. Ao trabalhar com LINQ to SQL, as tabelas de chave estrangeira são representadas no modelo de objeto como propriedades da tabela primária. Por exemplo, no banco de dados Northwind, a tabela Cliente tem uma relação de chave estrangeira com a tabela Pedidos. Quando você mapear as tabelas para o modelo de objeto, a classe Cliente terá uma propriedade de Pedidos contendo a coleção de Pedidos associados a esse Cliente. Na verdade, a junção já foi feita para você.

Para obter mais informações sobre como fazer consultas entre tabelas relacionadas no contexto do LINQ to SQL, veja [Como: mapear relações de banco de dados](#).

## Chaves compostas

Você pode testar a igualdade de vários valores, usando uma chave de composição. Para obter mais informações, consulte [Unir usando chaves compostas](#). As chaves compostas também podem ser usadas em uma cláusula `group`.

## Exemplo

O exemplo a seguir compara os resultados de uma junção interna, uma junção de grupo e uma junção externa esquerda nas mesmas fontes de dados, usando as mesmas chaves

correspondentes. Foi adicionado algum código extra nesses exemplos a fim de deixar os resultados na tela do console mais claros.

```
C#
```

```
class JoinDemonstration
{
    #region Data

        class Product
        {
            public string Name { get; set; }
            public int CategoryID { get; set; }
        }

        class Category
        {
            public string Name { get; set; }
            public int ID { get; set; }
        }

        // Specify the first data source.
        List<Category> categories = new List<Category>()
        {
            new Category {Name="Beverages", ID=001},
            new Category {Name="Condiments", ID=002},
            new Category {Name="Vegetables", ID=003},
            new Category {Name="Grains", ID=004},
            new Category {Name="Fruit", ID=005}
        };

        // Specify the second data source.
        List<Product> products = new List<Product>()
        {
            new Product {Name="Cola", CategoryID=001},
            new Product {Name="Tea", CategoryID=001},
            new Product {Name="Mustard", CategoryID=002},
            new Product {Name="Pickles", CategoryID=002},
            new Product {Name="Carrots", CategoryID=003},
            new Product {Name="Bok Choy", CategoryID=003},
            new Product {Name="Peaches", CategoryID=005},
            new Product {Name="Melons", CategoryID=005},
        };
    #endregion

    static void Main(string[] args)
    {
        JoinDemonstration app = new JoinDemonstration();

        app.InnerJoin();
        app.GroupJoin();
        app.GroupInnerJoin();
        app.GroupJoin3();
        app.LeftOuterJoin();
```

```

    app.LeftOuterJoin2();

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

void InnerJoin()
{
    // Create the query that selects
    // a property from each element.
    var innerJoinQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID
        select new { Category = category.ID, Product = prod.Name };

    Console.WriteLine("InnerJoin:");
    // Execute the query. Access results
    // with a simple foreach statement.
    foreach (var item in innerJoinQuery)
    {
        Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
    }
    Console.WriteLine("InnerJoin: {0} items in 1 group.", 
innerJoinQuery.Count());
    Console.WriteLine(System.Environment.NewLine);
}

void GroupJoin()
{
    // This is a demonstration query to show the output
    // of a "raw" group join. A more typical group join
    // is shown in the GroupInnerJoin method.
    var groupJoinQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into
prodGroup
        select prodGroup;

    // Store the count of total items (for demonstration only).
    int totalItems = 0;

    Console.WriteLine("Simple GroupJoin:");

    // A nested foreach statement is required to access group items.
    foreach (var prodGrouping in groupJoinQuery)
    {
        Console.WriteLine("Group:");
        foreach (var item in prodGrouping)
        {
            totalItems++;
            Console.WriteLine("    {0,-10}{1}", item.Name,
item.CategoryID);
        }
    }
}

```

```

        Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed
groups", totalItems, groupJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupInnerJoin()
    {
        var groupJoinQuery2 =
            from category in categories
            orderby category.ID
            join prod in products on category.ID equals prod.CategoryID into
prodGroup
            select new
            {
                Category = category.Name,
                Products = from prod2 in prodGroup
                            orderby prod2.Name
                            select prod2
            };
        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupInnerJoin:");
        foreach (var productGroup in groupJoinQuery2)
        {
            Console.WriteLine(productGroup.Category);
            foreach (var prodItem in productGroup.Products)
            {
                totalItems++;
                Console.WriteLine(" {0,-10} {1}", prodItem.Name,
prodItem.CategoryID);
            }
        }
        Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups",
totalItems, groupJoinQuery2.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin3()
    {

        var groupJoinQuery3 =
            from category in categories
            join product in products on category.ID equals
product.CategoryID into prodGroup
            from prod in prodGroup
            orderby prod.CategoryID
            select new { Category = prod.CategoryID, ProductName = prod.Name
};

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupJoin3:");
    }
}

```

```

        foreach (var item in groupJoinQuery3)
        {
            totalItems++;
            Console.WriteLine("    {0}:{1}", item.ProductName,
item.Category);
        }

        Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems);
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin()
    {
        // Create the query.
        var leftOuterQuery =
            from category in categories
            join prod in products on category.ID equals prod.CategoryID into
prodGroup
            select prodGroup.DefaultIfEmpty(new Product() { Name =
"Nothing!", CategoryID = category.ID });

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Left Outer Join:");

        // A nested foreach statement is required to access group items
        foreach (var prodGrouping in leftOuterQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("    {0,-10}{1}", item.Name,
item.CategoryID);
            }
        }
        Console.WriteLine("LeftOuterJoin: {0} items in {1} groups",
totalItems, leftOuterQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin2()
    {
        // Create the query.
        var leftOuterQuery2 =
            from category in categories
            join prod in products on category.ID equals prod.CategoryID into
prodGroup
            from item in prodGroup.DefaultIfEmpty()
            select new { Name = item == null ? "Nothing!" : item.Name,
CategoryID = category.ID };

        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group",
leftOuterQuery2.Count());
    }
}

```

```

    // Store the count of total items
    int totalItems = 0;

    Console.WriteLine("Left Outer Join 2:");

    // Groups have been flattened.
    foreach (var item in leftOuterQuery2)
    {
        totalItems++;
        Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
    }
    Console.WriteLine("LeftOuterJoin2: {0} items in 1 group",
totalItems);
}
/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.

Unshaped GroupJoin:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups

GroupInnerJoin:
Beverages
    Cola      1
    Tea       1
Condiments
    Mustard   2
    Pickles   2
Vegetables
    Bok Choy  3

```

```
    Carrots      3
Grains
Fruit
    Melons      5
    Peaches     5
GroupInnerJoin: 8 items in 5 named groups
```

```
GroupJoin3:
    Cola:1
    Tea:1
    Mustard:2
    Pickles:2
    Carrots:3
    Bok Choy:3
    Peaches:5
    Melons:5
GroupJoin3: 8 items in 1 group
```

```
Left Outer Join:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
    Nothing!  4
Group:
    Peaches   5
    Melons    5
LeftOuterJoin: 9 items in 5 groups
```

```
LeftOuterJoin2: 9 items in 1 group
Left Outer Join 2:
    Cola      1
    Tea       1
    Mustard   2
    Pickles   2
    Carrots   3
    Bok Choy  3
    Nothing!  4
    Peaches   5
    Melons    5
LeftOuterJoin2: 9 items in 1 group
Press any key to exit.
*/
```

# Comentários

Uma cláusula `join` que não é seguida por `into` é convertida em uma chamada de método `Join`. Uma cláusula `join` que é seguida por `into` é convertida em uma chamada de método `GroupJoin`.

## Confira também

- [Palavras-chave de Consulta \(LINQ\)](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Operações join](#)
- [Cláusula group](#)
- [Executar junções externas esquerdas](#)
- [Executar junções internas](#)
- [Executar junções agrupadas](#)
- [Ordenar os resultados de uma cláusula join](#)
- [Unir usando chaves compostas](#)
- [Sistemas de banco de dados compatíveis para Visual Studio](#)

# Cláusula let (Referência de C#)

Artigo • 07/04/2023

Em uma expressão de consulta, às vezes é útil armazenar o resultado de uma subexpressão para usá-lo em cláusulas subsequentes. É possível fazer isso com a palavra-chave `let`, que cria uma nova variável de intervalo e a inicializa com o resultado da expressão fornecida. Depois de inicializado com um valor, a variável de intervalo não pode ser usada para armazenar outro valor. No entanto, se a variável de intervalo mantiver um tipo passível de consulta, ela poderá ser consultada.

## Exemplo

No exemplo a seguir, `let` é usado de duas maneiras:

1. Para criar um tipo enumerável que pode ser pesquisado por si só.
2. Para permitir que a consulta chame `ToLower` apenas uma vez na variável de intervalo `word`. Sem usar `let`, seria necessário chamar `ToLower` em cada predicado na cláusula `where`.

```
C#  
  
class LetSample1  
{  
    static void Main()  
    {  
        string[] strings =  
        {  
            "A penny saved is a penny earned.",  
            "The early bird catches the worm.",  
            "The pen is mightier than the sword."  
        };  
  
        // Split the sentence into an array of words  
        // and select those whose first letter is a vowel.  
        var earlyBirdQuery =  
            from sentence in strings  
            let words = sentence.Split(' ')  
            from word in words  
            let w = word.ToLower()  
            where w[0] == 'a' || w[0] == 'e'  
                || w[0] == 'i' || w[0] == 'o'  
                || w[0] == 'u'  
            select word;  
  
        // Execute the query.
```

```
foreach (var v in earlyBirdQuery)
{
    Console.WriteLine("\"{0}\" starts with a vowel", v);
}

// Keep the console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

/* Output:
   "A" starts with a vowel
   "is" starts with a vowel
   "a" starts with a vowel
   "earned." starts with a vowel
   "early" starts with a vowel
   "is" starts with a vowel
*/
```

## Confira também

- [Referência de C#](#)
- [Palavras-chave de Consulta \(LINQ\)](#)
- [LINQ em C#](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Tratar exceções em expressões de consulta](#)

# ascending (Referência de C#)

Artigo • 07/04/2023

A palavra-chave contextual `ascending` é usada na [cláusula orderby](#) em expressões de consulta para especificar que a ordem de classificação é do menor para o maior. Como `ascending` é a ordem de classificação padrão, não é necessário especificá-la.

## Exemplo

O exemplo a seguir mostra o uso de `ascending` em uma [cláusula orderby](#).

C#

```
IEnumerable<string> sortAscendingQuery =  
    from vegetable in vegetables  
    orderby vegetable ascending  
    select vegetable;
```

## Confira também

- [Referência de C#](#)
- [LINQ em C#](#)
- [descending](#)

# descending (Referência de C#)

Artigo • 07/04/2023

A palavra-chave contextual `descending` é usada na cláusula `orderby` em expressões de consulta para especificar que a ordem de classificação é do maior para o menor.

## Exemplo

O exemplo a seguir mostra o uso de `descending` em uma cláusula `orderby`.

C#

```
IEnumerable<string> sortDescendingQuery =  
    from vegetable in vegetables  
    orderby vegetable descending  
    select vegetable;
```

## Confira também

- [Referência de C#](#)
- [LINQ em C#](#)
- [ascending](#)

# on (Referência de C#)

Artigo • 07/04/2023

A palavra-chave contextual `on` é usada na [cláusula `join`](#) de uma expressão de consulta a fim de especificar a condição de união.

## Exemplo

O exemplo a seguir mostra o uso de `on` em uma cláusula `join`.

C#

```
var innerJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID  
    select new { ProductName = prod.Name, Category = category.Name };
```

## Confira também

- [Referência de C#](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)

# equals (Referência de C#)

Artigo • 07/04/2023

A palavra-chave contextual `equals` é usada uma cláusula `join` em uma expressão de consulta a fim de comparar os elementos de duas sequências. Para obter mais informações, consulte [Cláusula join](#).

## Exemplo

O exemplo a seguir mostra o uso da palavra-chave `equals` em uma cláusula `join`.

C#

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

## Confira também

- [LINQ \(Consulta Integrada à Linguagem\)](#)

# by (Referência de C#)

Artigo • 07/04/2023

A palavra-chave contextual `by` é usada na cláusula `group` de uma expressão de consulta para especificar como os itens retornados devem ser agrupados. Para obter mais informações, consulte [Cláusula group](#).

## Exemplo

O exemplo a seguir mostra o uso da palavra-chave contextual `by` em uma cláusula `group` para especificar que os alunos devem ser agrupados de acordo com a primeira letra do sobrenome.

C#

```
var query = from student in students
            group student by student.LastName[0];
```

## Confira também

- [LINQ em C#](#)

# in (Referência de C#)

Artigo • 07/04/2023

A palavra-chave `in` é usada nos seguintes contextos:

- [parâmetros de tipo genérico](#) em interfaces e delegados genéricos.
- Como um [modificador de parâmetro](#), que permite passar um argumento para um método por referência, não por valor.
- Instruções [foreach](#).
- [Cláusulas from](#) em expressões de consulta LINQ.
- [Cláusulas de junção](#) em expressões de consulta LINQ.

## Confira também

- [Palavras-chave do C#](#)
- [Referência de C#](#)

# Operadores e expressões C# (referência em C#)

Artigo • 15/02/2023

O C# fornece vários operadores. Muitos deles têm suporte pelos [tipos internos](#) e permitem que você execute operações básicas com valores desses tipos. Esses operadores incluem os seguintes grupos:

- [Operadores aritméticos](#) que executam operações aritméticas com operandos numéricos
- [Operadores de comparação](#) que comparam operandos numéricos
- [Operadores lógicos booleanos](#) que executam operando lógicos com operandos `bool`
- [Operadores shift e bit a bit](#) que rrealizam operações de deslocamento ou bit a bit com operandos de tipos inteiros:
- [Operadores de igualdade](#) que verificam se os operandos são iguais ou não

Normalmente, você pode [sobreclarregar](#) esses operadores, ou seja, especificar o comportamento do operador para os operandos de um tipo definido pelo usuário.

As expressões C# mais simples são literais (por exemplo, números [inteiros](#) e [reais](#)) e nomes de variáveis. Você pode combiná-las em expressões complexas usando operadores. O operador [precedence](#) e [associativity](#), determinam a ordem na qual as operações em uma expressão são executadas. Você pode usar parênteses para alterar a ordem de avaliação imposta pela prioridade e pela associação dos operadores.

No código a seguir, exemplos de expressões estão no lado direito das atribuições:

C#

```
int a, b, c;
a = 7;
b = a;
c = b++;
b = a + b * c;
c = a >= 100 ? b : c / 10;
a = (int)Math.Sqrt(b * b + c * c);

string s = "String literal";
char l = s[s.Length - 1];

var numbers = new List<int>(new[] { 1, 2, 3 });
b = numbers.FindLast(n => n > 1);
```

Normalmente, uma expressão produz um resultado e pode ser incluída em outra expressão. Uma chamada de método `void` é um exemplo de uma expressão que não produz um resultado. Ele pode ser usado apenas como uma [instrução](#), como mostra o exemplo a seguir:

```
C#
```

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

Aqui estão alguns outros tipos de expressões que o C# fornece:

- Expressões de cadeia de caracteres [interpoladas](#) que fornecem sintaxe conveniente para criar cadeias de caracteres formatadas:

```
C#
```

```
var r = 2.3;
var message = $"The area of a circle with radius {r} is {Math.PI * r *
r:F3}.";
Console.WriteLine(message);
// Output:
// The area of a circle with radius 2.3 is 16.619.
```

- Expressões [lambda](#) que permitem criar funções anônimas:

```
C#
```

```
int[] numbers = { 2, 3, 4, 5 };
var maximumSquare = numbers.Max(x => x * x);
Console.WriteLine(maximumSquare);
// Output:
// 25
```

- Expressões de [consulta](#) que permitem que você use recursos de consulta diretamente em C#:

```
C#
```

```
var scores = new[] { 90, 97, 78, 68, 85 };
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
Console.WriteLine(string.Join(" ", highScoresQuery));
// Output:
// 97 90 85
```

Você pode usar uma [definição de corpo de expressão](#) para fornecer uma definição concisa para um método, construtor, propriedade, indexador ou finalizador.

## Precedência do operador

Em uma expressão com vários operadores, os operadores com maior precedência são avaliados antes dos operadores com menor precedência. No exemplo a seguir, a multiplicação é executada primeiro porque tem uma precedência mais alta do que a adição:

```
C#  
  
var a = 2 + 2 * 2;  
Console.WriteLine(a); // output: 6
```

Use parênteses para alterar a ordem de avaliação imposta pela precedência do operador:

```
C#  
  
var a = (2 + 2) * 2;  
Console.WriteLine(a); // output: 8
```

A tabela a seguir lista os operadores C#, começando com a precedência mais alta até a mais baixa. Os operadores em cada linha têm a mesma precedência.

Operadores	Categoria ou nome
x.y, f(x), a[i], x?.y, x?[y], x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x->y	Primário
+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, verdadeiro e falso	Unário
x..y	Intervalo
switch, com	switch e expressões with
x * y, x / y, x % y	Multiplicativo
x + y, x - y	Aditiva
x << y, x >> y	Shift
x < y, x > y, x <= y, x >= y, is, as	Teste de tipo e relacional

Operadores	Categoria ou nome
<code>x == y, x != y</code>	Igualitário
<code>x &amp; y</code>	AND lógico booliano ou AND lógico bit a bit
<code>x ^ y</code>	XOR lógico booliano ou XOR lógico bit a bit
<code>x   y</code>	OR lógico booliano ou OR lógico bit a bit
<code>x &amp;&amp; y</code>	AND condicional
<code>x    y</code>	OR condicional
<code>x ?? a</code>	Operador de coalescência nula
<code>c ? t : f</code>	Operador condicional
<code>x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &amp;= y, x  = y, x ^= y, x &lt;&lt;= y, x &gt;&gt;= y, x ??= y, =&gt;</code>	Declaração de atribuição e lambda

## Associação de operador

Quando os operadores têm a mesma precedência, a associação dos operadores determina a ordem na qual as operações são executadas:

- Os operadores *associativos esquerdos* são avaliados na ordem da esquerda para a direita. Com exceção dos [operadores de atribuição](#) e do [operador de avaliação de nulo](#), todos os operadores binários são associativos esquerdos. Por exemplo, `a + b - c` é avaliado como `(a + b) - c`.
- Os operadores *associativos direitos* são avaliados na ordem da direita para a esquerda. Os operadores de atribuição, o operador de avaliação de nulo, lambdas e o [operador condicional?:](#) são associativos direitos. Por exemplo, `x = y = z` é avaliado como `x = (y = z)`.

Use parênteses para alterar a ordem de avaliação imposta pela associação de operador:

C#

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}"); // output: a = 1, b = 6
```

# Avaliação do operando

Sem considerar a relação com a precedência e a associação de operadores, os operandos em uma expressão são avaliados da esquerda para a direita. Os exemplos a seguir demonstram a ordem em que os operadores e os operandos são avaliados:

Expression	Ordem de avaliação
a + b	a, b, +
a + b * c	a, b, c, *, +
a / b + c * d	a, b, /, c, d, *, +
a / (b + c) * d	a, b, c, +, /, d, *

Normalmente, todos os operandos do operador são avaliados. Alguns operadores avaliam os operandos condicionalmente. Ou seja, o valor do operando mais à esquerda de tal operador define se (ou quais) outros operandos devem ser avaliados. Esses operadores são os operadores lógicos **AND** (`&&`) e **OR** (`||`), o **operador de avaliação de nulo??** e `??=`, os **operadores condicionais nulos ?..e?[]** e o **operador condicional ?:.** Para obter mais informações, consulte a descrição de cada operador.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Expressões](#)
- [Operadores](#)

## Confira também

- [Referência de C#](#)
- [Sobrecarga de operador](#)
- [Árvores de expressão](#)

# Operadores aritméticos (referência do C#)

Artigo • 10/05/2023

Os seguintes operadores executam operações aritméticas com operandos de tipos numéricos:

- Operadores unários [++ \(incremento\)](#), [-- \(decremento\)](#), [+ \(adição\)](#) e [- \(subtração\)](#)
- Operadores binários [\\* \(multiplicação\)](#), [/ \(divisão\)](#), [% \(resto\)](#), [+ \(adição\)](#) e [- \(subtração\)](#)

Esses operadores são suportados por todos os tipos numéricos [integrais](#) e de [ponto flutuante](#).

No caso de tipos integrais, esses operadores (exceto os operadores `++` e `--`) são definidos para os tipos `int`, `uint`, `long` e `ulong`. Quando os operandos são de outros tipos integrais (`sbyte`, `byte`, `short`, `ushort` ou `char`), seus valores são convertidos no tipo `int`, que também é o tipo de resultado de uma operação. Quando operandos são de tipos diferentes de ponto flutuante ou integral, seus valores são convertidos para o tipo recipiente mais próximo, se esse tipo existir. Para saber mais, confira a seção [Promoções numéricas](#) da [Especificação da linguagem C#](#). Os operadores `++` e `--` são definidos para todos os tipos numéricos de ponto integral e flutuante e para o tipo `char`. O tipo de resultado de uma [expressão de atribuição composta](#) é o tipo do operando à esquerda.

## Operador de incremento `++`

O operador de incremento unário `++` incrementa seu operando em 1. O operando precisa ser uma variável, um acesso de [propriedade](#) ou um acesso de [indexador](#).

Há duas formas de suporte para o operador de incremento: o operador de incremento pós-fixado, `x++`, e o operador de incremento pré-fixado, `++x`.

## Operador de incremento pós-fixado

O resultado de `x++` é o valor de `x` *antes* da operação, como mostra o exemplo a seguir:

C#

```
int i = 3;
Console.WriteLine(i);    // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);    // output: 4
```

## Operador de incremento de prefixo

O resultado de `++x` é o valor de `x` *após* a operação, como mostra o exemplo a seguir:

C#

```
double a = 1.5;
Console.WriteLine(a);    // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);    // output: 2.5
```

## Operador de decremento --

O operador de decremento unário `--` decremente o operando em 1. O operando precisa ser uma variável, um acesso de [propriedade](#) ou um acesso de [indexador](#).

Há duas formas de suporte para o operador de decremento: o operador de decremento pós-fixado, `x--`, e o operador de decremento pré-fixado, `--x`.

## Operador de decremento pós-fixado

O resultado de `x--` é o valor de `x` *antes* da operação, como mostra o exemplo a seguir:

C#

```
int i = 3;
Console.WriteLine(i);    // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);    // output: 2
```

## Operador de decremento de prefixo

O resultado de `--x` é o valor de `x` *após* a operação, como mostra o exemplo a seguir:

C#

```
double a = 1.5;
Console.WriteLine(a);    // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);    // output: 0.5
```

## Operadores unários de adição e subtração

O operador unário `+` retorna o valor do operando. O operador unário `-` calcula a negação numérica do operando.

C#

```
Console.WriteLine(+4);      // output: 4

Console.WriteLine(-4);      // output: -4
Console.WriteLine(-(-4));  // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);       // output: -5
Console.WriteLine(b.GetType()); // output: System.Int64

Console.WriteLine(-double.NaN); // output: NaN
```

O tipo `ulong` não dá suporte ao operador unário `-`.

## Operador de multiplicação \*

O operador de multiplicação `*` calcula o produto dos operandos:

C#

```
Console.WriteLine(5 * 2);        // output: 10
Console.WriteLine(0.5 * 2.5);    // output: 1.25
Console.WriteLine(0.1m * 23.4m); // output: 2.34
```

O operador unário `*` é o [operador de indireção de ponteiro](#).

## Operador de divisão /

O operador de divisão `/` divide o operando à esquerda pelo operando à direita.

## Divisão de inteiros

Para os operandos de tipos inteiros, o resultado do operador `/` é de um tipo inteiro e igual ao quociente dos dois operandos arredondados para zero:

C#

```
Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5);  // output: 2
```

Para obter o quociente dos dois operandos como um número de ponto flutuante, use o tipo `float`, `double` ou `decimal`:

C#

```
Console.WriteLine(13 / 5.0);      // output: 2.6
int a = 13;
int b = 5;
Console.WriteLine((double)a / b); // output: 2.6
```

## Divisão de ponto flutuante

Para os tipos `float`, `double` e `decimal`, o resultado do operador `/` é o quociente dos dois operandos:

C#

```
Console.WriteLine(16.8f / 4.1f);  // output: 4.097561
Console.WriteLine(16.8d / 4.1d);  // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m);  // output: 4.0975609756097560975609756098
```

Se um dos operandos é `decimal`, outro operando não pode ser `float` nem `double`, porque nem `float` ou `double` é implicitamente conversível para `decimal`. Você deve converter explicitamente o operando `float` ou `double` para o tipo `decimal`. Para saber mais sobre as conversões entre tipos numéricos, confira a as [Conversões numéricas internas](#).

## Operador de resto %

O operador de resto `%` calcula o resto após dividir o operando à esquerda pelo à direita.

## Resto inteiro

Para os operandos de tipos inteiros, o resultado de `a % b` é o valor produzido por  $a - (a / b) * b$ . O sinal do resto diferente de zero é o mesmo do operando à esquerda, conforme mostra o seguinte exemplo:

C#

```
Console.WriteLine(5 % 4); // output: 1
Console.WriteLine(5 % -4); // output: 1
Console.WriteLine(-5 % 4); // output: -1
Console.WriteLine(-5 % -4); // output: -1
```

Use o método [Math.DivRem](#) para calcular a divisão de inteiros e os resultados do resto.

## Resto de ponto flutuante

Para os operandos `float` e `double`, o resultado de `x % y` para o `x` e o `y` finitos é o valor `z` tal que

- O sinal de `z`, se diferente de zero, é o mesmo que o sinal de `x`.
- O valor absoluto de `z` é o valor produzido por  $|x| - n * |y|$ , em que `n` é o maior inteiro possível que é inferior ou igual a  $|x| / |y|$ , e  $|x|$  e  $|y|$  são os valores absolutos de `x` e `y`, respectivamente.

### ⓘ Observação

Esse método de calcular o resto é semelhante ao usado para operandos inteiros, mas é diferente da especificação IEEE 754. Se precisar que a operação de resto esteja em conformidade com a especificação IEEE 754, use o método [Math.IEEEremainder](#).

Para saber mais sobre o comportamento do operador `%` com operandos não finitos, confira a seção [Operador de restante](#) da [especificação da linguagem C#](#).

Para os operandos `decimal`, o operador de resto `%` é equivalente ao [operador de resto](#) do tipo [System.Decimal](#).

O seguinte exemplo demonstra o comportamento do operador de resto com os operandos de ponto flutuante:

C#

```
Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m); // output: 2.8
```

## Operador de adição +

O operador de adição `+` calcula a soma dos operandos:

C#

```
Console.WriteLine(5 + 4);      // output: 9
Console.WriteLine(5 + 4.3);    // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3
```

Você também pode usar o operador `+` para a combinação de delegado e concatenação de cadeia de caracteres. Para obter mais informações, confira o artigo [Operadores + e +=](#).

## Operador de subtração -

O operador de subtração `-` subtrai o operando à direita do operando à esquerda:

C#

```
Console.WriteLine(47 - 3);     // output: 44
Console.WriteLine(5 - 4.3);    // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

Você também pode usar o operador `-` para a remoção de delegado. Para obter mais informações, confira o artigo [Operadores - e -=](#).

## Atribuição composta

Para um operador binário `op`, uma expressão de atribuição composta do formato

C#

```
x op= y
```

é equivalente a

```
C#
```

```
x = x op y
```

exceto que `x` é avaliado apenas uma vez.

O seguinte exemplo demonstra o uso da atribuição composta com operadores aritméticos:

```
C#
```

```
int a = 5;
a += 9;
Console.WriteLine(a); // output: 14

a -= 4;
Console.WriteLine(a); // output: 10

a *= 2;
Console.WriteLine(a); // output: 20

a /= 4;
Console.WriteLine(a); // output: 5

a %= 3;
Console.WriteLine(a); // output: 2
```

Devido a [promoções numéricas](#), o resultado da operação `op` pode não ser implicitamente conversível no tipo `T` de `x`. Nesse caso, se `op` for um operador predefinido e o resultado da operação for explicitamente convertido no tipo `T` de `x`, uma expressão de atribuição composta da forma `x op= y` será equivalente a `x = (T)(x op y)`, exceto que `x` será avaliada apenas uma vez. O exemplo a seguir demonstra esse comportamento:

```
C#
```

```
byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType()); // output: System.Int32
Console.WriteLine(c); // output: 300

a += b;
Console.WriteLine(a); // output: 44
```

Use também os operadores `+=` e `-=` para assinar e cancelar a assinatura de [eventos](#), respectivamente. Para obter mais informações, confira [Como assinar e cancelar a assinatura de eventos](#).

## Precedência e associatividade do operador

A seguinte lista ordena os operadores aritméticos da precedência mais alta para a mais baixa:

- Incluir um pós-fixo a operadores de incremento `x++` e decremento `x--`
- Incluir um prefixo a operadores de incremento `++x` e de decremento `--x` e operadores unários `+` e `-`
- Operadores de multiplicação `*`, `/` e `%`
- Operadores de adição `+` e `-`

Operadores aritméticos binários são associativos à esquerda. Ou seja, os operadores com o mesmo nível de precedência são avaliados da esquerda para a direita.

Use parênteses, `( )`, para alterar a ordem de avaliação imposta pela precedência e pela capacidade de associação do operador.

C#

```
Console.WriteLine(2 + 2 * 2);    // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2);    // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4
```

Para obter a lista completa de operadores C# ordenados por nível de precedência, consulte a seção [Precedência de operador](#) do artigo [Operadores C#](#).

## Estouro aritmético e divisão por zero

Quando o resultado de uma operação aritmética está fora do intervalo de valores finitos possíveis do tipo numérico envolvido, o comportamento de um operador aritmético depende do tipo dos operandos.

## Estouro aritmético de inteiros

A divisão de inteiro por zero sempre lança um [DivideByZeroException](#).

Se ocorrer um estouro aritmético inteiro, o contexto de verificação de estouro, que pode ser [checked](#) ou [unchecked](#), controlará o comportamento resultante:

- Em um contexto verificado, se o estouro acontece em uma expressão de constante, ocorre um erro em tempo de compilação. Caso contrário, quando a operação é executada em tempo de execução, uma [OverflowException](#) é gerada.
- Em um contexto não verificado, o resultado é truncado pelo descarte dos bits de ordem superior que não se ajustam ao tipo de destino.

Juntamente com as instruções [checked](#) e [unchecked](#), você pode usar os operadores [checked](#) e [unchecked](#) para controlar o contexto de verificação de estouro, no qual uma expressão é avaliada:

C#

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b)); // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

Por padrão, as operações aritméticas ocorrem em um contexto *não verificado*.

## Estouro aritmético de ponto flutuante

As operações aritméticas com os tipos [float](#) e [double](#) nunca geram uma exceção. O resultado de operações aritméticas com esses tipos pode ser um dos valores especiais que representam o infinito e um valor não é um número:

C#

```
double a = 1.0 / 0.0;
Console.WriteLine(a); // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b); // output: NaN
Console.WriteLine(double.IsNaN(b)); // output: True
```

Para os operandos do tipo `decimal`, o estouro aritmético sempre gera uma `OverflowException`. A divisão por zero sempre lança uma `DivideByZeroException`.

## Erros de arredondamento

Devido às limitações gerais de uma representação de ponto flutuante de números reais e da aritmética de ponto flutuante, os erros de arredondamento podem ocorrer em cálculos com tipos de ponto flutuante. Ou seja, o resultado produzido de uma expressão pode diferir do resultado matemático esperado. O seguinte exemplo demonstra vários casos desse tipo:

C#

```
Console.WriteLine(.41f % .2f); // output: 0.00999999  
  
double a = 0.1;  
double b = 3 * a;  
Console.WriteLine(b == 0.3);    // output: False  
Console.WriteLine(b - 0.3);    // output: 5.55111512312578E-17  
  
decimal c = 1 / 3.0m;  
decimal d = 3 * c;  
Console.WriteLine(d == 1.0m);   // output: False  
Console.WriteLine(d);        // output: 0.99999999999999999999999999999999
```

Para obter mais informações, confira os comentários nas páginas de referência [System.Double](#), [System.Single](#) ou [System.Decimal](#).

## Capacidade de sobrecarga do operador

Um tipo definido pelo usuário pode **sobrepor** os operadores aritméticos unários (`++`, `--`, `+` e `-`) e binários (`*`, `/`, `%`, `+` e `-`). Quando um operador binário está sobrepor, o operador de atribuição composta correspondente também é implicitamente sobrepor. Um tipo definido pelo usuário não pode sobrepor explicitamente um operador de atribuição composta.

## Operadores verificados definidos pelo usuário

A partir do C# 11, quando você sobrecarrega um operador aritmético, pode usar a `checked` palavra-chave para definir a versão *marcada* desse operador. O seguinte exemplo mostra como fazer isso:

C#

```
public record struct Point(int X, int Y)
{
    public static Point operator checked +(Point left, Point right)
    {
        checked
        {
            return new Point(left.X + right.X, left.Y + right.Y);
        }
    }

    public static Point operator +(Point left, Point right)
    {
        return new Point(left.X + right.X, left.Y + right.Y);
    }
}
```

Ao definir um operador verificado, você também deve definir o operador correspondente sem o modificador `checked`. O operador verificado é chamado em um contexto verificado; o operador sem o modificador `checked` é chamado em um contexto não verificado. Se você fornecer apenas o operador sem o modificador `checked`, ele será chamado em um contexto `checked` e `unchecked`.

Quando você define ambas as versões de um operador, espera-se que o comportamento delas seja diferente somente quando o resultado de uma operação for muito grande para representar no tipo de resultado da seguinte maneira:

- Um operador verificado lança um `OverflowException`.
- Um operador sem o modificador `checked` retornará uma instância que representa um resultado *truncado*.

Para obter informações sobre a diferença no comportamento dos operadores aritméticos internos, consulte a seção [Estouro aritmético e divisão por zero](#).

Você só poderá usar o `checked` modificador quando sobrecarregar qualquer um dos seguintes operadores:

- Operadores unários `++`, `--` e `-`
- Operadores binários `*`, `/`, `+` e `-`
- [Operadores de conversão explícita](#)

### ⚠ Observação

O contexto de verificação de estouro dentro do corpo de um operador verificado não é afetado pela presença do modificador `checked`. O contexto padrão é

definido pelo valor da opção do compilador `CheckForOverflowUnderflow`. Use as instruções `checked` e `unchecked` para especificar explicitamente o contexto de verificação de estouro, como demonstra o exemplo no início desta seção.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- Operadores de incremento e decremento pós-fixados
- Operadores de incremento e decremento pré-fixados
- Operador de adição de unário
- Operador de subtração de unário
- Operador de multiplicação
- Operador de divisão
- Operador de resto
- Operador de adição
- Operador de subtração
- Atribuição composta
- Os operadores verificados e não verificados
- Promoções numéricas

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [System.Math](#)
- [System.MathF](#)
- [Numéricos no .NET](#)

# Operadores lógicos booleanos – AND, OR, NOT, XOR

Artigo • 07/04/2023

Os operadores booleanos lógicos executam operações lógicas com [operandos bool](#). Os operadores incluem a negação lógica unária (!), and () lógico binário (&), OR (|) e OR exclusivo (^) e os AND () e OR (&&) lógicos condicionais binários ||.

- Operador unário ! (negação lógica).
- Operadores binários & (AND lógico), | (OR lógico) e ^ (OR exclusivo lógico). Esses operadores sempre avaliam os dois operandos.
- Operadores binários && (AND lógico condicional) e || (OR lógico condicional). Esses operadores avaliam o operando à direita apenas se for necessário.

Para os operandos dos [tipos numéricos inteiros](#), os operadores &, | e ^ executam operações lógicas bit a bit. Para obter mais informações, veja [Operadores bit a bit e shift](#).

## Operador de negação lógica !

O prefixo unário ! calcula a negação lógica de seu operando. Ou seja, ele produz `true`, se o operando for avaliado como `false`, e `false`, se o operando for avaliado como `true`:

C#

```
bool passed = false;
Console.WriteLine(!passed); // output: True
Console.WriteLine(!true); // output: False
```

O operador de postfixo ! unário é o [operador que perdoa nulos](#).

## Operador AND lógico &

O operador & computa o AND lógico de seus operandos. O resultado de `x & y` será `true` se ambos `x` e `y` forem avaliados como `true`. Caso contrário, o resultado será `false`.

O operador `&` avalia os dois operandos, mesmo se o operando à esquerda for avaliado como `false`, de modo que o resultado da operação seja `false`, independentemente do valor do operando à direita.

No exemplo a seguir, o operando à direita do operador `&` é uma chamada de método, que é executada independentemente do valor do operando à esquerda:

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

O [operador AND lógico condicional](#) `&&` também computa o AND lógico e seus operandos, mas não avalia o operando à direita se o operando à esquerda for avaliado como `false`.

Para os operandos de [tipos numéricos integras](#), o operador `&` computa o AND lógico bit a bit de seus operandos. O operador `&` unário é o [operador address-of](#).

## Operador OR exclusivo lógico `^`

O operador `^` computa o OR exclusivo lógico, também conhecido como o XOR lógico, de seus operandos. O resultado de `x ^ y` é `true` se `x` é avaliado como `true` e `y` avaliado como `false`, ou `x` avaliado como `false` e `y` avaliado como `true`. Caso contrário, o resultado será `false`. Ou seja, para os operandos `bool`, o operador `^` computa o mesmo resultado que o [operador de desigualdade](#) `!=`.

C#

```
Console.WriteLine(true ^ true); // output: False
Console.WriteLine(true ^ false); // output: True
Console.WriteLine(false ^ true); // output: True
Console.WriteLine(false ^ false); // output: False
```

Para os operandos de [tipos numéricos integrais](#), o operador `^` computa o [OR exclusivo lógico bit a bit](#) de seus operandos.

## Operador OR lógico |

O operador `|` computa o OR lógico de seus operandos. O resultado de `x | y` será `true` se `x` ou `y` for avaliado como `true`. Caso contrário, o resultado será `false`.

O operador `|` avalia os dois operandos, mesmo se o operando à esquerda for avaliado como `true`, de modo que o resultado da operação seja `true`, independentemente do valor do operando à direita.

No exemplo a seguir, o operando à direita do operador `|` é uma chamada de método, que é executada independentemente do valor do operando à esquerda:

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

O [operador OR lógico condicional](#) `||` também computa o OR lógico e seus operandos, mas não avalia o operando à direita se o operando à esquerda for avaliado como `true`.

Para os operandos de [tipos numéricos integrais](#), o operador `|` computa o [OR lógico bit a bit](#) de seus operandos.

# Operador AND lógico condicional &&

O operador AND lógico condicional `&&`, também conhecido como operador AND lógico de "curto-circuito", computa o AND lógico de seus operandos. O resultado de `x && y` será `true` se ambos `x` e `y` forem avaliados como `true`. Caso contrário, o resultado será `false`. Se `x` for avaliado como `false`, `y` não será avaliado.

No exemplo a seguir, o operando à direita do operador `&&` é uma chamada de método, que não é executada se o operando à esquerda for avaliado como `false`:

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

O [operador AND lógico &](#) também computa o AND lógico de seus operandos, mas sempre avalia os dois operandos.

# Operador OR lógico condicional ||

O operador OR lógico condicional `||`, também conhecido como operador OR lógico de "curto-circuito", computa o OR lógico de seus operandos. O resultado de `x || y` será `true` se `x` ou `y` for avaliado como `true`. Caso contrário, o resultado será `false`. Se `x` for avaliado como `true`, `y` não será avaliado.

No exemplo a seguir, o operando à direita do operador `||` é uma chamada de método, que não é executada se o operando à esquerda for avaliado como `true`:

C#

```

bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True

```

O operador OR lógico `|` também computa o OR lógico de seus operandos, mas sempre avalia os dois operandos.

## Operadores lógicos booleanos anuláveis

Para operandos `bool?`, os operadores `&` (AND lógico) e `|` (OR lógico) dão suporte à lógica de três valores da seguinte maneira:

- O operador `&` produz `true` somente se ambos os operandos forem avaliados como `true`. Se `x` ou `y` forem avaliados como `false`, `x & y` produzirá `false` (mesmo que outro operando seja avaliado como `null`). Caso contrário, o resultado de `x & y` será `null`.
- O operador `|` produz `false` somente se ambos os operandos forem avaliados como `false`. Se `x` ou `y` forem avaliados como `true`, `x | y` produzirá `true` (mesmo que outro operando seja avaliado como `null`). Caso contrário, o resultado de `x | y` será `null`.

A tabela a seguir apresenta essa semântica:

<code>x</code>	<code>a</code>	<code>x&amp;y</code>	<code>x y</code>
<code>true</code>	<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>
<code>true</code>	<code>null</code>	<code>null</code>	<code>true</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>

x	a	x&y	x y
false	false	false	false
false	nulo	false	nulo
null	true	null	true
nulo	false	false	nulo
nulo	nulo	nulo	nulo

O comportamento desses operadores difere do comportamento típico do operador com tipos de valores anuláveis. Normalmente, um operador definido para operandos de um tipo de valor também pode ser usado com operandos do tipo de valor anulável correspondente. Esse operador produz `null` se algum de seus operandos avaliar para `null`. No entanto, os operadores `&` e `|` podem produzir não nulos, mesmo se um dos operandos avaliar para `null`. Para obter mais informações sobre o comportamento do operador com tipos de valor nulos, confira a seção [Operadores suspensos](#) no artigo [Tipos de valor anulável](#).

Você também pode usar os operadores `!` e `^` com os operandos `bool?`, como mostra o exemplo a seguir:

```
C#
bool? test = null;
Display(!test);           // output: null
Display(test ^ false);   // output: null
Display(test ^ null);    // output: null
Display(true ^ null);   // output: null

void Display(bool? b) => Console.WriteLine(b is null ? "null" :
b.Value.ToString());
```

Os operadores lógicos condicionais `&&` e `||` não oferecem suporte a operandos `bool?`.

## Atribuição composta

Para um operador binário `op`, uma expressão de atribuição composta do formato

```
C#
x op= y
```

é equivalente a

```
C#
```

```
x = x op y
```

exceto que `x` é avaliado apenas uma vez.

Os operadores `&`, `|` e `^` suportam a atribuição de compostos, conforme mostrado no exemplo a seguir:

```
C#
```

```
bool test = true;
test &= false;
Console.WriteLine(test); // output: False

test |= true;
Console.WriteLine(test); // output: True

test ^= false;
Console.WriteLine(test); // output: True
```

### ① Observação

Os operadores lógicos condicionais `&&` e `||` não suportam a atribuição composta.

## Precedência do operador

A lista a seguir ordena os operadores lógicos, começando da mais alta precedência até a mais baixa:

- Operador de negação lógica `!`
- Operador AND lógico `&`
- Operador OR exclusivo lógico `^`
- Operador OR lógico `|`
- Operador AND lógico condicional `&&`
- Operador OR lógico condicional `||`

Use parênteses, `()`, para alterar a ordem de avaliação imposta pela precedência do operador:

```
C#
```

```

Console.WriteLine(true | true & false);    // output: True
Console.WriteLine((true | true) & false); // output: False

bool Operand(string name, bool value)
{
    Console.WriteLine($"Operand {name} is evaluated.");
    return value;
}

var byDefaultPrecedence = Operand("A", true) || Operand("B", true) &&
Operand("C", false);
Console.WriteLine(byDefaultPrecedence);
// Output:
// Operand A is evaluated.
// True

var changedOrder = (Operand("A", true) || Operand("B", true)) &&
Operand("C", false);
Console.WriteLine(changedOrder);
// Output:
// Operand A is evaluated.
// Operand C is evaluated.
// False

```

Para obter a lista completa de operadores C# ordenados por nível de precedência, consulte a seção [Precedência de operador](#) do artigo [Operadores C#](#).

## Capacidade de sobrecarga do operador

Os tipos definidos pelo usuário podem [sobrecarregar](#) os operadores `!`, `&`, `|` e `^`.

Quando um operador binário está sobrecarregado, o operador de atribuição composta correspondente também é implicitamente sobrecarregado. Um tipo definido pelo usuário não pode sobrecarregar explicitamente um operador de atribuição composta.

Um tipo definido pelo usuário não pode sobrecarregar os operadores lógicos `&&` condicionais e `||`. No entanto, se um tipo definido pelo usuário sobrecarregar os operadores `true` e `false` e o operador `&` ou `|` de uma determinada maneira, a operação `&&` ou `||`, respectivamente, pode ser avaliada para os operandos desse tipo. Para obter mais informações, veja a seção [Operadores lógicos condicionais definidos pelo usuário](#) na [especificação da linguagem C#](#).

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- Operador de negação lógica
- Operadores lógicos
- Operadores lógicos condicionais
- Atribuição composta

## Confira também

- Referência de C#
- Operadores e expressões C#
- Operadores shift e bit a bit

# Operadores bit a bit e de deslocamento (referência do C#)

Artigo • 03/04/2023

Os operadores bit a bit e de deslocamento incluem complemento bit a bit unário, deslocamento binário para a esquerda e a direita, deslocamento sem sinal para a direita e os operadores binários lógicos AND, OR e OR exclusivo. Esses operandos usam operandos dos [tipos numéricos integrais](#) ou do tipo `char`.

- Operador unário `~` ([complemento bit a bit](#))
- Operadores binários `<<` ([deslocamento para a esquerda](#)), `>>` ([deslocamento para a direita](#)) e `>>>` ([sem sinal de deslocamento para a direita](#))
- Operadores binários `&` ([AND lógico](#)), `|` ([OR lógico](#)) e `^` ([OR exclusivo lógico](#))

Esses operadores são definidos para os tipos `int`, `uint`, `long` e `ulong`. Quando ambos os operandos são de outros tipos integrais (`sbyte`, `byte`, `short`, `ushort` ou `char`), seus valores são convertidos no tipo `int`, que também é o tipo de resultado de uma operação. Quando os operandos são de tipos integrais diferentes, seus valores são convertidos no tipo integral mais próximo que o contém. Para saber mais, confira a seção [Promoções numéricas da Especificação da linguagem C#](#). Os operadores compostos (como `>>=`) não convertem seus argumentos em `int` ou têm o tipo de resultado como `int`.

Os operadores `&`, `|` e `^` também são definidos para os operandos do tipo `bool`. Para obter mais informações, veja [Operadores lógicos booleanos](#).

As operações de deslocamento e bit a bit nunca causam estouro e produzem os mesmos resultados nos contextos [marcados e desmarcados](#).

## Operador de complemento bit a bit `~`

O operador `~` produz um complemento bit a bit de seu operando invertendo cada bit:

C#

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

Você também pode usar o símbolo `~` para declarar finalizadores. Para mais informações, consulte [Finalizadores](#).

## Operador de deslocamento à esquerda `<<`

O operador `<<` desloca para esquerda o operando à esquerda pelo número de bits definido pelo seu operando à direita. Para saber mais sobre como o operando à direita define a contagem de deslocamento, veja a seção [Contagem de deslocamento dos operadores de deslocamento](#).

A operação de deslocamento à esquerda descarta os bits de ordem superior que estão fora do intervalo do tipo de resultado e define as posições de bits vazios de ordem inferior como zero, como mostra o exemplo a seguir:

```
C#  
  
uint x = 0b_1100_1001_0000_0000_0000_0001_0001;  
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");  
  
uint y = x << 4;  
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2)}");  
// Output:  
// Before: 110010010000000000000000000000010001  
// After: 1001000000000000000000000000000100010000
```

Como os operadores de deslocamento são definidos apenas para os tipos `int`, `uint`, `long` e `ulong`, o resultado de uma operação sempre contém pelo menos 32 bits. Se o operando à esquerda for de outro tipo integral (`sbyte`, `byte`, `short`, `ushort` ou `char`), seu valor será convertido no tipo `int`, como mostra o exemplo a seguir:

```
C#  
  
byte a = 0b_1111_0001;  
  
var b = a << 8;  
Console.WriteLine(b.GetType());  
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");  
// Output:  
// System.Int32  
// Shifted byte: 1111000100000000
```

## Operador de deslocamento à direita `>>`

O operador `>>` desloca para direita o operando à esquerda pelo número de bits definido pelo seu operando à direita. Para saber mais sobre como o operando à direita define a contagem de deslocamento, veja a seção [Contagem de deslocamento dos operadores de deslocamento](#).

A operação de deslocamento à direita descarta os bits de ordem inferior, como mostra o exemplo a seguir:

C#

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2).PadLeft(4, '0'), 4}");
// Output:
// Before: 1001
// After: 0010
```

As posições vazias de bit de ordem superior são definidas com base no tipo do operando à esquerda da seguinte maneira:

- Se o operando à esquerda for do tipo `int` ou `long`, o operador de deslocamento à direita executará um deslocamento *aritmético*: o valor do bit mais significativo (o bit de sinal) do operando à esquerda é propagado para as posições vazias de bits de ordem superior. Ou seja, as posições vazias de bit de ordem superior são definidas como zero se o operando à esquerda for positivo e definidas como um se ele for negativo.

C#

- Se o operando à esquerda for do tipo `uint` ou `ulong`, o operador de deslocamento à direita executará um deslocamento *lógico*: as posições vazias de bit de ordem superior são sempre definidas como zero.

C#

```
uint c = 0b_1000_0000_0000_0000_0000_0000_0000;
Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

uint d = c >> 3;
Console.WriteLine($"After: {Convert.ToString(d, toBase: 2).PadLeft(32,
'0'), 32}");
// Output:
// Before: 10000000000000000000000000000000
// After: 00010000000000000000000000000000
```

### ① Observação

Use o operador de deslocamento à direita sem sinal para executar um deslocamento *lógica* em operandos de tipos inteiros com sinal. Isso é preferido para converter um operando à esquerda em um tipo não assinado e, em seguida, converter o resultado de uma operação de deslocamento de volta para um tipo assinado.

## Operador de deslocamento para a direita sem sinal `>>`

Disponível em C# 11 e versões posteriores, o operador `>>>` desloca para direita o operando à esquerda pelo número de bits definido pelo seu operando à direita. Para saber mais sobre como o operando à direita define a contagem de deslocamento, veja a seção [Contagem de deslocamento dos operadores de deslocamento](#).

O operador `>>>` sempre realiza um deslocamento *lógico*. Ou seja, as posições de bits vazias de alta ordem são sempre definidas como zero, independentemente do tipo do operando à esquerda. O operador `>>` executa um deslocamento *aritmético* (ou seja, o valor do bit mais significativo é propagado para as posições de bit vazias de alta ordem) se o operando à esquerda for de um tipo assinado. O seguinte exemplo demonstra a diferença entre os operadores `>>` e `>>>` para um operando à esquerda negativo:

C#

```
int x = -8;
Console.WriteLine($"Before: {x,11}, hex: {x,8:x}, binary:
{Convert.ToString(x, toBase: 2), 32}");

int y = x >> 2;
Console.WriteLine($"After >>: {y,11}, hex: {y,8:x}, binary:
{Convert.ToString(y, toBase: 2), 32}");
```

```
int z = x >>> 2;
Console.WriteLine($"After >>>: {z,11}, hex: {z,8:x}, binary:
{Convert.ToString(z, toBase: 2).PadLeft(32, '0'), 32}");
// Output:
// Before:          -8, hex: fffffff8, binary:
1111111111111111111111111111111000
// After >>:        -2, hex: ffffffe, binary:
1111111111111111111111111111111110
// After >>>: 1073741822, hex: 3fffffe, binary:
00111111111111111111111111111110
```

## Operador AND lógico &

O operador `&` computa o AND lógico bit a bit de seus operandos inteiros:

C#

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

Para operandos `bool`, o operador `&` computa o [AND lógico](#) de seus operandos. O operador `&` unário é o [operador address-of](#).

## Operador OR exclusivo lógico ^

O operador `^` computa o OR exclusivo lógico bit a bit, também conhecido como o XOR lógico bit a bit, de seus operandos inteiros:

C#

```
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

Para operandos `bool`, o operador `^` computa o [OR exclusivo lógico](#) de seus operandos.

## Operador OR lógico |

O operador `|` computa o OR lógico bit a bit de seus operandos inteiros:

```
C#
```

```
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

Para operandos `bool`, o operador `|` computa o [OR lógico](#) de seus operandos.

## Atribuição composta

Para um operador binário `op`, uma expressão de atribuição composta do formato

```
C#
```

```
x op= y
```

é equivalente a

```
C#
```

```
x = x op y
```

exceto que `x` é avaliado apenas uma vez.

O seguinte exemplo demonstra o uso da atribuição composta com operadores bit a bit e de deslocamento:

```
C#
```

```
uint INITIAL_VALUE = 0b_1111_1000;

uint a = INITIAL_VALUE;
a &= 0b_1001_1101;
Display(a); // output: 10011000

a = INITIAL_VALUE;
a |= 0b_0011_0001;
Display(a); // output: 11111001

a = INITIAL_VALUE;
a ^= 0b_1000_0000;
```

```

Display(a); // output: 01111000

a = INITIAL_VALUE;
a <= 2;
Display(a); // output: 1111100000

a = INITIAL_VALUE;
a >= 4;
Display(a); // output: 00001111

a = INITIAL_VALUE;
a >>= 4;
Display(a); // output: 00001111

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2).PadLeft(8, '0')}, 8}");

```

Devido a [promoções numéricas](#), o resultado da operação `op` pode não ser implicitamente conversível no tipo `T` de `x`. Nesse caso, se `op` for um operador predefinido e o resultado da operação for explicitamente convertido no tipo `T` de `x`, uma expressão de atribuição composta da forma `x op= y` será equivalente a `x = (T)(x op y)`, exceto que `x` será avaliada apenas uma vez. O exemplo a seguir demonstra esse comportamento:

C#

```

byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}"); // output:
1111000100000000

x <= 8;
Console.WriteLine(x); // output: 0

```

## Precedência do operador

A lista a seguir ordena os operadores lógicos, começando da mais alta precedência até a mais baixa:

- Operador de complemento bit a bit `~`
- Operadores de deslocamento `<<`, `>>` e `>>>`
- Operador AND lógico `&`
- Operador OR exclusivo lógico `^`
- Operador OR lógico `|`

Use parênteses, `()`, para alterar a ordem de avaliação imposta pela precedência do operador:

```
C#  
  
uint a = 0b_1101;  
uint b = 0b_1001;  
uint c = 0b_1010;  
  
uint d1 = a | b & c;  
Display(d1); // output: 1101  
  
uint d2 = (a | b) & c;  
Display(d2); // output: 1000  
  
void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2),  
4}");
```

Para obter a lista completa de operadores C# ordenados por nível de precedência, consulte a seção [Precedência de operador](#) do artigo [Operadores C#](#).

## Contagem de deslocamento dos operadores de deslocamento

Para os operadores de deslocamento `<<`, `>>` e `>>>` o tipo do operando à direita deve ser `int` ou um tipo que tenha uma [conversão numérica implícita predefinida](#) para `int`.

Para as expressões `x << count`, `x >> count` e `x >>> count`, a contagem real de deslocamento depende do tipo de `x` da seguinte maneira:

- Se o tipo de `x` for `int` ou `uint`, a contagem de deslocamentos será definida pelos *cinco* bits de ordem inferior do operando à direita. Ou seja, a contagem de deslocamentos é calculada a partir de `count & 0x1F` (ou `count & 0b_1_1111`).
- Se o tipo de `x` for `long` ou `ulong`, a contagem de deslocamentos será definida pelos *seis* bits de ordem inferior do operando à direita. Ou seja, a contagem de deslocamentos é calculada a partir de `count & 0x3F` (ou `count & 0b_11_1111`).

O exemplo a seguir demonstra esse comportamento:

```
C#  
  
int count1 = 0b_0000_0001;  
int count2 = 0b_1110_0001;
```

```
int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a
<< count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {count2} is {b
>> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2

int count = -31;
int c = 0b_0001;
Console.WriteLine($"{c} << {count} is {c << count}");
// Output:
// 1 << -31 is 2
```

### Observação

Como mostra o exemplo anterior, o resultado de uma operação de deslocamento pode ser diferente de zero mesmo que o valor do operando à direita seja maior do que o número de bits no operando à esquerda.

## Operadores lógicos de enumeração

Os operadores `~`, `&`, `|` e `^` também têm suporte em qualquer tipo de enumeração. Para operandos do mesmo tipo de enumeração, uma operação lógica é executada nos valores correspondentes do tipo integral subjacente. Por exemplo, para qualquer `x` e `y` de um tipo de enumeração `T` com um tipo subjacente `U`, a expressão `x & y` produz o mesmo resultado que a expressão `(T)((U)x & (U)y)`.

Geralmente, você usa os operadores lógicos bit a bit com um tipo de enumeração definido com o atributo [sinalizadores](#). Para obter mais informações, veja a seção [Tipos de enumeração como sinalizadores de bit](#) do artigo [Tipos de enumeração](#).

## Capacidade de sobrecarga do operador

Um tipo definido pelo usuário pode [sobrepor](#) os operadores `~`, `<<`, `>>`, `>>>`, `&`, `|` e `^`. Quando um operador binário está sobrepor, o operador de atribuição composta correspondente também é implicitamente sobrepor. Um tipo definido pelo usuário não pode sobrepor explicitamente um operador de atribuição composta.

Se um tipo definido pelo usuário `T` sobrecarregar o operador `<<`, `>>` ou `>>>`, o tipo do operando à esquerda deverá ser `T`. No C# 10 e anterior, o tipo do operando à direita deve ser `int`; começando com C# 11, o tipo do operando à direita de um operador de deslocamento sobrecarregado pode ser qualquer um.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- Operador de complemento bit a bit
- Operadores shift
- Operadores lógicos
- Atribuição composta
- Promoções numéricas
- C# 11 – Requisitos de deslocamento flexíveis
- C# 11 – Operador de deslocamento lógico à direita

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Operadores lógicos boolianos](#)

# Operadores de igualdade – teste se dois objetos são iguais ou não

Artigo • 07/04/2023

Os operadores `==` (igualdade) e `!=` (desigualdade) verificam se os operandos são iguais ou não. Os tipos de valor são iguais quando seu conteúdo é igual. Os tipos de referência são iguais quando as duas variáveis se referem ao mesmo armazenamento.

## Operador de igualdade `==`

O operador de igualdade `==` retornará `true` se seus operandos forem iguais; caso contrário, `false`.

## Igualdade de tipos de valor

Os operandos dos [tipos de valor internos](#) serão iguais se seus valores forem iguais:

C#

```
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b); // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2); // output: False
Console.WriteLine(c1 == char.ToLower(c2)); // output: True
```

### ⓘ Observação

Para os operadores `==`, `<`, `>`, `<=` e `>=`, se nenhum dos operandos for um número (`Double.NaN` ou `Single.NaN`), o resultado da operação será `false`. Isso significa que o valor `NaN` não é superior, inferior nem igual a nenhum outro valor `double` (ou `float`), incluindo `NaN`. Para obter mais informações e exemplos, consulte o artigo de referência [Double.NaN ou Single.NaN](#).

Dois operandos do mesmo tipo de [enum](#) serão iguais se os valores correspondentes do tipo integral subjacente forem iguais.

Os tipos `struct` definidos pelo usuário não dão suporte ao operador `==`, por padrão. Para dar suporte ao operador `==`, um `struct` definido pelo usuário precisa [sobrecarregá-lo](#).

Os `==` operadores e `!=` são compatíveis com [tuplas C#](#). Para obter mais informações, consulte a seção [Igualdade de tupla](#) do artigo [Tipos de tupla](#).

## Igualdade de tipos de referência

Por padrão, dois operandos do tipo de referência não registrados são iguais quando se referem ao mesmo objeto:

```
C#  
  
public class ReferenceTypesEquality  
{  
    public class MyClass  
    {  
        private int id;  
  
        public MyClass(int id) => this.id = id;  
    }  
  
    public static void Main()  
    {  
        var a = new MyClass(1);  
        var b = new MyClass(1);  
        var c = a;  
        Console.WriteLine(a == b); // output: False  
        Console.WriteLine(a == c); // output: True  
    }  
}
```

Como mostra o exemplo, os tipos de referência definidos pelo usuário dão suporte ao operador `==`, por padrão. No entanto, um tipo de referência pode sobrecarregar o operador `==`. Se um tipo de referência sobrecarregar o operador `==`, use o método [Object.ReferenceEquals](#) para verificar se as duas referências desse tipo dizem respeito ao mesmo objeto.

## Igualdade de tipos de registro

Disponíveis no C# 9.0 e posterior, os [tipos de registro](#) dão suporte aos operadores `==` e `!=` que, por padrão, fornecem semântica de igualdade de valor. Ou seja, dois operandos de registro são iguais quando ambos são `null` ou os valores correspondentes de todos os campos e propriedades implementadas automaticamente são iguais.

C#

```
public class RecordTypesEquality
{
    public record Point(int X, int Y, string Name);
    public record TaggedNumber(int Number, List<string> Tags);

    public static void Main()
    {
        var p1 = new Point(2, 3, "A");
        var p2 = new Point(1, 3, "B");
        var p3 = new Point(2, 3, "A");

        Console.WriteLine(p1 == p2); // output: False
        Console.WriteLine(p1 == p3); // output: True

        var n1 = new TaggedNumber(2, new List<string>() { "A" });
        var n2 = new TaggedNumber(2, new List<string>() { "A" });
        Console.WriteLine(n1 == n2); // output: False
    }
}
```

Como mostra o exemplo anterior, para membros do tipo de referência não registrados, seus valores de referência são comparados, não as instâncias referenciadas.

## Igualdade da cadeia de caracteres

Dois operandos da [cadeia de caracteres](#) serão iguais quando ambos forem `null` ou ambas as instâncias da cadeia de caracteres tiverem o mesmo comprimento e caracteres idênticos em cada posição de caractere:

C#

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower()); // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3); // output: False
```

Comparações de igualdade de cadeia de caracteres são comparações ordinais que diferenciam maiúsculas de minúsculas. Para obter mais informações sobre a comparação de cadeias de caracteres, confira [Como comparar cadeias de caracteres no C#](#).

## Igualdade de delegado

Os dois operandos `delegate` do mesmo tipo de tempo de execução são iguais quando ambos são `null` ou suas listas de invocação são do mesmo comprimento e tem entradas iguais em cada posição:

C#

```
Action a = () => Console.WriteLine("a");

Action b = a + a;
Action c = a + a;
Console.WriteLine(object.ReferenceEquals(b, c)); // output: False
Console.WriteLine(b == c); // output: True
```

Saiba mais na seção [Operadores de igualdade de delegados](#) na [Especificação da linguagem C#](#).

Delegados produzidos a partir da avaliação de [expressões lambda](#) semanticamente idênticas não são iguais, como mostra o exemplo a seguir:

C#

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("a");

Console.WriteLine(a == b); // output: False
Console.WriteLine(a + b == a + b); // output: True
Console.WriteLine(b + a == a + b); // output: False
```

## Operador de desigualdade !=

O operador `!=` de desigualdade retornará `true` se seus operandos não forem iguais, `false` caso contrário. No caso dos operandos de [tipos internos](#), a expressão `x != y` gera o mesmo resultado que a expressão `!(x == y)`. Para obter mais informações sobre a igualdade de tipos, confira a seção [Operador de igualdade](#).

O exemplo a seguir demonstra o uso do operador `!=`:

C#

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b); // output: True

string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2); // output: False
```

```
object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2); // output: True
```

## Capacidade de sobrecarga do operador

Os tipos definidos pelo usuário podem [sobrecarregar](#) os operadores `==` e `!=`. Se um tipo sobrecarregar um dos dois operadores, ele também precisará sobrecarregar o outro.

Um tipo de registro não pode sobrecarregar explicitamente os `==` operadores e `!=`. Se você precisar alterar o comportamento dos operadores `==` e `!=` para o tipo de registro `T`, implemente o método [IEquatable<T>.Equals](#) com a seguinte assinatura:

C#

```
public virtual bool Equals(T? other);
```

## Especificação da linguagem C#

Para obter mais informações, consulte a seção [Operadores de teste de tipo e relacional](#) na [Especificação da linguagem C#](#).

Para obter mais informações sobre a igualdade de tipos de registro, consulte a seção [Membros de igualdade](#) da [nota de proposta de recurso de registros](#).

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [System.IEquatable<T>](#)
- [Object.Equals](#)
- [Object.ReferenceEquals](#)
- [Comparações de igualdade](#)
- [Operadores de comparação](#)

# Operadores de comparação (referência do C#)

Artigo • 07/04/2023

Os operadores de comparação < (menor que), > (maior que), <= (menor ou igual) e >= (maior que ou igual), também conhecida como relacionais, comparam seus operandos. Esses operadores são suportados por todos os tipos numéricos [integrais](#) e de [ponto flutuante](#).

## Observação

Para os operadores ==, <, >, <= e >=, se nenhum dos operandos for um número ([Double.NaN](#) ou [Single.NaN](#)), o resultado da operação será [false](#). Isso significa que o valor [NaN](#) não é superior, inferior nem igual a nenhum outro valor [double](#) (ou [float](#)), incluindo [NaN](#). Para obter mais informações e exemplos, consulte o artigo de referência [Double.NaN ou Single.NaN](#).

O tipo [char](#) também dá suporte a operadores de comparação. No caso de operandos [char](#), os códigos de caractere correspondentes são comparados.

Tipos de enumeração também dão suporte a operadores de comparação. No caso dos operandos do mesmo tipo de [enum](#), os valores correspondentes do tipo integral subjacente são comparados.

Os operadores == e != verificam se seus operandos são iguais ou não.

## Operador menor que <

O operador < retornará [true](#) se o operando à esquerda for menor do que o operando à direita, caso contrário, [false](#):

C#

```
Console.WriteLine(7.0 < 5.1);    // output: False
Console.WriteLine(5.1 < 5.1);    // output: False
Console.WriteLine(0.0 < 5.1);    // output: True

Console.WriteLine(double.NaN < 5.1);    // output: False
Console.WriteLine(double.NaN >= 5.1);    // output: False
```

# Operador maior que >

O operador `>` retornará `true` se o operando à esquerda for maior do que o operando à direita, caso contrário, `false`:

C#

```
Console.WriteLine(7.0 > 5.1);    // output: True
Console.WriteLine(5.1 > 5.1);    // output: False
Console.WriteLine(0.0 > 5.1);    // output: False

Console.WriteLine(double.NaN > 5.1);    // output: False
Console.WriteLine(double.NaN <= 5.1);    // output: False
```

# Operador menor ou igual a <=

O operador `<=` retornará `true` se o operando à esquerda for menor ou igual ao operando à direita, caso contrário, `false`:

C#

```
Console.WriteLine(7.0 <= 5.1);    // output: False
Console.WriteLine(5.1 <= 5.1);    // output: True
Console.WriteLine(0.0 <= 5.1);    // output: True

Console.WriteLine(double.NaN > 5.1);    // output: False
Console.WriteLine(double.NaN <= 5.1);    // output: False
```

# Operador maior ou igual >=

O operador `>=` retornará `true` se o operando à esquerda for maior ou igual ao operando à direita, caso contrário, `false`:

C#

```
Console.WriteLine(7.0 >= 5.1);    // output: True
Console.WriteLine(5.1 >= 5.1);    // output: True
Console.WriteLine(0.0 >= 5.1);    // output: False

Console.WriteLine(double.NaN < 5.1);    // output: False
Console.WriteLine(double.NaN >= 5.1);    // output: False
```

# Capacidade de sobrecarga do operador

Os tipos definidos pelo usuário podem [sobrecarregar](#) os operadores `<`, `>`, `<=` e `>=`.

Se um tipo sobrepor um dos operadores `<` ou `>`, ele deverá sobrepor tanto `<` quanto `>`. Se um tipo sobrepor um dos operadores `<=` ou `>=`, ele deverá sobrepor tanto `<=` quanto `>=`.

## Especificação da linguagem C#

Para obter mais informações, consulte a seção [Operadores de teste de tipo e relacional](#) na [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [System.IComparable<T>](#)
- [Operadores de igualdade](#)

# Operadores e expressões de acesso a membro – os operadores de ponto, indexador e invocação.

Artigo • 20/03/2023

Você usa vários operadores e expressões para acessar um membro de tipo. Esses operadores incluem acesso de membro (.), acesso de elemento de matriz ou indexador ([]), índice de ponta (^), intervalo (..), operadores condicionais nulos (?.) e ?[], e invocação de método (). Eles incluem os operadores de acesso de membro *condicional nulo* (?.) e de acesso do indexador (?[]).

- [.\(acesso a membro\)](#): para acessar um membro de um namespace ou um tipo
- [\[\] \(acesso a um indexador ou elemento de matriz\)](#): para acessar um elemento de matriz ou um indexador de tipo
- [?. e ?\[\] \(operadores condicionais nulos\)](#): para executar uma operação de acesso a membro ou elemento somente se um operando for não nulo
- [\(\) \(invocação\)](#): chamar um método acessado ou invocar um delegado
- [^ \(índice do final\)](#): para indicar que a posição do elemento é do final de uma sequência
- [.. \(intervalo\)](#): para especificar um intervalo de índices que você pode usar para obter um intervalo de elementos de sequência

## Expressão de acesso a membro .

Use o token . para acessar um membro de um namespace ou um tipo, como demonstram os exemplos a seguir:

- Use . para acessar um namespace aninhado dentro de um namespace, como mostra o exemplo a seguir de uma [diretiva using](#):

```
C#
```

```
using System.Collections.Generic;
```

- Use . para formar um *nome qualificado* para acessar um tipo dentro de um namespace, como mostra o código a seguir:

```
C#
```

```
System.Collections.Generic.IEnumerable<int> numbers = new int[] { 1, 2, 3 };
```

Use uma [diretiva using](#) para tornar o uso de nomes qualificados opcional.

- Use `.` para acessar [membros de tipo](#), estático e não-estático, como mostra o código a seguir:

C#

```
var constants = new List<double>();
constants.Add(Math.PI);
constants.Add(Math.E);
Console.WriteLine($"{constants.Count} values to show:");
Console.WriteLine(string.Join(", ", constants));
// Output:
// 2 values to show:
// 3.14159265358979, 2.71828182845905
```

Você também pode usar `.` para acessar um [método de extensão](#).

## Operador de indexador []

Os colchetes, `[]`, normalmente são usados para acesso de elemento de matriz, indexador ou ponteiro.

## Acesso de matriz

O exemplo a seguir demonstra como acessar elementos de matriz:

C#

```
int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]); // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant); // output: -3
```

Se um índice de matriz estiver fora dos limites da dimensão correspondente de uma matriz, uma [IndexOutOfRangeException](#) será gerada.

Como mostra o exemplo anterior, você também usar colchetes quando declara um tipo de matriz ou instancia uma instância de matriz.

Para obter mais informações sobre matrizes, confira [Matrizes](#).

## Acesso de indexador

O exemplo a seguir usa o tipo [Dictionary<TKey,TValue>](#) do .NET para demonstrar o acesso de indexador:

C#

```
var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]); // output: 4.14159265358979
```

Os indexadores permitem indexar instâncias de um tipo definido pelo usuário de maneira semelhante à indexação de matriz. Ao contrário dos índices da matriz, que precisam ser um inteiro, os parâmetros do indexador podem ser declarados como qualquer tipo.

Para obter mais informações sobre indexadores, confira [Indexadores](#).

## Outros usos de []

Para saber mais sobre o acesso a elemento de ponteiro, confira a seção [Operador de acesso a elemento de ponteiro \[\]](#) do artigo [Operadores relacionados a ponteiro](#).

Os colchetes também são usados para especificar [atributos](#):

C#

```
[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}
```

## Operadores condicionais nulos ?. e ?[]

Um operador condicional nulo aplicará uma operação de [acesso de membro](#), `?.` ou de [acesso de elemento](#), `?[]` ao operando somente se esse operando for avaliado como

não nulo; caso contrário, ele retornará `null`. Ou seja:

- Se `a` for avaliado como `null`, o resultado de `a?.x` ou `a?[]` será `null`.
- Se `a` for avaliado como não nulo, o resultado de `a?.x` ou `a?[]` será o mesmo resultado de `a.x` ou `a[]`, respectivamente.

### ! Observação

Se `a.x` ou `a[]` lançarem uma exceção, `a?.x` ou `a?[]` lançarão a mesma exceção para não nulo `a`. Por exemplo, se `a` for uma instância de matriz não nula e `x` estiver fora dos limites de `a`, `a?[]` lançará um `IndexOutOfRangeException`.

Os operadores condicionais nulos estão entrando em curto-circuito. Ou seja, se uma operação em uma cadeia de membro operações condicionais de acesso a membro ou elemento retornar `null`, o restante da cadeia não será executado. No exemplo a seguir, `B` não será avaliado se `A` for avaliado como `null` e `C` não será avaliado se `A` ou `B` for avaliado como `null`:

C#

```
A?.B?.Do(C);  
A?.B?[C];
```

Se `A` puder ser nulo, mas `B` e `C` não forem nulos se `A` não for nulo, você só precisará aplicar o operador condicional nulo a `A`:

C#

```
A?.B.C();
```

No exemplo anterior, `B` não é avaliado e `c()` não será chamado se `A` for nulo. No entanto, se o acesso de membro encadeado for interrompido, por exemplo, por parênteses como em `(A?.B).C()`, o curto-circuito não ocorrerá.

Os exemplos a seguir demonstram o uso dos operadores `?.` e `?[]`:

C#

```
double SumNumbers(List<double> setsOfNumbers, int indexOfSetToSum)  
{  
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
```

```
}
```

```
var sum1 = SumNumbers(null, 0);
Console.WriteLine(sum1); // output: NaN
```

```
var numberSets = new List<double[]>
{
    new[] { 1.0, 2.0, 3.0 },
    null
};
```

```
var sum2 = SumNumbers(numberSets, 0);
Console.WriteLine(sum2); // output: 6
```

```
var sum3 = SumNumbers(numberSets, 1);
Console.WriteLine(sum3); // output: NaN
```

C#

```
namespace MemberAccessOperators2;

public static class NullConditionalShortCircuiting
{
    public static void Main()
    {
        Person person = null;
        person?.Name.Write(); // no output: Write() is not called due to
short-circuit.
        try
        {
            (person?.Name).Write();
        }
        catch (NullReferenceException)
        {
            Console.WriteLine("NullReferenceException");
        }; // output: NullReferenceException
    }
}

public class Person
{
    public FullName Name { get; set; }
}

public class FullName
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public void Write()
    {
        Console.WriteLine($"{FirstName} {LastName}");
    }
}
```

O primeiro dos dois exemplos anteriores também usa o [operador de avaliação de nulo??](#) para especificar uma expressão alternativa a ser avaliada caso o resultado de uma operação condicional nula seja `null`.

Se `a.x` ou `a[x]` forem de um tipo de valor não anulável `T`, `a?.x` ou `a? [x]` serão do [tipo de valor anulável T?](#) correspondente. Se você precisar de uma expressão de tipo `T`, aplique o operador de avaliação de nulo `??` a uma expressão condicional nula, como mostra o exemplo a seguir:

C#

```
int GetSumOfFirstTwoOrDefault(int[] numbers)
{
    if ((numbers?.Length ?? 0) < 2)
    {
        return 0;
    }
    return numbers[0] + numbers[1];
}

Console.WriteLine(GetSumOfFirstTwoOrDefault(null)); // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new int[0])); // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new[] { 3, 4, 5 })); // output:
7
```

No exemplo anterior, se você não usar o operador `??`, `numbers?.Length < 2` será avaliado como `false` quando `numbers` for `null`.

### ⓘ Observação

O operador `?.` avalia seu operando à esquerda não mais do que uma vez, garantindo que ele não possa ser alterado para `null` após ser verificado como não nulo.

O operador de acesso do membro condicional nulo `?.` também é conhecido como o operador Elvis.

## Invocação de delegado thread-safe

Use o operador `?.` para verificar se um delegado é não nulo e chame-o de uma forma thread-safe (por exemplo, quando você [aciona um evento](#)), conforme mostrado no código a seguir:

```
C#
```

```
PropertyChanged?.Invoke(...)
```

Esse código é equivalente ao seguinte:

```
C#
```

```
var handler = this.PropertyChanged;
if (handler != null)
{
    handler(...);
}
```

O exemplo anterior é uma maneira thread-safe para garantir que apenas um `handler` não nulo seja invocado. Como as instâncias de delegado são imutáveis, nenhum thread pode alterar o objeto referenciado pela variável local `handler`. Em especial, se o código executado por outro thread cancelar a assinatura do evento `PropertyChanged` e `PropertyChanged` se tornar `null` antes de `handler` ser invocado, o objeto referenciado por `handler` permanecerá não afetado.

## Expressão de invocação ()

Use parênteses, `()`, para chamar um [método](#) ou invocar um [delegado](#).

O exemplo a seguir demonstra como chamar um método (com ou sem argumentos) e invocar um delegado:

```
C#
```

```
Action<int> display = s => Console.WriteLine(s);

var numbers = new List<int>();
numbers.Add(10);
numbers.Add(17);
display(numbers.Count);    // output: 2

numbers.Clear();
display(numbers.Count);    // output: 0
```

Você também pode usar parênteses ao invocar um [construtor](#) com o operador `new`.

## Outros usos de ()

Você também pode usar parênteses para ajustar a ordem na qual as operações em uma expressão são avaliadas. Para saber mais, confira [Operadores C#](#).

[Expressões de conversão](#), que executam conversões de tipo explícitas, também usam parênteses.

## Operador índice do final ^

O operador `^` indica a posição do elemento a partir do final de uma sequência. Para uma sequência de comprimento `length`, `^n` aponta para o elemento com deslocamento `length - n` desde o início de uma sequência. Por exemplo, `^1` aponta para o último elemento de uma sequência, e `^length` aponta para o primeiro elemento de uma sequência.

C#

```
int[] xs = new[] { 0, 10, 20, 30, 40 };
int last = xs[^1];
Console.WriteLine(last); // output: 40

var lines = new List<string> { "one", "two", "three", "four" };
string prelast = lines[^2];
Console.WriteLine(prelast); // output: three

string word = "Twenty";
Index toFirst = ^word.Length;
char first = word[toFirst];
Console.WriteLine(first); // output: T
```

Como mostra o exemplo anterior, a expressão `^e` é do tipo [System.Index](#). Na expressão `^e`, o resultado de `e` deve ser implicitamente conversível para `int`.

Você também pode usar o operador `^` com o [operador de intervalo](#) para criar um intervalo de índices. Para obter mais informações, consulte [Índices e intervalos](#).

## Operador de intervalo ..

O operador `..` especifica o início e o fim de um intervalo de índices como seus operandos. O operando à esquerda é um início *inclusivo* de um intervalo. O operando à direita é um final *exclusivo* de um intervalo. Qualquer um dos operandos pode ser um índice desde o início ou do final de uma sequência, como mostra o exemplo a seguir:

C#

```

int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int start = 1;
int amountToTake = 3;
int[] subset = numbers[start..(start + amountToTake)];
Display(subset); // output: 10 20 30

int margin = 1;
int[] inner = numbers[margin..^margin];
Display(inner); // output: 10 20 30 40

string line = "one two three";
int amountToTakeFromEnd = 5;
Range endIndices = ^amountToTakeFromEnd..^0;
string end = line[endIndices];
Console.WriteLine(end); // output: three

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));

```

Como mostra o exemplo anterior, a expressão `a..b` é do tipo `System.Range`. Na expressão `a..b`, o resultado de `a` e `b` deve ser implicitamente conversível para `Int32` ou `Index`.

### ⓘ Importante

Conversões implícitas de `int` para `Index` gerar um `ArgumentOutOfRangeException` quando o valor é negativo.

Você pode omitir qualquer um dos operandos do operador `..` para obter um intervalo aberto:

- `a..` equivale a `a..^0`
- `..b` equivale a `0..b`
- `..` equivale a `0..^0`

C#

```

int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int amountToDrop = numbers.Length / 2;

int[] rightHalf = numbers[amountToDrop..];
Display(rightHalf); // output: 30 40 50

int[] leftHalf = numbers[..^amountToDrop];
Display(leftHalf); // output: 0 10 20

int[] all = numbers[...];

```

```

Display(all); // output: 0 10 20 30 40 50

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));

```

A tabela a seguir mostra várias maneiras de expressar intervalos de coleções:

Expressão do operador de intervalo	Descrição
..	Todos os valores na coleção.
..end	Valores do início a end exclusivamente.
start..	Valores de start até e inclusive o final.
start..end	Valores de start até e inclusive o end exclusivamente.
^start..	Valores de start até e inclusive a contagem final.
..^end	Valores do início até a end excluindo a contagem final.
start..^end	Valores de start até e inclusive end excluindo a contagem final.
^start..^end	Valores de start até inclusive end excluindo ambas as contagens finais.

O exemplo a seguir demonstra o efeito de usar todos os intervalos apresentados na tabela anterior:

```

C#

int[] oneThroughTen =
{
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};

Write(oneThroughTen, ..);
Write(oneThroughTen, ..3);
Write(oneThroughTen, 2..);
Write(oneThroughTen, 3..5);
Write(oneThroughTen, ^2..);
Write(oneThroughTen, ..^3);
Write(oneThroughTen, 3..^4);
Write(oneThroughTen, ^4..^2);

static void Write(int[] values, Range range) =>
    Console.WriteLine($"{range}: {string.Join(", ", values[range])}");
// Sample output:
//      0..^0:      1, 2, 3, 4, 5, 6, 7, 8, 9, 10

```

```
//    0..3:      1, 2, 3
//    2..^0:      3, 4, 5, 6, 7, 8, 9, 10
//    3..5:       4, 5
//    ^2..^0:     9, 10
//    0..^3:      1, 2, 3, 4, 5, 6, 7
//    3..^4:       4, 5, 6
//    ^4..^2:     7, 8
```

Para obter mais informações, consulte [Índices e intervalos](#).

## Capacidade de sobrecarga do operador

Os operadores `.`, `()`, `^` e `..` não podem ser sobreescritos. O operador `[]` também é considerado um operador não sobreescravável. Use [indexadores](#) para permitir a indexação com tipos definidos pelo usuário.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Acesso de membros](#)
- [Acesso a elemento](#)
- [Acesso de membro condicional nulo](#)
- [Expressões de invocação](#)

Para obter mais informações sobre índices e intervalos, confira a [nota da proposta do recurso](#).

## Confira também

- [Usar operador de índice \(regra de estilo IDE0056\)](#)
- [Usar operador de intervalo \(regra de estilo IDE0057\)](#)
- [Usar chamada de delegado condicional \(regra de estilo IDE1005\)](#)
- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [?? \(operador de avaliação de nulo\)](#)
- [operador ::](#)

# Operadores de teste de tipo e expressão de conversão `is`, `as`, `typeof` e conversões

Artigo • 05/06/2023

Esses operadores e expressões executam a verificação de tipo ou a conversão de tipo. O `is` operador verifica se o tipo do runtime de uma expressão é compatível com um determinado tipo. O `as` operador converterá explicitamente uma expressão para determinado tipo se o tipo de runtime dele for compatível com esse tipo. As [expressões de conversão](#) executam uma conversão explícita para um tipo de destino. O `typeof` operador obtém a instância `System.Type` para um tipo.

## Operador `is`

O operador `is` verifica se o tipo do resultado de uma expressão em tempo de execução é compatível com determinado tipo. O operador `is` também testa um resultado de expressão em relação a um padrão.

A expressão com o operador `is` de teste de tipo tem o seguinte formato

C#

`E is T`

em que `E` é uma expressão que retorna um valor e `T` é o nome de um tipo ou um parâmetro de tipo. `E` não pode ser um método anônimo ou uma expressão lambda.

O operador `is` retorna `true` quando o resultado de uma expressão não é nulo e qualquer uma das seguintes condições é verdadeira:

- O tipo do resultado de uma expressão em tempo de execução é `T`.
- O tipo do resultado de uma expressão em tempo de execução deriva do tipo `T`, implementa a interface `T` ou existe outra [conversão de referência implícita](#) dele para `T`.
- O tipo do resultado de uma expressão em tempo de execução é um [tipo de valor anulável](#) com o tipo subjacente `T` e o `Nullable<T>.HasValue` é `true`.

- Existe uma [conversão boxing](#) ou uma [conversão unboxing](#) do tipo do resultado de uma expressão em tempo de execução para o tipo `T`.

O operador `is` não considera conversões definidas pelo usuário.

O seguinte exemplo demonstra que o operador `is` retorna `true` se o tipo do resultado de uma expressão em tempo de execução é derivado de determinado tipo, ou seja, existe uma conversão de referência entre tipos:

C#

```
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base); // output: True
        Console.WriteLine(b is Derived); // output: False

        object d = new Derived();
        Console.WriteLine(d is Base); // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

O próximo exemplo mostra que o operador `is` leva em conta conversões boxing e unboxing, mas não considera [conversões numéricas](#):

C#

```
int i = 27;
Console.WriteLine(i is System.IFormattable); // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int); // output: True
Console.WriteLine(iBoxed is long); // output: False
```

Para saber mais sobre conversões em C#, confira o capítulo [Conversões da Especificação da linguagem C#](#).

## Teste de tipo com correspondência de padrões

O operador `is` também testa um resultado de expressão em relação a um padrão. O seguinte exemplo mostra como usar um [padrão de declaração](#) para verificar o tipo de uma expressão em tempo de execução:

C#

```
int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b); // output 30
}
```

Para obter informações sobre os padrões com suporte, confira [Padrões](#).

## Operador as

O operador `as` converte explicitamente o resultado de uma expressão para uma determinada referência ou tipo de valor anulável. Se a conversão não for possível, o operador `as` retornará `null`. Ao contrário de uma [expressão de conversão](#), o operador `as` nunca gera uma exceção.

A expressão da forma

C#

```
E as T
```

em que `E` é uma expressão que retorna um valor e `T` é o nome de um tipo ou um parâmetro de tipo, produz o mesmo resultado que

C#

```
E is T ? (T)(E) : (T)null
```

exceto que `E` é avaliado apenas uma vez.

O operador `as` considera apenas as conversões de referência, anulável, boxing e unboxing. Não é possível usar o operador `as` para executar uma conversão definida pelo usuário. Para fazer isso, use uma [expressão de conversão](#).

O exemplo a seguir demonstra o uso do operador `as`:

C#

```
IEnumerable<int> numbers = new[] { 10, 20, 30 };
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]); // 
output: 40
}
```

### Observação

Como mostrado no exemplo anterior, você precisa comparar o resultado da expressão `as` com `null` para verificar se uma conversão foi bem-sucedida. Você pode usar o **operador is** para testar se a conversão foi bem-sucedida e, se for bem-sucedida, atribuir seu resultado a uma nova variável.

## Expressão de conversão

Uma expressão de conversão do formulário `(T)E` realiza uma conversão explícita do resultado da expressão `E` para o tipo `T`. Se não existir nenhuma conversão explícita do tipo `E` para o tipo `T`, ocorrerá um erro em tempo de compilação. No tempo de execução, uma conversão explícita pode não ter êxito e uma expressão de conversão pode lançar uma exceção.

O exemplo a seguir demonstra conversões numéricas e de referência explícitas:

C#

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a); // output: 1234

IEnumerable<int> numbers = new int[] { 10, 20, 30 };
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count); // output: 3
Console.WriteLine(list[1]); // output: 20
```

Para saber mais sobre conversões explícitas sem suporte, confira a seção [Conversões explícitas](#) da [Especificação da linguagem C#](#). Para saber mais sobre como definir uma conversão de tipo explícito ou implícito personalizado, confira [Operadores de conversão definidos pelo usuário](#).

## Outros usos de ()

Você também usa parênteses para [chamar um método ou chamar um delegado](#).

Outro uso dos parênteses é ajustar a ordem na qual as operações em uma expressão são avaliadas. Para saber mais, confira [Operadores C#](#).

## Operador `typeof`

O operador `typeof` obtém a instância [System.Type](#) para um tipo. O argumento do operador `typeof` deve ser o nome de um tipo ou um parâmetro de tipo, como mostra o exemplo a seguir:

C#

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

O argumento não deve ser um tipo que exija anotações de metadados. Como exemplo, confira os seguintes tipos:

- `dynamic`
- `string?` (ou qualquer tipo de referência anulável)

Esses tipos não são representados diretamente em metadados. Os tipos incluem atributos que descrevem o tipo subjacente. Em ambos os casos, você pode usar o tipo subjacente. Em vez de `dynamic`, você pode usar `object`. Em vez de `string?`, você pode usar `string`.

Você também pode usar o operador `typeof` com tipos genéricos não associados. O nome de um tipo genérico não associado deve conter o número apropriado de vírgulas, que é um a menos que o número de parâmetros de tipo. O exemplo a seguir mostra o uso do operador `typeof` com um tipo genérico não associado:

C#

```
Console.WriteLine(typeof(Dictionary<,>));  
// Output:  
// System.Collections.Generic.Dictionary`2[TKey,TValue]
```

O operador `typeof` não pode ter como argumento uma expressão. Para obter a instância `System.Type` para o tipo do resultado de uma expressão em tempo de execução, use o método `Object.GetType`.

## Teste de tipo com o operador `typeof`

Use o operador `typeof` para verificar se o tipo do resultado da expressão em tempo de execução corresponde exatamente a determinado tipo. O seguinte exemplo demonstra as diferenças entre as verificações de tipo executadas com o operador `typeof` e com o operador `is`:

C#

```
public class Animal { }  
  
public class Giraffe : Animal { }  
  
public static class TypeOfExample  
{  
    public static void Main()  
    {  
        object b = new Giraffe();  
        Console.WriteLine(b is Animal); // output: True  
        Console.WriteLine(b.GetType() == typeof(Animal)); // output: False  
  
        Console.WriteLine(b is Giraffe); // output: True  
        Console.WriteLine(b.GetType() == typeof(Giraffe)); // output: True  
    }  
}
```

## Capacidade de sobrecarga do operador

Os operadores `is`, `as` e `typeof` não podem ser sobre carregados.

Um tipo definido pelo usuário não pode criar uma sobre carregar para o operador `()`, mas pode definir conversões de tipo personalizadas, que podem ser executadas por uma expressão de conversão. Para saber mais, confira [Operadores de conversão definidos pelo usuário](#).

# Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- Operador `is`
- Operador `as`
- Expressões `cast`
- Operador `typeof`

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Como converter com segurança usando padrões correspondentes e os operadores "is" e "as"](#)
- [Generics in .NET \(Genéricos no .NET\)](#)

# Operadores de conversão explícitos e implícitos definidos pelo usuário

Artigo • 07/04/2023

Um tipo definido pelo usuário pode definir uma conversão implícita ou explícita personalizada de outro tipo ou para outro. Conversões implícitas não exigem a invocação de sintaxe especial e podem ocorrer em várias situações, por exemplo, em invocações de atribuições e métodos. Conversões implícitas em C# predefinidas sempre têm êxito e nunca geram uma exceção. Conversões implícitas definidas pelo usuário devem se comportar dessa forma também. Se uma conversão personalizada puder gerar uma exceção ou perder informações, defina-a como uma conversão explícita.

As conversões definidas pelo usuário não são consideradas pelos operadores [is](#) e [como](#). Use uma [expressão de conversão](#) para invocar uma conversão explícita definida pelo usuário.

Use `operator` e as palavras-chave `implicit` ou `explicit` para definir uma conversão implícita ou explícita, respectivamente. O tipo que define uma conversão deve ser um tipo de origem ou destino dessa conversão. Uma conversão entre os dois tipos definidos pelo usuário pode ser definida em qualquer um dos dois tipos.

O exemplo a seguir demonstra como definir uma conversão implícita e explícita:

C#

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);

    public override string ToString() => $"{digit}";
}
```

```
public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number); // output: 7

        Digit digit = (Digit)number;
        Console.WriteLine(digit); // output: 7
    }
}
```

A partir do C# 11, você pode definir operadores de conversão explícita *verificados*. Saiba mais na seção [Operadores verificados definidos pelo usuário](#) do artigo [Operadores aritméticos](#).

Use também a palavra-chave `operator` para sobrepor um operador C# predefinido. Para obter mais informações, consulte [Sobrepor operador](#).

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Operadores de conversão](#)
- [Conversões definidas pelo usuário](#)
- [Conversões implícitas](#)
- [Conversões explícitas](#)

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Sobrepor operador](#)
- [Operadores cast e teste de tipo](#)
- [Conversão e conversão de tipo](#)
- [Diretrizes de design - Operadores de conversão](#)
- [Conversões explícitas encadeadas definidas pelo usuário em C#](#)

# Operadores relacionados ao ponteiro – pegue o endereço de variáveis, locais de armazenamento de desreferência e acesse locais de memória

Artigo • 07/04/2023

Os operadores de ponteiro permitem que você pegue o endereço de uma variável (&), desreferenciar um ponteiro (\*), comparar valores de ponteiro e adicionar ou subtrair ponteiros e inteiros.

Use os seguintes operadores para trabalhar com ponteiros:

- Operador unário [& \(address-of\)](#): para obter o endereço de uma variável
- Operador unário [\\* \(indireção de ponteiro\)](#): para obter a variável apontada por um ponteiro
- Os operadores [-> \(acesso de membro\)](#) e [\[\] \(acesso de elemento\)](#)
- Operadores aritméticos [+, -, ++ e --](#)
- Operadores de comparação [==, !=, <, >, <= e >=](#)

Para obter informações sobre tipos de ponteiros, veja [Tipos de ponteiro](#).

## ⓘ Observação

Qualquer operação com ponteiros exige um contexto `unsafe`. O código que contém blocos não seguros deve ser compilado com a opção do compilador `AllowUnsafeBlocks`.

## Operador address-of &

O operador unário `&` retorna o endereço de seu operando:

C#

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable: {&number}");
}
```

```
((long)pointerToNumber:X});  
}  
// Output is similar to:  
// Value of the variable: 27  
// Address of the variable: 6C1457DBD4
```

O operando do operador `&` deve ser uma variável fixa. Variáveis *fixas* são variáveis que residem em locais de armazenamento não afetados pela operação do [coletor de lixo](#). No exemplo anterior, a variável local `number` é uma variável fixa, pois reside na pilha. Variáveis que residem em locais de armazenamento que podem ser afetados pelo coletor de lixo (por exemplo, realocado) são chamadas de variáveis *móveis*. Campos de objeto e elementos de matriz são exemplos de variáveis móveis. Você poderá obter o endereço de uma variável móvel se a "firmar" ou "fixar" com uma instrução `fixed`. O endereço obtido é válido somente dentro do bloco de uma instrução `fixed`. O exemplo a seguir mostra como usar uma instrução `fixed` e o operador `&`:

C#

```
unsafe  
{  
    byte[] bytes = { 1, 2, 3 };  
    fixed (byte* pointerToFirst = &bytes[0])  
    {  
        // The address stored in pointerToFirst  
        // is valid only inside this fixed statement block.  
    }  
}
```

Não é possível obter o endereço de uma constante nem de um valor.

Para obter mais informações sobre variáveis fixas e móveis, veja a seção [Variáveis fixas e móveis](#) da [Especificação de linguagem C#](#).

O operador binário `&` computa o [AND lógico](#) de seus operandos Booleanos ou o [AND lógico bit a bit](#) de seus operandos integrais.

## Operador de indireção de ponteiro\*

O operador unário de indireção de ponteiro `*` obtém a variável para a qual o operando aponta. Também é conhecido como o operador de desreferenciar. O operando do operador `*` deve ser de um tipo de ponteiro.

C#

```

unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable:
{((long)pointerToLetter:X}"));

    *pointerToLetter = 'Z';
    Console.WriteLine($"Value of the `letter` variable after update:
{letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z

```

Não é possível aplicar o `*` operador a uma expressão do tipo `void*`.

O operador binário `*` computa o [produto](#) de seus operandos numéricos.

## Operador de acesso a membro do ponteiro ->

O operador `->` combina [indireção do ponteiro](#) e [acesso de membro](#). Ou seja, se `x` for um ponteiro do tipo `T*` e `y` for um membro acessível de `T`, uma expressão do formulário

C#

`x->y`

é equivalente a

C#

`(*x).y`

O exemplo a seguir demonstra o uso do operador `->`:

C#

```

public struct Coords
{
    public int X;
    public int Y;
    public override string ToString() => $"({X}, {Y})";

```

```
}

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
        Coords coords;
        Coords* p = &coords;
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString()); // output: (3, 4)
    }
}
```

Não é possível aplicar o `->` operador a uma expressão do tipo `void*`.

## Operador de acesso a elemento do ponteiro []

Para uma expressão `p` de um tipo de ponteiro, um acesso de elemento de ponteiro da forma `p[n]` é avaliado como `*(p + n)`, em que `n` deve ser do tipo implicitamente conversível em `int`, `uint`, `long` ou `ulong`. Para obter informações sobre o comportamento do operador `+` com ponteiros, veja a seção [Adição ou subtração de um valor integral para ou de um ponteiro](#).

O exemplo a seguir demonstra como acessar elementos da matriz com um ponteiro e o operador `[]`:

```
C#  
  
unsafe  
{  
    char* pointerToChars = stackalloc char[123];  
  
    for (int i = 65; i < 123; i++)  
    {  
        pointerToChars[i] = (char)i;  
    }  
  
    Console.Write("Uppercase letters: ");  
    for (int i = 65; i < 91; i++)  
    {  
        Console.Write(pointerToChars[i]);  
    }  
}  
// Output:  
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

No exemplo anterior, uma expressão `stackalloc` aloca um bloco de memória na pilha.

## ⓘ Observação

O operador de acesso de elemento de ponteiro não verifica se há erros fora dos limites.

Você não pode usar `[]` para acesso de elemento de ponteiro com uma expressão do tipo `void*`.

Você também pode usar o operador `[]` para [acesso de indexador ou elemento de matriz](#).

## Operadores aritméticos de ponteiro

Você pode executar as seguintes operações aritméticas com ponteiros:

- Adicionar ou subtrair um valor integral de ou para um ponteiro
- Subtrair dois ponteiros
- Incrementar ou decrementar um ponteiro

Não é possível executar essas operações com ponteiros do tipo `void*`.

Para obter informações sobre operações aritméticas com suporte com tipos numéricos, veja [Operadores aritméticos](#).

## Adição ou subtração de um valor integral a ou de um ponteiro

Para um ponteiro `p` do tipo `T*` e uma expressão `n` de um tipo implicitamente conversível em `int`, `uint`, `long` ou `ulong`, adição e subtração são definidas da seguinte maneira:

- As expressões `p + n` e `n + p` produzem um ponteiro do tipo `T*` que resulta da adição de `n * sizeof(T)` ao endereço fornecido pelo `p`.
- A expressão `p - n` produz um ponteiro do tipo `T*` que resulta da subtração de `n * sizeof(T)` ao endereço fornecido pelo `p`.

O [operador sizeof](#) obtém o tamanho de um tipo em bytes.

O exemplo a seguir demonstra o uso do operador `+` com um ponteiro:

C#

```

unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };
    fixed (int* pointerToFirst = &numbers[0])
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {*pointerToFirst} at address
{((long)pointerToFirst)}");
        Console.WriteLine($"Value {*pointerToLast} at address
{((long)pointerToLast)}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144

```

## Subtração de ponteiro

Para dois ponteiros `p1` e `p2` do tipo `T*`, a expressão `p1 - p2` produz a diferença entre os endereços dados por `p1` e `p2` divididos por `sizeof(T)`. O tipo de resultado é `long`. Ou seja, `p1 - p2` é computado como `((long)(p1) - (long)(p2)) / sizeof(T)`.

O exemplo a seguir demonstra a subtração de ponteiro:

C#

```

unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1); // output: 4
}

```

## Incrementar e decrementar ponteiros

O operador de incremento `++` adiciona 1 ao operando do ponteiro. O operador de decremento `--` subtrai 1 do operando do ponteiro.

Os dois operadores têm suporte em duas formas: sufixo (`p++` e `p--`) e prefixo (`++p` e `--p`). O resultado de `p++` e `p--` é o valor de `p` antes da operação. O resultado de `++p` e `--p` é o valor de `p` depois da operação.

O exemplo a seguir demonstra o comportamento dos operadores de incremento de sufixo e prefixo:

```
C#  
  
unsafe  
{  
    int* numbers = stackalloc int[] { 0, 1, 2 };  
    int* p1 = &numbers[0];  
    int* p2 = p1;  
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 -  
{((long)p2)}");  
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");  
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");  
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");  
}  
// Output is similar to  
// Before operation: p1 - 816489946512, p2 - 816489946512  
// Postfix increment of p1: 816489946512  
// Prefix increment of p2: 816489946516  
// After operation: p1 - 816489946516, p2 - 816489946516
```

## Operadores de comparação de ponteiro

Você pode usar os operadores `==`, `!=`, `<`, `>`, `<=` e `>=` para comparar os operandos de qualquer tipo de ponteiro, incluindo `void*`. Esses operadores comparam os endereços fornecidos pelos dois operandos como se fossem inteiros sem sinal.

Para obter informações sobre o comportamento desses operadores para operandos de outros tipos, veja os artigos [Operadores de igualdade](#) e [Operadores de comparação](#).

## Precedência do operador

A lista a seguir ordena operadores relacionados a ponteiro começando da precedência mais alta até a mais baixa:

- Operadores de incremento `x++` e decremento `x--` sufixados e os operadores `->` e `[ ]`
- Operadores de incremento `++x` e decremento `--x` prefixados e os operadores `&` e `*`
- Operadores de adição `+` e `-`
- Operadores de comparação `<`, `>`, `<=` e `>=`
- Operadores de igualdade `==` e `!=`

Use parênteses, `()`, para alterar a ordem de avaliação imposta pela precedência do operador.

Para obter a lista completa de operadores C# ordenados por nível de precedência, consulte a seção [Precedência de operador](#) do artigo [Operadores C#](#).

## Capacidade de sobrecarga do operador

Um tipo definido pelo usuário não pode sobrepor os operadores & relacionados ao ponteiro `*`, `->` e `[]`.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Variáveis fixas e móveis](#)
- [O operador address-of](#)
- [Indireção de ponteiro](#)
- [Acesso de membro do ponteiro](#)
- [Acesso de elemento do ponteiro](#)
- [Aritmética do ponteiro](#)
- [Incrementar e decrementar ponteiros](#)
- [Comparação de ponteiros](#)

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Código não seguro, tipos de ponteiro e ponteiros de função](#)
- [Palavra-chave unsafe](#)
- [Instrução fixed](#)
- [expressão stackalloc](#)
- [Operador sizeof](#)

# Operadores de atribuição (referência C#)

Artigo • 17/07/2023

O operador de atribuição `=` atribui o *valor* do operando do lado direito a uma variável, uma [propriedade](#) ou um elemento [indexador](#) fornecido pelo operando do lado esquerdo. O resultado de uma expressão de atribuição é o valor atribuído a um operando do lado esquerdo. O tipo do operandos do lado direito deve ser do mesmo tipo ou implicitamente conversível para o operando do lado esquerdo.

O operador de atribuição `=` é associativo direito, ou seja, uma expressão da forma

```
C#
```

```
a = b = c
```

é avaliada como

```
C#
```

```
a = (b = c)
```

O exemplo a seguir demonstra o uso do operador de atribuição com uma variável local, uma propriedade e um elemento do indexador como seu operando esquerdo:

```
C#
```

```
var numbers = new List<double>() { 1.0, 2.0, 3.0 };

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
```

```
// 1  
// 5
```

O operando à esquerda de uma atribuição recebe o *valor* do operando à direita.

Quando os operandos são de [tipos de valor](#), a atribuição copia o conteúdo do operando à direita. Quando os operandos são de [tipos de referência](#), a atribuição copia a referência ao objeto.

Isso é chamado de *atribuição de valor*: o valor é atribuído.

## atribuição de referência

A *atribuição de referência* = `ref` torna seu operando da esquerda um alias para o operando à direita, conforme demonstrado no exemplo a seguir:

C#

```
void Display(double[] s) => Console.WriteLine(string.Join(" ", s));  
  
double[] arr = { 0.0, 0.0, 0.0 };  
Display(arr);  
  
ref double arrayElement = ref arr[0];  
arrayElement = 3.0;  
Display(arr);  
  
arrayElement = ref arr[arr.Length - 1];  
arrayElement = 5.0;  
Display(arr);  
// Output:  
// 0 0 0  
// 3 0 0  
// 3 0 5
```

No exemplo anterior, a [variável de referência local](#) `arrayElement` é inicializada como um alias para o primeiro elemento da matriz. Em seguida, `ref` será reatribuído para se referir ao último elemento da matriz. Como é um alias, quando você atualiza seu valor com um operador = de atribuição comum, o elemento de matriz correspondente também é atualizado.

O operando esquerdo da atribuição `ref` pode ser uma [variável de referência local](#), um [ref campo](#) e um parâmetro de método `ref`, `out` ou `in`. Ambos os operandos devem ter o mesmo tipo.

# Atribuição composta

Para um operador binário `op`, uma expressão de atribuição composta do formato

```
C#
```

```
x op= y
```

é equivalente a

```
C#
```

```
x = x op y
```

exceto que `x` é avaliado apenas uma vez.

A atribuição composta é suportada por operadores aritmético, lógico booleano e bit a bit lógico e shift.

## Atribuição de avaliação de nulo

Você pode usar o operador de avaliação de nulo `??=` para atribuir o valor do seu operando direito ao operando esquerdo somente se o operando esquerdo for avaliado como `null`. Para obter mais informações, confira o artigo [Operadores ?? e ??=](#).

## Capacidade de sobrecarga do operador

Um tipo definido pelo usuário não pode [sobrecarregar](#) o operador de atribuição. No entanto, um tipo definido pelo usuário pode definir uma conversão implícita em outro tipo. Dessa forma, o valor de um tipo definido pelo usuário pode ser atribuído a uma variável, uma propriedade ou um elemento do indexador de outro tipo. Para saber mais, confira [Operadores de conversão definidos pelo usuário](#).

Um tipo definido pelo usuário não pode sobrecarregar explicitamente um operador de atribuição composta. No entanto, se um tipo definido pelo usuário sobrecarregar um operador binário `op`, o operador `op=`, se houver, também será implicitamente sobrecarregado.

## Especificação da linguagem C#

Saiba mais na seção [Operadores de atribuição](#) na [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [ref keyword](#)
- [Usar atribuição composta \(regras de estilo IDE0054 e IDE0074\)](#)

# Expressões lambda e funções anônimas

Artigo • 09/03/2023

Use uma *expressão lambda* para criar uma função anônima. Use o [operador de declaração lambda =>](#) para separar a lista de parâmetros de lambda do corpo. Uma expressão lambda pode ser de qualquer uma das duas formas a seguir:

- [Expressão lambda](#) que tem uma expressão como corpo:

C#

```
(input-parameters) => expression
```

- [Instrução lambda](#) que tem um bloco de instrução como corpo:

C#

```
(input-parameters) => { <sequence-of-statements> }
```

Para criar uma expressão lambda, especifique os parâmetros de entrada (se houver) no lado esquerdo do operador lambda e uma expressão ou um bloco de instrução do outro lado.

Qualquer expressão lambda pode ser convertida para um tipo [delegado](#). O tipo delegado no qual uma expressão lambda pode ser convertida é definido pelos tipos de parâmetros e pelo valor retornado. Se uma expressão lambda não retornar um valor, ela poderá ser convertida em um dos tipos delegados `Action`; caso contrário, ela poderá ser convertida em um dos tipos delegados `Func`. Por exemplo, uma expressão lambda que tem dois parâmetros e não retorna nenhum valor pode ser convertida em um delegado `Action<T1,T2>`. Uma expressão lambda que tem um parâmetro e retorna um valor pode ser convertida em um delegado `Func<T,TResult>`. No seguinte exemplo, a expressão lambda `x => x * x`, que especifica um parâmetro chamado `x` e retorna o valor de `x` quadrado, é atribuída a uma variável de um tipo delegado:

C#

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

As expressões lambdas também podem ser convertidas nos tipos de [árvore de expressão](#), como mostra o seguinte exemplo:

```
C#
```

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

Use expressões lambda em qualquer código que exija instâncias de tipos delegados ou árvores de expressão, por exemplo, como um argumento ao método [Task.Run\(Action\)](#) para passar o código que deve ser executado em segundo plano. Use também expressões lambda ao gravar [LINQ no C#](#), conforme mostrado no exemplo a seguir:

```
C#
```

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

Quando você usa a sintaxe baseada em método para chamar o método [Enumerable.Select](#) na classe [System.Linq.Enumerable](#), por exemplo, no LINQ to Objects e no LINQ to XML, o parâmetro é um tipo delegado [System.Func<T,TResult>](#). Quando você chama o método [Queryable.Select](#) na classe [System.Linq.Queryable](#), por exemplo, no LINQ to SQL, o tipo de parâmetro é um tipo de árvore de expressão [Expression<Func<TSource,TResult>>](#). Em ambos os casos, você pode usar a mesma expressão lambda para especificar o valor do parâmetro. Isso faz com que as duas chamadas `Select` pareçam semelhantes, embora, na verdade, o tipo de objetos criados das lambdas seja diferente.

## Lambdas de expressão

Uma expressão lambda com uma expressão no lado direito do operador `=>` é chamada de *lambda de expressão*. Uma expressão lambda retorna o resultado da expressão e tem o seguinte formato básico:

```
C#
```

```
(input-parameters) => expression
```

O corpo de um lambda de expressão pode consistir em uma chamada de método. No entanto, se você estiver criando [árvore de expressão](#) que serão avaliadas fora contexto do .NET CLR (Common Language Runtime), como no SQL Server, você não deverá usar chamadas de método em lambdas de expressão. Os métodos não terão significado fora do contexto do .NET CLR (Common Language Runtime).

## Lambdas de instrução

Um lambda de instrução é semelhante a um lambda de expressão, exceto que as instruções são colocadas entre chaves:

C#

```
(input-parameters) => { <sequence-of-statements> }
```

O corpo de uma instrução lambda pode consistir de qualquer número de instruções; no entanto, na prática, normalmente não há mais de duas ou três.

C#

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

Você não pode usar os lambdas de instrução para criar árvores de expressão.

## Parâmetros de entrada de uma expressão lambda

Coloque os parâmetros de entrada de uma expressão lambda entre parênteses. Especifique parâmetros de entrada zero com parênteses vazios:

C#

```
Action line = () => Console.WriteLine();
```

Se uma expressão lambda tiver apenas um parâmetro de entrada, os parênteses serão opcionais:

```
C#
```

```
Func<double, double> cube = x => x * x * x;
```

Dois ou mais parâmetros de entrada são separados por vírgulas:

```
C#
```

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Às vezes, o compilador não pode inferir os tipos de parâmetros de entrada. Você pode especificar os tipos de maneira explícita conforme mostrado neste exemplo:

```
C#
```

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Os tipos de parâmetro de entrada devem ser todos explícitos ou implícitos; caso contrário, ocorrerá o erro [CS0748](#) de compilador.

A partir do C# 9.0, é possível usar [descartes](#) para especificar dois ou mais parâmetros de entrada de uma expressão lambda, os quais não são usados na expressão:

```
C#
```

```
Func<int, int, int> constant = (_, _) => 42;
```

Os parâmetros de descarte do lambda podem ser úteis quando você usa uma expressão lambda para [fornecer um manipulador de eventos](#).

#### ⓘ Observação

Para compatibilidade com versões anteriores, se apenas um único parâmetro de entrada for chamado de `_`, em uma expressão lambda, `_` será tratado como o nome desse parâmetro.

## Lambdas assíncronos

Você pode facilmente criar expressões e instruções lambda que incorporem processamento assíncrono, ao usar as palavras-chaves `async` e `await`. Por exemplo, o exemplo do Windows Forms a seguir contém um manipulador de eventos que chama e espera um método assíncrono `ExampleMethodAsync`.

C#

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous
process.
        await Task.Delay(1000);
    }
}
```

Você pode adicionar o mesmo manipulador de eventos ao usar um lambda assíncrono. Para adicionar esse manipulador, adicione um modificador `async` antes da lista de parâmetros lambda, como mostra o exemplo a seguir:

C#

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event
handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
```

```
// The following line simulates a task-returning asynchronous
process.
    await Task.Delay(1000);
}
}
```

Para obter mais informações sobre como criar e usar os métodos assíncronos, consulte [Programação assíncrona com `async` e `await`](#).

## Expressões lambda e tuplas

A linguagem C# fornece suporte interno para [tuplas](#). Você pode fornecer uma tupla como um argumento para uma expressão lambda e a expressão lambda também pode retornar uma tupla. Em alguns casos, o compilador do C# usa a inferência de tipos para determinar os tipos dos componentes da tupla.

Você pode definir uma tupla, colocando entre parênteses uma lista delimitada por vírgulas de seus componentes. O exemplo a seguir usa a tupla com três componentes para passar uma sequência de números para uma expressão lambda, que dobra cada valor e retorna uma tupla com três componentes que contém o resultado das multiplicações.

C#

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 *
ns.Item2, 2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

Normalmente, os campos de uma tupla são chamados de `Item1`, `Item2` e assim por diante. No entanto, você pode definir uma tupla com componentes nomeados, como é feito no exemplo a seguir.

C#

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 *
ns.n1, 2 * ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

Para saber mais sobre as tuplas do C#, confira [Tipos de tuplas](#).

# Lambdas com os operadores de consulta padrão

O LINQ to Objects, entre outras implementações, tem um parâmetro de entrada cujo tipo faz parte da família de delegados genéricos `Func<TResult>`. Esses delegados usam parâmetros de tipo para definir o número e o tipo de parâmetros de entrada e o tipo de retorno do delegado. Delegados `Func` são úteis para encapsular expressões definidas pelo usuário aplicadas a cada elemento em um conjunto de dados de origem. Por exemplo, considere o seguinte tipo delegado `Func<T,TResult>`:

C#

```
public delegate TResult Func<in T, out TResult>(T arg)
```

O delegado pode ser instanciado como um `Func<int, bool>`, em que `int` é um parâmetro de entrada e `bool` é o valor de retorno. O valor de retorno é sempre especificado no último parâmetro de tipo. Por exemplo, `Func<int, string, bool>` define um delegado com dois parâmetros de entrada, `int` e `string`, e um tipo de retorno de `bool`. O delegado `Func` a seguir, quando é invocado, retornará um valor booleano que indica se o parâmetro de entrada é ou não igual a cinco:

C#

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result); // False
```

Você também pode fornecer uma expressão lambda quando o tipo de argumento é um `Expression<TDelegate>`. Por exemplo, nos operadores de consulta padrão que são definidos no tipo `Queryable`. Quando você especifica um argumento `Expression<TDelegate>`, o lambda é compilado em uma árvore de expressão.

O exemplo a seguir usa o operador padrão de consulta `Count`:

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ", numbers)}");
```

O compilador pode inferir o tipo de parâmetro de entrada ou você também pode especificá-lo explicitamente. Essa expressão lambda em particular conta esses inteiros ( $n$ ) que, quando dividida por dois, tem um resto 1.

O exemplo a seguir gera uma sequência que contém todos os elementos da matriz `numbers` que precedem o 9, porque esse é o primeiro número na sequência que não satisfaz a condição:

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

O exemplo a seguir especifica vários parâmetros de entrada, colocando-os entre parênteses. O método retorna todos os elementos na matriz `numbers` até encontrar um número cujo valor seja inferior à sua posição ordinal na matriz:

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

Você não usa expressões lambda diretamente em [expressões de consulta](#), mas pode usá-las em chamadas de método nas expressões de consulta, conforme mostrado no exemplo a seguir:

C#

```
var numberSets = new List<int[]>
{
    new[] { 1, 2, 3, 4, 5 },
    new[] { 0, 0, 0 },
    new[] { 9, 8 },
    new[] { 1, 0, 1, 0, 1, 0, 1, 0 }
};

var setsWithManyPositives =
    from numberSet in numberSets
    where numberSet.Count(n => n > 0) > 3
    select numberSet;

foreach (var numberSet in setsWithManyPositives)
```

```
{  
    Console.WriteLine(string.Join(" ", numberSet));  
}  
// Output:  
// 1 2 3 4 5  
// 1 0 1 0 1 0 1 0
```

## Inferência de tipos em expressões lambda

Ao escrever lambdas, você geralmente não precisa especificar um tipo para os parâmetros de entrada porque o compilador pode inferir o tipo com base no corpo do lambda, nos tipos de parâmetro e em outros fatores, conforme descrito na especificação da linguagem C#. Para a maioria dos operadores de consulta padrão, a primeira entrada é o tipo dos elementos na sequência de origem. Se você estiver consultando um `IEnumerable<Customer>`, a variável de entrada será inferida para ser um objeto `Customer`, o que significa que você terá acesso aos seus métodos e propriedades:

```
C#  
  
customers.Where(c => c.City == "London");
```

As regras gerais para a inferência de tipos para lambdas são as seguintes:

- O lambda deve conter o mesmo número de parâmetros do tipo delegado.
- Cada parâmetro de entrada no lambda deve ser implicitamente conversível em seu parâmetro delegado correspondente.
- O valor de retorno do lambda (se houver) deve ser implicitamente conversível para o tipo de retorno do delegado.

## Tipo natural de uma expressão lambda

Uma expressão lambda em si não tem um tipo, pois o sistema de tipo comum não tem conceitos intrínsecos de "expressão lambda". No entanto, às vezes convém falar informalmente do "tipo" de uma expressão lambda. Esse "tipo" informal se refere ao tipo delegado ou tipo `Expression` ao qual a expressão lambda é convertida.

A partir do C# 10, uma expressão lambda pode ter um *tipo natural*. Em vez de forçar você a declarar um tipo delegado, como `Func<...>` ou `Action<...>`, para uma expressão lambda, o compilador pode inferir o tipo delegado da expressão lambda. Por exemplo, considere a seguinte declaração:

```
C#
```

```
var parse = (string s) => int.Parse(s);
```

O compilador pode inferir `parse` para ser um `Func<string, int>`. O compilador escolhe um delegado `Func` ou `Action` disponível, se houver um adequado. Caso contrário, ele sintetiza um tipo delegado. Por exemplo, o tipo delegado será sintetizado se a expressão lambda tiver os parâmetros `ref`. Quando uma expressão lambda tem um tipo natural, pode ser atribuída a um tipo menos explícito, como `System.Object` ou `System.Delegate`:

C#

```
object parse = (string s) => int.Parse(s); // Func<string, int>
Delegate parse = (string s) => int.Parse(s); // Func<string, int>
```

Os grupos de métodos (ou seja, nomes de método sem listas de parâmetros) com exatamente uma sobrecarga têm um tipo natural:

C#

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

Se você atribuir uma expressão lambda a `System.Linq.Expressions.LambdaExpression` ou `System.Linq.Expressions.Expression` e o lambda tiver um tipo delegado natural, a expressão terá um tipo natural de `System.Linq.Expressions.Expression<TDelegate>`, e o tipo delegado natural será usado como o argumento para o parâmetro de tipo:

C#

```
LambdaExpression parseExpr = (string s) => int.Parse(s); //
Expression<Func<string, int>>
Expression parseExpr = (string s) => int.Parse(s);           //
Expression<Func<string, int>>
```

Nem todas as expressões lambda têm um tipo natural. Considere a seguinte declaração:

C#

```
var parse = s => int.Parse(s); // ERROR: Not enough type info in the lambda
```

O compilador não pode inferir um tipo de parâmetro para `s`. Quando o compilador não pode inferir um tipo natural, você deve declarar o tipo:

```
C#
```

```
Func<string, int> parse = s => int.Parse(s);
```

## Tipo de retorno explícito

Normalmente, o tipo de retorno de uma expressão lambda é óbvio e inferido. Para algumas expressões, isso não dá certo:

```
C#
```

```
var choose = (bool b) => b ? 1 : "two"; // ERROR: Can't infer return type
```

A partir do C# 10, você pode especificar o tipo de retorno de uma expressão lambda, antes dos parâmetros de entrada. Ao especificar um tipo de retorno explícito, você deve colocar entre parênteses os parâmetros de entrada:

```
C#
```

```
var choose = object (bool b) => b ? 1 : "two"; // Func<bool, object>
```

## Atributos

A partir do C# 10, você pode adicionar atributos a uma expressão lambda e seus parâmetros. O exemplo a seguir mostra como adicionar atributos a uma expressão lambda:

```
C#
```

```
Func<string?, int?> parse = [ProvidesNullCheck] (s) => (s is not null) ?  
    int.Parse(s) : null;
```

Você também pode adicionar atributos aos parâmetros de entrada ou ao valor de retorno, conforme mostrado no exemplo a seguir:

```
C#
```

```
var concat = ([DisallowNull] string a, [DisallowNull] string b) => a + b;  
var inc = [return: NotNullifNotNull(nameof(s))] (int? s) => s.HasValue ? s++  
    : null;
```

Conforme mostrado nos exemplos anteriores, você deve colocar entre parênteses os parâmetros de entrada, ao adicionar atributos a uma expressão lambda ou seus parâmetros.

### ⓘ Importante

As expressões lambda são invocadas por meio do tipo delegado subjacente. É diferente dos métodos e funções locais. O método `Invoke` do delegado não verifica os atributos na expressão lambda. Os atributos não têm efeito quando a expressão lambda é invocada. Atributos das expressões lambda são úteis para análise de código e podem ser descobertos por meio da reflexão. Uma consequência dessa decisão é que `System.Diagnostics.ConditionalAttribute` não pode ser aplicado a uma expressão lambda.

## Captura de variáveis externas e escopo variável em expressões lambda

Os lambdas podem fazer referência a *variáveis externas*. Essas *variáveis externas* são as variáveis que estão no escopo do método que define a expressão lambda ou no escopo do tipo que contém a expressão lambda. As variáveis que são capturadas dessa forma são armazenadas para uso na expressão lambda mesmo que de alguma outra forma elas saíssem do escopo e fossem coletadas como lixo. Uma variável externa deve ser definitivamente atribuída para que possa ser consumida em uma expressão lambda. O exemplo a seguir demonstra estas regras:

C#

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int>? updateCapturedLocalVariable;
        internal Func<int, bool>? isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };
        }
    }
}
```

```

        isEqualToCapturedLocalVariable = x => x == j;

        Console.WriteLine($"Local variable before lambda invocation:
{j}");
        updateCapturedLocalVariable(10);
        Console.WriteLine($"Local variable after lambda invocation:
{j}");
    }

}

public static void Main()
{
    var game = new VariableCaptureGame();

    int gameInput = 5;
    game.Run(gameInput);

    int jTry = 10;
    bool result = game.isEqualToCapturedLocalVariable!(jTry);
    Console.WriteLine($"Captured local variable is equal to {jTry}:
{result}");

    int anotherJ = 3;
    game.updateCapturedLocalVariable!(anotherJ);

    bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
    Console.WriteLine($"Another lambda observes a new value of captured
variable: {equalToAnother}");
}
// Output:
// Local variable before lambda invocation: 0
// 10 is greater than 5: True
// Local variable after lambda invocation: 10
// Captured local variable is equal to 10: True
// 3 is greater than 5: False
// Another lambda observes a new value of captured variable: True
}

```

As seguintes regras se aplicam ao escopo variável em expressões lambda:

- Uma variável capturada não será coletada do lixo até que o delegado que faz referência a ela se qualifique para coleta de lixo.
- As variáveis introduzidas em uma expressão lambda não são visíveis no método delimitador.
- Uma expressão lambda não pode capturar um parâmetro `in`, `ref` ou `out` diretamente de um método delimitador.
- Uma instrução `return` em uma expressão lambda não faz com que o método delimitador retorne.

- Uma expressão lambda não pode conter uma instrução `goto`, `break` ou `continue` se o destino daquela instrução de salto estiver fora do bloco da expressão lambda. Também será um erro ter uma instrução de salto fora do bloco da expressão lambda se o destino estiver dentro do bloco.

A partir do C# 9.0, você pode aplicar o modificador `static` a uma expressão lambda, para evitar a captura não intencional de variáveis locais ou do estado da instância pelo lambda:

C#

```
Func<double, double> square = static x => x * x;
```

Um lambda estático não pode capturar variáveis locais ou o estado da instância dos escopos delimitadores, mas pode fazer referência a membros estáticos e definições constantes.

## Especificação da linguagem C#

Para obter mais informações, confira a seção [Expressões de função anônima](#) da [Especificação da linguagem C#](#).

Para obter mais informações sobre os recursos adicionados ao C# 9.0 e posteriores, confira as notas sobre a proposta do recurso a seguir:

- [Parâmetros de descarte do lambda \(C# 9.0\)](#)
- [Funções estáticas anônimas \(C# 9.0\)](#)
- [Melhorias do lambda \(C# 10\)](#)

## Confira também

- [Usar função local em vez do lambda \(regra de estilo IDE0039\)](#)
- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [LINQ \(Consulta Integrada à Linguagem\)](#)
- [Árvores de expressão](#)
- [Funções locais vs. expressões lambda](#)
- [Consultas de exemplo do LINQ ↗](#)
- [Exemplo do XQuery ↗](#)
- [101 exemplos do LINQ](#)

# Padrões correspondentes: as expressões `is`, `e switch`, e os operadores `and`, `or` e `not` em padrões

Artigo • 28/07/2023

Você usa a [expressão `is`](#), a [instrução `switch`](#) e a [expressão `switch`](#) para fazer a correspondência de uma expressão de entrada com qualquer número de características. O C# dá suporte a vários padrões, incluindo declaração, tipo, constante, relacional, propriedade, lista, var e descarte. Os padrões podem ser combinados usando palavras-chave da lógica booleana `and`, `or` e `not`.

As seguintes expressões e instruções C# dão suporte à correspondência de padrões:

- [Expressão `is`](#)
- [instrução `switch`](#)
- [expressão `switch`](#)

Nesses constructos, você pode corresponder uma expressão de entrada com qualquer um dos seguintes padrões:

- [Padrão de declaração](#): para verificar o tipo de tempo de execução de uma expressão e, se uma correspondência for bem-sucedida, atribua um resultado de expressão a uma variável declarada.
- [Padrão de tipo](#): para verificar o tipo de tempo de execução de uma expressão. Introduzido no C# 9.0.
- [Padrão constante](#): para testar se um resultado de expressão é igual a uma constante especificada.
- [Padrões relacionais](#): para comparar um resultado de expressão com uma constante especificada. Introduzido no C# 9.0.
- [Padrões lógicos](#): para testar se uma expressão corresponde a uma combinação lógica de padrões. Introduzido no C# 9.0.
- [Padrão de propriedade](#): para testar se as propriedades ou campos de uma expressão correspondem a padrões aninhados.
- [Padrão posicional](#): para desconstruir um resultado de expressão e testar se os valores resultantes correspondem a padrões aninhados.
- [var padrão](#): para corresponder a qualquer expressão e atribuir seu resultado a uma variável declarada.
- [Descartar padrão](#): para corresponder a qualquer expressão.

- **Padrões de lista:** para testar se os elementos de sequência correspondem aos padrões aninhados correspondentes. Introduzido em C# 11.

[Logic](#), [property](#), [positionale](#) [list](#) são padrões *recursivos*. Ou seja, eles podem conter padrões *aninhados*.

Para obter o exemplo de como usar esses padrões para criar um algoritmo controlado por dados, consulte [Tutorial: Usar a correspondência de padrões para criar algoritmos controlados por tipo e controlados por dados](#).

## Padrões de declaração e tipo

Você usa padrões de declaração e tipo para verificar se o tipo de tempo de execução de uma expressão é compatível com um determinado tipo. Com um padrão de declaração, você também pode declarar uma nova variável local. Quando um padrão de declaração corresponde a uma expressão, essa variável recebe um resultado de expressão convertida, como mostra o exemplo a seguir:

C#

```
object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower()); // output: hello, world!
}
```

Um *padrão de declaração* com tipo `T` corresponde a uma expressão quando um resultado de expressão não é nulo e qualquer uma das seguintes condições é verdadeira:

- O tipo do resultado de uma expressão em tempo de execução é `T`.
- O tipo do resultado de uma expressão em tempo de execução deriva do tipo `T`, implementa a interface `T` ou existe outra [conversão de referência implícita](#) dele para `T`. O exemplo a seguir demonstra dois casos em que essa condição é verdadeira:

C#

```
var numbers = new int[] { 10, 20, 30 };
Console.WriteLine(GetSourceLabel(numbers)); // output: 1

var letters = new List<char> { 'a', 'b', 'c', 'd' };
Console.WriteLine(GetSourceLabel(letters)); // output: 2
```

```
static int GetSourceLabel<T>(IEnumerable<T> source) => source switch
{
    Array array => 1,
    ICollection<T> collection => 2,
    _ => 3,
};
```

No exemplo anterior, na primeira chamada ao método `GetSourceLabel`, o primeiro padrão corresponde a um valor de argumento porque o tipo `int[]` de tempo de execução do argumento deriva do tipo `Array`. Na segunda chamada para o método `GetSourceLabel`, o tipo `List<T>` de tempo de execução do argumento não deriva do tipo `Array`, mas implementa a interface `ICollection<T>`.

- O tipo do resultado de uma expressão em tempo de execução é um [tipo de valor anulável](#) com o tipo subjacente `T`.
- Existe uma [conversão boxing](#) ou uma [conversão unboxing](#) do tipo do resultado de uma expressão em tempo de execução para o tipo `T`.

O exemplo a seguir demonstra as duas últimas condições:

C#

```
int? xNullable = 7;
int y = 23;
object yBoxed = y;
if (xNullable is int a && yBoxed is int b)
{
    Console.WriteLine(a + b); // output: 30
}
```

Se você quiser verificar apenas o tipo de uma expressão, poderá usar um descarte `_` no lugar do nome de uma variável, como mostra o exemplo a seguir:

C#

```
public abstract class Vehicle {}
public class Car : Vehicle {}
public class Truck : Vehicle {}

public static class TollCalculator
{
    public static decimal CalculateToll(this Vehicle vehicle) => vehicle
    switch
    {
        Car _ => 2.00m,
        Truck _ => 7.50m,
        null => throw new ArgumentNullException(nameof(vehicle)),
```

```
    _ => throw new ArgumentException("Unknown type of a vehicle",
nameof(vehicle)),
};

}
```

A partir do C# 9.0, para essa finalidade, você pode usar um *padrão de tipo*, como mostra o exemplo a seguir:

C#

```
public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
{
    Car => 2.00m,
    Truck => 7.50m,
    null => throw new ArgumentNullException(nameof(vehicle)),
    _ => throw new ArgumentException("Unknown type of a vehicle",
nameof(vehicle)),
};
```

Como um padrão de declaração, um padrão de tipo corresponde a uma expressão quando um resultado de expressão não é nulo e seu tipo de tempo de execução atende a qualquer uma das condições listadas acima.

Para verificar se há não nulo, você pode usar um padrão **negated null constant**, como mostra o exemplo a seguir:

C#

```
if (input is not null)
{
    // ...
}
```

Para obter mais informações, consulte as seções [padrão declaração](#) e [padrão de tipo](#) das notas de proposta do recurso.

## Padrão de constante

Você usa um *padrão constante* para testar se um resultado de expressão é igual a uma constante especificada, como mostra o exemplo a seguir:

C#

```
public static decimal GetGroupTicketPrice(int visitorCount) => visitorCount
switch
{
```

```
1 => 12.0m,  
2 => 20.0m,  
3 => 27.0m,  
4 => 32.0m,  
0 => 0.0m,  
_ => throw new ArgumentException($"Not supported number of visitors:  
{visitorCount}", nameof(visitorCount)),  
};
```

Em um padrão constante, você pode usar qualquer expressão constante, como:

- um literal numérico de [inteiro](#) ou [ponto flutuante](#)
- um [char](#)
- um literal de [cadeia de caracteres](#).
- um valor booleano [true](#) ou [false](#)
- um valor [enum](#)
- o nome de um campo [const](#) declarado ou local
- [null](#)

A expressão deve ser um tipo que seja conversível para o tipo constante, com uma exceção: uma expressão cujo tipo é [Span<char>](#) ou [ReadOnlySpan<char>](#) pode ser correspondida com cadeias de caracteres constantes em C# 11 e versões posteriores.

Use um padrão constante para verificar [null](#), como mostra o exemplo a seguir:

C#

```
if (input is null)  
{  
    return;  
}
```

O compilador garante que nenhum operador de igualdade sobrecarregado pelo usuário [==](#) seja invocado quando a expressão [x is null](#) for avaliada.

A partir do C# 9.0, você pode usar um padrão de constante [negado](#)[null](#) para verificar se não é nulo, como mostra o exemplo a seguir:

C#

```
if (input is not null)  
{  
    // ...  
}
```

Para obter mais informações, consulte a seção [Padrão constante](#) da nota da proposta do recurso.

## Padrões relacionais

A partir do C# 9.0, você usa um *padrão relacional* para comparar um resultado de expressão com uma constante, como mostra o exemplo a seguir:

```
C#  
  
Console.WriteLine(Classify(13)); // output: Too high  
Console.WriteLine(Classify(double.NaN)); // output: Unknown  
Console.WriteLine(Classify(2.4)); // output: Acceptable  
  
static string Classify(double measurement) => measurement switch  
{  
    < -4.0 => "Too low",  
    > 10.0 => "Too high",  
    double.NaN => "Unknown",  
    _ => "Acceptable",  
};
```

Em um padrão relacional, você pode usar qualquer um dos operadores relacionais `<`, `>`, `<=` ou `>=`. A parte direita de um padrão relacional deve ser uma expressão constante. A expressão constante pode ser de um tipo [inteiro](#), [ponto flutuante](#), [char](#) ou [enumerado](#).

Para verificar se um resultado de expressão está em um determinado intervalo, corresponda-o a um [padrão conjuntivo](#) and, como mostra o exemplo a seguir:

```
C#  
  
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 3, 14))); // output:  
spring  
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 7, 19))); // output:  
summer  
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 2, 17))); // output:  
winter  
  
static string GetCalendarSeason(DateTime date) => date.Month switch  
{  
    >= 3 and < 6 => "spring",  
    >= 6 and < 9 => "summer",  
    >= 9 and < 12 => "autumn",  
    12 or (>= 1 and < 3) => "winter",  
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with  
unexpected month: {date.Month}.")  
};
```

Se um resultado de expressão for `null` ou não for convertido no tipo de uma constante por uma conversão anulável ou de unboxing, um padrão relacional não corresponderá a uma expressão.

Para obter mais informações, consulte a seção [Padrões relacionais](#) da nota da proposta do recurso.

## Padrões lógicos

A partir do C# 9.0, você usa os combinadores de padrões `not`, `and` e `or` para criar os seguintes *padrões lógicos*:

- *Negação not* padrão que corresponde a uma expressão quando o padrão negado não corresponde à expressão. O exemplo a seguir mostra como você pode negar um padrão `constante null` para verificar se uma expressão não é nula:

C#

```
if (input is not null)
{
    // ...
}
```

- *Negação and* padrão que corresponde a uma expressão quando o padrão negado não corresponde à expressão. O exemplo a seguir mostra como você pode combinar [padrões relacionais](#) para verificar se um valor está em um determinado intervalo:

C#

```
Console.WriteLine(Classify(13)); // output: High
Console.WriteLine(Classify(-100)); // output: Too low
Console.WriteLine(Classify(5.7)); // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -40.0 => "Too low",
    >= -40.0 and < 0 => "Low",
    >= 0 and < 10.0 => "Acceptable",
    >= 10.0 and < 20.0 => "High",
    >= 20.0 => "Too high",
    double.NaN => "Unknown",
};
```

- **Disjuntivo or** padrão que corresponde a uma expressão quando um dos padrões corresponde à expressão, como mostra o exemplo a seguir:

```
C#  
  
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 1, 19))); //  
output: winter  
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 10, 9))); //  
output: autumn  
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 5, 11))); //  
output: spring  
  
static string GetCalendarSeason(DateTime date) => date.Month switch  
{  
    3 or 4 or 5 => "spring",  
    6 or 7 or 8 => "summer",  
    9 or 10 or 11 => "autumn",  
    12 or 1 or 2 => "winter",  
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date  
with unexpected month: {date.Month}.")  
};
```

Como mostra o exemplo anterior, você pode usar repetidamente os combinadores de padrão em um padrão.

## Precedência e ordem da avaliação

Os combinadores de padrão são ordenados da precedência mais alta para a mais baixa da seguinte maneira:

- **not**
- **and**
- **or**

Quando um padrão lógico for um padrão de uma **is** expressão, a precedência de combinadores de padrões lógicos é **maior** do que a precedência dos operadores lógicos (operadores **lógicos bit a bit** e **boolianos lógicos**). Caso contrário, a precedência dos combinadores de padrões lógicos é **menor** do que a precedência de operadores lógicos e lógicos condicionais. Para obter a lista completa de operadores C# ordenados por nível de precedência, consulte a seção [Precedência de operador](#) do artigo [Operadores C#](#).

Para especificar explicitamente a precedência, use parênteses, como mostra o exemplo a seguir:

```
C#
```

```
static bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');
```

### ⓘ Observação

A ordem na qual os padrões são verificados é indefinida. Em tempo de execução, os padrões aninhados à direita dos padrões `or` e `and` podem ser verificados primeiro.

Para obter mais informações, consulte a seção [Padrão constante](#) da nota da proposta do recurso.

## Padrão de propriedade

Você usa um *padrão de propriedade* para corresponder às propriedades ou campos de uma expressão com padrões aninhados, como mostra o exemplo a seguir:

C#

```
static bool IsConferenceDay(DateTime date) => date is { Year: 2020, Month: 5, Day: 19 or 20 or 21 };
```

Um padrão de propriedade corresponde a uma expressão quando um resultado de expressão não é nulo e cada padrão aninhado corresponde à propriedade ou campo correspondente do resultado da expressão.

Você também pode adicionar uma verificação de tipo em tempo de execução e uma declaração de variável a um padrão de propriedade, como mostra o exemplo a seguir:

C#

```
Console.WriteLine(TakeFive("Hello, world!")); // output: Hello
Console.WriteLine(TakeFive("Hi!")); // output: Hi!
Console.WriteLine(TakeFive(new[] { '1', '2', '3', '4', '5', '6', '7' }));
// output: 12345
Console.WriteLine(TakeFive(new[] { 'a', 'b', 'c' })); // output: abc

static string TakeFive(object input) => input switch
{
    string { Length: >= 5 } s => s.Substring(0, 5),
    string s => s,
    ICollection<char> { Count: >= 5 } symbols => new
        string(symbols.Take(5).ToArray()),
```

```
ICollection<char> symbols => new string(symbols.ToArray()),  
    null => throw new ArgumentNullException(nameof(input)),  
    _ => throw new ArgumentException("Not supported input type."),  
};
```

Um padrão de propriedade é um padrão recursivo. Ou seja, você pode usar qualquer padrão como aninhado. Use um padrão de propriedade para corresponder partes de dados com padrões aninhados, como mostra o exemplo a seguir:

C#

```
public record Point(int X, int Y);  
public record Segment(Point Start, Point End);  
  
static bool IsAnyEndOnXAxis(Segment segment) =>  
    segment is { Start: { Y: 0 } } or { End: { Y: 0 } };
```

O exemplo anterior usa dois recursos disponíveis no C# 9.0 e posterior: [or combinador de padrões](#) e [tipos de registro](#).

A partir do C# 10, você pode referenciar propriedades aninhadas ou campos dentro de um padrão de propriedade. Essa capacidade é conhecida como um *padrão de propriedade estendido*. Por exemplo, você pode refatorar o método do exemplo anterior para o seguinte código equivalente:

C#

```
static bool IsAnyEndOnXAxis(Segment segment) =>  
    segment is { Start.Y: 0 } or { End.Y: 0 };
```

Para obter mais informações, consulte a seção [Padrão de propriedade](#) da nota de proposta do recurso e a nota de proposta de recurso [Padrões de propriedade estendida](#).

### 💡 Dica

Você pode usar a regra de estilo [Simplificar padrão de propriedade \(IDE0170\)](#) para melhorar a legibilidade do código sugerindo locais para usar padrões de propriedade estendidos.

## Padrão posicional

Você usa um *padrão posicional* para desconstruir um resultado de expressão e corresponder aos valores resultantes com os padrões aninhados correspondentes, como mostra o exemplo a seguir:

C#

```
public readonly struct 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);
}

static string Classify(Point point) => point switch
{
    (0, 0) => "Origin",
    (1, 0) => "positive X basis end",
    (0, 1) => "positive Y basis end",
    _ => "Just a point",
};
```

No exemplo anterior, o tipo de expressão contém o método [Deconstrução](#), que é usado para desconstruir um resultado de expressão. Você também pode corresponder expressões de [tipos de tupla](#) com padrões posicionais. Dessa forma, você pode corresponder várias entradas com vários padrões, como mostra o exemplo a seguir:

C#

```
static decimal GetGroupTicketPriceDiscount(int groupSize, DateTime
visitDate)
=> (groupSize, visitDate.DayOfWeek) switch
{
    (<= 0, _) => throw new ArgumentException("Group size must be
positive."),
    (_, DayOfWeek.Saturday or DayOfWeek.Sunday) => 0.0m,
    (>= 5 and < 10, DayOfWeek.Monday) => 20.0m,
    (>= 10, DayOfWeek.Monday) => 30.0m,
    (>= 5 and < 10, _) => 12.0m,
    (>= 10, _) => 15.0m,
    _ => 0.0m,
};
```

O exemplo anterior usa padrões [relacionais](#) e [lógicos](#), que estão disponíveis no C# 9.0 e posterior.

Você pode usar os nomes de elementos de tupla e parâmetros `Deconstruct` em um padrão posicional, como mostra o exemplo a seguir:

```
C#  
  
var numbers = new List<int> { 1, 2, 3 };  
if (SumAndCount(numbers) is (Sum: var sum, Count: > 0))  
{  
    Console.WriteLine($"Sum of [{string.Join(" ", numbers)}] is {sum}"); //  
    output: Sum of [1 2 3] is 6  
}  
  
static (double Sum, int Count) SumAndCount(IEnumerable<int> numbers)  
{  
    int sum = 0;  
    int count = 0;  
    foreach (int number in numbers)  
    {  
        sum += number;  
        count++;  
    }  
    return (sum, count);  
}
```

Você também pode estender um padrão posicional de qualquer uma das seguintes maneiras:

- Adicione uma verificação de tipo de tempo de execução e uma declaração de variável, como mostra o exemplo a seguir:

```
C#  
  
public record Point2D(int X, int Y);  
public record Point3D(int X, int Y, int Z);  
  
static string PrintIfAllCoordinatesArePositive(object point) => point  
switch  
{  
    Point2D (> 0, > 0) p => p.ToString(),  
    Point3D (> 0, > 0, > 0) p => p.ToString(),  
    _ => string.Empty,  
};
```

O exemplo anterior usa [registros posicionais](#) que fornecem implicitamente o método `Deconstruct`.

- Use um [padrão de propriedade](#) dentro de um padrão posicional, como mostra o exemplo a seguir:

C#

```
public record WeightedPoint(int X, int Y)
{
    public double Weight { get; set; }
}

static bool IsInDomain(WeightedPoint point) => point is (>= 0, >= 0) {
    Weight: >= 0.0 };
```

- Combine dois usos anteriores, como mostra o exemplo a seguir:

C#

```
if (input is WeightedPoint (> 0, > 0) { Weight: > 0.0 } p)
{
    // ..
}
```

Um padrão posicional é um padrão recursivo. Ou seja, você pode usar qualquer padrão como aninhado.

Para obter mais informações, consulte a seção [Padrão constante](#) da nota da proposta do recurso.

## Padrão var

Você usa um `var` padrão para corresponder a qualquer expressão, incluindo `null`, e atribuir seu resultado a uma nova variável local, como mostra o exemplo a seguir:

C#

```
static bool IsAcceptable(int id, int absLimit) =>
    SimulateDataFetch(id) is var results
    && results.Min() >= -absLimit
    && results.Max() <= absLimit;

static int[] SimulateDataFetch(int id)
{
    var rand = new Random();
    return Enumerable
        .Range(start: 0, count: 5)
        .Select(s => rand.Next(minValue: -10, maxValue: 11))
        .ToArray();
}
```

Um padrão `var` é útil quando você precisa de uma variável temporária dentro de uma expressão booliana para manter o resultado de cálculos intermediários. Você também poderá usar um padrão `var` quando precisar realizar mais verificações em proteções de caso `when` de uma expressão ou instrução `switch`, como mostra o exemplo a seguir:

```
C#  
  
public record Point(int X, int Y);  
  
static Point Transform(Point point) => point switch  
{  
    var (x, y) when x < y => new Point(-x, y),  
    var (x, y) when x > y => new Point(x, -y),  
    var (x, y) => new Point(x, y),  
};  
  
static void TestTransform()  
{  
    Console.WriteLine(Transform(new Point(1, 2))); // output: Point { X =  
    -1, Y = 2 }  
    Console.WriteLine(Transform(new Point(5, 2))); // output: Point { X =  
    5, Y = -2 }  
}
```

No exemplo anterior, o padrão `var (x, y)` é equivalente a um [padrão posicional](#) (`var x, var y`).

Em um padrão `var`, o tipo de uma variável declarada é o tipo de tempo de compilação da expressão que corresponde ao padrão.

Para obter mais informações, consulte a seção [Padrão constante](#) da nota da proposta do recurso.

## Padrão de descarte

Você usa um *padrão de descarte* para corresponder a qualquer expressão, incluindo `null`, conforme mostra o exemplo a seguir:

```
C#  
  
Console.WriteLine(GetDiscountInPercent(DayOfWeek.Friday)); // output: 5.0  
Console.WriteLine(GetDiscountInPercent(null)); // output: 0.0  
Console.WriteLine(GetDiscountInPercent((DayOfWeek)10)); // output: 0.0  
  
static decimal GetDiscountInPercent(DayOfWeek? dayOfWeek) => dayOfWeek  
switch  
{
```

```
DayOfWeek.Monday => 0.5m,  
DayOfWeek.Tuesday => 12.5m,  
DayOfWeek.Wednesday => 7.5m,  
DayOfWeek.Thursday => 12.5m,  
DayOfWeek.Friday => 5.0m,  
DayOfWeek.Saturday => 2.5m,  
DayOfWeek.Sunday => 2.0m,  
_ => 0.0m,  
};
```

Um `null` padrão de descarte [DayOfWeek](#): para manipular qualquer valor inteiro que não tenha o membro correspondente da enumeração (por exemplo, `_`). Isso garante que uma expressão `switch` no exemplo manipule todos os valores de entrada possíveis. Se você não usar um padrão de descarte em uma expressão `switch` e nenhum dos padrões da expressão corresponder a uma entrada, o runtime [gerará uma exceção](#). O compilador gera um aviso se uma expressão `switch` não manipular todos os valores de entrada possíveis.

Um padrão de descarte não pode ser um padrão em uma expressão `is` ou uma instrução `switch`. Nesses casos, para corresponder a qualquer expressão, use um [var padrão](#) com um descarte: `var _`. Um padrão de descarte não pode ser um padrão em uma `switch` expressão.

Para obter mais informações, consulte a seção [Padrão constante](#) da nota da proposta do recurso.

## Padrão entre parênteses

A partir do C# 9.0, você pode colocar parênteses em torno de qualquer padrão. Normalmente, você faz isso para enfatizar ou alterar a precedência em [padrões lógicos](#), como mostra o exemplo a seguir:

C#

```
if (input is not (float or double))  
{  
    return;  
}
```

## Padrões de lista

A partir do C# 11, você pode fazer a correspondência de uma matriz ou de uma lista com uma *sequência* como mostra o exemplo a seguir:

C#

```
int[] numbers = { 1, 2, 3 };

Console.WriteLine(numbers is [1, 2, 3]); // True
Console.WriteLine(numbers is [1, 2, 4]); // False
Console.WriteLine(numbers is [1, 2, 3, 4]); // False
Console.WriteLine(numbers is [0 or 1, <= 2, >= 3]); // True
```

Como mostra o exemplo anterior, um padrão de lista é correspondido quando cada padrão aninhado é correspondido pelo elemento correspondente de uma sequência de entrada. Você pode usar qualquer padrão dentro de um padrão de lista. Para corresponder a qualquer elemento, use o [padrão discard](#) ou, se você também quiser capturar o elemento, o [padrão var](#), como mostra o exemplo a seguir:

C#

```
List<int> numbers = new() { 1, 2, 3 };

if (numbers is [var first, _, _])
{
    Console.WriteLine($"The first element of a three-item list is
{first}.");
}
// Output:
// The first element of a three-item list is 1.
```

Os exemplos anteriores correspondem a uma sequência de entrada inteira em relação a um padrão de lista. Para corresponder elementos somente no início ou/e no final de uma sequência de entrada, use o [padrão de fatia ..](#) como mostra o seguinte exemplo:

C#

```
Console.WriteLine(new[] { 1, 2, 3, 4, 5 } is [> 0, > 0, ...]); // True
Console.WriteLine(new[] { 1, 1 } is [..., ...]); // True
Console.WriteLine(new[] { 0, 1, 2, 3, 4 } is [> 0, > 0, ...]); // False
Console.WriteLine(new[] { 1 } is [1, 2, ...]); // False

Console.WriteLine(new[] { 1, 2, 3, 4 } is [..., > 0, > 0]); // True
Console.WriteLine(new[] { 2, 4 } is [..., > 0, 2, 4]); // False
Console.WriteLine(new[] { 2, 4 } is [..., 2, 4]); // True

Console.WriteLine(new[] { 1, 2, 3, 4 } is [>= 0, ..., 2 or 4]); // True
Console.WriteLine(new[] { 1, 0, 0, 1 } is [1, 0, ..., 0, 1]); // True
Console.WriteLine(new[] { 1, 0, 1 } is [1, 0, ..., 0, 1]); // False
```

Um padrão de fatia corresponde a zero ou mais elementos. Você pode usar no máximo um padrão de fatia em um padrão de lista. O padrão de fatia só pode aparecer em um padrão de lista.

Você também pode aninhar um subpadrão dentro de um padrão de fatia, como mostra o exemplo a seguir:

C#

```
void MatchMessage(string message)
{
    var result = message is ['a' or 'A', .. var s, 'a' or 'A']
        ? $"Message {message} matches; inner part is {s}."
        : $"Message {message} doesn't match.";
    Console.WriteLine(result);
}

MatchMessage("aBBA"); // output: Message aBBA matches; inner part is BB.
MatchMessage("apron"); // output: Message apron doesn't match.

void Validate(int[] numbers)
{
    var result = numbers is [< 0, .. { Length: 2 or 4 }, > 0] ? "valid" :
    "not valid";
    Console.WriteLine(result);
}

Validate(new[] { -1, 0, 1 }); // output: not valid
Validate(new[] { -1, 0, 0, 1 }); // output: valid
```

Para obter mais informações, confira a nota de proposta de recursos de [Padrões de lista](#).

## Especificação da linguagem C#

Para obter mais informações, confira a seção [Padrões e padrões correspondentes](#) da especificação da linguagem C#.

Para informações sobre os recursos adicionados ao C# 8 e versões posteriores, confira as seguintes notas sobre a proposta de recursos:

- [C# 8 – Padrões correspondentes recursivos](#)
- [C# 9 - Atualizações de correspondência de padrão](#)
- [C# 10 – Padrões de propriedade estendida](#)
- [C# 11 – Padrões de lista](#)
- [C# 11 – Correspondência de padrão Span<char>no literal de cadeia de caracteres](#)

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Visão geral dos padrões correspondentes](#)
- [Tutorial: Usar padrões correspondentes para criar algoritmos controlados por tipo e controlados por dados](#)

# Operadores de adição – + e +=

Artigo • 07/04/2023

Os operadores + e += são compatíveis com os tipos numéricos internos [integrais](#) e [ponto flutuante](#), o tipo [cadeia de caracteres](#) e os tipos [delegados](#).

Para obter informações sobre o operador + aritmético, consulte as seções [Operadores de adição e subtração unários](#) e [Operador de adição +](#) do artigo [Operadores aritméticos](#).

## Concatenação de cadeia de caracteres

Quando um ou ambos os operandos forem do tipo [cadeia de caracteres](#), o operador + concatenará as representações de cadeia de caracteres de seus operandos (a representação de cadeia de caracteres de `null` é uma cadeia de caracteres vazia):

C#

```
Console.WriteLine("Forgot" + "white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
Console.WriteLine(null + "Nothing to add.");
// Output:
// Forgotwhite space
// Probably the oldest constant: 3.14159265358979
// Nothing to add.
```

A [interpolação de cadeia de caracteres](#) fornece uma maneira mais conveniente de formatar cadeias de caracteres:

C#

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

A partir do C# 10, você pode usar a interpolação de cadeia de caracteres para inicializar uma cadeia de caracteres constante quando todas as expressões usadas para espaços reservados também são cadeias de caracteres constantes.

A partir do C# 11, o operador + executa a concatenação de cadeia de caracteres para cadeias de caracteres literais UTF-8. Esse operador concatena dois objetos `ReadOnlySpan<byte>`.

# Combinação de delegados

Para operandos do mesmo tipo [delegado](#), o operador `+` retorna uma nova instância de delegado que, quando invocada, chama o operando esquerdo e, em seguida, chama o operando direito. Se qualquer um dos operandos for `null`, o operador `+` retornará o valor de outro operando (que também pode ser `null`). O exemplo a seguir mostra como os delegados podem ser combinados com o operador `+`:

C#

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("b");
Action ab = a + b;
ab(); // output: ab
```

Para executar a remoção de delegado, use o [operador `-`](#).

Para obter mais informações sobre tipos de delegado, veja [Delegados](#).

## Operador de atribuição de adição `+=`

Uma expressão que usa o operador `+=`, como

C#

```
x += y
```

é equivalente a

C#

```
x = x + y
```

exceto que `x` é avaliado apenas uma vez.

O exemplo a seguir demonstra o uso do operador `+=`:

C#

```
int i = 5;
i += 9;
Console.WriteLine(i);
// Output: 14
```

```
string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action printer = () => Console.Write("a");
printer(); // output: a

Console.WriteLine();
printer += () => Console.Write("b");
printer(); // output: ab
```

Você também usará o operador `+=` para especificar um método de manipulador de eventos ao assinar um [evento](#). Para obter mais informações, confira [Como assinar e cancelar a assinatura de eventos](#).

## Capacidade de sobrecarga do operador

Um tipo definido pelo usuário pode [sobreclarregar](#) o operador `+`. Quando um operador `+` binário é sobreclarregado, o operador `+=` também é implicitamente sobreclarregado. Um tipo definido pelo usuário não pode sobreclarregar explicitamente o operador `+=`.

## Especificação da linguagem C#

Para obter mais informações, veja as seções [Operador de adição unário](#) e [Operador de adição](#) da [Especificação de linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Como concatenar várias cadeias de caracteres](#)
- [Eventos](#)
- [Operadores aritméticos](#)
- [Operadores - e -=](#)

# Operadores - e -= - subtração (menos)

Artigo • 07/04/2023

Os operadores `-` e `-=` são compatíveis com os tipos numéricos [integrais](#) e [flutuantes](#) internos e os tipos [delegados](#).

Para obter informações sobre o operador `-` aritmético, consulte as seções [Operadores de adição e subtração unários](#) e [Operador de subtração -](#) do artigo [Operadores aritméticos](#).

## Remoção de delegado

Para operandos do mesmo tipo [delegado](#), o operador `-` retorna uma instância de delegado que é calculada da seguinte maneira:

- Se ambos os operandos forem não nulos e a lista de invocação do operando à direita for uma sublista contígua apropriada da lista de invocação do operando à esquerda, o resultado da operação será uma nova lista de invocação obtida removendo-se as entradas do operando à direita da lista de invocação do operando à esquerda. Se a lista do operando à direita corresponder a várias sublistas contíguas na lista do operando à esquerda, somente a sublista correspondente mais à direita será removida. Se a remoção resultar em uma lista vazia, o resultado será `null`.

C#

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");

var abbaab = a + b + b + a + a + b;
abbaab(); // output: abbaab
Console.WriteLine();

var ab = a + b;
var abba = abbaab - ab;
abba(); // output: abba
Console.WriteLine();

var nihil = abbaab - abbaab;
Console.WriteLine(nihil is null); // output: True
```

- Se a lista de invocação do operando à direita não for uma sublista contígua adequada da lista de invocação do operando à esquerda, o resultado da operação

será o operando à esquerda. Por exemplo, remover um delegado que não faz parte do delegado multicast não faz nada e resulta no delegado multicast inalterado.

```
C#  
  
Action a = () => Console.Write("a");  
Action b = () => Console.Write("b");  
  
var abbaab = a + b + b + a + a + b;  
var aba = a + b + a;  
  
var first = abbaab - aba;  
first(); // output: abbaab  
Console.WriteLine();  
Console.WriteLine(object.ReferenceEquals(abbaab, first)); // output:  
True  
  
Action a2 = () => Console.Write("a");  
var changed = aba - a;  
changed(); // output: ab  
Console.WriteLine();  
var unchanged = aba - a2;  
unchanged(); // output: aba  
Console.WriteLine();  
Console.WriteLine(object.ReferenceEquals(aba, unchanged)); // output:  
True
```

O exemplo anterior também demonstra que, durante a remoção de delegados, as instâncias de delegado são comparadas. Por exemplo, delegados produzidos a partir da avaliação de [expressões lambda idênticas não são iguais](#). Para saber mais sobre a igualdade de delegados, confira a seção [Operadores de igualdade de delegados](#) da [Especificação da linguagem C#](#).

- Se o operando à esquerda for `null`, o resultado da operação será `null`. Se o operando à direita for `null`, o resultado da operação será o operando à esquerda.

```
C#  
  
Action a = () => Console.Write("a");  
  
var nothing = null - a;  
Console.WriteLine(nothing is null); // output: True  
  
var first = a - null;  
a(); // output: a  
Console.WriteLine();  
Console.WriteLine(object.ReferenceEquals(first, a)); // output: True
```

Para combinar delegados, use o [operador +](#).

Para obter mais informações sobre tipos de delegado, veja [Delegados](#).

## Operador de atribuição de subtração -=

Uma expressão que usa o operador -=, como

```
C#
```

```
x -= y
```

é equivalente a

```
C#
```

```
x = x - y
```

exceto que x é avaliado apenas uma vez.

O exemplo a seguir demonstra o uso do operador -=:

```
C#
```

```
int i = 5;
i -= 9;
Console.WriteLine(i);
// Output: -4

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
var printer = a + b + a;
printer(); // output: aba

Console.WriteLine();
printer -= a;
printer(); // output: ab
```

Você também usará o operador -= para especificar um método de manipulador de eventos a remover ao cancelar a assinatura de um [evento](#). Para obter mais informações, confira [Como assinar e cancelar a assinatura de eventos](#).

## Capacidade de sobrecarga do operador

Um tipo definido pelo usuário pode [sobreregar](#) o operador `-`. Quando um operador

`-` binário é sobrecarregado, o operador `-=` também é implicitamente sobrecarregado.

Um tipo definido pelo usuário não pode sobrecarregar explicitamente o operador `-=`.

## Especificação da linguagem C#

Para obter mais informações, veja as seções [Operador de subtração unário](#) e [Operador de subtração](#) da [Especificação de linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Eventos](#)
- [Operadores aritméticos](#)
- [Operadores + e +=](#)

# Operador ?: – o operador condicional ternário

Artigo • 26/07/2023

O operador condicional `?:`, também conhecido como o operador condicional ternário, avalia uma expressão booliana e retorna o resultado de uma das duas expressões, dependendo se a expressão booliana é avaliada como `true` ou `false`, conforme mostra o exemplo a seguir:

C#

```
string GetWeatherDisplay(double tempInCelsius) => tempInCelsius < 20.0 ?  
    "Cold." : "Perfect!";  
  
Console.WriteLine(GetWeatherDisplay(15)); // output: Cold.  
Console.WriteLine(GetWeatherDisplay(27)); // output: Perfect!
```

Como mostra o exemplo anterior, a sintaxe do operador condicional é a seguinte:

C#

```
condition ? consequent : alternative
```

A expressão `condition` deve ser avaliada para `true` ou `false`. Se `condition` for avaliada como `true`, a expressão `consequent` será avaliada e seu resultado se tornará o resultado da operação. Se `condition` for avaliada como `false`, a expressão `alternative` será avaliada e seu resultado se tornará o resultado da operação. Somente `consequent` ou `alternative` é avaliada.

A partir do C# 9.0, as expressões condicionais são com tipo de destino. Ou seja, se um tipo de destino de uma expressão condicional for conhecido, os tipos de `consequent` e `alternative` devem ser implicitamente conversíveis para o tipo de destino, como mostra o exemplo a seguir:

C#

```
var rand = new Random();  
var condition = rand.NextDouble() > 0.5;  
  
int? x = condition ? 12 : null;
```

```
IEnumerable<int> xs = x is null ? new List<int>() { 0, 1 } : new int[] { 2, 3 };
```

Se um tipo de destino de uma expressão condicional for desconhecido (por exemplo, quando você usa a palavra-chave `var`) ou o tipo `consequent` e `alternative` precisa ser o mesmo ou precisa haver uma conversão implícita de um tipo para outro:

C#

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

var x = condition ? 12 : (int?)null;
```

O operador condicional é associativo direito, ou seja, uma expressão da forma

C#

```
a ? b : c ? d : e
```

é avaliada como

C#

```
a ? b : (c ? d : e)
```

### 💡 Dica

Você pode usar o seguinte dispositivo mnemônico para se lembrar de como o operador condicional é avaliado:

text

```
is this condition true ? yes : no
```

## Expressão condicional ref

Uma expressão ref condicional retorna condicionalmente uma referência de variável, como mostra o exemplo a seguir:

C#

```
var smallArray = new int[] { 1, 2, 3, 4, 5 };
var largeArray = new int[] { 10, 20, 30, 40, 50 };

int index = 7;
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref
largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
// 1 2 100 4 5
// 10 20 0 40 50
```

Você pode [refatribuir](#) o resultado de uma expressão ref condicional, usá-lo como um [retorno de referência](#) ou passá-lo como um parâmetro de método `ref`, `out` ou `in`. Você também pode atribuir ao resultado de uma expressão ref condicional, como mostra o exemplo anterior.

A sintaxe de uma expressão condicional ref é a seguinte:

C#

```
condition ? ref consequent : ref alternative
```

Como o operador condicional, uma expressão ref condicional avalia apenas uma das duas expressões: `consequent` ou `alternative`.

Em uma expressão ref condicional, o tipo de `consequent` e `alternative` devem ser iguais. Expressões ref condicionais não são com tipo de destino.

## Operador condicional e uma instrução if

O uso do operador condicional em vez de uma [instrução if](#) pode resultar em código mais conciso em casos onde você precisa calcular um valor condicionalmente. O exemplo a seguir demonstra duas maneiras de classificar um inteiro como negativo ou não negativo:

C#

```
int input = new Random().Next(-5, 5);
```

```
string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

## Capacidade de sobrecarga do operador

Um tipo definido pelo usuário não pode sobrepor o operador condicional.

## Especificação da linguagem C#

Para saber mais, confira a seção [Operador condicional](#) na [especificação da linguagem C#](#).

As especificações para recursos mais recentes são:

- [Expressão condicional com tipo de destino \(C# 9.0\)](#)

## Confira também

- [Simplificar expressão condicional \(regra de estilo IDE0075\)](#)
- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [instrução if](#)
- [?. e operadores ?\[\]](#)
- [?? e operadores ??=](#)
- [ref keyword](#)

# ! ! (tolerante a nulo) operador (referência do C#)

Artigo • 16/06/2023

O operador `!` de sufixo unário é o operador tolerante a nulo ou de supressão de nulo. Em um [contexto de anotação nulo](#) habilitado, você usa o operador de tolerância nula para suprimir todos os avisos anuláveis para a expressão anterior. O operador `!` de prefixo unário é o [operador de negação lógica](#). O operador tolerante a nulo não tem efeito em tempo de execução. Ele afeta apenas a análise de fluxo estático do compilador alterando o estado nulo da expressão. Em tempo de execução, a expressão `x!` é avaliada como o resultado da expressão `x` subjacente.

Para obter mais informações sobre o recurso de tipos de referência que permitem valor nulo, consulte [Tipos de referência que permitem valor nulo](#).

## Exemplos

Um dos casos de uso do operador tolerante a nulo está em testar a lógica de validação do argumento. Por exemplo, considere a seguinte classe:

C#

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new
ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

Usando a [estrutura de teste MSTest](#), você pode criar o seguinte teste para a lógica de validação no construtor:

C#

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
```

Sem o operador tolerante a nulo, o compilador gera o seguinte aviso para o código anterior: `Warning CS8625: Cannot convert null literal to non-nullable reference type`. Ao usar o operador tolerante a nulo, informe ao compilador que a passagem `null` é esperada e não deve ser avisada.

Você também pode usar o operador de tolerância nula se souber definitivamente que uma expressão não pode ser `null`, mas o compilador não consegue reconhecer isso. No exemplo a seguir, se o método `IsValid` retornar `true`, seu argumento não será `null` e você poderá desreferenciá-lo com segurança:

```
C#  
  
public static void Main()  
{  
    Person? p = Find("John");  
    if (IsValid(p))  
    {  
        Console.WriteLine($"Found {p!.Name}");  
    }  
}  
  
public static bool IsValid(Person? person)  
=> person is not null && person.Name is not null;
```

Sem o operador tolerante a nulo, o compilador gera o seguinte aviso para o código `p.Name`: `Warning CS8602: Dereference of a possibly null reference`.

Se você puder modificar o método `IsValid`, poderá usar o atributo `NotNullWhen` para informar ao compilador que um argumento do método `IsValid` não pode ser `null` quando o método retornar `true`:

```
C#  
  
public static void Main()  
{  
    Person? p = Find("John");  
    if (IsValid(p))  
    {  
        Console.WriteLine($"Found {p.Name}");  
    }  
}  
  
public static bool IsValid([NotNullWhen(true)] Person? person)  
=> person is not null && person.Name is not null;
```

No exemplo anterior, você não precisa usar o operador de tolerância nula porque o compilador tem informações suficientes para descobrir que `p` não pode ser `null` dentro da instrução `if`. Para obter mais informações sobre os atributos que permitem fornecer informações adicionais sobre o estado nulo de uma variável, consulte [As APIs de Atualização com atributos para definir expectativas quanto a nulos](#).

## Especificação da linguagem C#

Para obter mais informações, consulte a seção [Operador tolerante a nulo](#) do [Rascunho da especificação de tipos de referência que permitem valor nulo](#).

## Confira também

- [Remover o operador de supressão desnecessário \(regra de estilo IDE0080\)](#)
- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Tutorial: Criar com tipos de referência que permitem valor nulo](#)

# ?? e os operadores ?? = - os operadores de união nula

Artigo • 28/07/2023

O operador de coalescência nula `??` retornará o valor do operando esquerdo se não for `null`; caso contrário, ele avaliará o operando direito e retornará seu resultado. O operador `??=` não avaliará o operando do lado direito se o operando esquerdo for avaliado como não nulo. O operador de atribuição de avaliação de nulo `??=` atribuirá o valor do operando do lado direito para o operando esquerdo somente se o operando esquerdo for avaliado como `null`. O operador `??=` não avaliará o operando do lado direito se o operando esquerdo for avaliado como não nulo.

C#

```
List<int> numbers = null;
int? a = null;

Console.WriteLine((numbers is null)); // expected: true
// if numbers is null, initialize it. Then, add 5 to numbers
(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers)); // output: 5
Console.WriteLine((numbers is null)); // expected: false

Console.WriteLine((a is null)); // expected: true
Console.WriteLine((a ?? 3)); // expected: 3 since a is still null
// if a is null then assign 0 to a and add a to the list
numbers.Add(a ??= 0);
Console.WriteLine((a is null)); // expected: false
Console.WriteLine(string.Join(" ", numbers)); // output: 5 0
Console.WriteLine(a); // output: 0
```

O operando à esquerda do operador `??=` deve ser uma variável, uma [propriedade](#) ou um elemento [indexador](#).

O tipo do operando à esquerda dos `??` e `??=` operadores não pode ser um tipo de valor não anulável. Em particular, você pode usar os operadores de avaliação de nulo com parâmetros de tipo não treinados:

C#

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

```
        Console.WriteLine(a ?? backup);
    }
```

Os operadores de avaliação de nulo são associativos à direita. Ou seja, expressões do formulário

C#

```
a ?? b ?? c  
d ??= e ??= f
```

são avaliadas como

C#

```
a ?? (b ?? c)  
d ??= (e ??= f)
```

## Exemplos

Os operadores `??` e `??=` podem ser úteis nos seguintes cenários:

- Em expressões com os [operadores condicionais nulos `?.` e `?\[\]`](#), você pode usar o `??` operador para fornecer uma expressão alternativa a ser avaliada caso o resultado da expressão com operação condicional nula seja `null`:

```
C#  
  
double SumNumbers(List<double> setsOfNumbers, int indexOfSetToSum)  
{  
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;  
}  
  
var sum = SumNumbers(null, 0);  
Console.WriteLine(sum); // output: NaN
```

- Quando você trabalhar com [tipos que permitem valor nulo](#) e precisar fornecer o valor de um tipo subjacente, use o operador `??` para especificar o valor a ser fornecido caso o tipo que permite valor nulo seja `null`:

```
C#  
  
int? a = null;  
int b = a ?? -1;
```

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

Use o método `Nullable<T>.GetValueOrDefault()` se o valor a ser usado quando um valor de tipo que permite valor nulo for `null` tiver que ser o valor padrão do tipo de valor subjacente.

- Você pode usar uma [expressão throw](#) como o operando direito do operador `??` para tornar o código de verificação de argumento mais conciso:

C#

```
public string Name
{
    get => name;
    set => name = value ?? throw new
ArgumentNullException(nameof(value), "Name cannot be null");
}
```

O exemplo anterior também demonstra como usar [membros aptos para expressão](#) para definir uma propriedade.

- Você pode usar o operador `??=` para substituir o código do formulário

C#

```
if (variable is null)
{
    variable = expression;
}
```

pelo código a seguir:

C#

```
variable ??= expression;
```

## Capacidade de sobrecarga do operador

Os operadores `??` e `??=` não podem ser sobre carregados.

## Especificação da linguagem C#

Para obter mais informações sobre o operador `??`, consulte a seção [O operador de avaliação de nulo da Especificação da linguagem c#](#).

Para obter mais informações sobre o operador `??=`, confira a [nota da proposta do recurso](#).

## Confira também

- A verificação de nulidade pode ser simplificada (IDE0029, IDE0030 e IDE0270)
- Referência de C#
- Operadores e expressões C#
- `?.` e operadores `?[]`
- Operador `:`

" é usado para definir uma expressão lambda em C# | Microsoft Learn" />

# O operador de expressão lambda (`=>`) define uma expressão lambda

Artigo • 20/07/2023

Há suporte para o token `=>` de duas formas: como o [operador lambda](#) e como um separador de um nome de membro e a implementação de membro em uma [definição de corpo da expressão](#).

## Operador lambda

Em [expressões lambda](#), o operador lambda `=>` separa os parâmetros de entrada do lado esquerdo do corpo lambda no lado direito.

O seguinte exemplo usa o recurso [LINQ](#) com a sintaxe de método para demonstrar o uso de expressões lambda:

C#

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
    .Where(w => w.StartsWith("a"))
    .Min(w => w.Length);
Console.WriteLine(minimalLength); // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product); // output: 280
```

Os parâmetros de entrada de uma expressão lambda são fortemente digitados no tempo de compilação. Quando o compilador pode inferir os tipos de parâmetros de entrada, como no exemplo anterior, você pode omitir as declarações de tipo. Caso precise especificar o tipo de parâmetros de entrada, faça isso para cada parâmetro, como mostra o seguinte exemplo:

C#

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim *
next);
Console.WriteLine(product); // output: 280
```

O seguinte exemplo mostra como definir uma expressão lambda sem parâmetros de entrada:

```
C#
```

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

Para obter mais informações, confira [Expressões lambda](#).

## Definição de corpo da expressão

Uma definição de corpo da expressão tem a seguinte sintaxe geral:

```
C#
```

```
member => expression;
```

em que `expression` é uma expressão válida. O tipo de retorno de `expression` deve ser implicitamente conversível para o tipo de retorno do membro. Se o membro:

- Tem um tipo de retorno `void` ou
- for um:
  - Construtor
  - Finalizer
  - Acessador `set` de propriedade ou indexador

`expression` deve ser uma *expressão de instrução*. Como o resultado da expressão é descartado, o tipo de retorno dessa expressão pode ser qualquer tipo.

O seguinte exemplo mostra uma definição de corpo da expressão para um método `Person.ToString`:

```
C#
```

```
public override string ToString() => $"{fname} {lname}".Trim();
```

É uma versão abreviada da seguinte definição de método:

```
C#
```

```
public override string ToString()
{
```

```
    return $"{fname} {lname}".Trim();  
}
```

É possível criar definições de corpo da expressão para métodos, operadores, propriedades somente leitura, constructos, finalizadores e acessadores e indexadores de propriedade. Para obter mais informações, consulte [Membros aptos para expressão](#).

## Capacidade de sobrecarga do operador

O operador `=>` não pode ser sobreescrito.

## Especificação da linguagem C#

Para obter mais informações sobre o operador lambda, confira a seção [Expressões de função anônima](#) da [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)

# Operador :: – o operador de alias de namespace

Artigo • 05/06/2023

Use o qualificador de alias de namespace `::` para acessar membros de um namespace com alias. O qualificador `::` só pode ser usado entre dois identificadores. O identificador à esquerda pode ser um alias de namespace, um alias externo ou o alias `global`. Por exemplo:

- Um alias de namespace criado com uma [diretiva using de alias](#):

```
C#  
  
using forwinforms = System.Drawing;  
using forwpf = System.Windows;  
  
public class Converters  
{  
    public static forwpf::Point Convert(forwinforms::Point point) =>  
        new forwpf::Point(point.X, point.Y);  
}
```

- Um [alias externo](#).
- O alias `global`, que é o alias do namespace global. O namespace global é o namespace que contém namespaces e tipos que não são declarados dentro de um namespace com nome. Quando usado com o qualificador `::`, o alias `global` sempre faz referência ao namespace global, mesmo se houver o alias de namespace `global` definido pelo usuário.

O exemplo a seguir usa o alias `global` para acessar o namespace .NET `System`, que é um membro do namespace global. Sem o alias `global`, o namespace `System` definido pelo usuário, que é um membro do namespace `MyCompany.MyProduct`, seria acessado:

```
C#  
  
namespace MyCompany.MyProduct.System  
{  
    class Program  
    {  
        static void Main() => global::System.Console.WriteLine("Using  
        global alias");  
    }  
}
```

```
class Console
{
    string Suggestion => "Consider renaming this class";
}
```

### ➊ Observação

A palavra-chave `global` é o alias do namespace global apenas quando é o identificador à esquerda do qualificador `::`.

Você também pode usar o token `.` para acessar membros de um namespace com alias. No entanto, o token `.` também é usado para acessar um membro do tipo. O qualificador `::` garante que o identificador à esquerda dele sempre faça referência a um alias de namespace, mesmo que exista um tipo ou namespace com o mesmo nome.

## Especificação da linguagem C#

Saiba mais na seção [Qualificadores de alias de namespace](#) da [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)

# Operador await – aguardar de forma assíncrona a conclusão da tarefa

Artigo • 27/03/2023

O operador `await` suspende a avaliação do método `async` delimitador enquanto a operação assíncrona representada por seu operando não é concluída. Quando a operação assíncrona for concluída, o operador `await` retornará o resultado da operação, se houver. Quando o operador `await` é aplicado ao operando que representa uma operação já concluída, ele retorna o resultado da operação imediatamente sem a suspensão do método delimitador. O operador `await` não bloqueia o thread que avalia o método assíncrono. Quando o operador `await` suspende o método assíncrono delimitador, o controle é retornado ao chamador do método.

No exemplo a seguir, o método `HttpClient.GetByteArrayAsync` retorna a instância `Task<byte[]>`, que representa uma operação assíncrona que produz uma matriz de bytes quando é concluída. O operador `await` suspende o método `DownloadDocs MainPageAsync` até que a operação seja concluída. Quando `DownloadDocs MainPageAsync` é suspenso, o controle é retornado ao método `Main`, que é o chamador de `DownloadDocs MainPageAsync`. O método `Main` é executado até precisar do resultado da operação assíncrona executada pelo método `DownloadDocs MainPageAsync`. Quando `GetByteArrayAsync` obtém todos os bytes, o restante do método `DownloadDocs MainPageAsync` é avaliado. Depois disso, o restante do método `Main` é avaliado.

C#

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class AwaitOperator
{
    public static async Task Main()
    {
        Task<int> downloading = DownloadDocs MainPageAsync();
        Console.WriteLine($"{nameof(Main)}: Launched downloading.");

        int bytesLoaded = await downloading;
        Console.WriteLine($"{nameof(Main)}: Downloaded {bytesLoaded} bytes.");
    }

    private static async Task<int> DownloadDocs MainPageAsync()
```

```
{  
    Console.WriteLine($"{nameof(DownloadDocs MainPageAsync)}: About to  
start downloading.");  
  
    var client = new HttpClient();  
    byte[] content = await  
client.GetByteArrayAsync("https://learn.microsoft.com/en-us/");  
  
    Console.WriteLine($"{nameof(DownloadDocs MainPageAsync)}: Finished  
downloading.");  
    return content.Length;  
}  
}  
// Output similar to:  
// DownloadDocsMainPageAsync: About to start downloading.  
// Main: Launched downloading.  
// DownloadDocsMainPageAsync: Finished downloading.  
// Main: Downloaded 27700 bytes.
```

O exemplo anterior usa o [método assíncrono Main](#). Para obter mais informações, confira a seção [Operador await no método Main](#).

### ⓘ Observação

Para obter uma introdução à programação assíncrona, confira [Programação assíncrona com async e await](#). A programação assíncrona com `async` e `await` segue o [padrão assíncrono baseado em tarefas](#).

Use o operador `await` somente em um método, uma [expressão lambda](#) ou um [método anônimo](#) que seja modificado pela palavra-chave `async`. Em um método assíncrono, não é possível usar o operador `await` no corpo de uma função síncrona, dentro do bloco de uma [instrução lock](#) e em um contexto [desprotegido](#).

O operando `await` do operador geralmente é de um dos seguintes tipos .NET: [Task](#), [Task<TResult>](#), [ValueTask](#) ou [ValueTask<TResult>](#). No entanto, qualquer expressão aguardável pode ser o operando do operador `await`. Para obter mais informações, confira a seção [Expressões aguardáveis](#) da [Especificação da linguagem C#](#).

O tipo de expressão `await t` é `TResult` se o tipo de expressão `t` é [Task<TResult>](#) ou [ValueTask<TResult>](#). Se o tipo de `t` é [Task](#) ou [ValueTask](#), o tipo de `await t` é `void`. Em ambos os casos, se `t` gera uma exceção, `await t` gera a exceção novamente.

## Fluxos assíncronos e descartáveis

Você usa a instrução `await foreach` para consumir um fluxo assíncrono de dados. Para obter mais informações, consulte a seção sobre a instrução `foreach` do artigo [Instruções de iteração](#).

Você usa a instrução `await using` para trabalhar com um objeto assíncrono descartável, ou seja, um objeto de um tipo que implementa uma interface `IAsyncDisposable`. Para obter mais informações, consulte a seção [Usar descartáveis assíncronos](#) do artigo [Implementar um método `DisposeAsync`](#).

## Operador `await` no método Main

O método `Main`, que é o ponto de entrada do aplicativo, pode retornar `Task` ou `Task<int>`, permitindo que ele seja assíncrono, de modo que você possa usar o operador `await` no corpo. Em versões anteriores do C#, para garantir que o método `Main` aguarde a conclusão de uma operação assíncrona, você pode recuperar o valor da propriedade `Task<TResult>.Result` da instância `Task<TResult>` retornada pelo método assíncrono correspondente. Para operações assíncronas que não produzem um valor, você pode chamar o método `Task.Wait`. Para saber mais sobre como selecionar a versão da linguagem, consulte [Controle de versão da linguagem C#](#).

## Especificação da linguagem C#

Para obter mais informações, confira a seção [Expressões `await`](#) da [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [async](#)
- [Modelo de programação assíncrona de tarefa](#)
- [Programação assíncrona](#)
- [Passo a passo: Como acessar a Web usando `async` e `await`](#)
- [Tutorial: Gerar e consumir fluxos assíncronos](#)
- [Blog do .NET: como assíncrono/espera realmente funciona em C #](#) ↗

# expressões de valor padrão – produzem o valor padrão

Artigo • 05/06/2023

Uma expressão de valor padrão produz o [valor padrão](#) de um tipo. Há dois tipos de expressões de valor padrão: a chamada de [operador padrão](#) e um [literal padrão](#).

Você também usa a palavra-chave `default` como o rótulo de caso padrão em uma [switch instrução](#).

## operador default

O argumento do operador `default` deve ser o nome de um tipo ou um parâmetro de tipo, como mostra o exemplo a seguir:

C#

```
Console.WriteLine(default(int)); // output: 0
Console.WriteLine(default(object) is null); // output: True

void DisplayDefaultOf<T>()
{
    var val = default(T);
    Console.WriteLine($"Default value of {typeof(T)} is {(val == null ? "null" : val.ToString())}.");
}

DisplayDefaultOf<int?>();
DisplayDefaultOf<System.Numerics.Complex>();
DisplayDefaultOf<System.Collections.Generic.List<int>>();
// Output:
// Default value of System.Nullable`1[System.Int32] is null.
// Default value of System.Numerics.Complex is (0, 0).
// Default value of System.Collections.Generic.List`1[System.Int32] is null.
```

## literal padrão

Você pode usar o literal `default` para produzir o valor padrão de um tipo quando o compilador inferir o tipo de expressão. A expressão literal `default` produz o mesmo valor que a expressão `default(T)`, em que `T` é o tipo inferido. Você pode usar o literal `default` em qualquer um dos seguintes casos:

- Na atribuição ou inicialização de uma variável.

- Na declaração do valor padrão para um [parâmetro de método opcional](#).
- Em uma chamada de método para fornecer um valor de argumento.
- Em uma [returninstrução](#) ou, como expressão em um [membro apto para expressão](#).

O exemplo a seguir mostra o uso do literal `default`:

C#

```
T[] InitializeArray<T>(int length, T initialValue = default)
{
    if (length < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(length), "Array length
must be nonnegative.");
    }

    var array = new T[length];
    for (var i = 0; i < length; i++)
    {
        array[i] = initialValue;
    }
    return array;
}

void Display<T>(T[] values) => Console.WriteLine($"[ {string.Join(", ", values)} ]");

Display(InitializeArray<int>(3)); // output: [ 0, 0, 0 ]
Display(InitializeArray<bool>(4, default)); // output: [ False, False,
False, False ]

System.Numerics.Complex fillValue = default;
Display(InitializeArray(3, fillValue)); // output: [ (0, 0), (0, 0), (0, 0)
]
```

### 💡 Dica

Use a regra de estilo .NET [IDE0034](#) para especificar uma preferência sobre o uso do literal `default` na sua base de código.

## Especificação da linguagem C#

Para saber mais, confira a seção [Expressões de valor padrão](#) da [Especificação da linguagem C#](#).

## Confira também

- Referência de C#
- Operadores e expressões C#
- Valores padrão de tipos C#
- Generics in .NET (Genéricos no .NET)

# operador delegate

Artigo • 05/06/2023

O operador `delegate` cria um método anônimo que pode ser convertido em um tipo delegado. Um método anônimo pode ser convertido em tipos como `System.Action` e `System.Func<TResult>` usados como argumentos em muitos métodos.

C#

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4)); // output: 7
```

## ⓘ Observação

As expressões lambda fornecem uma forma mais concisa e expressiva de criar uma função anônima. Use o `=>` (operador) para construir uma expressão lambda:

C#

```
Func<int, int, int> sum = (a, b) => a + b;
Console.WriteLine(sum(3, 4)); // output: 7
```

Para saber mais sobre recursos de expressões lambda, por exemplo, que capturam variáveis externas, confira [Expressões lambda](#).

Você pode omitir a lista de parâmetros quando usa o operador `delegate`. Se você fizer isso, o método anônimo criado poderá ser convertido em um tipo delegado com qualquer lista de parâmetros, como mostra o exemplo a seguir:

C#

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is
world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```

Essa é a única funcionalidade de métodos anônimos que não tem suporte por expressões lambda. Em todos os outros casos, uma expressão lambda é a forma preferida de gravar código embutido.

A partir do C# 9.0, você pode usar [descartes](#) para especificar dois ou mais parâmetros de entrada de um método anônimo que não sejam usados pelo método:

C#

```
Func<int, int, int> constant = delegate (int _, int _) { return 42; };
Console.WriteLine(constant(3, 4)); // output: 42
```

Para compatibilidade com versões anteriores, se apenas um parâmetro for nomeado `_`, `_` será tratado como o nome desse parâmetro em um método anônimo.

Também a partir do C# 9.0, você pode usar o modificador `static` na declaração de um método anônimo:

C#

```
Func<int, int, int> sum = static delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(10, 4)); // output: 14
```

Um método anônimo estático não pode capturar variáveis locais ou o estado da instância entre escopos.

Você também usa a palavra-chave `delegate` para declarar um [tipo delegado](#).

A partir do C# 11, o compilador pode armazenar em cache o objeto delegado criado a partir de um grupo de métodos. Considere o método a seguir:

C#

```
static void StaticFunction() { }
```

Quando você atribui o grupo de métodos a um delegado, o compilador armazena em cache o delegado:

C#

```
Action a = StaticFunction;
```

Antes do C# 11, você precisaria usar uma expressão lambda para reutilizar um único objeto delegado:

C#

```
Action a = () => StaticFunction();
```

## Especificação da linguagem C#

Para obter mais informações, confira a seção [Expressões de função anônima](#) da [Especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [operador =>](#)

# é operador (referência C#)

Artigo • 07/04/2023

O operador `is` verifica se o resultado de expressão é compatível com um determinado tipo. Saiba mais sobre o operador `is` de teste de tipo na seção [operador is](#) do artigo [Operadores cast e teste de tipo](#). Você também pode usar o `is` operador para corresponder uma expressão com um padrão, como mostra o exemplo a seguir:

C#

```
static bool IsFirstFridayOfOctober(DateTime date) =>
    date is { Month: 10, Day: <=7, DayOfWeek: DayOfWeek.Friday };
```

No exemplo anterior, o operador corresponde a `is` uma expressão em relação a um [padrão de propriedade com padrões de constante](#) aninhada e [relacional](#) (disponível no C# 9.0 e posterior).

O operador `is` pode ser útil nos seguintes cenários:

- Para verificar o tipo de tempo de execução de uma expressão, como mostra o exemplo a seguir:

C#

```
int i = 34;
object iBoxed = i;
int? jNullable = 42;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b); // output 76
}
```

O exemplo anterior mostra o uso de um [padrão de declaração](#).

- Para verificar `null`, como mostra o exemplo a seguir:

C#

```
if (input is null)
{
    return;
}
```

Quando você compara a uma expressão em relação `null`, o compilador garante que nenhum operador ou sobrecarga do usuário `==` ou `!=` seja invocada.

- A partir do C# 9.0, você pode usar um [padrão de negação](#) para fazer uma verificação não nula, como mostra o exemplo a seguir:

C#

```
if (result is not null)
{
    Console.WriteLine(result.ToString());
}
```

- A partir do C# 11, você pode usar [padrões de lista](#) para comparar elementos de uma lista ou matriz. O código a seguir verifica matrizes para valores inteiros em posições esperadas:

C#

```
int[] empty = { };
int[] one = { 1 };
int[] odd = { 1, 3, 5 };
int[] even = { 2, 4, 6 };
int[] fib = { 1, 1, 2, 3, 5 };

Console.WriteLine(odd is [1, _, 2, ...]);    // false
Console.WriteLine(fib is [1, _, 2, ...]);    // true
Console.WriteLine(fib is [_, 1, 2, 3, ...]); // true
Console.WriteLine(fib is [..., 1, 2, 3, _]); // true
Console.WriteLine(even is [2, _, 6]);        // true
Console.WriteLine(even is [2, ..., 6]);        // true
Console.WriteLine(odd is [..., 3, 5]);        // true
Console.WriteLine(even is [..., 3, 5]);        // false
Console.WriteLine(fib is [..., 3, 5]);        // true
```

### ⓘ Observação

Para obter a lista completa de padrões compatíveis com o operador `is`, confira [Padrões](#).

## Especificação da linguagem C#

Para saber mais, confira a seção [O operador is](#) da [especificação da linguagem C#](#) e as seguintes propostas da linguagem C#:

- Correspondência de padrões
- Correspondência de padrões com genéricos

## Confira também

- Referência de C#
- Operadores e expressões C#
- Padrões
- Tutorial: Usar padrões correspondentes para criar algoritmos controlados por tipo e controlados por dados
- Operadores cast e teste de tipo

# expressão nameof (referência do C#)

Artigo • 20/03/2023

A expressão `nameof` produz o nome de uma variável, de um tipo ou membro como a constante de cadeia de caracteres. A expressão `nameof` é avaliada em tempo de compilação e não tem efeitos em tempo de execução. Quando o operando é um tipo ou um namespace, o nome produzido não é **totalmente qualificado**. O seguinte exemplo mostra o uso de uma expressão `nameof`:

C#

```
Console.WriteLine(nameof(System.Collections.Generic)); // output: Generic
Console.WriteLine(nameof(List<int>)); // output: List
Console.WriteLine(nameof(List<int>.Count)); // output: Count
Console.WriteLine(nameof(List<int>.Add)); // output: Add

var numbers = new List<int> { 1, 2, 3 };
Console.WriteLine(nameof(numbers)); // output: numbers
Console.WriteLine(nameof(numbers.Count)); // output: Count
Console.WriteLine(nameof(numbers.Add)); // output: Add
```

Você pode usar uma expressão `nameof` para tornar o código de verificação de argumentos mais passível de manutenção:

C#

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), $""
{nameof(Name)} cannot be null");
}
```

A partir do C# 11, você pode usar uma expressão `nameof` com um parâmetro de método dentro de um **atributo** em um método ou seu parâmetro. O código a seguir mostra como fazer isso para um atributo em um método, uma função local e o parâmetro de uma expressão lambda:

C#

```
[ParameterString(nameof(msg))]
public static void Method(string msg)
{
    [ParameterString(nameof(T))]
    void LocalFunction<T>(T param) { }
```

```
var lambdaExpression = ([ParameterString(nameof(aNumber))] int aNumber)
=> aNumber.ToString();
}
```

Uma expressão `nameof` com um parâmetro é útil quando você usa os [atributos de análise anuláveis](#) ou o [atributo CallerArgumentExpression](#).

Quando o operando é um [identificador verbatim](#), o caractere `@` não faz parte de um nome, como mostra o seguinte exemplo:

C#

```
var @new = 5;
Console.WriteLine(nameof(@new)); // output: new
```

## Especificação da linguagem C#

Para obter mais informações, confira a seção [Expressões Nameof](#) da [Especificação da linguagem C#](#) e a especificação do recurso [C# 11 – Escoponameof estendido](#).

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Converter typeof em nameof \(regra de estilo IDE0082\)](#)

# Operador new – o operador `new` cria uma instância de um tipo

Artigo • 20/03/2023

O operador `new` cria uma nova instância de um tipo. Você também pode usar a palavra-chave `new` como um [modificador de declaração de membro](#) ou uma [restrição de tipo genérico](#).

## Chamada de construtor

Para criar uma nova instância de um tipo, você normalmente invoca um dos [construtores](#) desse tipo usando o operador `new`:

C#

```
var dict = new Dictionary<string, int>();
dict["first"] = 10;
dict["second"] = 20;
dict["third"] = 30;

Console.WriteLine(string.Join(" ", dict.Select(entry => $"{entry.Key}:
{entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

Você pode usar um [inicializador de objeto ou coleção](#) com o operador `new` para instanciar e inicializar um objeto em uma instrução, como mostra o exemplo a seguir:

C#

```
var dict = new Dictionary<string, int>
{
    ["first"] = 10,
    ["second"] = 20,
    ["third"] = 30
};

Console.WriteLine(string.Join(" ", dict.Select(entry => $"{entry.Key}:
{entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

Do C# 9.0 em diante, as expressões de invocação do construtor são tipadas como destino. Ou seja, se um tipo de destino de uma expressão for conhecido, você poderá

omitir um nome de tipo, como mostra o seguinte exemplo:

C#

```
List<int> xs = new();
List<int> ys = new(capacity: 10_000);
List<int> zs = new() { Capacity = 20_000 };

Dictionary<int, List<int>> lookup = new()
{
    [1] = new() { 1, 2, 3 },
    [2] = new() { 5, 8, 3 },
    [5] = new() { 1, 0, 4 }
};
```

Como mostra o exemplo anterior, você sempre usa parênteses em uma expressão `new` com o tipo de destino.

Se o tipo de destino de uma expressão `new` for desconhecido (por exemplo, quando você usa a palavra-chave `var`), deverá especificar um nome de tipo.

## Criação de matriz

Você também usar o operador `new` para criar uma instância de matriz, como mostra o exemplo a seguir:

C#

```
var numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;

Console.WriteLine(string.Join(", ", numbers));
// Output:
// 10, 20, 30
```

Use a sintaxe de inicialização de matriz para criar uma instância de matriz e preenchê-la com os elementos em uma instrução. O exemplo a seguir mostra várias maneiras de como fazer isso:

C#

```
var a = new int[3] { 10, 20, 30 };
var b = new int[] { 10, 20, 30 };
var c = new[] { 10, 20, 30 };
Console.WriteLine(c.GetType()); // output: System.Int32[]
```

Para obter mais informações sobre matrizes, confira [Matrizes](#).

## Instanciação de tipos anônimos

Para criar uma instância de um [tipo anônimo](#), use o operador `new` e a sintaxe do inicializador de objeto:

C#

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

## Destruição de instâncias do tipo

Você não precisa destruir as instâncias do tipo criadas anteriormente. As instâncias dos tipos de referência e de valor são destruídas automaticamente. As instâncias dos tipos de valor serão destruídas assim que o contexto que as contém for destruído. As instâncias dos tipos de referência serão destruídas pelo [coletor de lixo](#) em algum momento não especificado depois que a última referência a eles for removida.

Para instâncias de tipos que contêm recursos não gerenciados, como um identificador de arquivo, é recomendável empregar limpeza determinística para garantir que os recursos sejam liberados assim que possível. Para obter mais informações, veja o artigo [System.IDisposable Referência da API](#) e a [instrução de uso](#).

## Capacidade de sobrecarga do operador

Um tipo definido pelo usuário não pode sobrecarregar o operador `new`.

## Especificação da linguagem C#

Para saber mais, confira a seção [O operador new](#) na [especificação da linguagem C#](#).

Para obter mais informações sobre uma expressão de tipo de destino `new`, confira a [nota de proposta de recurso](#).

## Confira também

- Referência de C#
- Operadores e expressões C#
- Inicializadores de objeto e de coleção

# operador sizeof – determinar as necessidades de memória para um determinado tipo

Artigo • 05/06/2023

O operador `sizeof` retorna o número de bytes ocupados por uma variável de um determinado tipo. O argumento do operador `sizeof` deve ser o nome de um [tipo não gerenciado](#) ou um parâmetro de tipo que seja [restrito](#) a um tipo não gerenciado.

O operador `sizeof` exige um contexto [não seguro](#). No entanto, as expressões apresentadas na tabela a seguir são avaliadas em tempo de compilação para os valores constantes correspondentes e não exigem um contexto não seguro:

Expression	Valor constante
<code>sizeof(sbyte)</code>	1
<code>sizeof(byte)</code>	1
<code>sizeof(short)</code>	2
<code>sizeof(ushort)</code>	2
<code>sizeof(int)</code>	4
<code>sizeof(uint)</code>	4
<code>sizeof(long)</code>	8
<code>sizeof(ulong)</code>	8
<code>sizeof(char)</code>	2
<code>sizeof(float)</code>	4
<code>sizeof(double)</code>	8
<code>sizeof(decimal)</code>	16
<code>sizeof(bool)</code>	1

Você também não precisará usar um contexto não seguro quando o operando do operador `sizeof` for o nome de um tipo [enumerado](#).

O exemplo a seguir demonstra o uso do operador `sizeof`:

C#

```
public struct Point
{
    public Point(byte tag, double x, double y) => (Tag, X, Y) = (tag, x, y);

    public byte Tag { get; }
    public double X { get; }
    public double Y { get; }
}

public class SizeOfOperator
{
    public static void Main()
    {
        Console.WriteLine(sizeof(byte)); // output: 1
        Console.WriteLine(sizeof(double)); // output: 8

        DisplaySizeOf<Point>(); // output: Size of Point is 24
        DisplaySizeOf<decimal>(); // output: Size of System.Decimal is 16

        unsafe
        {
            Console.WriteLine(sizeof(Point*)); // output: 8
        }
    }

    static unsafe void DisplaySizeOf<T>() where T : unmanaged
    {
        Console.WriteLine($"Size of {typeof(T)} is {sizeof(T)}");
    }
}
```

O operador `sizeof` retorna o número de bytes que seriam alocados pelo Common Language Runtime na memória gerenciada. Para tipos `struct`, esse valor inclui todo o preenchimento, como demonstra o exemplo anterior. O resultado do operador `sizeof` pode ser diferente do resultado do método `Marshal.SizeOf`, que retorna o tamanho de um tipo na memória *não gerenciada*.

## Especificação da linguagem C#

Para obter mais informações, confira a seção [O operador `sizeof`](#), nas [especificações da linguagem C#](#).

## Confira também

- [Referência de C#](#)

- Operadores e expressões C#
- Operadores relacionados a ponteiro
- Tipos de ponteiro
- Tipos relacionados a memória e extensão
- Generics in .NET (Genéricos no .NET)

# Expressão stackalloc (referência de C#)

Artigo • 07/04/2023

A expressão `stackalloc` aloca um bloco de memória na pilha. Um bloco de memória alocado na pilha criado durante a execução do método é descartado automaticamente quando esse método é retornado. Não é possível liberar explicitamente a memória alocada com `stackalloc`. Um bloco de memória alocado por pilha não está sujeito à [coleta de lixo](#) e não precisa ser fixado com uma [fixed instrução](#).

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

- [System.Span<T>](#) ou [System.ReadOnlySpan<T>](#), como mostra o exemplo a seguir:

C#

```
int length = 3;
Span<int> numbers = stackalloc int[length];
for (var i = 0; i < length; i++)
{
    numbers[i] = i;
}
```

Você não precisa usar um contexto [unsafe](#) quando atribui um bloco de memória alocado na pilha a uma variável [Span<T>](#) ou [ReadOnlySpan<T>](#).

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

C#

```
int length = 1000;
Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new
byte[length];
```

Você pode usar uma `stackalloc` expressão dentro de outras expressões sempre que uma [Span<T>](#) variável ou [ReadOnlySpan<T>](#) é permitida, como mostra o exemplo a seguir:

C#

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind); // output: 1
```

### ➊ Observação

É recomendável usar os tipos `Span<T>` ou `ReadOnlySpan<T>` sempre que possível para trabalhar com memória alocada na pilha.

- Um [tipo de ponteiro](#), como mostra o seguinte exemplo:

C#

```
unsafe
{
    int length = 3;
    int* numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
}
```

Como mostra o exemplo anterior, você precisa usar um contexto `unsafe` ao trabalhar com tipos de ponteiro.

No caso de tipos de ponteiro, você pode usar uma expressão `stackalloc` apenas em uma declaração de variável local para inicializar a variável.

A quantidade de memória disponível na pilha é limitada. Se você alocar muita memória na pilha, um [StackOverflowException](#) será lançado. Para evitar isso, siga as regras abaixo:

- Limite a quantidade de memória alocada com `stackalloc`. Por exemplo, se o tamanho do buffer pretendido estiver abaixo de um determinado limite, você alocará a memória na pilha; caso contrário, use uma matriz do comprimento necessário, como mostra o seguinte código:

C#

```
const int MaxStackLimit = 1024;
Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc
byte[MaxStackLimit] : new byte[inputLength];
```

### ➊ Observação

Como a quantidade de memória disponível na pilha depende do ambiente no qual o código é executado, seja conservador ao definir o valor de limite real.

- Evite usar `stackalloc` dentro de loops. 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 método `Span<T>.Clear` que define todos os itens com o valor padrão do tipo `T`.

Você pode usar a sintaxe do inicializador de matriz para definir o conteúdo da memória recém-alocada. O seguinte exemplo demonstra várias maneiras de fazer isso:

C#

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

Na expressão `stackalloc T[E]`, `T` deve ser um tipo não gerenciado e `E` deve ser avaliado como um valor `int` não negativo.

## Segurança

O uso de `stackalloc` habilita automaticamente os recursos de detecção de estouro de buffer no CLR (Common Language Runtime). Se for detectada uma estouro de buffer, o processo será encerrado assim que possível para minimizar a chance de o código mal-intencionado ser executado.

## Especificação da linguagem C#

Para obter mais informações, consulte a seção de [Alocação de pilha](#) da [Especificação da linguagem C#](#) e a nota de proposta do recurso [Permitir stackalloc em contextos aninhados](#).

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Operadores relacionados a ponteiro](#)
- [Tipos de ponteiro](#)
- [Tipos relacionados a memória e extensão](#)
- [O que FAZER e o que NÃO FAZER quanto ao stackalloc ↗](#)

# expressão switch – expressões de padrões correspondentes usando a palavra-chave `switch`

Artigo • 05/06/2023

Você pode usar a expressão `switch` para avaliar uma única expressão de uma lista de expressões candidatas com base em uma correspondência de padrão com uma expressão de saída. Para obter informações sobre a instrução `switch` que dá suporte à semântica semelhante a `switch` em um contexto de instrução, consulte a seção de [switchinstrução](#) do artigo [Instruções de seleção](#).

O exemplo a seguir demonstra uma expressão `switch`, que converte valores de um `enum` representando direções visuais em um mapa online para as direções cardinais correspondentes:

C#

```
public static class SwitchExample
{
    public enum Direction
    {
        Up,
        Down,
        Right,
        Left
    }

    public enum Orientation
    {
        North,
        South,
        East,
        West
    }

    public static Orientation ToOrientation(Direction direction) =>
        direction switch
    {
        Direction.Up      => Orientation.North,
        Direction.Right   => Orientation.East,
        Direction.Down    => Orientation.South,
        Direction.Left    => Orientation.West,
        _ => throw new ArgumentOutOfRangeException(nameof(direction), $"Not
expected direction value: {direction}"),
    };
}
```

```
public static void Main()
{
    var direction = Direction.Right;
    Console.WriteLine($"Map view direction is {direction}");
    Console.WriteLine($"Cardinal orientation is
{ToOrientation(direction)}");
    // Output:
    // Map view direction is Right
    // Cardinal orientation is East
}
}
```

O exemplo anterior mostra os elementos básicos de uma expressão `switch`:

- Uma expressão seguida pela palavra-chave `switch`. No exemplo anterior, é o parâmetro do método `direction`.
- Os `switch`braços da expressão, separados por vírgulas. Cada braço de expressão `switch` contém um padrão, um *protetor de maiúsculas e minúsculas* opcional, o token `=>` e uma expressão.

No exemplo anterior, uma expressão `switch` usa os seguintes padrões:

- Um *padrão constante*: para lidar com os valores definidos da enumeração `Direction`.
- Um *padrão de descarte*: para manipular qualquer valor inteiro que não tenha o membro correspondente da enumeração `Direction` (por exemplo, `(Direction)10`). Isso torna a expressão `switch` exaustiva.

### ⓘ Importante

Para obter informações sobre os padrões compatíveis com a expressão `switch` e mais exemplos, consulte [Padrões](#).

O resultado de uma expressão `switch` é o valor da expressão do primeiro braço da expressão `switch` cujo padrão corresponde à expressão de entrada, e cujo protetor de maiúsculas e minúsculas, se presente, é avaliado como `true`. Os braços da expressão `switch` são avaliados em ordem de texto.

O compilador gera um erro quando um braço de expressão inferior `switch` não pode ser escolhido porque um braço de expressão superior `switch` corresponde a todos os seus valores.

# Protetores de maiúsculas e minúsculas

Um padrão pode não ser expressivo o suficiente para especificar a condição para a avaliação da expressão de um braço. Nesse caso, você pode usar um *protetor de maiúsculas e minúsculas*. Uma *proteção de caso* é outra condição que deve ser atendida junto com um padrão correspondente. Um protetor de maiúsculas e minúsculas deve ser uma expressão booliana. Especifique um protetor de maiúsculas e minúsculas após a palavra-chave `when` que segue um padrão, como mostra o exemplo a seguir:

C#

```
public readonly struct Point
{
    public Point(int x, int y) => (X, Y) = (x, y);

    public int X { get; }
    public int Y { get; }
}

static Point Transform(Point point) => point switch
{
    { X: 0, Y: 0 }                  => new Point(0, 0),
    { X: var x, Y: var y } when x < y => new Point(x + y, y),
    { X: var x, Y: var y } when x > y => new Point(x - y, y),
    { X: var x, Y: var y }           => new Point(2 * x, 2 * y),
};
```

O exemplo anterior usa [padrões de propriedade](#) com [padrões aninhados do var](#).

## Expressões de switch não exaustivas

Se nenhum dos padrões de uma expressão `switch` corresponder a um valor de entrada, o runtime gerará uma exceção. No .NET Core 3.0 e versões posteriores, a exceção é um [System.Runtime.CompilerServices.SwitchExpressionException](#). No .NET Framework, a exceção é um [InvalidOperationException](#). Na maioria dos casos, o compilador gera um aviso se uma expressão `switch` não manipula todos os valores de entrada possíveis. Os [padrões de lista](#) não geram um aviso quando todas as entradas possíveis não são tratadas.

### 💡 Dica

Para garantir que uma expressão `switch` manipule todos os valores de entrada possíveis, forneça um braço de expressão `switch` com um [padrão de descarte](#).

# Especificação da linguagem C#

Para obter mais informações, consulte a seção da [switchexpressão](#) da [nota de proposta de recurso](#).

## Confira também

- [Usar expressão de switch \(regra de estilo IDE0066\)](#)
- [Adicionar casos ausentes para a expressão switch \(regra de estilo IDE0072\)](#)
- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Padrões](#)
- [Tutorial: Usar padrões correspondentes para criar algoritmos controlados por tipo e controlados por dados](#)
- [switchinstrução](#)

# operadores true e false – trate seus objetos como um valor booleano

Artigo • 07/04/2023

O operador `true` retorna `bool` value `true` para indicar que um operando é definitivamente true. O operador `false` retorna `bool` value `true` para indicar que um operando é definitivamente false. Os `true` operadores e `false` não têm garantia de complementar uns aos outros. Ou seja, ambos os operadores `true` e `false` podem retornar o valor `bool false` para o mesmo operando. Se um tipo define um dos dois operadores, ele também deve definir outro operador.

## 💡 Dica

Use o tipo `bool?` se você precisar oferecer suporte à lógica de três valores (por exemplo, ao trabalhar com bancos de dados que dão suporte a um tipo booleano de três valores). C# fornece os operadores `&` e `|` que suportam a lógica de três valores com os operandos `bool?`. Para obter mais informações, confira a seção [Operadores lógicos booleanos anuláveis](#) do artigo [Operadores lógicos booleanos](#).

## Expressões booleanas

Um tipo com o operador `true` definido pode ser o tipo de resultado de uma expressão condicional de controle nas instruções `if`, `do`, `while` e `for` e no operador condicional `?:`. Para saber mais, confira a seção [Expressões booleanas](#) da [Especificação da linguagem C#](#).

## Operadores lógicos condicionais definidos pelo usuário

Se um tipo com os operadores `true` e `false` definidos [sobrecarregar](#) o operador OR lógico `|` ou o operador AND lógico `&` de uma determinada maneira, o operador OR lógico condicional `||` ou operador AND lógico condicional `&&`, respectivamente, poderá ser avaliado para os operandos desse tipo. Para obter mais informações, veja a seção [Operadores lógicos condicionais definidos pelo usuário](#) na [especificação da linguagem C#](#).

# Exemplo

O exemplo a seguir apresenta o tipo que define os dois operadores, `true` e `false`. O tipo também sobrecarrega o operador AND lógico `&` de uma forma que o operador `&&` também possa ser avaliado para os operandos desse tipo.

C#

```
public struct LaunchStatus
{
    public static readonly LaunchStatus Green = new LaunchStatus(0);
    public static readonly LaunchStatus Yellow = new LaunchStatus(1);
    public static readonly LaunchStatus Red = new LaunchStatus(2);

    private int status;

    private LaunchStatus(int status)
    {
        this.status = status;
    }

    public static bool operator true(LaunchStatus x) => x == Green || x == Yellow;
    public static bool operator false(LaunchStatus x) => x == Red;

    public static LaunchStatus operator &(LaunchStatus x, LaunchStatus y)
    {
        if (x == Red || y == Red || (x == Yellow && y == Yellow))
        {
            return Red;
        }

        if (x == Yellow || y == Yellow)
        {
            return Yellow;
        }

        return Green;
    }

    public static bool operator ==(LaunchStatus x, LaunchStatus y) => x.status == y.status;
    public static bool operator !=(LaunchStatus x, LaunchStatus y) => !(x == y);

    public override bool Equals(object obj) => obj is LaunchStatus other && this == other;
    public override int GetHashCode() => status;
}

public class LaunchStatusTest
```

```
public static void Main()
{
    LaunchStatus okToLaunch = GetFuelLaunchStatus() &&
GetNavigationLaunchStatus();
    Console.WriteLine(okToLaunch ? "Ready to go!" : "Wait!");
}

static LaunchStatus GetFuelLaunchStatus()
{
    Console.WriteLine("Getting fuel launch status...");
    return LaunchStatus.Red;
}

static LaunchStatus GetNavigationLaunchStatus()
{
    Console.WriteLine("Getting navigation launch status...");
    return LaunchStatus.Yellow;
}
}
```

Observe o comportamento de curto-circuito do operador `&&`. Quando o `GetFuelLaunchStatus` método retorna `LaunchStatus.Red`, o operando à direita do `&&` operador não é avaliado. Isso ocorre porque `LaunchStatus.Red` é, definitivamente, false. Depois, o resultado do AND lógico não depende do valor do operando à direita. A saída do exemplo é a seguinte:

Console

```
Getting fuel launch status...
Wait!
```

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)

# Expressão with – a mutação não estruturativa cria um objeto com propriedades modificadas

Artigo • 27/03/2023

Disponível no C# 9.0 e posterior, uma expressão `with` produz uma cópia de seu operando com as propriedades e campos especificados modificados. Você usa a sintaxe do [inicializador de objeto](#) para especificar quais membros modificar e seus novos valores:

C#

```
using System;

public class WithExpressionBasicExample
{
    public record NamedPoint(string Name, int X, int Y);

    public static void Main()
    {
        var p1 = new NamedPoint("A", 0, 0);
        Console.WriteLine($"{nameof(p1)}: {p1}"); // output: p1: NamedPoint
        { Name = A, X = 0, Y = 0 }

        var p2 = p1 with { Name = "B", X = 5 };
        Console.WriteLine($"{nameof(p2)}: {p2}"); // output: p2: NamedPoint
        { Name = B, X = 5, Y = 0 }

        var p3 = p1 with
        {
            Name = "C",
            Y = 4
        };
        Console.WriteLine($"{nameof(p3)}: {p3}"); // output: p3: NamedPoint
        { Name = C, X = 0, Y = 4 }

        Console.WriteLine($"{nameof(p1)}: {p1}"); // output: p1: NamedPoint
        { Name = A, X = 0, Y = 0 }

        var apples = new { Item = "Apples", Price = 1.19m };
        Console.WriteLine($"Original: {apples}"); // output: Original: {
        Item = Apples, Price = 1.19 }
        var saleApples = apples with { Price = 0.79m };
        Console.WriteLine($"Sale: {saleApples}"); // output: Sale: { Item =
        Apples, Price = 0.79 }
    }
}
```

No C# 9.0, um operando à esquerda de uma expressão `with` deve ser de um [tipo de registro](#). A partir do C# 10, um operando à esquerda de uma expressão `with` também pode ser de um [tipo estrutura](#) ou de um [tipo anônimo](#).

O resultado de uma expressão `with` tem o mesmo tipo em tempo de execução que o operando da expressão, como mostra o exemplo a seguir:

C#

```
using System;

public class InheritanceExample
{
    public record Point(int X, int Y);
    public record NamedPoint(string Name, int X, int Y) : Point(X, Y);

    public static void Main()
    {
        Point p1 = new NamedPoint("A", 0, 0);
        Point p2 = p1 with { X = 5, Y = 3 };
        Console.WriteLine(p2 is NamedPoint); // output: True
        Console.WriteLine(p2); // output: NamedPoint { X = 5, Y = 3, Name =
A }
    }
}
```

No caso de um membro do tipo referência, somente a referência a uma instância de membro é copiada quando um operando é copiado. O operando da cópia e do original tem acesso à mesma instância de tipo referência. O exemplo a seguir demonstra esse comportamento:

C#

```
using System;
using System.Collections.Generic;

public class ExampleWithReferenceType
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}
```

```

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B, C
    }
}

```

## Semântica de cópia personalizada

Qualquer tipo de classe de registro tem o *construtor de cópia*. Um *construtor de cópia* é um construtor com um único parâmetro do tipo de registro que o contém. Ele copia o estado de seu argumento para uma nova instância de registro. Na avaliação de uma expressão `with`, o construtor de cópia é chamado para instanciar uma nova instância de registro com base em um registro original. Depois disso, a nova instância é atualizada de acordo com as modificações especificadas. Por padrão, o construtor de cópia é implícito, ou seja, gerado pelo compilador. Se você precisar personalizar a semântica de cópia de registro, declare explicitamente um construtor de cópia com o comportamento desejado. O exemplo a seguir atualiza o exemplo anterior com um construtor de cópia explícito. O novo comportamento de cópia é copiar itens de lista em vez de uma referência de lista quando um registro é copiado:

C#

```

using System;
using System.Collections.Generic;

public class UserDefinedCopyConstructorExample
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        protected TaggedNumber(TaggedNumber original)
        {
            Number = original.Number;
            Tags = new List<string>(original.Tags);
        }

        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
    }
}

```

```
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}
```

Não é possível personalizar a semântica de cópia para tipos estrutura.

## Especificação da linguagem C#

Para obter mais informações, consulte as seguintes seções da [observação da proposta do recurso de registros](#):

- [Expressão with](#)
- [Copiar e clonar membros](#)

## Confira também

- [Referência de C#](#)
- [Operadores e expressões C#](#)
- [Registros](#)
- [Tipos de estrutura](#)

# Sobrecarga de operador – operadores unários, aritméticos, de igualdade e de comparação predefinidos

Artigo • 07/04/2023

Um tipo definido pelo usuário pode sobrecarregar um operador C# predefinido. Ou seja, um tipo pode fornecer a implementação personalizada de uma operação caso um ou ambos os operandos sejam desse mesmo tipo. A seção [Operadores sobrecarregáveis](#) mostra quais operadores do C# podem ser sobrecarregados.

Use a palavra-chave `operator` para declarar um operador. Uma declaração de operador deve satisfazer as regras a seguir:

- Ela inclui os modificadores `public` e `static`.
- Um operador unário tem um parâmetro de entrada. Um operador binário tem dois parâmetros de entrada. Em cada caso, pelo menos um parâmetro deve ter o tipo `T` ou `T?`, em que `T` é o tipo que contém a declaração do operador.

O exemplo a seguir define uma estrutura simplificada para representar um número racional. A estrutura sobrecarrega alguns dos [operadores aritméticos](#):

C#

```
public readonly struct Fraction
{
    private readonly int num;
    private readonly int den;

    public Fraction(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new ArgumentException("Denominator cannot be zero.",
nameof(denominator));
        }
        num = numerator;
        den = denominator;
    }

    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num,
a.den);

    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);
```

```

public static Fraction operator -(Fraction a, Fraction b)
=> a + (-b);

public static Fraction operator *(Fraction a, Fraction b)
=> new Fraction(a.num * b.num, a.den * b.den);

public static Fraction operator /(Fraction a, Fraction b)
{
    if (b.num == 0)
    {
        throw new DivideByZeroException();
    }
    return new Fraction(a.num * b.den, a.den * b.num);
}

public override string ToString() => $"{num} / {den}";
}

public static class OperatorOverloading
{
    public static void Main()
    {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a); // output: -5 / 4
        Console.WriteLine(a + b); // output: 14 / 8
        Console.WriteLine(a - b); // output: 6 / 8
        Console.WriteLine(a * b); // output: 5 / 8
        Console.WriteLine(a / b); // output: 10 / 4
    }
}

```

Você pode estender o exemplo anterior definindo uma conversão implícita de `int` em `Fraction`. Em seguida, os operadores sobrecarregados seriam compatíveis com os argumentos desses dois tipos. Ou seja, tornaria-se possível adicionar um inteiro a uma fração e obter uma fração como um resultado.

Use também a palavra-chave `operator` para definir uma conversão de tipo personalizado. Para saber mais, confira [Operadores de conversão definidos pelo usuário](#).

## Operadores sobrecarregáveis

A tabela a seguir mostra os operadores que podem ser sobrecarregados:

Operadores	Anotações
<code>+x, -x, !x, ~x, ++, --, true, false</code>	Os <code>true</code> operadores e <code>false</code> devem ser sobrecarregados juntos.

Operadores	Anotações
<code>x + y, x - y, x * y, x / y, x % y, x &amp; y, x   y, x ^ y, x &lt;&lt; y, x &gt; y, x &gt;&gt; y</code>	
<code>x == y, x != y, x &lt; y, x &gt; y, x &lt;= y, x &gt;= y</code>	Deve ser sobreescrito em pares da seguinte maneira: <code>==</code> e <code>!=</code> , <code>&lt;</code> e <code>&gt;</code> , <code>&lt;=</code> e <code>&gt;=</code> .

## Operadores não sobreescrivíveis

A tabela a seguir mostra os operadores que não podem ser sobreescritos:

Operadores	Alternativas
<code>x &amp;&amp; y, x    y</code>	Sobreescrive os <code>true</code> operadores e <code>false</code> e os <code>&amp;</code> operadores ou <code> </code> . Para obter mais informações, consulte <a href="#">Operadores lógicos condicionais definidos pelo usuário</a> .
<code>a[i], a?[i]</code>	Defina um <a href="#">indexador</a> .
<code>(T)x</code>	Defina conversões de tipo personalizado que podem ser executadas por uma expressão de conversão. Para saber mais, confira <a href="#">Operadores de conversão definidos pelo usuário</a> .
<code>+=, -=, *=, /=, %=, &amp;=,  =, ^=, &lt;&lt;=, &gt;&gt;=, &gt;&gt;&gt;=</code>	Sobreescrive o operador binário correspondente. Por exemplo, quando você sobreescrava o operador binário <code>+</code> , <code>+=</code> é implicitamente sobreescrito.
<code>^x, x = y, x.y, x?.y, c ? t : f, x ?? y, x.y, x-&gt;y, =&gt;, f(x), as, await, checked, unchecked, default, delegate, is, nameof, new, sizeof, stackalloc, switch, typeof, with</code>	Nenhum.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Sobrecarga de operador](#)
- [Operadores](#)

## Confira também

- Referência de C#
- Operadores e expressões C#
- Operadores de conversões definidas pelo usuário
- Diretrizes de design – Sobrecargas do operador
- Diretrizes de design – Operadores de igualdade
- Por que os operadores sobrecarregados sempre são estáticos em C#?

# Instruções de iteração – `for`, `foreach`, `do` e `while`

Artigo • 07/04/2023

As instruções de iteração executam uma instrução ou um bloco de instruções repetidas vezes. [Instrução `for`](#): executa o bloco correspondente enquanto uma expressão booleana especificada é avaliada como `true`. [Instrução `foreach`](#): enumera os elementos de uma coleção e executa o bloco correspondente para cada elemento dessa coleção. [Instrução `do`](#): executa condicionalmente o bloco correspondente uma ou mais vezes. [Instrução `while`](#): executa condicionalmente o bloco correspondente zero ou mais vezes.

A qualquer momento dentro do corpo da instrução de iteração, interrompa o loop usando a instrução [break](#). Você pode passar para a próxima iteração no loop usando a instrução [continue](#).

## A instrução `for`

A instrução `for` executa uma instrução ou um bloco de instruções enquanto uma expressão booleana especificada é avaliada como `true`. O seguinte exemplo mostra a instrução `for` que executa o corpo dela enquanto um contador inteiro é menor que três:

```
C#  
  
for (int i = 0; i < 3; i++)  
{  
    Console.Write(i);  
}  
// Output:  
// 012
```

O exemplo anterior mostra os elementos da instrução `for`:

- A seção *inicializador* é executada apenas uma vez, antes de entrar no loop. Normalmente, você declara e inicializa a variável local do loop nessa seção. A variável declarada não pode ser acessada de fora da instrução `for`.

A seção *inicializador* do exemplo anterior declara e inicializa a variável do contador inteiro:

```
C#
```

```
int i = 0
```

- A seção *condição* determina se a próxima iteração do loop deve ser executada. Se ela for avaliada como `true`, ou se não estiver presente, a próxima iteração será executada; caso contrário, o loop será encerrado. A seção *condição* deve corresponder a uma expressão booliana.

A seção *condição* do exemplo anterior verifica se o valor do contador é menor que três:

```
C#
```

```
i < 3
```

- A seção *iterador* define o que acontece após cada execução do corpo do loop.

A seção *iterador* do exemplo anterior incrementa o contador:

```
C#
```

```
i++
```

- O corpo do loop deve corresponder a uma instrução ou a um bloco de instruções.

A seção iterador pode conter zero ou mais das seguintes expressões de instrução, separadas por vírgulas:

- prefixo ou sufixo da expressão **incrementar**, como `++i` ou `i++`
- prefixo ou sufixo da expressão **decrementar**, como `--i` ou `i--`
- **atribuição**
- invocação de um método
- expressão **await**
- criação de um objeto usando o operador **new**

Se não declarar uma variável de loop na seção inicializador, você poderá usar zero ou mais das expressões da lista anterior na seção inicializador também. O seguinte exemplo mostra vários usos menos comuns das seções inicializador e iterador: atribuindo um valor a uma variável externa na seção do inicializador, invocando um método nas seções inicializador e iterador e alterando os valores de duas variáveis na seção iterador:

```
C#
```

```
int i;
int j = 3;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--,
Console.WriteLine($"Step: i={i}, j={j}"))
{
    //...
}
// Output:
// Start: i=0, j=3
// Step: i=1, j=2
// Step: i=2, j=1
```

Todas as seções da instrução `for` são opcionais. Por exemplo, o seguinte código define um loop `for` infinito:

```
C#
for ( ; ; )
{
    //...
}
```

## A instrução `foreach`

A instrução `foreach` executa uma instrução ou um bloco de instruções para cada elemento em uma instância do tipo que implementa a interface `System.Collections.IEnumerable` ou `System.Collections.Generic.IEnumerable<T>`, como mostra o seguinte exemplo:

```
C#
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int element in fibNumbers)
{
    Console.Write($"{element} ");
}
// Output:
// 0 1 1 2 3 5 8 13
```

A instrução `foreach` não se limita a esses tipos. Você pode usá-la com uma instância de qualquer tipo que satisfaça as seguintes condições:

- Um tipo tem o método público sem parâmetros `GetEnumerator`. Do C# 9.0 em diante, o método `GetEnumerator` pode ser o [método de extensão](#) de um tipo.

- O tipo de retorno do método `GetEnumerator` tem a propriedade pública `Current` e o método público sem parâmetros `MoveNext`, cujo tipo de retorno é `bool`.

O seguinte exemplo usa a instrução `foreach` com uma instância do tipo `System.Span<T>`, que não implementa nenhuma interface:

C#

```
Span<int> numbers = new int[] { 3, 14, 15, 92, 6 };
foreach (int number in numbers)
{
    Console.WriteLine($"{number} ");
}
// Output:
// 3 14 15 92 6
```

Se a propriedade `Current` do enumerador retornar um [valor de retorno de referência](#) (`ref T` em que `T` é o tipo de elemento de uma coleção), você poderá declarar uma variável de iteração com o modificador `ref` ou `ref readonly`, como mostra o seguinte exemplo:

C#

```
Span<int> storage = stackalloc int[10];
int num = 0;
foreach (ref int item in storage)
{
    item = num++;
}
foreach (ref readonly var item in storage)
{
    Console.WriteLine($"{item} ");
}
// Output:
// 0 1 2 3 4 5 6 7 8 9
```

Se a instrução `foreach` for aplicada a `null`, uma [NullReferenceException](#) será gerada. Se a coleção de origem da instrução `foreach` estiver vazia, o corpo da instrução `foreach` não será executado e será ignorado.

## await foreach

Você pode usar a instrução `await foreach` para consumir um fluxo assíncrono de dados, ou seja, o tipo de coleção que implementa a interface `IAsyncEnumerable<T>`. Cada

iteração do loop pode ser suspensa enquanto o próximo elemento é recuperado de maneira assíncrona. O seguinte exemplo mostra como usar a instrução `await foreach`:

C#

```
await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}
```

Você também pode usar a instrução `await foreach` com uma instância de qualquer tipo que atenda às seguintes condições:

- Um tipo tem o método público sem parâmetros `GetAsyncEnumerator`. Esse método pode ser o [método de extensão](#) de um tipo.
- O tipo de retorno do método `GetAsyncEnumerator` tem a propriedade pública `Current` e o método público sem parâmetros `MoveNextAsync`, cujo tipo de retorno é `Task<bool>`, `ValueTask<bool>` ou qualquer outro tipo aguardável cujo método `GetResult` do awaiter retorna um valor `bool`.

Por padrão, os elementos de fluxo são processados no contexto capturado. Se você quiser desabilitar a captura do contexto, use o método de extensão [TaskAsyncEnumerableExtensions.ConfigureAwait](#). Para obter mais informações sobre contextos de sincronização e capturar o contexto atual, confira [Como consumir o padrão assíncrono baseado em tarefa](#). Para obter mais informações sobre fluxos assíncronos, confira o [Tutorial sobre fluxos assíncronos](#).

## Tipo de variável de iteração

Você pode usar a [palavra-chave var](#) para permitir que o compilador infira o tipo da variável de iteração na instrução `foreach`, como mostra o seguinte código:

C#

```
foreach (var item in collection) { }
```

Você também pode especificar explicitamente o tipo da variável de iteração, como mostra o seguinte código:

C#

```
IEnumerable<T> collection = new T[5];
foreach (V item in collection) { }
```

No formulário anterior, o tipo `T` de elemento da coleção deve ser implicitamente ou explicitamente conversível para o tipo `V` da variável de iteração. Se a conversão explícita de `T` para `V` falhar em tempo de execução, a instrução `foreach` vai gerar o erro [InvalidCastException](#). Por exemplo, se `T` for um tipo de classe não selado, `V` pode ser de qualquer tipo de interface, mesmo aquele que `T` não implementa. Em tempo de execução, o tipo de elemento da coleção pode ser aquele que deriva de `T` e, na prática, implementa `V`. Se esse não for o caso, o erro [InvalidCastException](#) é gerado.

## A instrução `do`

A instrução `do` executa uma instrução ou um bloco de instruções enquanto uma expressão booliana especificada é avaliada como `true`. Como essa expressão é avaliada após cada execução do loop, um loop `do` é executado uma ou mais vezes. A instrução `do` é diferente de um loop `while`, que pode ser executado zero ou mais vezes.

O seguinte exemplo mostra o uso da instrução `do`:

C#

```
int n = 0;
do
{
    Console.WriteLine(n);
    n++;
} while (n < 5);
// Output:
// 01234
```

## A instrução `while`

A instrução `while` executa uma instrução ou um bloco de instruções enquanto uma expressão booliana especificada é avaliada como `true`. Como essa expressão é avaliada antes de cada execução do loop, um loop `while` é executado zero ou mais vezes. A instrução `while` é diferente de um loop `do`, que é executado uma ou mais vezes.

O seguinte exemplo mostra o uso da instrução `while`:

C#

```
int n = 0;
while (n < 5)
{
```

```
    Console.WriteLine(n);
    n++;
}
// Output:
// 01234
```

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Instrução for](#)
- [Instrução foreach](#)
- [Instrução do](#)
- [Instrução while](#)

Para obter mais informações sobre os recursos adicionados ao C# 8.0 e versões posteriores, confira as seguintes notas sobre a proposta de recursos:

- [Fluxos assíncronos \(C# 8.0\)](#)
- [Extensão GetEnumerator suporte para loops foreach \(C# 9.0\)](#)

## Confira também

- [Referência de C#](#)
- [Usar foreach com matrizes](#)
- [Iteradores](#)

# Instruções de seleção – `if`, `if-else` e `switch`

Artigo • 05/06/2023

As instruções `if`, `if-else` e `switch` selecionam instruções a serem executadas de vários caminhos possíveis com base no valor de uma expressão. A [instrução `if`](#) executará uma instrução somente se uma expressão booliana fornecida for avaliada como `true`. A [instrução `if-else`](#) permite que você escolha qual dos dois caminhos de código seguir com base em uma expressão booliana. A [instrução `switch`](#) seleciona uma lista de instruções a ser executada com base em uma correspondência de padrão com uma expressão.

## A instrução `if`

Uma instrução `if` pode ter uma das duas formas a seguir:

- Uma instrução `if` com uma parte `else` seleciona uma das duas instruções a serem executadas com base no valor de uma expressão booliana, como mostra o exemplo a seguir:

```
C#  
  
DisplayWeatherReport(15.0); // Output: Cold.  
DisplayWeatherReport(24.0); // Output: Perfect!  
  
void DisplayWeatherReport(double tempInCelsius)  
{  
    if (tempInCelsius < 20.0)  
    {  
        Console.WriteLine("Cold.");  
    }  
    else  
    {  
        Console.WriteLine("Perfect!");  
    }  
}
```

- Uma instrução `if` sem uma parte `else` executará seu corpo somente se uma expressão booliana for avaliada como `true`, conforme mostrado no exemplo a seguir:

```
C#
```

```

DisplayMeasurement(45); // Output: The measurement value is 45
DisplayMeasurement(-3); // Output: Warning: not acceptable value! The
measurement value is -3

void DisplayMeasurement(double value)
{
    if (value < 0 || value > 100)
    {
        Console.WriteLine("Warning: not acceptable value! ");
    }

    Console.WriteLine($"The measurement value is {value}");
}

```

Você pode aninhar instruções `if` para verificar várias condições, como mostra o exemplo a seguir:

C#

```

DisplayCharacter('f'); // Output: A lowercase letter: f
DisplayCharacter('R'); // Output: An uppercase letter: R
DisplayCharacter('8'); // Output: A digit: 8
DisplayCharacter(',') // Output: Not alphanumeric character: ,

void DisplayCharacter(char ch)
{
    if (char.IsUpper(ch))
    {
        Console.WriteLine($"An uppercase letter: {ch}");
    }
    else if (char.IsLower(ch))
    {
        Console.WriteLine($"A lowercase letter: {ch}");
    }
    else if (char.IsDigit(ch))
    {
        Console.WriteLine($"A digit: {ch}");
    }
    else
    {
        Console.WriteLine($"Not alphanumeric character: {ch}");
    }
}

```

Em um contexto de expressão, você pode usar o [operador condicional ?: para avaliar uma das duas expressões com base no valor de uma expressão booleana.](#)

## A instrução `switch`

A instrução `switch` seleciona uma lista de instruções a ser executada com base em uma correspondência de padrão com uma expressão de correspondência, como mostra o exemplo a seguir:

C#

```
DisplayMeasurement(-4); // Output: Measured value is -4; too low.  
DisplayMeasurement(5); // Output: Measured value is 5.  
DisplayMeasurement(30); // Output: Measured value is 30; too high.  
DisplayMeasurement(double.NaN); // Output: Failed measurement.  
  
void DisplayMeasurement(double measurement)  
{  
    switch (measurement)  
    {  
        case < 0.0:  
            Console.WriteLine($"Measured value is {measurement}; too low.");  
            break;  
  
        case > 15.0:  
            Console.WriteLine($"Measured value is {measurement}; too  
high.");  
            break;  
  
        case double.NaN:  
            Console.WriteLine("Failed measurement.");  
            break;  
  
        default:  
            Console.WriteLine($"Measured value is {measurement}.");  
            break;  
    }  
}
```

No exemplo anterior, a instrução `switch` usa os seguintes padrões:

- Um **padrão relacional** (disponível em C# 9.0 e posterior): para comparar um resultado de expressão com uma constante.
- Um **padrão constante**: testa se o resultado de uma expressão é igual a uma constante.

### ⓘ Importante

Para obter informações sobre os padrões com suporte na instrução `switch`, consulte [Padrões](#).

O exemplo anterior também demonstra o caso `default`. O caso `default` especifica instruções a serem executadas quando uma expressão de correspondência não corresponde a nenhum outro padrão de caso. Se uma expressão de correspondência não corresponder a nenhum padrão de caso e não houver nenhum caso de `default`, o controle passará por uma instrução `switch`.

Uma instrução `switch` executa a *lista de instruções* na primeira *seção de comutador* cujo *padrão de caso* corresponder a uma expressão de correspondência e cujo [protetor de maiúsculas e minúsculas](#), se presente, for avaliado como `true`. Uma instrução `switch` avalia padrões de caso na ordem de texto de cima para baixo. O compilador gera um erro quando uma instrução `switch` contém um caso inacessível. Esse é um caso já tratado por um caso superior ou cujo padrão é impossível de corresponder.

### ① Observação

O caso `default` pode aparecer em qualquer lugar dentro de uma instrução `switch`. Qualquer que seja sua posição, o caso `default` será avaliado somente se todos os outros padrões de caso não forem correspondentes ou se a instrução `goto default;` for executada em uma das seções do comutador.

Você pode especificar vários padrões de caso para uma seção de uma instrução `switch`, como mostra o exemplo a seguir:

```
C#  
  
DisplayMeasurement(-4); // Output: Measured value is -4; out of an  
acceptable range.  
DisplayMeasurement(50); // Output: Measured value is 50.  
DisplayMeasurement(132); // Output: Measured value is 132; out of an  
acceptable range.  
  
void DisplayMeasurement(int measurement)  
{  
    switch (measurement)  
    {  
        case < 0:  
        case > 100:  
            Console.WriteLine($"Measured value is {measurement}; out of an  
acceptable range.");  
            break;  
  
        default:  
            Console.WriteLine($"Measured value is {measurement}.");  
            break;  
    }  
}
```

Dentro de uma instrução `switch`, o controle não pode passar de uma seção de comutador para a próxima. Como os exemplos nesta seção mostram, normalmente você usa a instrução `break` no final de cada seção de comutador para passar o controle para fora de uma instrução `switch`. Você também pode usar as instruções `return` e `throw` para passar o controle para fora de uma instrução `switch`. Para imitar o comportamento de fall-through e passar o controle para outra seção de comutador, você pode usar a instrução `goto`.

Em um contexto de expressão, você pode usar a [expressão switch](#) para avaliar uma única expressão de uma lista de expressões candidatas com base em uma correspondência de padrão com uma expressão.

## Protetor de maiúsculas e minúsculas

Um padrão de caso pode não ser expressivo o bastante para especificar a condição da execução da seção de comutador. Nesse caso, você pode usar um *protetor de maiúsculas e minúsculas*. Essa é uma condição adicional que deve ser atendida junto com um padrão correspondente. Um protetor de maiúsculas e minúsculas deve ser uma expressão booliana. Especifique um protetor de maiúsculas e minúsculas após a palavra-chave `when` que segue um padrão, como mostra o exemplo a seguir:

C#

```
DisplayMeasurements(3, 4); // Output: First measurement is 3, second
                           measurement is 4.
DisplayMeasurements(5, 5); // Output: Both measurements are valid and equal
                           to 5.

void DisplayMeasurements(int a, int b)
{
    switch ((a, b))
    {
        case (> 0, > 0) when a == b:
            Console.WriteLine($"Both measurements are valid and equal to
{a}.");
            break;

        case (> 0, > 0):
            Console.WriteLine($"First measurement is {a}, second measurement
is {b}.");
            break;

        default:
            Console.WriteLine("One or both measurements are not valid.");
            break;
    }
}
```

```
    }  
}
```

O exemplo anterior usa [padrões posicionais](#) com [padrões relacionais](#) aninhados.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Instrução if](#)
- [Instrução switch](#)

Para saber mais sobre padrões, confira a seção [Padrões e padrões correspondentes](#) da [especificação da linguagem C#](#).

## Confira também

- [Referência de C#](#)
- [Operador condicional ?:](#)
- [Operadores lógicos](#)
- [Padrões](#)
- [Expressão switch](#)
- [Adicionar casos ausentes à instrução switch \(regra de estilo IDE0010\)](#)

# Instruções de salto – `break`, `continue`, `return` e `goto`

Artigo • 20/03/2023

Quatro instruções C# transferem incondicionalmente o controle. A `break` instrução encerra a [instrução de iteração](#) ou `switch` instrução mais próxima. A `continue` instrução inicia uma nova iteração da [instrução de iteração](#) mais próxima. A `return` instrução encerra a execução da função na qual ela aparece e retorna o controle ao chamador. O modificador `ref` em uma instrução `return` indica que a expressão retornada é retornada *por referência*, não *por valor*. A `goto` instrução transfere o controle para uma instrução marcada por um rótulo.

Para obter informações sobre a instrução `throw` que gera uma exceção e também transfere incondicionalmente o controle, confira [throw](#).

## A instrução `break`

A instrução `break` encerra a [instrução de iteração](#) mais próxima (ou seja, loop `for`, `foreach`, `while` ou `do`) ou [instrução switch](#). A instrução `break` transfere o controle para a instrução que segue a instrução encerrada, se houver.

C#

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int number in numbers)
{
    if (number == 3)
    {
        break;
    }

    Console.WriteLine($"{number} ");
}
Console.WriteLine();
Console.WriteLine("End of the example.");
// Output:
// 0 1 2
// End of the example.
```

Em loops aninhados, a instrução `break` encerra apenas o loop mais interno que a contém, como mostra o seguinte exemplo:

C#

```
for (int outer = 0; outer < 5; outer++)
{
    for (int inner = 0; inner < 5; inner++)
    {
        if (inner > outer)
        {
            break;
        }

        Console.WriteLine($"{inner} ");
    }
    Console.WriteLine();
}

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

Quando você usa a instrução `switch` dentro de um loop, uma instrução `break` ao final de uma seção `switch` transfere o controle somente para fora da instrução `switch`. O loop que contém a instrução `switch` não é afetado, como mostra o seguinte exemplo:

C#

```
double[] measurements = { -4, 5, 30, double.NaN };
foreach (double measurement in measurements)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too
high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}

// Output:
```

```
// Measured value is -4; too low.  
// Measured value is 5.  
// Measured value is 30; too high.  
// Failed measurement.
```

## A instrução continue

A instrução `continue` inicia uma nova iteração da [instrução de iteração](#) mais próxima (ou seja, loop `for`, `foreach`, `while` ou `do`), como mostra o seguinte exemplo:

C#

```
for (int i = 0; i < 5; i++)  
{  
    Console.WriteLine($"Iteration {i}: ");  
  
    if (i < 3)  
    {  
        Console.WriteLine("skip");  
        continue;  
    }  
  
    Console.WriteLine("done");  
}  
// Output:  
// Iteration 0: skip  
// Iteration 1: skip  
// Iteration 2: skip  
// Iteration 3: done  
// Iteration 4: done
```

## A instrução return

A instrução `return` encerra a execução da função em que aparece e devolve o controle e o resultado da função, se houver, ao chamador.

Se um membro da função não computar um valor, você usará a instrução `return` sem expressão, como mostra o seguinte exemplo:

C#

```
Console.WriteLine("First call:");  
DisplayIfNecessary(6);  
  
Console.WriteLine("Second call:");  
DisplayIfNecessary(5);
```

```
void DisplayIfNecessary(int number)
{
    if (number % 2 == 0)
    {
        return;
    }

    Console.WriteLine(number);
}
// Output:
// First call:
// Second call:
// 5
```

Como mostra o exemplo anterior, normalmente você usa a instrução `return` sem expressão para encerrar um membro da função antecipadamente. Se um membro da função não contiver a instrução `return`, ele será encerrado após a última instrução ser executada.

Se um membro da função calcular um valor, você usará a instrução `return` com uma expressão, como mostra o seguinte exemplo:

C#

```
double surfaceArea = CalculateCylinderSurfaceArea(1, 1);
Console.WriteLine($"{surfaceArea:F2}"); // output: 12.57

double CalculateCylinderSurfaceArea(double baseRadius, double height)
{
    double baseArea = Math.PI * baseRadius * baseRadius;
    double sideArea = 2 * Math.PI * baseRadius * height;
    return 2 * baseArea + sideArea;
}
```

Quando a instrução `return` tem uma expressão, essa expressão deve ser implicitamente conversível para o tipo de retorno de um membro da função, a menos que seja [assíncrona](#). A expressão retornada de uma função `async` deve ser implicitamente conversível para o argumento de tipo de `Task<TResult>` ou `ValueTask<TResult>`, seja qual for o tipo de retorno da função. Se o tipo de retorno de uma função `async` for `Task` ou `ValueTask`, você usará a instrução `return` sem expressão.

Por padrão, a instrução `return` retorna o valor de uma expressão. Você pode retornar uma referência a uma variável. Para fazer isso, use a instrução `return` com a [palavra-chave ref](#), como mostra o seguinte exemplo:

C#

```

var xs = new int[] { 10, 20, 30, 40 };
ref int found = ref FindFirst(xs, s => s == 30);
found = 0;
Console.WriteLine(string.Join(" ", xs)); // output: 10 20 0 40

ref int FindFirst(int[] numbers, Func<int, bool> predicate)
{
    for (int i = 0; i < numbers.Length; i++)
    {
        if (predicate(numbers[i]))
        {
            return ref numbers[i];
        }
    }
    throw new InvalidOperationException("No element satisfies the given
condition.");
}

```

## Retornos de referências

Os valores retornados podem ser retornados por referência (`ref` retorna). Um valor retornado por referência permite que um método retorne uma referência a uma variável, em vez de um valor, de volta para um chamador. O chamador pode optar por tratar a variável retornada como se tivesse sido retornada por valor ou referência. O chamador pode criar uma nova variável que seja uma referência ao valor retornado, chamado de `ref local`. Um *valor retornado de referência* significa que um método retorna uma *referência* (ou um alias) para alguma variável. O escopo da variável deve incluir o método. O tempo de vida da variável deve ultrapassar o retorno do método. As modificações no valor retornado do método pelo chamador são feitas na variável que é retornada pelo método.

Declarar que um método retorna um *valor retornado de referência* indica que o método retorna um alias para uma variável. A intenção de design geralmente é que chamar código acessa essa variável por meio do alias, inclusive para modificá-la. Métodos retornados por referência não podem ter o tipo de retorno `void`.

O valor retornado `ref` é um alias para outra variável no escopo do método chamado. Você pode interpretar qualquer uso do retorno de `ref` como usando a variável da qual ele é um alias:

- Ao atribuir o valor, você atribui um valor à variável da qual ele é um alias.
- Ao ler o valor, você lê o valor da variável da qual ele é um alias.
- Se o retornar *por referência*, você retornará um alias para a mesma variável.
- Se o passar para outro método *por referência*, você passará uma referência à variável da qual ele é um alias.

- Ao criar um alias de [referência local](#), você cria um novo alias para a mesma variável.

Um retorno de referência deve ser [ref\\_safe\\_to\\_escape](#) ao método de chamada. Isso significa que:

- O valor retornado deve ter um tempo de vida que ultrapasse a execução do método. Em outras palavras, não pode ser uma variável local no método que o retorna. Ele pode ser uma instância ou um campo estático de uma classe ou pode ser um argumento passado para o método. Tentar retornar a uma variável local gera o erro do compilador CS8168, "não é possível retornar o 'obj' local por referência porque ele não é um ref local."
- O valor retornado não pode ser um `null` literal. Um método com um retorno de referência pode retornar um alias para uma variável cujo valor é atualmente o valor nulo `null` (não instanciado) ou um [tipo de valor anulável](#) para um tipo de valor.
- O valor retornado não pode ser uma constante, um membro de enumeração, o valor retornado por valor de uma propriedade ou um método `class` ou `struct`.

Além disso, valores retornados de referência não são permitidos em métodos assíncronos. Um método assíncrono pode retornar antes de concluir a execução, enquanto o valor retornado ainda é desconhecido.

Um método que retorna um *valor retornado de referência* deve:

- Inclua a palavra-chave `ref` na frente do tipo de retorno.
- Cada instrução `return` no corpo do método inclui a palavra-chave `ref` antes do nome da instância retornada.

O exemplo a seguir mostra um método que satisfaz essas condições e retorna uma referência a um objeto `Person` chamado `p`:

C#

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

## A instrução goto

A instrução `goto` transfere o controle para uma instrução marcada por um rótulo, como mostra o seguinte exemplo:

C#

```
var matrices = new Dictionary<string, int[][]>
{
    ["A"] = new[]
    {
        new[] { 1, 2, 3, 4 },
        new[] { 4, 3, 2, 1 }
    },
    ["B"] = new[]
    {
        new[] { 5, 6, 7, 8 },
        new[] { 8, 7, 6, 5 }
    },
};

CheckMatrices(matrices, 4);

void CheckMatrices(Dictionary<string, int[][]> matrixLookup, int target)
{
    foreach (var (key, matrix) in matrixLookup)
    {
        for (int row = 0; row < matrix.Length; row++)
        {
            for (int col = 0; col < matrix[row].Length; col++)
            {
                if (matrix[row][col] == target)
                {
                    goto Found;
                }
            }
        }
        Console.WriteLine($"Not found {target} in matrix {key}.");
        continue;

    Found:
        Console.WriteLine($"Found {target} in matrix {key}.");
    }
}

// Output:
// Found 4 in matrix A.
// Not found 4 in matrix B.
```

Como mostra o exemplo anterior, você pode usar a instrução `goto` para sair de um loop aninhado.

### 💡 Dica

Ao trabalhar com loops aninhados, considere refatorar loops separados em métodos separados. Isso pode resultar em um código mais simples e legível, sem a instrução `goto`.

Você também pode usar a instrução `goto` na [instrução switch](#) a fim de transferir o controle para uma seção switch com um rótulo case constante, como mostra o seguinte exemplo:

```
C#  
  
using System;  
  
public enum CoffeeChoice  
{  
    Plain,  
    WithMilk,  
    WithIceCream,  
}  
  
public class GotoInSwitchExample  
{  
    public static void Main()  
    {  
        Console.WriteLine(CalculatePrice(CoffeeChoice.Plain)); // output:  
10.0  
        Console.WriteLine(CalculatePrice(CoffeeChoice.WithMilk)); //  
output: 15.0  
        Console.WriteLine(CalculatePrice(CoffeeChoice.WithIceCream)); //  
output: 17.0  
    }  
  
    private static decimal CalculatePrice(CoffeeChoice choice)  
    {  
        decimal price = 0;  
        switch (choice)  
        {  
            case CoffeeChoice.Plain:  
                price += 10.0m;  
                break;  
  
            case CoffeeChoice.WithMilk:  
                price += 5.0m;  
                goto case CoffeeChoice.Plain;  
  
            case CoffeeChoice.WithIceCream:  
                price += 7.0m;  
                goto case CoffeeChoice.Plain;  
        }  
        return price;  
    }  
}
```

Na instrução `switch`, você também pode usar a instrução `goto default;` para transferir o controle para a seção switch com o rótulo `default`.

Se um rótulo com o nome fornecido não existir no membro da função atual ou se a instrução `goto` não estiver dentro do escopo do rótulo, ocorrerá um erro em tempo de compilação. Ou seja, você não pode usar a instrução `goto` para transferir o controle do membro da função atual ou para qualquer escopo aninhado, por exemplo, um bloco `try`.

## Especificação da linguagem C#

Para obter mais informações, confira as seguintes seções da [especificação da linguagem C#](#):

- [Instrução break](#)
- [Instrução continue](#)
- [Instrução return](#)
- [Instrução goto](#)

## Confira também

- [Referência de C#](#)
- [Instrução yield](#)

# Instrução lock – garantir o acesso exclusivo a um recurso compartilhado

Artigo • 05/06/2023

A instrução `lock` obtém o bloqueio de exclusão mútua para um determinado objeto, executa um bloco de instruções e, em seguida, libera o bloqueio. Embora um bloqueio seja mantido, o thread que mantém o bloqueio pode adquiri-lo novamente e liberá-lo. Qualquer outro thread é impedido de adquirir o bloqueio e aguarda até que ele seja liberado. A instrução `lock` garante que, no máximo, apenas um thread execute seu corpo a qualquer momento.

A instrução `lock` está no formato

C#

```
lock (x)
{
    // Your code...
}
```

em que `x` é uma expressão de um [tipo de referência](#). Ela é precisamente equivalente a

C#

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

Como o código usa uma [instrução try-finally](#), o bloqueio será liberado mesmo se uma exceção for gerada dentro do corpo de uma instrução `lock`.

Não é possível usar a [expressão await](#) no corpo de uma instrução `lock`.

## Diretrizes

Ao sincronizar o acesso de thread com um recurso compartilhado, bloqueie uma instância de objeto dedicada (por exemplo, `private readonly object balanceLock = new object();`) ou outra instância que provavelmente não será usada como um objeto de bloqueio por partes não relacionadas do código. Evite usar a mesma instância de objeto de bloqueio para diferentes recursos compartilhados, uma vez que ela poderia resultar em deadlock ou contenção de bloqueio. Especificamente, evite usar as seguintes instâncias como objetos de bloqueio:

- `this`, uma vez que pode ser usado pelos chamadores como um bloqueio.
- Instâncias `Type`, pois elas podem ser obtidas pelo operador ou reflexão `typeof`.
- Instâncias de cadeia de caracteres, incluindo literais de cadeia de caracteres, pois podem ser [internalizadas](#).

Mantenha um bloqueio pelo menor tempo possível para reduzir a contenção de bloqueio.

## Exemplo

O exemplo a seguir define uma classe `Account` que sincroniza o acesso com seu campo privado `balance` bloqueando uma instância `balanceLock` dedicada. Usar a mesma instância para bloquear garante que o campo `balance` não pode ser atualizado simultaneamente por dois threads que tentam chamar os métodos `Debit` ou `Credit` simultaneamente.

C#

```
using System;
using System.Threading.Tasks;

public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;

    public Account(decimal initialBalance) => balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit amount cannot be negative.");
        }

        decimal appliedAmount = 0;
        lock (balanceLock)
```

```
        {
            if (balance >= amount)
            {
                balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The
credit amount cannot be negative.");
        }

        lock (balanceLock)
        {
            balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (balanceLock)
        {
            return balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
    }

    static void Update(Account account)
    {
        decimal[] amounts = { 0, 2, -3, 6, -2, -1, 8, -5, 11, -6 };
        foreach (var amount in amounts)
        {
            if (amount >= 0)
```

```
        {
            account.Credit(amount);
        }
        else
        {
            account.Debit(Math.Abs(amount));
        }
    }
}
```

## Especificação da linguagem C#

Para saber mais, confira a seção [A instrução lock](#) na especificação da linguagem C#.

## Confira também

- [Referência de C#](#)
- [System.Threading.Monitor](#)
- [System.Threading.SpinLock](#)
- [System.Threading.Interlocked](#)
- [Visão geral dos primitivos de sincronização](#)
- [Introdução ao System.Threading.Channels ↗](#)

# Caracteres especiais de C#

Artigo • 07/04/2023

Os caracteres especiais são predefinidos e contextuais, e modificam o elemento de programa (uma cadeia de caracteres literal, um identificador ou um nome de atributo) ao qual eles são acrescentados. O C# dá suporte aos seguintes caracteres especiais:

- `@`, o caractere identificador do texto.
- `$`, o caractere da cadeia de caracteres interpolada.

Esta seção inclui apenas os tokens que não são operadores. Consulte a seção [operadores](#) para ver todos os operadores.

## Confira também

- [Referência de C#](#)
- [Guia de Programação em C#](#)

# Interpolação de cadeia de caracteres usando \$

Artigo • 05/06/2023

O caractere especial `$` identifica um literal de cadeia de caracteres como uma *cadeia de caracteres interpolada*. Uma cadeia de caracteres interpolada é um literal de cadeia de caracteres que pode conter *expressões de interpolação*. Quando uma cadeia de caracteres interpolada é resolvida em uma cadeia de caracteres de resultado, itens com expressões de interpolação são substituídos pelas representações de cadeia de caracteres dos resultados da expressão.

A interpolação de cadeia de caracteres fornece uma sintaxe mais acessível e prática para as cadeias de caracteres de formato. Ela é mais fácil de ler do que a [formatação composta de cadeia de caracteres](#). Compare o exemplo a seguir que usa os dois recursos para produzir a mesma saída:

C#

```
string name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name,
date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's
{date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

## Estrutura de uma cadeia de caracteres interpolada

Para identificar uma literal de cadeia de caracteres como uma cadeia de caracteres interpolada, preceda-o com o símbolo `$`. Não pode haver nenhum espaço em branco entre o `$` e `"` que iniciam um literal de cadeia de caracteres. Para concatenar várias cadeias de caracteres interpoladas, adicione o caractere especial `$` a cada literal de cadeia de caracteres.

A estrutura de um item com uma expressão de interpolação é da seguinte maneira:

C#

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

Os elementos entre colchetes são opcionais. A seguinte tabela descreve cada elemento:

Elemento	Descrição
interpolationExpression	A expressão que produz um resultado a ser formatado. A representação da cadeia de caracteres de <code>null</code> é <a href="#">String.Empty</a> .
alignment	A expressão de constante cujo valor define o número mínimo de caracteres da representação de cadeia de caracteres do resultado da expressão. Se for positiva, a representação de cadeia de caracteres será alinhada à direita; se for negativa, será alinhada à esquerda. Para obter mais informações, consulte <a href="#">Componente de alinhamento</a> .
formatString	Uma cadeia de caracteres de formato compatível com o tipo do resultado da expressão. Para obter mais informações, consulte <a href="#">Componente da cadeia de caracteres de formato</a> .

O exemplo a seguir usa os componentes opcionais de formatação descritos acima:

C#

```
Console.WriteLine($"|{ "Left", -7 }|{ "Right", 7 }|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of
the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three
decimal digits of the pi number");
// Expected output is:
// |Left    | Right|
//      3.14159265358979 - default formatting of the pi number
//                      3.142 - display only three decimal digits of the pi number
```

A partir do C# 10, você pode usar a interpolação de cadeia de caracteres para inicializar uma cadeia de caracteres constante. Todas as expressões usadas para espaços reservados devem ser cadeias de caracteres constantes. Em outras palavras, cada *expressão de interpolação* deve ser uma cadeia de caracteres e uma constante em tempo de compilação.

A partir do C# 11, as expressões interpoladas podem incluir novas linhas. O texto entre o `{` e o `}` deve ser o C# válido. Portanto, pode incluir novas linhas que melhoram a legibilidade. O exemplo a seguir mostra como as novas linhas podem melhorar a legibilidade de uma expressão que envolve padrões correspondentes:

C#

```
string message = $"The usage policy for {safetyScore} is {
    safetyScore switch
    {
        > 90 => "Unlimited usage",
        > 80 => "General usage, with daily safety check",
        > 70 => "Issues must be addressed within 1 week",
        > 50 => "Issues must be addressed within 1 day",
        _ => "Issues must be addressed before continued use",
    }
}";
```

Além disso, a partir do C# 11, você pode usar um [literal de cadeia de caracteres bruto](#) para a cadeia de caracteres de formato:

C#

```
int X = 2;
int Y = 3;

var pointMessage = $"""The point "{X}, {Y}" is {Math.Sqrt(X * X + Y * Y)}
from the origin""";

Console.WriteLine(pointMessage);
// output: The point "2, 3" is 3.605551275463989 from the origin.
```

Você pode usar vários caracteres `$` em um literal de cadeia de caracteres bruto interpolado, para inserir os caracteres `{` e `}` na cadeia de caracteres de saída, sem escapar deles:

C#

```
int X = 2;
int Y = 3;

var pointMessage = $$"""The point {{X}}, {{Y}} is {{Math.Sqrt(X * X + Y *
Y)}} from the origin""";
Console.WriteLine(pointMessage);
// output: The point {2, 3} is 3.605551275463989 from the origin.
```

Caso a cadeia de caracteres de saída deva conter os caracteres `{` ou `}` repetidos, você pode adicionar mais `$` para designar a cadeia de caracteres interpolada. Qualquer sequência de `{` ou `}` menor que o número de `$` será inserida na cadeia de caracteres de saída. Conforme mostrado no exemplo anterior, as sequências maiores que a sequência de caracteres `$` inserem os caracteres adicionais `{` ou `}` na saída. O

compilador emitirá um erro se a sequência de caracteres de chave for igual ou maior que o dobro do comprimento da sequência de caracteres `$`.

Você pode experimentar esses recursos usando o SDK do .NET 7. Ou, se você tiver o SDK do .NET 6.00.200 ou posterior, pode definir o elemento `<LangVersion>` no arquivo `csproj` como `preview`.

## Caracteres especiais

Para incluir uma chave, `"{"` ou `"}"`, no texto produzido por uma cadeia de caracteres interpolada, use duas chaves, `"{{" ou "}}"`. Para obter mais informações, consulte [Chaves de escape](#).

Como os dois-pontos (`:`) têm um significado especial em um item de expressão de interpolação, para usar um [operador condicional](#) em uma expressão de interpolação, coloque essa expressão entre parênteses.

O exemplo a seguir mostra como incluir uma chave em uma cadeia de caracteres de resultado. Ele também mostra como usar um operador condicional:

C#

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for
a reply :{-{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-
// Horace is 34 years old.
```

Uma cadeia de caracteres verbatim interpolada começa com o caractere `$`, seguido pelo caractere `@`. Você pode usar os tokens `$` e `@` em qualquer ordem: ambas, `$@"..."` e `@$"..."`, são cadeias de caracteres verbatim interpoladas válidas. Para obter mais informações sobre cadeias de caracteres textuais, confira os artigos [cadeia de caracteres](#) e [identificador textual](#).

## Conversões implícitas e como especificar a implementação `IFormatProvider`

Há três conversões implícitas de uma cadeia de caracteres interpolada:

1. Conversão de uma cadeia de caracteres interpolada em uma instância [String](#). A cadeia de caracteres é o resultado da resolução de cadeia de caracteres interpolada. Todos os itens de expressão de interpolação são substituídos pelas representações de cadeia de caracteres formatadas corretamente dos resultados. Essa conversão usa o [CurrentCulture](#) para formatar os resultados da expressão.
2. Conversão de uma cadeia de caracteres interpolada em uma instância de [FormattableString](#), que representa uma cadeia de caracteres de formato composto, juntamente com os resultados da expressão a ser formatada. Isso permite criar várias cadeias de caracteres de resultado com conteúdo específico da cultura com base em uma única instância [FormattableString](#). Para fazer isso, chame um dos seguintes métodos:
  - Uma sobrecarga [ToString\(\)](#) que produza uma cadeia de caracteres de resultado para a [CurrentCulture](#).
  - Um método [Invariant](#) que produz uma cadeia de caracteres de resultado para a [InvariantCulture](#).
  - Um método [ToString\(IFormatProvider\)](#) que produza uma cadeia de caracteres de resultado para uma cultura específica.

O [ToString\(IFormatProvider\)](#) fornece uma implementação definida pelo usuário da interface [IFormatProvider](#), que permite formatação personalizada. Para obter mais informações, confira a seção [Formatação personalizada com ICustomFormatter](#) do artigo [Formatação de tipos no .NET](#).

3. Conversão de uma cadeia de caracteres interpolada em uma instância [IFormattable](#), que também permite criar várias cadeias de caracteres de resultado com conteúdo específico da cultura com base em uma única instância [IFormattable](#).

O exemplo a seguir usa a conversão implícita em [FormattableString](#) para a criação de cadeias de caracteres de resultado específicas de cultura:

C#

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3}
km/s.;

System.Globalization.CultureInfo.CurrentCulture =
System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = message.ToString();

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-
IN");
string messageInSpecificCulture = message.ToString(specificCulture);
```

```
string messageInInvariantCulture = FormattableString.Invariant(message);

Console.WriteLine(${System.Globalization.CultureInfo.CurrentCulture,-10}
{messageInCurrentCulture});
Console.WriteLine(${specificCulture,-10} {messageInSpecificCulture});
Console.WriteLine(${"Invariant",-10} {messageInInvariantCulture});
// Expected output is:
// nl-NL      The speed of light is 299.792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant  The speed of light is 299,792.458 km/s.
```

## Outros recursos

Se não estiver familiarizado com a interpolação de cadeia de caracteres, confira o tutorial [Interpolação de cadeia de caracteres no C# Interativo](#). Você também pode verificar outro tutorial [Interpolação de cadeia de caracteres no C#](#). Esse tutorial demonstra como usar cadeias de caracteres interpoladas para produzir cadeias de caracteres formatadas.

## Compilação de cadeias de caracteres interpoladas

Se uma cadeia de caracteres interpolada tiver o tipo `string`, ela normalmente será transformada em uma chamada de método `String.Format`. O compilador pode substituir `String.Format` por `String.Concat` se o comportamento analisado for equivalente à concatenação.

Se uma cadeia de caracteres interpolada tiver o tipo `IFormattable` ou `FormattableString`, o compilador gerará uma chamada para o método `FormattableStringFactory.Create`.

A partir do C# 10, quando uma cadeia de caracteres interpolada é usada, o compilador verifica se a cadeia de caracteres interpolada é atribuída a um tipo que atende ao *padrão do manipulador de cadeia de caracteres interpolada*. Um *manipulador de cadeia de caracteres interpolada* é um tipo personalizado que converte a cadeia de caracteres interpolada em uma cadeia de caracteres. Um manipulador de cadeia de caracteres interpolada é um cenário avançado, normalmente usado por motivos de desempenho. Você pode saber mais sobre os requisitos para criar um manipulador de cadeia de caracteres interpolada na especificação de linguagem para [melhorias da cadeia de caracteres interpolada](#). Você pode compilar um ao seguir o [tutorial do manipulador de cadeia de caracteres interpolada](#) na seção Novidades do C#. No .NET 6, quando você usa uma cadeia de caracteres interpolada para um argumento do tipo `string`, a cadeia

de caracteres interpolada é processada pelo [System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#).

#### ⓘ Observação

Um efeito colateral dos manipuladores de cadeia de caracteres interpolada é que um manipulador personalizado, incluindo [System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#), pode não avaliar todas as expressões usadas como espaços reservados na cadeia de caracteres interpolada em todas as condições. Isso significa que os efeitos colaterais nessas expressões podem não ocorrer.

## Especificação da linguagem C#

Para obter mais informações, confira a seção [Cadeia de caracteres interpolada](#) da seção [Especificação de linguagem C#](#), a especificação de recurso [C# 11 - Literais de cadeia de caracteres bruta](#) especificação de recurso [C# 11 - Novas linhas em interpolações de cadeia de caracteres](#).

## Confira também

- [Simplificar a interpolação \(regra de estilo IDE0071\)](#)
- [Referência de C#](#)
- [Caracteres especiais de C#](#)
- [Cadeias de caracteres](#)
- [Cadeias de caracteres de formato numérico padrão](#)
- [Formatação de composição](#)
- [String.Format](#)

# Texto verbatim – @ em variáveis, atributos e literais de cadeia de caracteres

Artigo • 27/03/2023

O caractere especial `@` serve como um identificador textual. Ele é usado das seguintes maneiras:

1. Para indicar que um literal de cadeia de caracteres é interpretado de forma textual.

O caractere `@` neste exemplo define um *literal de cadeia de caracteres textual*.

Sequências de escape simples (como `"\\\"` para uma barra invertida), sequências de escape hexadecimais (como um `"\x0041"` para um A maiúsculo) e sequências de escape Unicode (como `"\u0041"` para um A maiúsculo) são interpretadas de forma textual. Somente uma sequência de escape de aspas (`""`) não é interpretada literalmente; ela produz aspas duplas. Além disso, no caso de uma **cadeia de caracteres interpolada** textual, as sequências de escape de chave (`{}{}` e `{}{}`) não são interpretadas de forma textual; elas geram caracteres de chave única. O exemplo a seguir define dois caminhos de arquivo idênticos, um usando um literal de cadeia de caracteres regular e o outro usando um literal de cadeia de caracteres textual. Este é um dos usos mais comuns de literais de cadeias de caracteres textuais.

C#

```
string filename1 = @"c:\documents\files\u0066.txt";
string filename2 = "c:\\documents\\files\\u0066.txt";

Console.WriteLine(filename1);
Console.WriteLine(filename2);
// The example displays the following output:
//      c:\documents\files\u0066.txt
//      c:\documents\files\u0066.txt
```

O exemplo a seguir ilustra o efeito de definir um literal de cadeia de caracteres regular e um literal de cadeia de caracteres textual que contêm sequências de caracteres idênticas.

C#

```
string s1 = "He said, \"This is the last \u0063hance\x0021\"";
string s2 = @"He said, ""This is the last \u0063hance\x0021""";
```

```
Console.WriteLine(s1);
Console.WriteLine(s2);
// The example displays the following output:
//      He said, "This is the last chance!"
//      He said, "This is the last \u0063hance\x0021"
```

2. Para usar palavras-chave C# como identificadores. O caractere @ atua como prefixo de um elemento de código que o compilador deve interpretar como um identificador e não como uma palavra-chave de C#. O exemplo a seguir usa o caractere @ para definir um identificador chamado `for` que usa em um loop `for`.

C#

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
    Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//      Here is your gift, John!
//      Here is your gift, James!
//      Here is your gift, Joan!
//      Here is your gift, Jamie!
```

3. Para habilitar o compilador a distinguir entre os atributos em caso de um conflito de nomenclatura. Um atributo é um tipo que deriva de `Attribute`. Seu nome de tipo normalmente inclui o sufixo `Attribute`, embora o compilador não imponha essa convenção. O atributo pode, então, ser referenciado no código por seu nome de tipo completo (por exemplo, `[InfoAttribute]` ou pelo nome abreviado (por exemplo, `[Info]`). No entanto, um conflito de nomenclatura ocorre se dois nomes de tipo abreviados forem idênticos e um nome de tipo incluir o sufixo `Attribute` e o outro não incluir. Por exemplo, o código a seguir não é compilado porque o compilador não consegue determinar se o atributo `Info` ou `InfoAttribute` é aplicado à classe `Example`. Para mais informações, confira [CS1614](#).

C#

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
    private string information;

    public Info(string info)
    {
```

```
        information = info;
    }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
    private string information;

    public InfoAttribute(string info)
    {
        information = info;
    }
}

[Info("A simple executable.")] // Generates compiler error CS1614.
// Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info' ([@Info("A simple executable.")]).
// Specify the full name 'InfoAttribute' to select it.
public class Example
{
    [InfoAttribute("The entry point.")]
    public static void Main()
    {
    }
}
```

## Confira também

- Referência de C#
- Guia de Programação em C#
- Caracteres especiais de C#

# Atributos de nível de assembly interpretados pelo compilador C#

Artigo • 10/05/2023

A maioria dos atributos são aplicados aos elementos específicos de linguagem, como classes ou métodos. No entanto, alguns atributos são globais. Eles se aplicam a um assembly inteiro ou módulo. Por exemplo, o atributo [AssemblyVersionAttribute](#) pode ser usado para inserir informações de versão em um assembly, desta maneira:

C#

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Os atributos globais aparecem no código-fonte depois de qualquer diretiva `using` de nível superior e antes de qualquer declaração de namespace, de módulo ou de tipo. Os atributos globais podem aparecer em vários arquivos de origem, mas os arquivos devem ser compilados em uma única passagem de compilação. O Visual Studio adiciona atributos globais ao arquivo `AssemblyInfo.cs` em projetos do .NET Framework. Esses atributos não são adicionados a projetos do .NET Core.

Os atributos de assembly são valores que fornecem informações sobre um assembly. Eles se enquadram nas seguintes categorias:

- Atributos de identidade do assembly
- Atributos informativos
- Atributos de manifesto do assembly

## Atributos de identidade do assembly

Três atributos (com um nome forte, se aplicável) determinam a identidade de um assembly: nome, versão e cultura. Esses atributos formam o nome completo do assembly e são necessários ao fazer referência a ele no código. Você pode definir a versão e a cultura de um assembly, usando atributos. No entanto, o valor do nome é definido pelo compilador, pelo Visual Studio IDE na [caixa de diálogo Informações do Assembly](#) ou pelo vinculador do assembly (Al.exe) quando o assembly é criado. O nome do assembly é baseado no manifesto do assembly. O atributo [AssemblyFlagsAttribute](#) especifica se várias cópias do assembly podem coexistir.

A tabela a seguir mostra os atributos de identidade.

Atributo	Finalidade
<a href="#">AssemblyVersionAttribute</a>	Especifica a versão de um assembly.
<a href="#">AssemblyCultureAttribute</a>	Especifica a qual cultura o assembly dá suporte.
<a href="#">AssemblyFlagsAttribute</a>	Especifica se um assembly dá suporte à execução lado a lado no mesmo computador, no mesmo processo ou no mesmo domínio do aplicativo.

## Atributos informativos

Você usa atributos informativos para fornecer informações adicionais da empresa ou do produto para um assembly. A tabela a seguir mostra os atributos informativos definidos no namespace [System.Reflection](#).

Atributo	Finalidade
<a href="#">AssemblyProductAttribute</a>	Especifica um nome de produto para um manifesto do assembly.
<a href="#">AssemblyTrademarkAttribute</a>	Especifica uma marca para um manifesto do assembly.
<a href="#">AssemblyInformationalVersionAttribute</a>	Especifica uma versão informativa para um manifesto do assembly.
<a href="#">AssemblyCompanyAttribute</a>	Especifica um nome de empresa para um manifesto do assembly.
<a href="#">AssemblyCopyrightAttribute</a>	Define um atributo personalizado que especifica os direitos autorais para um manifesto do assembly.
<a href="#">AssemblyFileVersionAttribute</a>	Define um número de versão específico para o recurso de versão do arquivo Win32.
<a href="#">CLSCCompliantAttribute</a>	Indica se o assembly está em conformidade com a CLS (Common Language Specification).

## Atributos de manifesto do assembly

Você pode usar atributos de manifesto do assembly para fornecer informações no manifesto do assembly. Os atributos incluem título, descrição, alias padrão e configuração. A tabela a seguir mostra os atributos de manifesto do assembly definidos no namespace [System.Reflection](#).

Atributo	Finalidade
----------	------------

Atributo	Finalidade
<a href="#">AssemblyTitleAttribute</a>	Especifica um título de assembly para um manifesto do assembly.
<a href="#">AssemblyDescriptionAttribute</a>	Especifica uma descrição do assembly para um manifesto do assembly.
<a href="#">AssemblyConfigurationAttribute</a>	Especifica uma configuração de assembly (como varejo ou depuração) para um manifesto do assembly.
<a href="#">AssemblyDefaultAliasAttribute</a>	Define um alias amigável padrão para um manifesto do assembly

# Determinar informações do chamador usando atributos interpretados pelo compilador C#

Artigo • 07/04/2023

Usando atributos de informações, você obtém informações sobre o chamador de um método. Você obtém o caminho do arquivo do código-fonte, o número de linha no código-fonte e o nome do membro do chamador. Para obter informações do chamador do membro, você usa os atributos que são aplicados aos parâmetros opcionais. Cada parâmetro opcional especifica um valor padrão. A tabela a seguir lista os atributos de informações do chamador que são definidos no namespace de [System.Runtime.CompilerServices](#):

Atributo	Descrição	Type
<a href="#">CallerFilePathAttribute</a>	O caminho completo do arquivo de origem que contém o chamador. O caminho completo é o caminho no tempo de compilação.	<code>String</code>
<a href="#">CallerLineNumberAttribute</a>	Número de linha no arquivo de origem do qual o método é chamado.	<code>Integer</code>
<a href="#">CallerMemberNameAttribute</a>	Nome do método ou nome da propriedade do chamador.	<code>String</code>
<a href="#">CallerArgumentExpressionAttribute</a>	Representação de cadeia de caracteres da expressão de argumento.	<code>String</code>

Essas informações ajudam você a rastrear e depurar e ajuda você a criar ferramentas de diagnóstico. O exemplo a seguir mostra como usar os atributos de informações do chamador. Em cada chamada para o método `TraceMessage`, as informações do chamador são inseridas para os argumentos nos parâmetros opcionais.

C#

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
```

```

{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
// message: Something happened.
// member name: DoProcessing
// source file path: c:\Visual Studio
Projects\CallerInfoCS\CallerInfoCS\Form1.cs
// source line number: 31

```

Especifique um valor padrão explícito para cada parâmetro opcional. Você não pode aplicar atributos de informações do chamador aos parâmetros que não são especificados como opcionais. Os atributos de informações do chamador não tornam um parâmetro opcional. Em vez disso, eles afetam o valor padrão que é passado quando o argumento é omitido. Os valores de informações do chamador são emitidos como literais em linguagem intermediária (IL) em tempo de compilação. Ao contrário dos resultados da propriedade [StackTrace](#) para exceções, os resultados não são afetados por ofuscamento. Você pode fornecer explicitamente os argumentos opcionais para controlar as informações do chamador ou ocultá-las.

## Nomes dos membros

Você pode usar o atributo `CallerMemberName` para evitar especificar o nome do membro como um argumento `String` ao método chamado. Ao usar essa técnica, você evita o problema de que a [Refatoração de Renomeação](#) não altera os valores de `String`. Esse benefício é especialmente útil para as seguintes tarefas:

- Usar rotinas de rastreamento e diagnóstico.
- Implementando a interface [INotifyPropertyChanged](#) ao associar dados. Essa interface permite que a propriedade de um objeto Notifique um controle ligado de que a propriedade foi alterada. O controle pode exibir as informações atualizadas. Sem o atributo `CallerMemberName`, você deve especificar o nome da propriedade como um literal.

O gráfico a seguir mostra os nomes de membros que são retornados quando você usa o atributo `CallerMemberName`.

<b>As chamadas ocorrem dentro</b>	<b>Resultado de nome de membro</b>
-----------------------------------	------------------------------------

<b>As chamadas ocorrem dentro</b>	<b>Resultado de nome de membro</b>
Método, propriedade ou evento	O nome do método, da propriedade ou do evento em que a chamada foi originada.
Construtor	A cadeia de caracteres ".ctor"
Construtor estático	A cadeia de caracteres ".cctor"
Finalizer	A cadeia de caracteres "Finalize"
Operadores usuário ou conversões definidos pelo usuário	O nome gerado para o membro, por exemplo, "op>Addition".
Construtor de atributos	O nome do método ou propriedade ao qual o atributo se aplica. Se o atributo é qualquer elemento dentro de um membro (como um parâmetro, um valor de retorno, ou um parâmetro de tipo genérico), esse resultado é o nome do membro associado a esse elemento.
Nenhum membro contentor (por exemplo, nível de assembly ou atributos que são aplicadas aos tipos)	O valor padrão do parâmetro opcional.

## Expressões de argumento

Use quando [System.Runtime.CompilerServices.CallerArgumentExpressionAttribute](#) quiser que a expressão seja passada como um argumento. As bibliotecas de diagnóstico podem querer fornecer mais detalhes sobre as *expressões* passadas para argumentos. Ao fornecer a expressão que disparou o diagnóstico, além do nome do parâmetro, os desenvolvedores têm mais detalhes sobre a condição que disparou o diagnóstico. Essas informações extras facilitam a correção.

O exemplo a seguir mostra como você pode fornecer informações detalhadas sobre o argumento quando ele é inválido:

C#

```
public static void ValidateArgument(string parameterName, bool condition,
[CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new ArgumentException($"Argument failed validation:
<{message}>", parameterName);
```

```
    }  
}
```

Você o invocaria conforme mostrado no exemplo a seguir:

C#

```
public void Operation(Action func)  
{  
    Utilities.ValidateArgument(nameof(func), func is not null);  
    func();  
}
```

A expressão usada para `condition` é injetada pelo compilador no argumento `message`. Quando um desenvolvedor chama `Operation` com um argumento `null`, a seguinte mensagem é armazenada no `ArgumentException`:

CLI do .NET

```
Argument failed validation: <func is not null>
```

Esse atributo permite que você escreva utilitários de diagnóstico que fornecem mais detalhes. Os desenvolvedores podem entender mais rapidamente quais alterações são necessárias. Você também pode usar o [CallerArgumentExpressionAttribute](#) para determinar qual expressão foi usada como receptor para métodos de extensão. O método a seguir amostra uma sequência em intervalos regulares. Se a sequência tiver menos elementos do que a frequência, ela relatará um erro:

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sequence, int  
frequency,  
    [CallerArgumentExpression(nameof(sequence))] string? message = null)  
{  
    if (sequence.Count() < frequency)  
        throw new ArgumentException($"Expression doesn't have enough  
elements: {message}", nameof(sequence));  
    int i = 0;  
    foreach (T item in sequence)  
    {  
        if (i++ % frequency == 0)  
            yield return item;  
    }  
}
```

O exemplo anterior usa o operador `nameof` para o parâmetro `sequence`. Este recurso está disponível no C# 11. Antes do C# 11, você precisará digitar o nome do parâmetro como uma cadeia de caracteres. Você pode chamar esse método da seguinte maneira:

```
C#
```

```
sample = Enumerable.Range(0, 10).Sample(100);
```

O exemplo anterior lançaria um [ArgumentException](#), cuja mensagem é o seguinte texto:

```
CLI do .NET
```

```
Expression doesn't have enough elements: Enumerable.Range(0, 10) (Parameter  
'sequence')
```

## Confira também

- [Argumentos nomeados e opcionais](#)
- [System.Reflection](#)
- [Attribute](#)
- [Atributos](#)

# Atributos para análise estática de estado nulo interpretados pelo compilador C#

Artigo • 05/06/2023

Em um contexto habilitado para anulável, o compilador executa a análise estática do código para determinar o *estado nulo* de todas as variáveis de tipo de referência:

- *not-null*: a análise estática determina que uma variável tem um valor não nulo.
- *maybe-null*: a análise estática não pode determinar que uma variável recebe um valor não nulo.

Esses estados permitem que o compilador forneça avisos quando você pode desreferenciar um valor nulo, gerando `System.NullReferenceException`. Esses atributos fornecem ao compilador informações semânticas sobre o *null-state* de argumentos, valores retornados e membros do objeto com base no estado dos argumentos e valores retornados. O compilador fornece avisos mais precisos quando suas APIs foram anotadas corretamente com essas informações semânticas.

Este artigo fornece uma breve descrição de cada um dos atributos de tipo de referência anuláveis e como usá-los.

Vamos começar com um exemplo. Imagine que sua biblioteca tem a API a seguir para recuperar uma cadeia de caracteres de recurso. Este método foi originalmente compilado em um contexto *alheio anulável*:

```
C#  
  
bool TryGetMessage(string key, out string message)  
{  
    if (_messageMap.ContainsKey(key))  
        message = _messageMap[key];  
    else  
        message = null;  
    return message != null;  
}
```

O exemplo anterior segue o padrão familiar `Try*` no .NET. Há dois parâmetros de referência para essa API: o `key` e o `message`. Essa API tem as seguintes regras relacionadas ao *null-state* desses parâmetros:

- Os chamadores não devem passar `null` como o argumento para `key`.

- Os chamadores podem passar uma variável cujo valor é `null` como o argumento para `message`.
- Se o método `TryGetMessage` retornar `true`, o valor `message` não será nulo. Se o valor de retorno for `false`, o valor de `message` será nulo.

A regra para `key` pode ser expressa de forma sucinta: `key` deve ser um tipo de referência não anulável. O parâmetro `message` é mais complexo. Ele permite uma variável que seja `null` como argumento, mas garante, em caso de sucesso, que o argumento `out` não seja `null`. Para esses cenários, você precisa de um vocabulário mais rico para descrever as expectativas. O atributo `NotNullWhen`, descrito abaixo, descreve o *null-state* para o argumento usado para o parâmetro `message`.

### ⓘ Observação

Adicionar esses atributos fornece ao compilador mais informações sobre as regras da API. Quando o código de chamada for compilado em um contexto habilitado para anulável, o compilador avisará os chamados quando eles violarem essas regras. Esses atributos não habilitam mais verificações em sua implementação.

Atributo	Categoria	Significado
<code>AllowNull</code>	Pré-condição	Um parâmetro, um campo ou uma propriedade não anulável pode ser nulo.
<code>DisallowNull</code>	Pré-condição	Um parâmetro, campo ou propriedade anulável nunca deve ser nulo.
<code>BeNull</code>	Pós-condição	Um parâmetro, campo, propriedade ou valor de retorno não anulável pode ser nulo.
<code>NotNull</code>	Pós-condição	Um parâmetro anulável, um campo, uma propriedade ou um valor retornado nunca será nulo.
<code>BeNullWhen</code>	Pós-condição condicional	Um argumento não anulável pode ser nulo quando o método retorna o valor <code>bool</code> especificado.
<code>NotNullWhen</code>	Pós-condição condicional	Um argumento anulável não pode ser nulo quando o método retorna o valor <code>bool</code> especificado.
<code>NotNullIfNotNull</code>	Pós-condição condicional	Um valor retornado, uma propriedade ou um argumento não será nulo se o argumento para o parâmetro especificado não for nulo.

Atributo	Categoria	Significado
MemberNotNull	Métodos auxiliares de método e Propriedade	O membro listado não será nulo quando o método retornar.
MemberNotNullWhen	Métodos auxiliares de método e Propriedade	O membro listado não será nulo quando o método retornar o valor <code>bool</code> especificado.
DoesNotReturn	Código inacessível	Um método ou propriedade nunca retorna. Em outras palavras, ele sempre gera uma exceção.
DoesNotReturnIf	Código inacessível	Esse método ou propriedade nunca retornará se o parâmetro associado <code>bool</code> tiver o valor especificado.

As descrições anteriores são uma referência rápida ao que cada atributo faz. As seções a seguir descrevem mais detalhadamente o comportamento e o significado desses atributos.

## Pré-condições: `AllowNull` e `DisallowNull`

Considere uma propriedade de leitura/gravação que nunca retorna `null` porque tem um valor padrão razoável. Os chamadores passam `null` para o acessador definido ao defini-lo para esse valor padrão. Por exemplo, considere um sistema de mensagens que solicita um nome de tela em uma sala de chat. Se nenhum for fornecido, o sistema gerará um nome aleatório:

C#

```
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

Quando você compila o código anterior em um contexto alheio anulável, tudo está bem. Depois de habilitar tipos de referência anuláveis, a propriedade `ScreenName` se tornará uma referência não anulável. Isso está correto para o acessador `get`: ele nunca retorna `null`. Os chamadores não precisam verificar a propriedade retornada para `null`. Mas agora definir a propriedade como `null` gera um aviso. Para dar suporte a esse tipo

de código, adicione o atributo `System.Diagnostics.CodeAnalysis.AllowNullAttribute` à propriedade, conforme mostrado no seguinte código:

```
C#  
  
[AllowNull]  
public string ScreenName  
{  
    get => _screenName;  
    set => _screenName = value ?? GenerateRandomScreenName();  
}  
private string _screenName = GenerateRandomScreenName();
```

Talvez seja necessário adicionar uma diretiva `using` para `System.Diagnostics.CodeAnalysis`, a fim de usar este e outros atributos discutidos neste artigo. O atributo é aplicado à propriedade, não ao acessador `set`. O atributo `AllowNull` especifica *pré-condições* e só se aplica a argumentos. O acessador `get` tem um valor retornado, mas nenhum parâmetro. Portanto, o atributo `AllowNull` só se aplica ao acessador `set`.

O exemplo anterior demonstra o que procurar ao adicionar o atributo `AllowNull` em um argumento:

1. O contrato geral dessa variável é que ela não deve ser `null`, portanto, você deseja um tipo de referência não anulável.
2. Há cenários para um chamador passar `null` como o argumento, embora não sejam o uso mais comum.

Na maioria das vezes, você precisará desse atributo para propriedades ou argumentos `in`, `out` e `ref`. O atributo `AllowNull` é a melhor opção quando uma variável normalmente não é nula, mas você precisa permitir `null` como pré-condição.

Contraste isso com cenários para uso `DisallowNull`: você usa esse atributo para especificar que um argumento de um tipo de referência anulável não deve ser `null`. Considere uma propriedade em que `null` é o valor padrão, mas os clientes só podem defini-lo como um valor não nulo. Considere o seguinte código:

```
C#  
  
public string ReviewComment  
{  
    get => _comment;  
    set => _comment = value ?? throw new  
ArgumentNullException(nameof(value), "Cannot set to null");  
}
```

```
}
```

```
string _comment;
```

O código anterior é a melhor maneira de expressar seu design que `ReviewComment` pode ser `null`, mas não pode ser definido como `null`. Depois que esse código for anulável, você poderá expressar esse conceito com mais clareza aos chamadores usando o [System.Diagnostics.CodeAnalysis.DisallowNullAttribute](#):

C#

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new
ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

Em um contexto anulável, o acessador `ReviewComment get` poderia retornar o valor padrão de `null`. O compilador avisa que ele deve ser verificado antes do acesso. Além disso, ele avisa os chamadores que, embora possa ser `null`, os chamadores não devem defini-lo explicitamente como `null`. O atributo `DisallowNull` também especifica uma *pré-condição*, não afeta o acessador `get`. Você usa o atributo `DisallowNull` quando observa essas características sobre:

1. A variável pode ser `null` em cenários principais, muitas vezes quando instanciada pela primeira vez.
2. A variável não deve ser definida explicitamente como `null`.

Essas situações são comuns em código originalmente *nulo alheio*. Pode ser que as propriedades do objeto sejam definidas em duas operações de inicialização distintas. Pode ser que algumas propriedades sejam definidas somente após a conclusão de algum trabalho assíncrono.

Os atributos `AllowNull` e `DisallowNull` permitem que você especifique que as pré-condições em variáveis podem não corresponder às anotações anuláveis nessas variáveis. Eles fornecem mais detalhes sobre as características de sua API. Essas informações adicionais ajudam os chamadores a usar sua API corretamente. Lembre-se de especificar pré-condições usando os seguintes atributos:

- **AllowNull**: Um argumento não anulável pode ser nulo.
- **DisallowNull**: um argumento anulável nunca deve ser nulo.

# Pós-condições: `MaybeNull` e `NotNull`

Considere um método com a seguinte assinatura:

C#

```
public Customer FindCustomer(string lastName, string firstName)
```

Você provavelmente escreveu um método como este para retornar `null` quando o nome procurado não foi encontrado. `null` indica claramente que o registro não foi encontrado. Neste exemplo, você provavelmente alteraria o tipo de retorno de `Customer` para `Customer?`. Declarar o valor retornado como um tipo de referência anulável especifica claramente a intenção dessa API:

C#

```
public Customer? FindCustomer(string lastName, string firstName)
```

Por motivos abordados em [Anulabilidade de genéricos](#), essa técnica pode não produzir a análise estática que corresponde à sua API. Você pode ter um método genérico que segue um padrão semelhante:

C#

```
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

O método retorna `null` quando o item procurado não é encontrado. Você pode esclarecer que o método retorna `null` quando um item não é encontrado adicionando a anotação `MaybeNull` ao retorno do método:

C#

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

O código anterior informa aos chamadores que o valor retornado *pode* realmente ser nulo. Ele também informa ao compilador que o método pode retornar uma expressão `null` mesmo que o tipo não seja anulável. Quando você tem um método genérico que retorna uma instância de seu parâmetro de tipo, `T` você pode expressar que ele nunca retorna `null` usando o atributo `NotNull`.

Você também pode especificar que um valor retornado ou um argumento não seja nulo, mesmo que o tipo seja um tipo de referência anulável. O método a seguir é um método auxiliar que gera se seu primeiro argumento for `null`:

C#

```
public static void ThrowWhenNull(object value, string valueExpression = "")  
{  
    if (value is null) throw new ArgumentNullException(nameof(value),  
    valueExpression);  
}
```

Você pode chamar essa rotina da seguinte maneira:

C#

```
public static void LogMessage(string? message)  
{  
    ThrowWhenNull(message, $"{nameof(message)} must not be null");  
  
    Console.WriteLine(message.Length);  
}
```

Depois de habilitar tipos de referência nulos, você deseja garantir que o código anterior seja compilado sem avisos. Quando o método retorna, o parâmetro `value` tem a garantia de não ser nulo. No entanto, é aceitável chamar `ThrowWhenNull` com uma referência nula. Você pode fazer `value` um tipo de referência anulável e adicionar a pós-condição `NotNull` à declaração de parâmetro:

C#

```
public static void ThrowWhenNull([NotNull] object? value, string  
valueExpression = "")  
{  
    _ = value ?? throw new ArgumentNullException(nameof(value),  
    valueExpression);  
    // other logic elided
```

O código anterior expressa claramente o contrato existente: os chamadores podem passar uma variável com o valor `null`, mas é garantido que o argumento nunca será nulo se o método retornar sem gerar uma exceção.

Especifique pós-condições incondicional usando os seguintes atributos:

- **MaybeNull**: Um valor de retorno não anulável pode ser nulo.
- **NotNull**: um valor de retorno anulável nunca será nulo.

## Pós-condições condicionais: `NotNullWhen`,

### `MaybeNullWhen` e `NotNullIfNotNull`

Você provavelmente está familiarizado com o `string` método

`String.IsNullOrEmpty(String)`. Esse método retorna `true` quando o argumento é nulo ou uma cadeia de caracteres vazia. É uma forma de verificação nula: os chamadores não precisam verificar nulo o argumento se o método retornar `false`. Para tornar um método como esse anulável consciente, você definiria o argumento como um tipo de referência anulável e adicionaria o atributo `NotNullWhen`:

C#

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value)
```

Isso informa ao compilador que qualquer código em que o valor retornado é `false` não precisa de verificações nulas. A adição do atributo informa a análise estática do compilador que `IsNullOrEmpty` executa a verificação nula necessária: quando ele retorna `false`, o argumento não é `null`.

C#

```
string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
    int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

#### ① Observação

O exemplo anterior só é válido no C# 11 e posterior. Começando no C# 11, a expressão `[nameof](../operators/nameof.md)` pode referenciar nomes de parâmetro de tipo e de parâmetro quando usada em um atributo aplicado a um método. No C# 10 e anteriores, você precisa usar um literal de cadeia de caracteres em vez da expressão `nameof`.

O método `String.IsNullOrEmpty(String)` será anotado conforme mostrado acima para o .NET Core 3.0. Você pode ter métodos semelhantes em sua base de código que verificam o estado dos objetos em busca de valores nulos. O compilador não reconhecerá métodos personalizados de verificação nula e você precisará adicionar as

anotações por conta própria. Quando você adiciona o atributo, a análise estática do compilador sabe quando a variável testada foi verificada nulo.

Outro uso para esses atributos é o padrão `Try*`. As pós-condições para argumentos `ref` e `out` são comunicados por meio do valor retornado. Considere esse método mostrado anteriormente (em um contexto nulo desabilitado):

C#

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

O método anterior segue um idioma .NET típico: o valor retornado indica se `message` foi definido como o valor encontrado ou, se nenhuma mensagem for encontrada, para o valor padrão. Se o método retornar `true`, o valor não `message` será nulo; caso contrário, o método definirá `message` como nulo.

Em um contexto habilitado para anulável, você pode comunicar esse idioma usando o atributo `NotNullWhen`. Quando você anotar parâmetros para tipos de referência anuláveis, faça de `message` um `string?` e adicione um atributo:

C#

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message is not null;
}
```

No exemplo anterior, o valor de `message` é conhecido por não ser nulo quando `TryGetMessage` retorna `true`. Você deve anotar métodos semelhantes em sua base de código da mesma maneira: os argumentos podem ser iguais `null` e são conhecidos por não serem nulos quando o método retorna `true`.

Há um atributo final que você também pode precisar. Às vezes, o estado nulo de um valor retornado depende do estado nulo de um ou mais argumentos. Esses métodos

retornarão um valor não nulo sempre que determinados argumentos não forem `null`. Para anotar corretamente esses métodos, use o `NotNullIfNotNull` atributo. Considere o método a seguir:

```
C#
```

```
string GetTopLevelDomainFromFullUrl(string url)
```

Se o argumento `url` não for nulo, a saída não será `null`. Depois que as referências anuláveis forem habilitadas, você precisará adicionar mais anotações se sua API puder aceitar um argumento nulo. Você pode anotar o tipo de retorno, conforme mostrado no código a seguir:

```
C#
```

```
string? GetTopLevelDomainFromFullUrl(string? url)
```

Isso também funciona, mas muitas vezes forçará os chamadores a implementar verificações `null` extras. O contrato é que o valor retornado seria `null` somente quando o argumento `url` for `null`. Para expressar esse contrato, você anotaria esse método, conforme mostrado no código a seguir:

```
C#
```

```
[return: NotNullIfNotNull(nameof(url))]  
string? GetTopLevelDomainFromFullUrl(string? url)
```

O exemplo anterior usa o operador `nameof` para o parâmetro `url`. Este recurso está disponível no C# 11. Antes do C# 11, você precisará digitar o nome do parâmetro como uma cadeia de caracteres. O valor retornado e o argumento foram anotados com a indicação `?` de que qualquer um poderia ser `null`. O atributo esclarece ainda mais que o valor retornado não será nulo quando o argumento `url` não for `null`.

Especifique pós-condições condicionais usando estes atributos:

- **MaybeNullWhen**: Um argumento não anulável pode ser nulo quando o método retorna o valor `bool` especificado.
- **NotNullWhen**: Um argumento anulável não será nulo quando o método retornar o valor `bool` especificado.
- **NotNullIfNotNull**: um valor de retorno não é nulo se o argumento para o parâmetro especificado não for nulo.

# Métodos auxiliares: `MemberNotNull` e

## `MemberNotNullWhen`

Esses atributos especificam sua intenção quando você refatora o código comum de construtores em métodos auxiliares. O compilador C# analisa construtores e inicializadores de campo para garantir que todos os campos de referência não anuláveis tenham sido inicializados antes de cada construtor retornar. No entanto, o compilador C# não acompanha as atribuições de campo em todos os métodos auxiliares. O compilador emite um aviso `CS8618` quando os campos não são inicializados diretamente no construtor, mas sim em um método auxiliar. Adicione `MemberNotNullAttribute` a uma declaração de método e especifique os campos inicializados para um valor não nulo no método. Por exemplo, considere o exemplo a seguir:

C#

```
public class Container
{
    private string _uniqueIdentifier; // must be initialized.
    private string? _optionalMessage;

    public Container()
    {
        Helper();
    }

    public Container(string message)
    {
        Helper();
        _optionalMessage = message;
    }

    [MemberNotNull(nameof(_uniqueIdentifier))]
    private void Helper()
    {
        _uniqueIdentifier = DateTime.Now.Ticks.ToString();
    }
}
```

Você pode especificar vários nomes de campo como argumentos para o construtor de atributo `MemberNotNull`.

`MemberNotNullWhenAttribute` tem um argumento `bool`. Você usa `MemberNotNullWhen` em situações em que seu método auxiliar retorna um `bool`, que indica se o método auxiliar inicializou campos.

# Parar a análise anulável quando o método chamado é lançado

Alguns métodos, normalmente auxiliares de exceção ou outros métodos utilitários, sempre saem lançando uma exceção. Ou, um auxiliar pode lançar uma exceção com base no valor de um argumento booleano.

No primeiro caso, você pode adicionar o atributo `DoesNotReturnAttribute` à declaração do método. A análise de *null-state* do compilador não verifica nenhum código em um método que segue uma chamada para um método anotado com `DoesNotReturn`.

Considere este método:

```
C#  
  
[DoesNotReturn]  
private void FailFast()  
{  
    throw new InvalidOperationException();  
}  
  
public void SetState(object containedField)  
{  
    if (containedField is null)  
    {  
        FailFast();  
    }  
  
    // containedField can't be null:  
    _field = containedField;  
}
```

O compilador não emite avisos após a chamada para `FailFast`.

No segundo caso, você adiciona o `System.Diagnostics.CodeAnalysis.DoesNotReturnIfAttribute` atributo a um parâmetro booleano do método. Você pode modificar o exemplo anterior da seguinte maneira:

```
C#  
  
private void FailFastIf([DoesNotReturnIf(true)] bool isNull)  
{  
    if (isNull)  
    {  
        throw new InvalidOperationException();  
    }  
}  
  
public void SetFieldState(object? containedField)
```

```
{  
    FailFastIf(containedField == null);  
    // No warning: containedField can't be null here:  
    _field = containedField;  
}
```

Quando o valor do argumento corresponde ao valor do construtor `DoesNotReturnIf`, o compilador não executa nenhuma análise *null-state* após esse método.

## Resumo

Adicionar tipos de referência anuláveis fornece um vocabulário inicial para descrever suas expectativas de APIs para variáveis que podem ser `null`. Os atributos fornecem um vocabulário mais avançado para descrever o estado nulo de variáveis como pré-condições e pós-condições. Esses atributos descrevem mais claramente suas expectativas e fornecem uma experiência melhor para os desenvolvedores que usam suas APIs.

À medida que você atualiza bibliotecas para um contexto anulável, adicione esses atributos para orientar os usuários de suas APIs ao uso correto. Esses atributos ajudam você a descrever totalmente o *null-state* dos argumentos e os valores retornados.

- `AllowNull`: um campo, parâmetro ou propriedade não anulável pode ser nulo.
- `DisallowNull`: um campo, parâmetro ou propriedade anulável nunca deve ser nulo.
- `MaybeNull`: um campo, parâmetro, propriedade ou valor retornado não anulável pode ser nulo.
- `NotNull`: um campo, parâmetro, propriedade ou valor retornado anulável nunca será nulo.
- `MaybeNullWhen`: Um argumento não anulável pode ser nulo quando o método retorna o valor `bool` especificado.
- `NotNullWhen`: Um argumento anulável não será nulo quando o método retornar o valor `bool` especificado.
- `NotNullIfNotNull`: um parâmetro, propriedade ou valor de retorno não é nulo se o argumento para o parâmetro especificado não for nulo.
- `DoesNotReturn`: um método ou propriedade nunca retorna. Em outras palavras, ele sempre gera uma exceção.
- `DoesNotReturnIf`: esse método ou propriedade nunca retornará se o parâmetro associado `bool` tiver o valor especificado.

# Atributos diversos interpretados pelo compilador C#

Artigo • 16/03/2023

Os atributos `Conditional`, `Obsolete`, `AttributeUsage`, `AsyncMethodBuilder`, `InterpolatedStringHandler` e `ModuleInitializer` podem ser aplicados a elementos do seu código. Eles adicionam significado semântico a esses elementos. O compilador usa esses significados semânticos para alterar a saída e relatar possíveis erros cometidos pelos desenvolvedores que usam seu código.

## Atributo `Conditional`

O atributo `Conditional` torna a execução de um método dependente de um identificador de pré-processamento. O atributo `Conditional` é um alias para `ConditionalAttribute` e pode ser aplicado a um método ou uma classe de atributo.

No exemplo seguinte, `Conditional` é aplicado a um método para habilitar ou desabilitar a exibição de informações de diagnóstico específicas do programa:

C#

```
#define TRACE_ON
using System.Diagnostics;

namespace AttributeExamples;

public class Trace
{
    [Conditional("TRACE_ON")]
    public static void Msg(string msg)
    {
        Console.WriteLine(msg);
    }
}

public class TraceExample
{
    public static void Main()
    {
        Trace.Msg("Now in Main...");
        Console.WriteLine("Done.");
    }
}
```

Se o identificador `TRACE_ON` não estiver definido, a saída de rastreamento não será exibida. Explore por conta própria na janela interativa.

O atributo `Conditional` é frequentemente usado com o identificador `DEBUG` para habilitar os recursos de rastreamento e log em builds de depuração, mas não em builds de versão, conforme mostrado no seguinte exemplo:

C#

```
[Conditional("DEBUG")]
static void DebugMethod()
{}
```

Quando um método marcado como condicional é chamado, a presença ou a ausência do símbolo de pré-processamento especificado determina se o compilador inclui ou omite chamadas para o método. Se o símbolo estiver definido, a chamada será incluída, caso contrário, a chamada será omitida. Um método condicional deve ser um método em uma declaração de classe ou struct e deve ter um tipo de retorno `void`. Usar `Conditional` é mais limpo, mais elegante e menos propenso a erros do que os métodos delimitadores dentro de blocos `#if...#endif`.

Se um método tiver vários atributos `Conditional`, o compilador incluirá chamadas ao método se um ou mais símbolos condicionais forem definidos (os símbolos serão vinculados logicamente entre si usando o operador OR). No seguinte exemplo, a presença de um `A` ou `B` resulta em uma chamada de método:

C#

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

## Usar `Conditional` com classes de atributo

O atributo `Conditional` também pode ser aplicado a uma definição de classe de atributos. No exemplo a seguir, o atributo personalizado `Documentation` só adicionará informações aos metadados se `DEBUG` for definido.

C#

```

[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
{
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}

```

## Atributo Obsolete

O atributo `Obsolete` marca um elemento de código como não mais recomendado para uso. O uso de uma entidade marcada como obsoleta gera um aviso ou um erro. O atributo `Obsolete` é um atributo de uso único e pode ser aplicado a qualquer entidade que permite atributos. `Obsolete` é um alias para `ObsoleteAttribute`.

No exemplo a seguir, o atributo `Obsolete` é aplicado à classe `A` e ao método `B.oldMethod`. Como o segundo argumento do construtor de atributo aplicado a `B.oldMethod` está definido como `true`, esse método causará um erro do compilador, enquanto que ao usar a classe `A`, produzirá apenas um aviso. Chamar `B.newMethod`, no entanto, não produz aviso nem erro. Por exemplo, ao usá-lo com as definições anteriores, o código a seguir gera um erro e dois avisos:

C#

```

namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }

    }

    public class B

```

```

{
    [Obsolete("use NewMethod", true)]
    public void OldMethod() { }

    public void NewMethod() { }
}

public static class ObsoleteProgram
{
    public static void Main()
    {
        // Generates 2 warnings:
        A a = new A();

        // Generate no errors or warnings:
        B b = new B();
        b.NewMethod();

        // Generates an error, compilation fails.
        // b.OldMethod();
    }
}

```

A cadeia de caracteres fornecida como primeiro argumento para o construtor do atributo será exibida como parte do aviso ou erro. São gerados dois avisos para a classe A: um para a declaração da referência de classe e outro para o construtor de classe. O atributo `Obsolete` pode ser usado sem argumentos, mas é recomendável incluir uma explicação do que deve ser usado no lugar.

No C# 10, você pode usar a interpolação de cadeia de caracteres constante e o operador `nameof` para garantir que os nomes correspondam:

```

C#

public class B
{
    [Obsolete($"use {nameof(NewMethod)} instead", true)]
    public void OldMethod() { }

    public void NewMethod() { }
}

```

## Atributo `SetsRequiredMembers`

O atributo `SetsRequiredMembers` informa ao compilador que um construtor define todos os membros `required` nessa classe ou estrutura. O compilador assume que qualquer

construtor com o atributo

`System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute` inicializa todos os membros `required`. Qualquer código que invoque tal construtor não precisa de inicializadores de objeto para definir os membros necessários. Isso é útil principalmente para registros posicionais e construtores primários.

## Atributo `AttributeUsage`

O atributo `AttributeUsage` determina como uma classe de atributos personalizados pode ser usada. `AttributeUsageAttribute` é um atributo aplicado a definições de atributo personalizado. O atributo `AttributeUsage` permite que você controle:

- A quais elementos do programa o atributo pode ser aplicado. A menos que você restrinja seu uso, um atributo poderá ser aplicado a qualquer um dos seguintes elementos do programa:
  - Assembly
  - Módulo
  - Campo
  - Evento
  - Método
  - Param
  - Propriedade
  - Retorno
  - Tipo
- Indica se um atributo pode ser aplicado a um único elemento do programa várias vezes.
- Indica se os atributos são herdados por classes derivadas.

As configurações padrão se parecem com o seguinte exemplo quando aplicadas explicitamente:

```
C#  
  
[AttributeUsage(AttributeTargets.All,  
                AllowMultiple = false,  
                Inherited = true)]  
class NewAttribute : Attribute { }
```

Neste exemplo, a classe `NewAttribute` pode ser aplicada a qualquer elemento de programa compatível. Porém, ele pode ser aplicado apenas uma vez para cada entidade. O atributo é herdado por classes derivadas quando aplicado a uma classe base.

Os argumentos `AllowMultiple` e `Inherited` são opcionais e, portanto, o seguinte código tem o mesmo efeito:

```
C#
```

```
[AttributeUsage(AttributeTargets.All)]  
class NewAttribute : Attribute { }
```

O primeiro argumento `AttributeUsageAttribute` deve ser um ou mais elementos da enumeração `AttributeTargets`. Vários tipos de destino podem ser vinculados junto com o operador OR, como mostra o seguinte exemplo:

```
C#
```

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]  
class NewPropertyOrFieldAttribute : Attribute { }
```

Os atributos podem ser aplicados à propriedade ou ao campo de suporte de uma propriedade autoimplementada. O atributo se aplica à propriedade, a menos que você especifique o especificador `field` no atributo. Ambos são mostrados no seguinte exemplo:

```
C#
```

```
class MyClass  
{  
    // Attribute attached to property:  
    [NewPropertyOrField]  
    public string Name { get; set; } = string.Empty;  
  
    // Attribute attached to backing field:  
    [field: NewPropertyOrField]  
    public string Description { get; set; } = string.Empty;  
}
```

Se o argumento `AllowMultiple` for `true`, o atributo resultante poderá ser aplicado mais de uma vez a uma única entidade, conforme mostrado no seguinte exemplo:

```
C#
```

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]  
class MultiUse : Attribute { }  
  
[MultiUse]  
[MultiUse]  
class Class1 { }
```

```
[MultiUse, MultiUse]  
class Class2 { }
```

Nesse caso, `MultiUseAttribute` pode ser aplicado repetidas vezes porque `AllowMultiple` está definido como `true`. Os dois formatos mostrados para a aplicação de vários atributos são válidos.

Se `Inherited` for `false`, o atributo não será herdado por classes derivadas de uma classe atribuída. Por exemplo:

C#

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]  
class NonInheritedAttribute : Attribute { }  
  
[NonInherited]  
class BClass { }  
  
class DClass : BClass { }
```

Nesse caso, `NonInheritedAttribute` não é aplicado a `DClass` por meio de herança.

Você também pode usar essas palavras-chave para especificar onde um atributo deve ser aplicado. Por exemplo, você pode usar o especificador `field:` para adicionar um atributo ao campo de suporte de uma [propriedade autoimplementada](#). Ou você pode usar o especificador `field:`, `property:` ou `param:` aplicar um atributo a qualquer um dos elementos gerados por meio de um registro posicional. Para obter um exemplo, confira a [Sintaxe posicional para definição de propriedade](#).

## Atributo `AsyncMethodBuilder`

Você adiciona o atributo

`System.Runtime.CompilerServices.AsyncMethodBuilderAttribute` a um tipo que pode ser um tipo de retorno assíncrono. O atributo especifica o tipo que cria a implementação do método assíncrono quando o tipo especificado é retornado de um método assíncrono. O atributo `AsyncMethodBuilder` pode ser aplicado a um tipo que:

- Tem um método `GetAwaiter` acessível.
- O objeto retornado pelo método `GetAwaiter` implementa a interface `System.Runtime.CompilerServices.ICriticalNotifyCompletion`.

O construtor do atributo `AsyncMethodBuilder` especifica o tipo do construtor associado. O construtor deve implementar os seguintes membros acessíveis:

- Um método estático `Create()` que retorna o tipo do construtor.
- Uma propriedade `Task` legível que retorna o tipo de retorno assíncrono.
- Um método `void SetException(Exception)` que define a exceção quando uma tarefa falha.
- Um método `void setResult()` ou `void setResult(T result)` que marca a tarefa como concluída e, opcionalmente, define o resultado da tarefa
- Um método `Start` com a seguinte assinatura de API:

C#

```
void Start<TStateMachine>(ref TStateMachine stateMachine)
    where TStateMachine :
        System.Runtime.CompilerServices.IAsyncStateMachine
```

- Um método `AwaitOnCompleted` com a seguinte assinatura:

C#

```
public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter
awaiter, ref TStateMachine stateMachine)
    where TAwaiter : System.Runtime.CompilerServices.INotifyCompletion
    where TStateMachine :
        System.Runtime.CompilerServices.IAsyncStateMachine
```

- Um método `AwaitUnsafeOnCompleted` com a seguinte assinatura:

C#

```
public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref
TAwaiter awaiter, ref TStateMachine stateMachine)
    where TAwaiter :
        System.Runtime.CompilerServices.ICriticalNotifyCompletion
    where TStateMachine :
        System.Runtime.CompilerServices.IAsyncStateMachine
```

Você pode aprender sobre construtores de métodos assíncronos lendo sobre os seguintes construtores fornecidos pelo .NET:

- [System.Runtime.CompilerServices.AsyncTaskMethodBuilder](#)
- [System.Runtime.CompilerServices.AsyncTaskMethodBuilder<TResult>](#)
- [System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder](#)
- [System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder<TResult>](#)

No C# 10 e versões posteriores, o atributo `AsyncMethodBuilder` pode ser aplicado a um método assíncrono para substituir o construtor por aquele tipo.

## Atributos `InterpolatedStringHandler` e `InterpolatedStringHandlerArguments`

Do C# 10 em diante, você usa esses atributos para especificar que um tipo é um *manipulador de cadeia de caracteres interpolada*. A biblioteca do .NET 6 já inclui `System.Runtime.CompilerServices.DefaultInterpolatedStringHandler` para cenários em que você usa uma cadeia de caracteres interpolada como argumento para um parâmetro `string`. Você pode ter outras instâncias em que deseja controlar como as cadeias de caracteres interpoladas são processadas. Você aplica `System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute` ao tipo que implementa seu manipulador. Você aplica `System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute` aos parâmetros do construtor desse tipo.

Você pode saber mais sobre como criar um manipulador de cadeia de caracteres interpolada na especificação de recursos do C# 10 para [melhorias de cadeia de caracteres interpolada](#).

## Atributo `ModuleInitializer`

Começando com C# 9, o atributo `ModuleInitializer` marca um método que o runtime chama quando o assembly é carregado. `ModuleInitializer` é um alias para `ModuleInitializerAttribute`.

O atributo `ModuleInitializer` só pode ser aplicado a um método que:

- É estático.
- Não tem parâmetros.
- Retorna `void`.
- É acessível por meio do módulo que o contém, ou seja, `internal` ou `public`.
- Não é um método genérico.
- Não está contido em uma classe genérica.
- Não é uma função local.

O atributo `ModuleInitializer` pode ser aplicado a vários métodos. Nesse caso, a ordem em que o runtime os chama é determinística, mas não especificada.

O exemplo a seguir ilustra o uso de vários métodos inicializadores de módulo. Os métodos `Init1` e `Init2` são executados antes de `Main` e cada um deles adiciona uma cadeia de caracteres à propriedade `Text`. Portanto, quando `Main` é executado, a propriedade `Text` já tem cadeias de caracteres de ambos os métodos inicializadores.

C#

```
using System;

internal class ModuleInitializerExampleMain
{
    public static void Main()
    {
        Console.WriteLine(ModuleInitializerExampleModule.Text);
        //output: Hello from Init1! Hello from Init2!
    }
}
```

C#

```
using System.Runtime.CompilerServices;

internal class ModuleInitializerExampleModule
{
    public static string? Text { get; set; }

    [ModuleInitializer]
    public static void Init1()
    {
        Text += "Hello from Init1! ";
    }

    [ModuleInitializer]
    public static void Init2()
    {
        Text += "Hello from Init2! ";
    }
}
```

Os geradores de código-fonte às vezes precisam gerar código de inicialização. Inicializadores de módulo fornecem um local padrão para esse código. Na maioria dos outros casos, você deve escrever um [construtor estático](#) em vez de um inicializador de módulo.

## Atributo `SkipLocalsInit`

Do C# 9 em diante, o atributo `SkipLocalsInit` impede que o compilador defina o sinalizador `.locals init` ao emitir metadados. O atributo `SkipLocalsInit` é um atributo de uso único e pode ser aplicado a um método, uma propriedade, uma classe, um struct, uma interface ou um módulo, mas não a um assembly. `SkipLocalsInit` é um alias para `SkipLocalsInitAttribute`.

O sinalizador `.locals init` faz com que o CLR inicialize todas as variáveis locais declaradas em um método com os respectivos valores padrão delas. Como o compilador também garante que você nunca use uma variável antes de atribuir algum valor a ela, `.locals init` normalmente não é necessário. No entanto, o recurso extra de inicialização com zero pode ter impacto mensurável no desempenho em alguns cenários, como quando você usa `stackalloc` para alocar uma matriz na pilha. Nesses casos, você pode adicionar o atributo `SkipLocalsInit`. Se aplicado diretamente a um método, o atributo afeta esse método e todas as funções aninhadas, incluindo lambdas e funções locais. Se aplicado a um tipo ou módulo, ele afeta todos os métodos aninhados dentro. Esse atributo não afeta métodos abstratos, mas afeta o código gerado para a implementação.

Esse atributo requer a opção do compilador `AllowUnsafeBlocks`. Esse requisito sinaliza que, em alguns casos, o código pode exibir memória não atribuída (por exemplo, leitura da memória alocada em pilha não inicializada).

O exemplo a seguir ilustra o efeito do atributo `SkipLocalsInit` em um método que usa `stackalloc`. O método exibe o que estava na memória quando a matriz de inteiros foi alocada.

C#

```
[SkipLocalsInit]
static void ReadUninitializedMemory()
{
    Span<int> numbers = stackalloc int[120];
    for (int i = 0; i < 120; i++)
    {
        Console.WriteLine(numbers[i]);
    }
}
// output depends on initial contents of memory, for example:
//0
//0
//0
//168
//0
// -1271631451
//32767
//38
```

```
//0  
//0  
//0  
//38  
// Remaining rows omitted for brevity.
```

Para experimentar esse código por conta própria, defina a opção do compilador `AllowUnsafeBlocks` no arquivo `.csproj`:

XML

```
<PropertyGroup>  
  ...  
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>  
</PropertyGroup>
```

## Confira também

- [Attribute](#)
- [System.Reflection](#)
- [Atributos](#)
- [Reflexão](#)

# Código não seguro, tipos de ponteiro e ponteiros de função

Artigo • 10/05/2023

A maior parte do código C# que você escreve é "código seguro verificável". *Código seguro verificável* significa que as ferramentas do .NET podem confirmar que o código é seguro. Em geral, o código seguro não acessa diretamente a memória usando ponteiros. Ele também não aloca memória bruta. Em vez disso, ele cria objetos gerenciados.

O C# dá suporte a um contexto `unsafe` em que você pode escrever códigos *não verificáveis*. Em um contexto `unsafe`, o código pode usar ponteiros, alocar e liberar blocos de memória e chamar métodos usando ponteiros de função. O código não seguro em C# não é necessariamente perigoso; é apenas um código cuja segurança não pode ser verificada.

O código não seguro tem as propriedades a seguir:

- Blocos de código, tipos e métodos podem ser definidos como não seguros.
- Em alguns casos, o código não seguro pode aumentar o desempenho de um aplicativo removendo as verificações de limites de matriz.
- O código não seguro é necessário quando você chama funções nativas que exigem ponteiros.
- Usar o código não seguro apresenta riscos de segurança e estabilidade.
- O código que contém blocos não seguros deve ser compilado com a opção do compilador `AllowUnsafeBlocks`.

## Tipos de ponteiro

Em um contexto não seguro, os tipos podem ser de ponteiro, além de tipo de valor ou tipo de referência. Uma declaração de tipo de ponteiro usa uma das seguintes formas:

C#

```
type* identifier;  
void* identifier; //allowed but not recommended
```

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 conversão entre tipos de ponteiro e `object`. Além disso, as conversões boxing e unboxing não oferecem suporte a ponteiros. No entanto, você pode converter entre diferentes tipos de ponteiro e tipos de ponteiro e tipos inteiros.

Ao declarar vários ponteiros na mesma declaração, você escreve o asterisco (\*) juntamente com o tipo subjacente apenas. Ele não é usado como um prefixo para cada nome de ponteiro. Por exemplo:

```
C#
```

```
int* p1, p2, p3; // Ok  
int *p1, *p2, *p3; // Invalid in C#
```

Um ponteiro não pode apontar para uma referência ou um `struct` que contenha referências, pois uma referência de objeto pode ser coletada como lixo mesmo se um ponteiro estiver apontando para ela. O coletor de lixo não se dá conta de que um objeto está sendo apontado por qualquer tipo de ponteiro.

O valor da variável de ponteiro do tipo `MyType*` é o endereço de uma variável do tipo `MyType`. Estes são exemplos de declarações de tipos de ponteiro:

- `int* p: p` é um ponteiro para um inteiro.
- `int** p: p` é um ponteiro para um ponteiro para um inteiro.
- `int*[] p: p` é uma matriz unidimensional de ponteiros para inteiros.
- `char* p: p` é um ponteiro para um caractere.
- `void* p: p` é um ponteiro para um tipo desconhecido.

O operador de indireção de ponteiro (\*) pode ser usado para acessar o conteúdo no local apontado pela variável de ponteiro. Por exemplo, considere a seguinte declaração:

```
C#
```

```
int* myVariable;
```

A expressão `*myVariable` denota a variável `int` encontrada no endereço contido em `myVariable`.

Há vários exemplos de ponteiros nos artigos sobre a instrução `fixed`. O exemplo a seguir usa a palavra-chave `unsafe` e a instrução `fixed` e mostra como incrementar um ponteiro interior. Você pode colar esse código na função principal de um aplicativo de

console para executá-lo. Estes exemplos precisam ser compilados com o conjunto de opções do compilador [AllowUnsafeBlocks](#).

C#

```
// Normal pointer to an object.  
int[] a = new int[5] { 10, 20, 30, 40, 50 };  
// Must be in unsafe code to use interior pointers.  
unsafe  
{  
    // Must pin object on heap so that it doesn't move while using interior  
    // pointers.  
    fixed (int* p = &a[0])  
    {  
        // p is pinned as well as object, so create another pointer to show  
        // incrementing it.  
        int* p2 = p;  
        Console.WriteLine(*p2);  
        // Incrementing p2 bumps the pointer by four bytes due to its type  
        ...  
        p2 += 1;  
        Console.WriteLine(*p2);  
        p2 += 1;  
        Console.WriteLine(*p2);  
        Console.WriteLine("-----");  
        Console.WriteLine(*p);  
        // Dereferencing p and incrementing changes the value of a[0] ...  
        *p += 1;  
        Console.WriteLine(*p);  
        *p += 1;  
        Console.WriteLine(*p);  
    }  
}  
  
Console.WriteLine("-----");  
Console.WriteLine(a[0]);  
  
/*  
Output:  
10  
20  
30  
-----  
10  
11  
12  
-----  
12  
*/
```

Você não pode aplicar o operador de indireção para um ponteiro do tipo `void*`. No entanto, você pode usar uma conversão para converter um ponteiro nulo em qualquer

outro tipo de ponteiro e vice-versa.

Um ponteiro pode ser `null`. Aplicar o operador de indireção a um ponteiro nulo causa um comportamento definido por implementação.

Passar ponteiros entre métodos pode causar um comportamento indefinido. Considere usar um método que retorne um ponteiro para uma variável local por meio de um parâmetro `in`, `out` ou `ref`, ou como o resultado da função. Se o ponteiro foi definido em um bloco fixo, a variável à qual ele aponta não pode mais ser corrigida.

A tabela a seguir lista os operadores e as instruções que podem operar em ponteiros em um contexto inseguro:

Operador/Instrução	Use
<code>*</code>	Executa indireção de ponteiro.
<code>-&gt;</code>	Acessa um membro de um struct através de um ponteiro.
<code>[]</code>	Indexa um ponteiro.
<code>&amp;</code>	Obtém o endereço de uma variável.
<code>++</code> e <code>--</code>	Incrementa e decrementa ponteiros.
<code>+ e -</code>	Executa aritmética de ponteiros.
<code>==</code> , <code>!=</code> , <code>&lt;</code> , <code>&gt;</code> , <code>&lt;=</code> e <code>&gt;=</code>	Compara ponteiros.
<code>stackalloc</code>	Aloca memória na pilha.
<code>fixed</code> instrução	Corrige temporariamente uma variável para que seu endereço possa ser encontrado.

Para obter mais informações sobre operadores relacionados a ponteiros, confira [Operadores relacionados a ponteiros](#).

Qualquer tipo de ponteiro pode ser convertido implicitamente em um tipo `void*`.

Qualquer tipo de ponteiro pode ser atribuído ao valor `null`. Qualquer tipo de ponteiro pode ser convertido explicitamente em qualquer outro tipo de ponteiro usando uma expressão de conversão. Também é possível converter qualquer tipo integral em um tipo de ponteiro ou qualquer tipo de ponteiro em um tipo integral. Essas conversões exigem uma conversão explícita.

O exemplo a seguir converte um `int*` em um `byte*`. Observe que o ponteiro aponta para o menor byte endereçado da variável. Quando você incrementar sucessivamente o

resultado, até o tamanho de `int` (4 bytes), você poderá exibir os bytes restantes da variável.

C#

```
int number = 1024;

unsafe
{
    // Convert to byte:
    byte* p = (byte*)&number;

    System.Console.Write("The 4 bytes of the integer:");

    // Display the 4 bytes of the int variable:
    for (int i = 0 ; i < sizeof(int) ; ++i)
    {
        System.Console.Write(" {0:X2}", *p);
        // Increment the pointer:
        p++;
    }
    System.Console.WriteLine();
    System.Console.WriteLine("The value of the integer: {0}", number);

    /* Output:
       The 4 bytes of the integer: 00 04 00 00
       The value of the integer: 1024
    */
}
```

## Buffers de tamanho fixo

Você pode usar a palavra-chave `fixed` para criar um buffer com uma matriz de tamanho fixo em uma estrutura de dados. Os buffers de tamanho fixo são úteis ao escrever métodos que interoperam com fontes de dados de outras linguagens ou plataformas. O buffer de tamanho fixo pode usar qualquer atributo ou modificador que for permitido para membros de struct regulares. A única restrição é que o tipo da matriz deve ser `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float` ou `double`.

C#

```
private fixed char name[30];
```

No código de seguro, um struct C# que contém uma matriz não contém os elementos da matriz. Em vez disso, o struct contém uma referência aos elementos. Você pode

inserir uma matriz de tamanho fixo em um `struct` quando ele é usado em um bloco de código [não seguro](#).

O tamanho do `struct` seguinte não depende do número de elementos da matriz, pois `pathName` é uma referência:

C#

```
public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

Um `struct` pode conter uma matriz inserida em código não seguro. No exemplo a seguir, a matriz `fixedBuffer` tem tamanho fixo. Use uma [instrução fixed](#) para estabelecer um ponteiro ao primeiro elemento. Os elementos da matriz são acessados por este ponteiro. A instrução `fixed` fixa o campo da instância `fixedBuffer` em um local específico na memória.

C#

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class Example
{
    public Buffer buffer = default;
}

private static void AccessEmbeddedArray()
{
    var example = new Example();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
        Console.WriteLine(c);

        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}
```

```
    }  
}
```

O tamanho da matriz `char` de 128 elementos é 256 bytes. Buffers de `char` de tamanho fixo sempre têm dois bytes por caractere, independentemente da codificação. Esse tamanho de matriz é igual até mesmo quando os buffers de `char` passam por marshaling para structs ou métodos de API com `CharSet = CharSet.Auto` ou `CharSet = CharSet.Ansi`. Para obter mais informações, consulte [CharSet](#).

O exemplo anterior demonstra o acesso a campos `fixed` sem fixação. Outra matriz de tamanho fixo comum é a matriz `bool`. Os elementos de uma matriz `bool` tem sempre um byte de tamanho. Matrizes `bool` não são adequadas para criar buffers ou matrizes de bits.

Os buffers de tamanho fixo são compilados com [System.Runtime.CompilerServices.UnsafeValueTypeAttribute](#), o que instrui o CLR (Common Language Runtime) que um tipo contém uma matriz não gerenciada com o potencial de estourar. A memória alocada usando `stackalloc` também habilita automaticamente os recursos de detecção de sobrecarga de buffer no CLR. O exemplo anterior mostra como um buffer de tamanho fixo poderia existir em um `unsafe struct`.

C#

```
internal unsafe struct Buffer  
{  
    public fixed char fixedBuffer[128];  
}
```

O C# gerado pelo compilador para `Buffer` é atribuído da seguinte maneira:

C#

```
internal struct Buffer  
{  
    [StructLayout(LayoutKind.Sequential, Size = 256)]  
    [CompilerGenerated]  
    [UnsafeValueType]  

```

Buffers de tamanho fixo diferem de matrizes regulares das seguintes maneiras:

- Só pode ser usado em contextos `unsafe`.
- Pode ser apenas campos de instância de structs.
- Eles são sempre vetores ou matrizes unidimensionais.
- A declaração deve incluir o comprimento, como `fixed char id[8]`. Não é possível usar `fixed char id[]`.

## Como usar ponteiros para copiar uma matriz de bytes

O exemplo a seguir usa ponteiros para copiar bytes de uma matriz para outra.

Este exemplo usa a palavra-chave `não seguro`, que permite que você use ponteiros no método `Copy`. A instrução `fixo` é usada para declarar ponteiros para as matrizes de origem e de destino. A instrução `fixed` fixa o local das matrizes de origem e de destino na memória para que elas não sejam movidas pela coleta de lixo. Os blocos de memória para as matrizes não serão fixado quando o bloco `fixed` for concluído. Como o método `Copy` neste exemplo usa a palavra-chave `unsafe`, ele deve ser compilado com a opção do compilador `AllowUnsafeBlocks`.

Este exemplo acessa os elementos das duas matrizes usando índices em vez de um segundo ponteiro não gerenciado. A declaração dos ponteiros `pSource` e `pTarget` fixa as matrizes.

C#

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException("source or target is null");
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException("offset or bytes to copy is
negative");
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
```

```
// the copy.
if ((source.Length - sourceOffset < count) ||
    (target.Length - targetOffset < count))
{
    throw new System.ArgumentException("offset to end of array is less
than bytes to be copied");
}

// The following fixed statement pins the location of the source and
// target objects in memory so that they will not be moved by garbage
// collection.
fixed (byte* pSource = source, pTarget = target)
{
    // Copy the specified number of bytes from source to target.
    for (int i = 0; i < count; i++)
    {
        pTarget[targetOffset + i] = pSource[sourceOffset + i];
    }
}
}

static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
```

```

// The offset specifies where the copying begins in the source array.
int offset = length - 10;
Copy(byteArray1, offset, byteArray2, 0, length - offset);

// Display the first 10 elements in the copy, byteArray2.
System.Console.WriteLine("The first 10 elements of the copy are:");
for (int i = 0; i < 10; ++i)
{
    System.Console.Write(byteArray2[i] + " ");
}
System.Console.WriteLine("\n");
/* Output:
   The first 10 elements of the original are:
   0 1 2 3 4 5 6 7 8 9

   The first 10 elements of the copy are:
   0 1 2 3 4 5 6 7 8 9

   The first 10 elements of the copy are:
   90 91 92 93 94 95 96 97 98 99
*/
}

```

## Ponteiros de função

O C# fornece tipos `delegate` para definir objetos de ponteiro de função seguros. Invocar um delegado envolve instanciar um tipo derivado de `System.Delegate` e fazer uma chamada de método virtual para o método `Invoke` dele. Essa chamada virtual usa a instrução IL `callvirt`. Em caminhos do código críticos quanto ao desempenho, usar a instrução IL `calli` é mais eficiente.

Você pode definir um ponteiro de função usando a sintaxe `delegate*`. O compilador chamará a função usando a instrução `calli` em vez de instanciar um objeto `delegate` e chamar `Invoke`. O código a seguir declara dois métodos que usam um `delegate` ou um `delegate*` para combinar dois objetos do mesmo tipo. O primeiro método usa um tipo de delegado `System.Func<T1,T2,TResult>`. O segundo método usa uma declaração `delegate*` com os mesmos parâmetros e tipo de retorno:

C#

```

public static T Combine<T>(Func<T, T, T> combinator, T left, T right) =>
    combinator(left, right);

public static T UnsafeCombine<T>(<code>delegate*</code><code>T, T, T</code> combinator, T left, T
right) =>
    combinator(left, right);

```

O seguinte código mostra como você declararia uma função local estática e invocaria o método `UnsafeCombine` usando um ponteiro para tal função local:

C#

```
static int localMultiply(int x, int y) => x * y;
int product = UnsafeCombine(&localMultiply, 3, 4);
```

O código anterior ilustra várias das regras da função acessada como um ponteiro de função:

- Ponteiros de função só podem ser declarados em contextos `unsafe`.
- Métodos que aceitam `delegate*` (ou retornam `delegate*`) só podem ser chamados em contextos `unsafe`.
- O operador `&` para obter o endereço de uma função é permitido somente em funções `static`. (Essa regra se aplica a funções membro e funções locais).

A sintaxe tem paralelos com a declaração de tipos `delegate` e o uso de ponteiros. O sufixo `*` em `delegate` indica que a declaração é um *ponteiro de função*. O `&` ao atribuir um grupo de métodos a um ponteiro de função indica que a operação usa o endereço do método.

Você pode especificar a convenção de chamada para `delegate*` usando as palavras-chave `managed` e `unmanaged`. Além disso, para ponteiros de função `unmanaged`, você pode especificar a convenção de chamada. As declarações a seguir mostram exemplos de cada um. A primeira declaração usa a convenção de chamada `managed`, que é o padrão. As quatro a seguir usam a convenção de chamada `unmanaged`. Cada uma especifica uma das convenções de chamada do ECMA 335: `Cdecl`, `Stdcall`, `Fastcall` ou `Thiscall`. A última declaração usa a convenção de chamada `unmanaged`, instruindo o CLR a escolher a convenção de chamada padrão para a plataforma. O CLR escolherá a convenção de chamada em tempo de execução.

C#

```
public static T ManagedCombine<T>(delegate* managed<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T CDeclCombine<T>(delegate* unmanaged[Cdecl]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T StdcallCombine<T>(delegate* unmanaged[Stdcall]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T FastcallCombine<T>(delegate* unmanaged[Fastcall]<T, T, T>
```

```
combinator, T left, T right) =>
    combinator(left, right);
public static T ThiscallCombine<T>(delegate* unmanaged[Thiscall]<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
public static T UnmanagedCombine<T>(delegate* unmanaged<T, T, T> combinator,
T left, T right) =>
    combinator(left, right);
```

Você pode saber mais sobre ponteiros de função na proposta [Ponteiro de função para C# 9.0](#).

## Especificação da linguagem C#

Para obter mais informações, confira o capítulo [Código não seguro](#) da [Especificação da linguagem C#](#).

# Diretivas de pré-processador do C#

Artigo • 05/06/2023

Embora o compilador não tenha um pré-processador separado, as diretivas descritas nesta seção são processadas como se houvesse um. Você as usa para ajudar na compilação condicional. Ao contrário das diretivas de C e C++, não é possível usar essas diretivas para criar macros. Uma diretiva de pré-processador deve ser a única instrução em uma linha.

## Contexto que permite valor nulo

A diretiva de pré-processador `#nullable` define o *contexto de anotação anulável* e o *contexto de aviso anulável*. Essa diretiva controla se as anotações anuláveis têm efeito e se os avisos de nulidade são dados. Cada contexto está *desabilitado* ou *habilitado*.

Ambos os contextos podem ser especificados no nível do projeto (fora do código-fonte C#). A diretiva `#nullable` controla os contextos de anotação e aviso e tem precedência sobre as configurações no nível do projeto. Uma diretiva define os contextos que controla até que outra diretiva a substitua ou até o final do arquivo de origem.

O efeito das diretivas é o seguinte:

- `#nullable disable`: define a anotação anulável e os contextos de aviso como *desabilitado*.
- `#nullable enable`: define os contextos de aviso e anotação anuláveis como *habilitado*.
- `#nullable restore`: restaura a anotação anulável e os contextos de aviso para as configurações do projeto.
- `#nullable disable annotations`: define o contexto de anotação anulada como *desabilitado*.
- `#nullable enable annotations`: define o contexto de anotação anulável como *habilitado*.
- `#nullable restore annotations`: restaura o contexto de anotação anulável para as configurações do projeto.
- `#nullable disable warnings`: define o contexto de aviso anulável como *desabilitado*.
- `#nullable enable warnings`: define o contexto de aviso anulável como *habilitado*.
- `#nullable restore warnings`: restaura o contexto de aviso anulável nas configurações do projeto.

# Compilação condicional

Você usa quatro diretivas de pré-processador para controlar a compilação condicional:

- `#if`: abre uma compilação condicional, em que o código é compilado somente se o símbolo especificado for definido.
- `#elif`: fecha a compilação condicional anterior e abre uma nova compilação condicional com base no caso do símbolo especificado ser definido.
- `#else`: fecha a compilação condicional anterior e abre uma nova compilação condicional se o símbolo especificado anteriormente não estiver definido.
- `#endif`: fecha a compilação condicional anterior.

O compilador C# compila o código entre a diretiva `#if` e a diretiva `#endif` somente se o símbolo especificado está definido ou não definido quando o operador `!` não é usado. Ao contrário do C e do C++, não é possível atribuir um valor numérico a um símbolo. A instrução `#if` em C# é booliana e testa apenas quando o símbolo foi definido ou não. Por exemplo, o seguinte código é compilado quando `DEBUG` é definido:

```
C#  
  
#if DEBUG  
    Console.WriteLine("Debug version");  
#endif
```

O seguinte código é compilado quando `MYTEST` não é definido:

```
C#  
  
#if !MYTEST  
    Console.WriteLine("MYTEST is not defined");  
#endif
```

Você pode usar os operadores `==` (igualdade) e `!=` (desigualdade) para testar os `bool` valores `true` ou `false`. `true` significa que o símbolo foi definido. A instrução `#if DEBUG` tem o mesmo significado que `#if (DEBUG == true)`. Você pode usar os `&&` operadores (`e`), `||` (`ou`) e `!` (`não`) para avaliar se vários símbolos foram definidos. Também é possível agrupar os símbolos e operadores com parênteses.

`#if`, juntamente com as diretivas `#else`, `#elif`, `#endif`, `#define` e `#undef` permite incluir ou excluir código com base na existência de um ou mais símbolos. A compilação condicional pode ser útil ao compilar código para uma compilação de depuração ou ao compilar para uma configuração específica.

Uma diretiva condicional que começa com `#if` deve ser terminada explicitamente com uma diretiva `#endif`. A diretiva `#define` permite definir um símbolo. Ao usar o símbolo como a expressão passada para a diretiva `#if`, a expressão será avaliada como `true`. Você também pode definir um símbolo com a opção do compilador [DefineConstants](#). Você pode anular a definição um símbolo com `#undef`. O escopo de um símbolo criado com `#define` é o arquivo no qual ele foi definido. Um símbolo que você define com [DefineConstants](#) ou com `#define` não entra em conflito com uma variável de mesmo nome. Ou seja, um nome de variável não deve ser passado para uma diretiva de pré-processador, e um símbolo só pode ser avaliado por uma diretiva de pré-processador.

O `#elif` permite criar uma diretiva condicional composta. A expressão `#elif` será avaliada se nem as expressões de diretiva anterior `#if` nem outras anteriores, opcionais `#elif` são avaliadas como `true`. Se uma expressão `#elif` for avaliada como `true`, o compilador avaliará todo o código entre `#elif` e a próxima diretiva condicional. Por exemplo:

```
C#  
  
#define VC7  
//...  
#if DEBUG  
    Console.WriteLine("Debug build");  
#elif VC7  
    Console.WriteLine("Visual Studio 7");  
#endif
```

`#else` permite que você crie uma diretiva condicional composta, para que, caso nenhuma das expressões nas diretivas anteriores `#if` ou (opcional) `#elif` seja avaliada como `true`, o compilador avalie todo o código entre `#else` e a próxima `#endif`. `#endif`(`#endif`) deve ser a próxima diretiva de pré-processador após `#else`.

`#endif` especifica o final de uma diretiva condicional, que começou com a diretiva `#if`.

O sistema de compilação também está ciente dos símbolos de pré-processamento predefinidos que representam várias [estruturas de destino](#) em projetos no estilo SDK. Eles são úteis ao criar aplicativos que podem ter como destino mais de uma versão do .NET.

<b>Frameworks</b>	<b>Símbolos de destino</b>	<b>Símbolos adicionais (disponível em SDKs do .NET 5+)</b>	<b>Símbolos de plataforma (disponíveis somente quando você especifica um TFM específico do sistema operacional)</b>
.NET Framework	NETFRAMEWORK, NET48, NET472, NET471, NET47, NET462, NET461, NET46, NET452, NET451, NET45, NET40, NET35, NET20	NET48_OR_GREATER, NET472_OR_GREATER, NET471_OR_GREATER, NET47_OR_GREATER, NET462_OR_GREATER, NET461_OR_GREATER, NET46_OR_GREATER, NET452_OR_GREATER, NET451_OR_GREATER, NET45_OR_GREATER, NET40_OR_GREATER, NET35_OR_GREATER, NET20_OR_GREATER	
.NET Standard	NETSTANDARD, NETSTANDARD2_1, NETSTANDARD2_0, NETSTANDARD1_6, NETSTANDARD1_5, NETSTANDARD1_4, NETSTANDARD1_3, NETSTANDARD1_2, NETSTANDARD1_1, NETSTANDARD1_0	NETSTANDARD2_1_OR_GREATER, NETSTANDARD2_0_OR_GREATER, NETSTANDARD1_6_OR_GREATER, NETSTANDARD1_5_OR_GREATER, NETSTANDARD1_4_OR_GREATER, NETSTANDARD1_3_OR_GREATER, NETSTANDARD1_2_OR_GREATER, NETSTANDARD1_1_OR_GREATER, NETSTANDARD1_0_OR_GREATER	
.NET 5+ (e.NET Core)	NET, NET7_0, NET6_0, NET5_0, NETCOREAPP, NETCOREAPP3_1, NETCOREAPP3_0, NETCOREAPP2_2, NETCOREAPP2_1, NETCOREAPP2_0, NETCOREAPP1_1, NETCOREAPP1_0	NET7_0_OR_GREATER, NET6_0_OR_GREATER, NET5_0_OR_GREATER, NETCOREAPP3_1_OR_GREATER, NETCOREAPP3_0_OR_GREATER, NETCOREAPP2_2_OR_GREATER, NETCOREAPP2_1_OR_GREATER, NETCOREAPP2_0_OR_GREATER, NETCOREAPP1_1_OR_GREATER, NETCOREAPP1_0_OR_GREATER	ANDROID, IOS, MACCATALYST, MACOS, TVOS, WINDOWS, [OS][version] (por exemplo, IOS15_1), [OS] [version]_OR_GREATER (por exemplo, IOS15_1_OR_GREATER)

① Observação

- Os símbolos sem versão são definidos independentemente da versão para a qual você está direcionando.
- Os símbolos específicos à versão são definidos apenas para a versão que você está direcionando.
- Os símbolos `<framework>_OR_GREATER` são definidos para a versão que você está direcionando e todas as versões anteriores. Por exemplo, se você estiver direcionando .NET Framework 2.0, os seguintes símbolos serão definidos:  
`NET20`, `NET20_OR_GREATER`, `NET11_OR_GREATER` e `NET10_OR_GREATER`.
- Eles são diferentes dos TFM's (Moniker da Estrutura de Destino) usados pela propriedade **MSBuild TargetFramework** e pelo **NuGet**.

## ① Observação

Para projetos tradicionais, não estilo SDK, você precisa configurar manualmente os símbolos de compilação condicional para as diferentes estruturas de destino no Visual Studio por meio das páginas de propriedades do projeto.

Outros símbolos predefinidos incluem as constantes `DEBUG` e `TRACE`. Para substituir os valores definidos no projeto, use a diretiva `#define`. Por exemplo, o símbolo `DEBUG` é definido automaticamente, de acordo com as propriedades de configuração do build (Modo de Depuração ou Modo de Versão).

O exemplo a seguir mostra como definir um símbolo `MYTEST` em um arquivo e testar os valores dos símbolos `MYTEST` e `DEBUG`. A saída deste exemplo depende da sua escolha ao compilar o projeto: se você optou pelo modo de configuração **Debug** ou **Release**.

C#

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
```

```
    }  
}
```

O exemplo a seguir mostra como testar várias estruturas de destino para que você possa usar APIs mais recentes, quando possível:

C#

```
public class MyClass  
{  
    static void Main()  
    {  
        #if NET40  
            WebClient _client = new WebClient();  
        #else  
            HttpClient _client = new HttpClient();  
        #endif  
    }  
    //...  
}
```

## Definindo símbolos

Use as duas diretivas de pré-processador seguintes para definir ou anular a definição de símbolos para compilação condicional:

- `#define`: define um símbolo.
- `#undef`: anula a definição de um símbolo.

Use `#define` para definir um símbolo. Quando você usa o símbolo como a expressão passada para a diretiva `#if`, a expressão será avaliada como `true`, conforme mostra o seguinte exemplo:

C#

```
#define VERBOSE  
  
#if VERBOSE  
    Console.WriteLine("Verbose output version");  
#endif
```

ⓘ Observação

A diretiva `#define` não pode ser usada para declarar valores constantes como normalmente é feito em C e C++. As constantes em C# são mais bem definidas como membros estáticos de uma classe ou struct. Se você tiver várias dessas constantes, considere criar uma classe "Constantes" separada para guardá-las.

Os símbolos podem ser usados para especificar condições para compilação. É possível testar o símbolo com `#if` ou `#elif`. Você também pode usar o [ConditionalAttribute](#) para executar uma compilação condicional. É possível definir um símbolo, mas não é possível atribuir um valor a um símbolo. A diretiva `#define` deve ser exibida no arquivo antes de usar as instruções que também não são diretivas de pré-processador. Você também pode definir um símbolo com a opção do compilador [DefineConstants](#). Você pode anular a definição um símbolo com `#undef`.

## Definindo regiões

Você pode definir regiões de código que podem ser recolhidas em uma estrutura de tópicos usando as duas diretivas de pré-processador seguintes:

- `#region`: inicia uma região.
- `#endregion`: encerra uma região.

`#region` permite que você especifique um bloco de código que pode ser expandido ou recolhido ao usar o recurso de [estrutura de tópicos](#) do editor de código. Em arquivos de código mais longos, é conveniente recolher ou ocultar uma ou mais regiões para que você possa se concentrar na parte do arquivo que está trabalhando no momento. O exemplo a seguir mostra como definir uma região:

C#

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

Um bloco `#region` deve ser encerrado com a diretiva `#endregion`. Um `#region` bloco não pode se sobrepor a um bloco `#if`. No entanto, um bloco `#region` pode ser aninhado em um bloco `#if` e um bloco `#if` pode ser aninhado em um bloco `#region`.

# Informações sobre erros e avisos

Você instrui o compilador a gerar erros e avisos do compilador definidos pelo usuário e controlar informações de linha usando as seguintes diretivas:

- `#error`: gere um erro do compilador com uma mensagem específica.
- `#warning`: gere um aviso do compilador com uma mensagem específica.
- `#line`: altere o número de linha impresso com mensagens do compilador.

`#error` permite gerar um erro definido pelo usuário [CS1029](#) de um local específico em seu código. Por exemplo:

```
C#
```

```
#error Deprecated code in this method.
```

## ⓘ Observação

O compilador trata `#error version` de maneira especial e relata um erro do compilador, CS8304, com uma mensagem que contém o compilador usado e as versões de idioma.

`#warning` permite gerar um aviso do compilador [CS1030](#) de nível um de um local específico no código. Por exemplo:

```
C#
```

```
#warning Deprecated code in this method.
```

O `#line` permite modificar o número de linha do compilador e (opcionalmente) a saída do nome de arquivo para erros e avisos.

O exemplo a seguir mostra como relatar dois avisos associados aos números de linha. A diretiva `#line 200` força o próximo número de linha a ser 200 (embora o padrão seja #6) e, até a próxima diretiva `#line`, o nome de arquivo será relatado como "Special". A diretiva `#line default` retorna a numeração de linhas à sua numeração padrão, que conta as linhas que foram renumeradas pela diretiva anterior.

```
C#
```

```
class MainClass
{
```

```
static void Main()
{
#line 200 "Special"
    int i;
    int j;
#line default
    char c;
    float f;
#line hidden // numbering not affected
    string s;
    double d;
}
}
```

A compilação produz a saída a seguir:

Console

```
Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never
used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never
used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never
used
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never
used
```

A diretiva `#line` pode ser usada em uma etapa intermediária e automatizada no processo de build. Por exemplo, se linhas fossem removidas do arquivo de código-fonte original, mas você ainda deseja que o compilador gere a saída com base na numeração de linha original no arquivo, seria possível remover as linhas e, em seguida, simular a numeração de linha original com `#line`.

A diretiva `#line hidden` oculta as linhas sucessivas do depurador, de modo que, quando o desenvolvedor percorrer o código, quaisquer linhas entre um `#line hidden` e a próxima diretiva `#line` (supondo que não seja outra diretiva `#line hidden`) serão puladas. Essa opção também pode ser usada para permitir que o ASP.NET diferencie entre o código gerado pelo computador e definido pelo usuário. Embora o ASP.NET seja o principal consumidor desse recurso, é provável que mais geradores de origem façam uso dele.

A diretiva `#line hidden` não afeta os nomes de arquivo ou números de linha nos relatórios de erro. Ou seja, se o compilador encontrar um erro em um bloco oculto, o compilador relatará o nome do arquivo atual e o número de linha do erro.

A diretiva `#line filename` especifica o nome de arquivo que você deseja que seja exibido na saída do compilador. Por padrão, é usado o nome real do arquivo de código-fonte. O nome de arquivo deve estar entre aspas duplas ("") e deve ser precedido por um número de linha.

Do C# 10 em diante, você pode usar uma nova forma da diretiva `#line`:

C#

```
#line (1, 1) - (5, 60) 10 "partial-class.cs"  
/*34567*/int b = 0;
```

Os componentes dessa nova forma são:

- `(1, 1)`: a linha de início e a coluna do primeiro caractere na linha que segue a diretiva. Neste exemplo, a próxima linha seria relatada como linha 1, coluna 1.
- `(5, 60)`: a linha de extremidade e a coluna da região marcada.
- `10`: o deslocamento da coluna para que a diretiva `#line` entre em vigor. Neste exemplo, a 10<sup>a</sup> coluna seria relatada como coluna um. É aí que começa a declaração `int b = 0;`. Esse campo é opcional. Se omitida, a diretiva vai entrar em vigor na primeira coluna.
- `"partial-class.cs"`: nome do arquivo de saída.

O exemplo anterior geraria o seguinte aviso:

CLI do .NET

```
partial-class.cs(1,5,1,6): warning CS0219: The variable 'b' is assigned but  
its value is never used
```

Depois de remapear, a variável `b` está na primeira linha, no caractere seis do arquivo `partial-class.cs`.

As DSLs (linguagens específicas do domínio) normalmente usam esse formato para fornecer um mapeamento melhor do arquivo de origem para a saída gerada em C#. O uso mais comum dessa diretiva `#line` estendida é mapear novamente avisos ou erros que aparecem em um arquivo gerado para a origem original. Por exemplo, considere esta página razor:

razor

```
@page "/"  
Time: @DateTime.NowAndThen
```

A propriedade `DateTime.Now` foi digitada incorretamente como `DateTime.NowAndThen`. O C# gerado para o snippet razor é semelhante ao seguinte, em `page.g.cs`:

```
C#  
  
_builder.Add("Time: ");  
#line (2, 6) (2, 27) 15 "page.razor"  
_builder.Add(DateTime.NowAndThen);
```

A saída do compilador para o snippet anterior é:

```
CLI do .NET  
  
page.razor(2, 2, 2, 27)error CS0117: 'DateTime' does not contain a  
definition for 'NowAndThen'
```

Linha 2, coluna 6 em `page.razor` é onde o texto `@DateTime.NowAndThen` começa. Isso é notado por `(2, 6)` na diretiva. Esse intervalo de `@DateTime.NowAndThen` termina na linha 2, coluna 27. Isso é notado pelo `(2, 27)` na diretiva. O texto para `DateTime.NowAndThen` começa na coluna 15 de `page.g.cs`. Isso é notado pelo `15` na diretiva. Juntando todos os argumentos, o compilador relata o erro em sua localização em `page.razor`. O desenvolvedor pode navegar diretamente até o erro no código-fonte, não na origem gerada.

Para ver mais exemplos desse formato, confira a [especificação de recurso](#) na seção sobre exemplos.

## Pragmas

O `#pragma` fornece ao compilador instruções especiais para a compilação do arquivo no qual ele é exibido. O compilador deve dar suporte às instruções. Em outras palavras, não é possível usar `#pragma` para criar instruções personalizadas de pré-processamento.

- `#pragma warning`: habilite ou desabilite avisos.
- `#pragma checksum`: gere uma soma de verificação.

```
C#  
  
#pragma pragma-name pragma-arguments
```

Em que `pragma-name` é o nome de um pragma reconhecido e `pragma-arguments` são os argumentos específicos do pragma.

# #pragma warning

O `#pragma warning` pode habilitar ou desabilitar determinados avisos.

C#

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

Em que `warning-list` é uma lista de números de aviso separada por vírgulas. O prefixo "CS" é opcional. Quando não houver números de aviso especificados, o `disable` desabilita todos os avisos e o `restore` habilita todos os avisos.

## ⓘ Observação

Para localizar números de aviso no Visual Studio, compile o projeto e, em seguida, procure os números de aviso na janela de **Saída**.

`disable` entra em vigor desde a próxima linha do arquivo de origem. O aviso é restaurado na linha que segue `restore`. Se não houver nenhum `restore` no arquivo, os avisos serão restaurados para o estado padrão na primeira linha de qualquer arquivo posterior na mesma compilação.

C#

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}
#pragma warning restore CS3021
[CLSCompliant(false)] // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

## #pragma checksum

Gera somas de verificação para os arquivos de origem para ajudar na depuração de páginas do ASP.NET.

C#

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

Em que `"filename"` é o nome do arquivo que requer monitoramento para alterações ou atualizações, `"{guid}"` é o GUID (Globally Unique Identifier) para o algoritmo hash e `"checksum_bytes"` é a cadeia de caracteres de dígitos hexadecimal que representam os bytes da soma de verificação. Deve ser um número par de dígitos hexadecimais. Um número ímpar de dígitos resulta em um aviso em tempo de compilação e a diretiva é ignorada.

O depurador do Visual Studio usa uma soma de verificação para garantir sempre a localização da fonte correta. O compilador calcula a soma de verificação para um arquivo de origem e, em seguida, emite a saída no arquivo PDB (banco de dados do programa). Em seguida, o depurador usa o PDB para comparar com a soma de verificação que ele calcula para o arquivo de origem.

Essa solução não funciona para projetos do ASP.NET, porque é a soma de verificação calculada para o arquivo de origem gerado e não para o arquivo .aspx. Para resolver esse problema, a `#pragma checksum` fornece suporte à soma de verificação para páginas do ASP.NET.

Quando você cria um projeto do ASP.NET em Visual C#, o arquivo de origem gerado contém uma soma de verificação para o arquivo .aspx, do qual a fonte é gerada. Então, o compilador grava essas informações no arquivo PDB.

Se o compilador não encontrar uma diretiva `#pragma checksum` no arquivo, ele calcula a soma de verificação e grava o valor no arquivo PDB.

C#

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}"
"ab007f1d23d9" // New checksum
    }
}
```

# Opções do compilador de C#

Artigo • 15/02/2023

Esta seção descreve as opções interpretadas pelo compilador de C#. As opções são agrupadas em artigos separados com base no que eles controlam, por exemplo, recursos de linguagem, geração de código e saída. Use o sumário para navegar entre elas.

## Como definir opções

Há duas maneiras de definir opções de compilador em projetos do .NET:

- *No arquivo \*.csproj*

Você pode adicionar propriedades do MSBuild a qualquer opção do compilador no arquivo `*.csproj` no formato XML. O nome da propriedade é o mesmo que a opção do compilador. O valor da propriedade define o valor da opção do compilador. Por exemplo, o snippet de arquivo de projeto a seguir define a propriedade `LangVersion`.

XML

```
<PropertyGroup>
  <LangVersion>preview</LangVersion>
</PropertyGroup>
```

Para obter mais informações sobre as opções de configuração em arquivos de projeto, consulte o artigo [Propriedades do MSBuild para projetos do SDK do .NET](#).

- *Usando as páginas de propriedades do Visual Studio*

O Visual Studio fornece páginas de propriedades para editar propriedades de build. Para saber mais sobre elas, consulte [Gerenciar propriedades do projeto e da solução – Windows](#) ou [Gerenciar propriedades do projeto e da solução – Mac](#).

## Projetos .NET Framework

### Importante

Esta seção se aplica somente a projetos .NET Framework.

Além dos mecanismos descritos acima, você pode definir opções de compilador usando dois métodos adicionais para projetos .NET Framework:

- **Argumentos de linha de comando para projetos .NET Framework:** projetos .NET Framework usam `csc.exe` em vez de `dotnet build` para criar projetos. Você pode especificar argumentos de linha de comando para `csc.exe` para projetos .NET Framework.
- **Páginas ASP.NET compiladas:** projetos .NET Framework usam uma seção do arquivo `web.config` para compilar páginas. Para o novo sistema de build e projetos ASP.NET Core, as opções são obtidas do arquivo de projeto.

A palavra para algumas opções do compilador foi alterada de `csc.exe` e projetos .NET Framework para o novo sistema MSBuild. A nova sintaxe é usada nesta seção. Ambas as versões são listadas na parte superior de cada página. Para `csc.exe`, todos os argumentos são listados seguindo a opção e dois-pontos. Por exemplo, a opção `-doc` seria:

Console

```
-doc:DocFile.xml
```

Você pode invocar o compilador do C#, digitando o nome do seu arquivo executável (`csc.exe`) em um prompt de comando.

Para projetos .NET Framework, você também pode executar `csc.exe` na linha de comando. Todas as opções do compilador estão disponíveis em duas formas: **-option** e **/option**. Em projetos web .NET Framework, você especifica opções para compilar code-behind no arquivo `web.config`. Para obter mais informações, consulte [<compilador> Elemento](#).

Se você usar a janela do **Prompt de Comando do Desenvolvedor do Visual Studio**, todas as variáveis de ambiente necessárias serão definidas para você. Para obter informações sobre como acessar essa ferramenta, consulte [Prompt de Comando do Desenvolvedor para o Visual Studio](#).

O arquivo executável `csc.exe` normalmente está localizado na pasta `Microsoft.NET\Framework\<Versão>` no diretório `Windows`. O local pode variar dependendo da configuração exata de um computador específico. Se mais de uma versão do .NET Framework estiver instalada em seu computador, você encontrará várias versões desse arquivo. Para obter mais informações sobre essas instalações, consulte [Determinando qual versão do .NET Framework está instalada](#).

# Opções do compilador C# para regras de recurso de idioma

Artigo • 05/06/2023

As opções a seguir controlam como o compilador interpreta os recursos de linguagem. A nova sintaxe MSBuild é mostrada em **Negrito**. A sintaxe `csc.exe` mais antiga é mostrada em `code style`.

- **CheckForOverflowUnderflow** / `-checked`: gerar verificações de estouro.
- **AllowUnsafeBlocks** / `-unsafe`: permitir código 'inseguro'.
- **DefineConstants** / `-define`: definir símbolos de compilação condicional.
- **LangVersion** / `-langversion`: especificar a versão de idioma como `default` (versão principal mais recente) ou `latest` (versão mais recente, incluindo versões secundárias).
- **Anulável** / `-nullable`: habilitar o contexto anulável ou avisos anuláveis.

## CheckForOverflowUnderflow

A opção **CheckForOverflowUnderflow** controla o contexto da verificação de estouro padrão que define o comportamento do programa no caso de estouros aritméticos inteiros.

XML

```
<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
```

Quando **CheckForOverflowUnderflow** é `true`, o contexto padrão é um contexto verificado e a verificação de estouro é habilitada; caso contrário, o contexto padrão é um contexto desmarcado. O valor padrão para essa opção é `false`, ou seja, verificação de estouro é desabilitada.

Você também pode controlar explicitamente o contexto de verificação de estouro para as partes do código usando as instruções `checked` e `unchecked`.

Para obter informações sobre como o contexto de verificação de estouro afeta as operações e quais operações são afetadas, confira o [artigo sobre checked e instruções unchecked](#).

# AllowUnsafeBlocks

A opção do compilador **AllowUnsafeBlocks** permite que o código que usa a palavra-chave `unsafe` seja compilado. O valor padrão para essa opção é `false`, o que significa que código não seguro não é permitido.

XML

```
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

Para obter mais informações sobre código não seguro, consulte [Código não seguro e ponteiros](#).

# DefineConstants

A opção **DefineConstants** define símbolos em todos os arquivos de código-fonte do seu programa.

XML

```
<DefineConstants>name;name2</DefineConstants>
```

Essa opção especifica os nomes de um ou mais símbolos que você deseja definir. A opção **DefineConstants** tem o mesmo efeito que usar uma diretiva de pré-processador `#define`, exceto que a opção do compilador está em vigor para todos os arquivos no projeto. Um símbolo permanece definido em um arquivo de origem até que uma diretiva `#undef` remova a definição no arquivo de origem. Quando você usa a opção `-define`, uma diretiva `#undef` em um arquivo não terá nenhum efeito em outros arquivos de código-fonte no projeto. Você pode usar os símbolos criados por essa opção com `#if`, `#else`, `#elif` e `#endif` para compilar os arquivos de origem condicionalmente. O compilador do C# não define símbolos ou macros que podem ser usados em seu código-fonte. Todas as definições de símbolo devem ser definidas pelo usuário.

## ① Observação

A diretiva `#define` do C# não permite que um valor seja dado a um símbolo, como nas linguagens como o C++. Por exemplo, `#define` não pode ser usado para criar uma macro ou para definir uma constante. Se você precisar definir uma constante, use uma variável `enum`. Se você quiser criar uma macro de estilo C++, considere

alternativas como os genéricos. Como as macros são notoriamente propensas a erros, o C# não permite o uso delas, mas oferece alternativas mais seguras.

## LangVersion

Faz com que o compilador aceite somente a sintaxe incluída na especificação de linguagem C# escolhida.

XML

```
<LangVersion>9.0</LangVersion>
```

Os seguintes valores são válidos:

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 ou default	O compilador aceita a sintaxe da versão principal mais recente lançada do compilador.
11.0	O compilador aceita somente a sintaxe incluída no C# 11 ou inferior.
10.0	O compilador aceita somente a sintaxe incluída no C# 10 ou inferior.
9.0	O compilador aceita somente a sintaxe incluída no C# 9 ou inferior.
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.

Valor	Significado
3	O compilador aceita somente a sintaxe incluída no C# 3.0 ou inferior.
ISO-2 ou 2	O compilador aceita somente a sintaxe incluída no ISO/IEC 23270:2006 C# (2.0).
ISO-1 ou 1	O compilador aceita somente a sintaxe incluída no ISO/IEC 23270:2003 C# (1.0/1.2).

A versão da linguagem padrão depende da estrutura de destino do aplicativo e da versão do SDK ou do Visual Studio instalado. Essas regras são definidas em [Controle de versão da linguagem C#](#).

### ⓘ Importante

O valor `latest` geralmente não é recomendado. Com ele, o compilador habilita os recursos mais recentes, mesmo que esses recursos dependam de atualizações não incluídas na estrutura de destino configurada. Sem essa configuração, o projeto usa a versão do compilador recomendada para a estrutura de destino. Você pode atualizar a estrutura de destino para acessar recursos de linguagem mais recentes.

Os metadados referenciados pelo seu aplicativo de C# não estão sujeitos à opção do compilador **Langversion**.

Como cada versão do compilador do C# contém extensões para a especificação de linguagem, **Langversion** não dá a funcionalidade equivalente de uma versão anterior do compilador.

Além disso, embora as atualizações de versão do C# geralmente coincidam com as versões principais do .NET, a nova sintaxe e as funcionalidades não estão necessariamente vinculadas a essa versão de estrutura específica. Cada funcionalidade específica tem os próprios requisitos mínimos de API ou do Common Language Runtime do .NET, que podem permitir que ela seja executada em estruturas de nível inferior com a inclusão de pacotes NuGet ou de outras bibliotecas.

Independentemente de qual configuração **Langversion** for usada, use a versão atual do Common Language Runtime para criar seu .exe ou .dll. Uma exceção são os assemblies amigáveis e **ModuleAssemblyName**, que funcionarão em-`langversion:ISO-1`.

Para descobrir outras maneiras de especificar a versão da linguagem C#, confira [Controle de versão da linguagem C#](#).

Para saber mais sobre como definir essa opção do compilador programaticamente, veja [LanguageVersion](#).

## Especificação da linguagem C#

Versão	Link	Descrição
C# 7.0 e posterior	<a href="#">link</a>	Especificação da Linguagem C# Versão 7 – Rascunho não oficial: .NET Foundation
C# 6.0	<a href="#">Baixar PDF</a>	Padrão ECMA-334 – 6ª Edição
C# 5.0	<a href="#">Baixar PDF</a>	Padrão ECMA-334 – 5ª Edição
C# 3.0	<a href="#">Baixar DOC</a>	Especificação da Linguagem C# Versão 3.0: Microsoft Corporation
C# 2.0	<a href="#">Baixar PDF</a>	Padrão ECMA-334 – 4ª Edição
C# 1.2	<a href="#">Baixar DOC</a>	Especificação da Linguagem C# Versão 1.2: Microsoft Corporation
C# 1.0	<a href="#">Baixar DOC</a>	Especificação da Linguagem C# Versão 1.0: Microsoft Corporation

## Versão mínima do SDK necessária para dar suporte a todos os recursos de idioma

A tabela a seguir lista as versões mínimas do SDK com o compilador C# que dá suporte à versão de idioma correspondente:

Versão do C#	Versão mínima do SDK
C# 11	Microsoft Visual Studio/Build Tools 2022, versão 17.4 ou SDK do .NET 7
C# 10	Microsoft Visual Studio/Build Tools 2022 ou SDK do .NET 6
C# 9.0	Microsoft Visual Studio/Build Tools 2019, versão 16.8 ou SDK do .NET 5
C# 8.0	Microsoft Visual Studio/Build Tools 2019, versão 16.3 ou SDK do .NET Core
C# 7.3	Microsoft Visual Studio/Ferramentas de Build 2017, versão 15.7
C# 7.2	Microsoft Visual Studio/Ferramentas de Build 2017, versão 15.5

Versão do C#	Versão mínima do SDK
C# 7.1	Microsoft Visual Studio/Ferramentas de Build 2017, versão 15.3
C# 7.0	Microsoft Visual Studio/Ferramentas de Build 2017
C# 6	Microsoft Visual Studio/Ferramentas de Build 2015
C# 5	Microsoft Visual Studio/Ferramentas de Build 2012 ou compilador do .NET Framework 4.5 em pacote
C# 4	Microsoft Visual Studio/Ferramentas de Build 2010 ou compilador do .NET Framework 4.0 em pacote
C# 3	Microsoft Visual Studio/Ferramentas de Build 2008 ou compilador do .NET Framework 3.5 em pacote
C# 2	Microsoft Visual Studio/Ferramentas de Build 2005 ou compilador do .NET Framework 2.0 em pacote
C# 1.0/1.2	Microsoft Visual Studio/Build Tools .NET 2002 ou compilador em pacote .NET Framework 1.0

## Nullable

A opção **Nullable** permite que você especifique o contexto anulável. Ele pode ser definido na configuração do projeto usando a marca `<Nullable>`:

XML

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

O argumento deve `enable`, `disable`, `warnings` OU `annotations`. O argumento `enable` habilita o contexto anulável. A especificação `disable` desabilitará o contexto anulável. Quando você especifica o argumento `warnings`, o contexto de aviso anulável é habilitado. Quando você especifica o argumento `annotations`, o contexto de anotação anulável é habilitado. Os valores são descritos e explicados no artigo sobre [Contextos anuláveis](#). Saiba mais sobre as tarefas envolvidas na habilitação de tipos de referência anuláveis em uma base de código existente no artigo sobre [estratégias de migração anuláveis](#).

ⓘ Observação

Quando não há nenhum valor definido, o valor `disable` padrão é aplicado, no entanto, os modelos .NET 6 são, por padrão, fornecidos com o valor **Nullable** definido como `enable`.

A análise de fluxo é usada para inferir a nulidade de variáveis no código executável. A nulidade inferida de uma variável é independente da nulidade declarada da variável. As chamadas de método são analisadas mesmo quando são condicionalmente omitidas. Por exemplo, `Debug.Assert` no modo de versão.

A invocação de métodos anotados com os seguintes atributos também afetará a análise de fluxo:

- Pré-condições simples: `AllowNullAttribute` e `DisallowNullAttribute`
- Pós-condições simples: `BeNullAttribute` e `NotNullAttribute`
- Pós-condições condicionais: `BeNullWhenAttribute` e `NotNullWhenAttribute`
- `DoesNotReturnIfAttribute` (por exemplo, `DoesNotReturnIf(false)` para `Debug.Assert`) e `DoesNotReturnAttribute`
- `NotNullIfNotNullAttribute`
- Pós-condições de membro: `MemberNotNullAttribute(String)` e `MemberNotNullAttribute(String[])`

### ⓘ Importante

O contexto global anulável não se aplica aos arquivos de código gerados. Independentemente dessa configuração, o contexto anulável é *desabilitado* para qualquer arquivo de origem marcado como gerado. Há quatro maneiras de um arquivo ser marcado como gerado:

1. No `.editorconfig`, especifique `generated_code = true` em uma seção que se aplica a esse arquivo.
2. Coloque `<auto-generated>` ou `<auto-generated/>` em um comentário na parte superior do arquivo. Ele pode estar em qualquer linha nesse comentário, mas o bloco de comentários deve ser o primeiro elemento do arquivo.
3. Inicie o nome do arquivo com `TemporaryGeneratedFile_`
4. Termine o nome do arquivo com `.designer.cs`, `.generated.cs`, `.g.cs` ou `.g.i.cs`.

Os geradores podem aceitar usando a diretiva de pré-processador `#nullable`.

# Opções do compilador C# que controlam a saída do compilador

Artigo • 10/05/2023

As seguintes opções controlam a geração de saída do compilador.

MSBuild	csc.exe	Descrição
DocumentationFile	-doc:	Gere o arquivo de documento XML a partir de comentários <code>///</code> .
OutputAssembly	-out:	Especifique o arquivo de assembly de saída.
PlatformTarget	- platform:	Especifique a CPU da plataforma de destino.
ProduceReferenceAssembly	-refout:	Gera um assembly de referência.
TargetType	-target:	Especifique o tipo do assembly de saída.

## DocumentationFile

A opção **DocumentationFile** permite colocar comentários de documentação em um arquivo XML. Para saber mais sobre como documentar seu código, consulte [Marcações Recomendadas para Comentários de Documentação](#). O valor especifica o caminho para o arquivo XML de saída. O arquivo XML contém os comentários nos arquivos de código-fonte da compilação.

XML

```
<DocumentationFile>path/to/file.xml</DocumentationFile>
```

O arquivo de código-fonte que contém as instruções principal ou de nível superior é apresentado primeiro no XML. Você geralmente deseja usar o arquivo .xml gerado com o [IntelliSense](#). O nome.xml deve ser o mesmo que o nome do assembly. O arquivo .xml deve estar no mesmo diretório que o assembly. Quando o assembly for referenciado em um projeto do Visual Studio, o arquivo .xml também será encontrado. Para obter mais informações sobre como gerar comentários de código, consulte [Fornecendo comentários de código](#). A menos que você compile com `<TargetType:Module>`, `file` conterá `<assembly>` e `</assembly>` e marcações especificando o nome do arquivo com o

manifesto do assembly para o arquivo de saída. Por exemplo, consulte [Como usar as funcionalidades da documentação XML](#).

#### ⓘ Observação

A opção **DocumentationFile** se aplica a todos os arquivos no projeto. Para desabilitar avisos relacionados aos comentários de documentação para um arquivo ou uma seção específica do código, use `#pragma warning`.

Essa opção pode ser usada em qualquer projeto no estilo SDK do .NET. Para obter mais informações, consulte [propriedade DocumentationFile](#).

## OutputAssembly

A opção **OutputAssembly** especifica o nome do arquivo de saída. O caminho de saída especifica a pasta em que a saída do compilador é colocada.

XML

```
<OutputAssembly>folder</OutputAssembly>
```

Especifique o nome completo e a extensão do arquivo que você deseja criar. Se você não especificar o nome do arquivo de saída, o MSBuild usará o nome do projeto para especificar o nome do assembly de saída. Projetos de estilo antigo usam as seguintes regras:

- Um .exe extrairá seu nome do arquivo do código-fonte que contém o método `Main` ou instruções de alto nível.
- Um arquivo .dll ou .netmodule extrairá seu nome do primeiro arquivo de código-fonte.

Os módulos produzidos como parte de uma compilação se tornam arquivos associados a qualquer assembly também produzido na compilação. Use [ildasm.exe](#) para exibir o manifesto do assembly para ver os arquivos associados.

A opção do compilador **OutputAssembly** é necessária para que um exe seja o destino de um [assembly amigável](#).

## PlatformTarget

Especifica qual versão do CLR pode executar o assembly.

```
<PlatformTarget>anycpu</PlatformTarget>
```

- O **anycpu** (padrão) compila seu assembly para que ele seja executado em qualquer plataforma. Seu aplicativo será executado como um processo de 64 bits sempre que possível e realizará fallback para 32 bits quando apenas esse modo estiver disponível.
- **anycpu32bitpreferred** compila seu assembly para que ele seja executado em qualquer plataforma. Seu aplicativo é executado no modo 32 bits em sistemas que dão suporte para aplicativos de 32 bits e 64 bits. É possível especificar essa opção apenas para projetos que definem como destino o .NET Framework 4.5 ou posterior.
- **ARM** compila seu assembly para que ele seja executado em um computador que tem um processador ARM (Advanced RISC Machine).
- **ARM64** compila o assembly para execução pelo CLR de 64 bits em um computador que tem um processador ARM (Máquina RISC Avançada) que dá suporte ao conjunto de instruções A64.
- **x64** compila o assembly para ser executado pelo CLR de 64 bits em um computador que dá suporte ao conjunto de instruções AMD64 ou EM64T.
- **x86** compila o assembly para ser executado pelo CLR compatível com x86 de 32 bits.
- **Itanium** compila o assembly para ser executado pelo CLR de 64 bits em um computador com um processador Itanium.

Em um sistema operacional do Windows de 64 bits:

- Assemblies compilados com **x86** serão executados no CLR de 32 bits em execução no x86.
- Uma DLL compilada com o **anycpu** é executada no mesmo CLR que o processo no qual ela é carregada.
- Os executáveis compilados com **anycpu** são executados no CLR de 64 bits.
- Os executáveis compilados com **anycpu32bitpreferred** são executados no CLR de 32 bits.

A configuração **anycpu32bitpreferred** é válida apenas para arquivos .EXE (executáveis) e requer o .NET Framework 4.5 ou posterior. Para obter mais informações sobre o desenvolvimento de um aplicativo para ser executado em um sistema operacional Windows de 64 bits, consulte [Aplicativos de 64 bits](#).

Defina a opção **PlatformTarget** na página de propriedades da **Build** para seu projeto no Visual Studio.

O comportamento de `anycpu` tem algumas nuances adicionais no .NET Core e no .NET 5 e versões posteriores. Quando você definir `anycpu`, publique seu aplicativo e execute-o com o `dotnet.exe x86` ou o `dotnet.exe x64`. Para aplicativos autocontidos, a etapa `dotnet publish` empacota o executável para configurar RID.

## ProduceReferenceAssembly

A opção `ProduceReferenceAssembly` controla se o compilador produz assemblies de referência.

XML

```
<ProduceReferenceAssembly>true</ProduceReferenceAssembly>
```

Os assemblies de referência são um tipo especial de assembly que contém apenas a quantidade mínima de metadados necessários para representar a superfície de API pública da biblioteca. Elas incluem declarações para todos os membros que são significativas ao referenciar um assembly em ferramentas de build. Os assemblies de referência excluem todas as implementações e declarações de membros privados. Esses membros não têm impacto observável em seu contrato de API. Para obter mais informações, consulte [Assemblies de referência](#) no Guia do .NET.

As opções `ProduceReferenceAssembly` e `ProduceOnlyReferenceAssembly` são mutuamente exclusivas.

Geralmente, você não precisa trabalhar diretamente com arquivos assembly de referência. Por padrão, assemblies de referência são gerados em uma `ref` subpasta do caminho intermediário (ou seja, `obj/ref/`). Para gerá-los no diretório de saída (ou seja `bin/ref/`, ) configure `ProduceReferenceAssemblyInOutDir` para `true` no seu projeto.

O SDK do .NET 6.0.200 fez uma [alteração](#) que moveu assemblies de referência do diretório de saída para o diretório intermediário por padrão.

## TargetType

A opção do compilador `TargetType` pode ser especificada em uma das seguintes formas:

- `library`: para criar uma biblioteca de códigos. `library` é o valor padrão.
- `exe`: para criar um arquivo .exe.
- `module` para criar um módulo.

- **winexe** para criar um programa do Windows.
- **winmdobj** para criar um arquivo *.winmdobj* intermediário.
- **appcontainerexe** para criar um arquivo *.exe* para aplicativos Windows 8.x Store.

### ⓘ Observação

Para destinos .NET Framework, a menos que você especifique o **módulo**, essa opção faz com que um manifesto do assembly .NET Framework seja colocado em um arquivo de saída. Para obter mais informações, consulte **Assemblies no .NET e atributos comuns**.

XML

```
<TargetType>library</TargetType>
```

O compilador cria apenas um manifesto do assembly por compilação. As informações sobre todos os arquivos em uma compilação são colocados no manifesto do assembly. Ao gerar vários arquivos de saída na linha de comando, apenas um manifesto do assembly pode ser criado e ele deve ir para o primeiro arquivo de saída especificado na linha de comando.

Se você criar um assembly, pode indicar que todo ou parte do seu código está em conformidade com CLS com o atributo [CLSCompliantAttribute](#).

## biblioteca

A opção **library** faz com que o compilador crie uma DLL (biblioteca de vínculo dinâmico) em vez de um EXE (arquivo executável). A DLL será criada com a extensão *.dll*. A menos que seja especificado de outra forma com a opção [OutputAssembly](#), o nome do arquivo de saída usa o nome do primeiro arquivo de entrada. Ao criar um arquivo *.dll*, um método [Main](#) não é necessário.

## exe

A opção **exe** faz com que o compilador crie um aplicativo de console executável (EXE). O arquivo executável será criado com a extensão *.exe*. Use **winexe** para criar um executável de programa do Windows. A menos que seja especificado de outra forma com a opção [OutputAssembly](#), o nome do arquivo de saída usará o nome do arquivo de entrada que contém o método [Main](#) ou instruções de alto nível). Somente um método ponto de entrada é necessário nos arquivos de código-fonte que são compilados em um arquivo *.exe*. A opção do compilador [StartupObject](#) permite

especificar qual classe contém o método `Main`, nos casos em que o código tem mais de uma classe com um método `Main`.

## module

Essa opção faz com que o compilador não gere um manifesto do assembly. Por padrão, o arquivo de saída criado por meio da compilação com essa opção terá uma extensão `.netmodule`. Um arquivo que não tem um manifesto do assembly não pode ser carregado pelo tempo de execução do .NET. No entanto, esse arquivo pode ser incorporado no manifesto do assembly de um assembly com [AddModules](#). Se mais de um módulo for criado em uma única compilação, tipos [internos](#) em um módulo estarão disponíveis para outros módulos na compilação. Quando o código em um módulo referencia tipos `internal` em outro módulo, os dois módulos deverão ser incorporados em um manifesto do assembly, com [AddModules](#). Não há suporte para a criação de um módulo no ambiente de desenvolvimento do Visual Studio.

## winexe

A opção `winexe` faz com que o compilador crie um programa do Windows executável (EXE). O arquivo executável será criado com a extensão .exe. Um programa do Windows é aquele que fornece uma interface do usuário da biblioteca do .NET ou com as APIs do Windows. Use `exe` para criar um aplicativo de console. A menos que seja especificado de outra forma com a opção [OutputAssembly](#), o nome do arquivo de saída usará o nome do arquivo de entrada que contém o método `Main`. Somente um método `Main` é necessário nos arquivos de código-fonte que são compilados em um arquivo .exe. A opção [StartupObject](#) permite especificar qual classe contém o método `Main`, nos casos em que o código tem mais de uma classe com um método `Main`.

## winmdobj

Se você usar a opção `winmdobj`, o compilador criará um arquivo `.winmdobj` intermediário que pode ser convertido em um arquivo binário do Windows Runtime (`.winmd`). O arquivo `.winmd` pode, então, ser consumido por programas JavaScript e C++, bem como programas de linguagem gerenciada.

A configuração `winmdobj` indica para o compilador que um módulo intermediário é necessário. O arquivo `.winmdobj` pode, então, ser alimentado por meio da ferramenta de exportação [WinMDExp](#) para produzir um arquivo de metadados do Windows (`.winmd`). O arquivo `.winmd` contém o código da biblioteca original e os metadados do WinMD que são usados pelo JavaScript ou C++ e pelo Windows Runtime. A saída de um

arquivo que é compilado usando a opção do compilador `winmdobj` é usada apenas como entrada para a ferramenta de exportação WimMDExp. O arquivo `.winmdobj` em si não é referenciado diretamente. A menos que você use a opção [OutputAssembly](#), o nome do arquivo de saída usará o nome do primeiro arquivo de entrada. Um método [Main](#) não é necessário.

## appcontainerexe

Se você usar a opção do compilador `appcontainerexe`, o compilador criará um arquivo executável (`.exe`) do Windows que deverá ser executado em um contêiner de aplicativos. Essa opção é equivalente a [-target:winexe](#), mas foi projetada para aplicativos Windows 8.x Store.

Para exigir que o aplicativo seja executado em um contêiner de aplicativos, esta opção define um bit no arquivo [PE](#). Quando esse bit estiver definido, ocorrerá um erro se o método `CreateProcess` tentar inicializar o arquivo executável fora de um contêiner de aplicativos. A menos que você use a opção [OutputAssembly](#), o nome do arquivo de saída usará o nome do arquivo de entrada que contém o método [Main](#).

# Opções do compilador C# que especificam entradas

Artigo • 10/05/2023

As opções a seguir controlam entradas do compilador. A nova sintaxe MSBuild é mostrada em **Negrito**. A sintaxe `csc.exe` mais antiga é mostrada em `code style`.

- **References** / `-reference` ou `-references`: referenciar metadados do arquivo ou arquivos de assembly especificados.
- **AddModules** / `-addmodule`: adicionar um módulo (criado com `target:module` para este assembly).
- **EmbedInteropTypes** / `-link`: inserir metadados dos arquivos de assembly de interoperabilidade especificados.

## Referências

A opção **References** faz com que o compilador importe informações de tipo `public` no arquivo especificado para o projeto atual, permitindo que você refcrcie metadados dos arquivos do assembly especificado.

XML

```
<Reference Include="filename" />
```

`filename` é o nome de um arquivo que contém um manifesto do assembly. Para importar mais de um arquivo, inclua um elemento **Reference** separado para cada arquivo. Você pode definir um alias como um elemento filho do elemento **Reference**:

XML

```
<Reference Include="filename.dll">
  <Aliases>LS</Aliases>
</Reference>
```

No exemplo anterior, `LS` é o identificador C# válido que representa um namespace raiz que conterá todos os namespaces do assembly `filename.dll`. Os arquivos importados devem conter um manifesto. Use **AdditionalLibPaths** para especificar o diretório no qual uma ou mais das suas referências de assembly estão localizadas. O tópico **AdditionalLibPaths** também discute os diretórios nos quais o compilador pesquisa assemblies. Para que o compilador reconheça um tipo em um assembly e não um

módulo, ele precisa ser forçado a resolver o tipo, que você pode fazer definindo uma instância do tipo. Há outras maneiras de resolver nomes de tipo em um assembly para o compilador: por exemplo, se você herda de um tipo em um assembly, o nome do tipo será reconhecido pelo compilador. Às vezes, é necessário referenciar duas versões diferentes do mesmo componente de dentro de um assembly. Para fazer isso, use o elemento **Alias** no elemento **References** para cada arquivo, a fim de diferenciar entre os dois arquivos. Esse alias será usado como um qualificador do nome do componente e será resolvido para o componente em um dos arquivos.

#### ⓘ Observação

No Visual Studio, use o comando **Adicionar Referência**. Para obter mais informações, consulte [Como adicionar ou remover referências usando o Gerenciador de Referências](#).

## AddModules

Essa opção adiciona à compilação atual um módulo que foi criado com a opção `<TargetType>module</TargetType>`:

### XML

```
<AddModule Include=file1 />
<AddModule Include=file2 />
```

Em que `file1` e `file2` são arquivos de saída que contêm metadados. O arquivo não pode conter um manifesto do assembly. Para importar mais de um arquivo, separe os nomes de arquivo com vírgula ou ponto e vírgula. Todos os módulos adicionados com **AddModules** devem estar no mesmo diretório que o arquivo de saída em tempo de execução. Ou seja, é possível especificar um módulo em qualquer diretório em tempo de compilação, mas o módulo deve estar no diretório do aplicativo em tempo de execução. Se o módulo não estiver no diretório do aplicativo em tempo de execução, você obterá um [TypeLoadException](#). `file` não pode conter um assembly. Por exemplo, se o arquivo de saída tiver sido criado com a opção **TargetType** igual a **module**, os metadados dele poderão ser importados com **AddModules**.

Se o arquivo de saída tiver sido criado com uma opção **TargetType** diferente de **module**, os metadados dele não poderão ser importados com **AddModules**, mas poderão ser importados com a opção **References**.

# EmbedInteropTypes

Faz com que o compilador disponibilize as informações de tipo COM nos assemblies especificados para o projeto sendo compilado no momento.

XML

```
<References>
  <EmbedInteropTypes>file1;file2;file3</EmbedInteropTypes>
</References>
```

Em que `file1;file2;file3` é uma lista de nomes de arquivo de assembly delimitada por ponto-e-vírgula. Se o nome do arquivo contém um espaço, coloque o nome entre aspas. A opção **EmbedInteropTypes** permite que você implante um aplicativo que tenha informações de tipo inserido. O aplicativo pode usar tipos em um assembly de runtime que implementa as informações de tipo inseridas sem a necessidade de uma referência ao assembly de runtime. Se forem publicadas várias versões do assembly de runtime, o aplicativo que contém as informações de tipo inseridas poderá trabalhar com as várias versões sem precisar ser recompilado. Para obter um exemplo, consulte [Instruções passo a passo: Inserindo tipos de assemblies gerenciado](#).

O uso da opção **EmbedInteropTypes** é especialmente útil quando você está trabalhando com a interoperabilidade COM. Você pode inserir tipos COM para que seu aplicativo não precise mais de um PIA (assembly de interoperabilidade primário) no computador de destino. A opção **EmbedInteropTypes** instrui o compilador a inserir as informações de tipo COM do assembly de interoperabilidade referenciado no código compilado resultante. O tipo COM é identificado pelo valor CLSID (GUID). Como resultado, o aplicativo pode ser executado em um computador de destino que tem os mesmos tipos COM instalados com os mesmos valores CLSID. Os aplicativos que automatizam o Microsoft Office são um bom exemplo. Como aplicativos como o Office normalmente mantêm o mesmo valor CLSID entre diferentes versões, seu aplicativo pode usar os tipos COM referenciados contanto que o .NET Framework 4 ou posterior esteja instalado no computador de destino e seu aplicativo use métodos, propriedades ou eventos que estão incluídos nos tipos COM referenciados. A opção **EmbedInteropTypes** incorpora apenas interfaces, estruturas e delegados. Não há suporte para a incorporação de classes COM.

## ⓘ Observação

Quando você cria uma instância de um tipo COM inserido no seu código, você deve criar a instância usando a interface apropriada. Tentar criar uma instância de um tipo COM inserido usando o CoClass causa um erro.

Assim como a opção de compilador [References](#), a opção de compilador [EmbedInteropTypes](#) usa o arquivo de resposta Csc.rsp, que faz referência a assemblies .NET usados com frequência. Use a opção do compilador [NoConfig](#) se não quiser que o compilador use o arquivo Csc.rsp.

C#

```
// The following code causes an error if ISampleInterface is an embedded
// interop type.
ISampleInterface<SampleType> sample;
```

Os tipos que têm um parâmetro genérico cujo tipo é inserido de um assembly de interoperabilidade não poderão ser usados se o tipo for de um assembly externo. Essa restrição não se aplica a interfaces. Por exemplo, considere a interface [Range](#) que é definida no assembly [Microsoft.Office.Interop.Excel](#). Se uma biblioteca insere tipos de interoperabilidade do assembly [Microsoft.Office.Interop.Excel](#) e expõe um método que retorna um tipo genérico que tem um parâmetro cujo tipo é a interface [Range](#), esse método deve retornar uma interface genérica, como mostrado no exemplo de código a seguir.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
    // The following code causes an error when called by a client assembly.
    public List<Range> GetRange1()
    {
        return null;
    }

    // The following code is valid for calls from a client assembly.
    public IList<Range> GetRange2()
    {
        return null;
    }
}
```

No exemplo a seguir, o código do cliente pode chamar o método que retorna a interface genérica [IList](#) sem erros.

C#

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();

        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

# Opções do Compilador do C# para relatar erros e avisos

Artigo • 10/05/2023

As opções a seguir controlam a maneira como o compilador relata erros e avisos. A nova sintaxe MSBuild é mostrada em **Negrito**. A sintaxe `csc.exe` mais antiga é mostrada em `code style`.

- **WarningLevel** / `-warn`: define o nível de aviso.
- **AnalysisLevel**: define o nível de aviso opcional.
- **TreatWarningsAsErrors** / `-warnaserror`: trata todos os avisos como erros
- **WarningsAsErrors** / `-warnaserror`: tratar um ou mais avisos como erros
- **WarningsNotAsErrors** / `-warnnotaserror`: não trata um ou mais avisos como erros
- **NoWarn** / `-nowarn`: define uma lista de avisos desabilitados.
- **CodeAnalysisRuleSet** / `-ruleset`: especifica um arquivo de conjunto de regras que desabilite o diagnóstico específico.
- **ErrorLog** / `-errorlog`: especifica um arquivo para registrar todos os diagnósticos do compilador e do analisador.
- **ReportAnalyzer** / `-reportanalyzer`: relata informações adicionais do analisador, como o tempo de execução.

## WarningLevel

A opção **WarningLevel** especifica o nível de aviso a ser exibido pelo compilador.

XML

```
<WarningLevel>3</WarningLevel>
```

O valor do elemento é o nível de aviso que você deseja que seja exibido para a compilação: números mais baixos mostram apenas avisos de gravidade alta. Números mais altos mostram mais avisos. O valor deve ser zero ou um número inteiro positivo:

Nível de aviso	Significado
0	Desativa a emissão de todas as mensagens de aviso.
1	Exibe mensagens de aviso graves.

Nível de aviso	Significado
2	Exibe os avisos do nível 1 e mais alguns avisos menos graves, como avisos sobre membros de classe ocultos.
3	Exibe os avisos do nível 2 e mais alguns avisos menos graves, como avisos sobre expressões que sempre resultam em <code>true</code> ou <code>false</code> .
4 (o padrão)	Exibe todos os avisos do nível 3 e também avisos informativos.

### ⚠ Aviso

A linha de comando do compilador aceita valores maiores que 4, para habilitar os avisos do ciclo de aviso. No entanto, o SDK do .NET define o *WarningLevel* para corresponder ao *AnalysisLevel* no arquivo de projeto.

Para obter informações sobre um erro ou aviso, você pode procurar o código de erro no [Índice da Ajuda](#). Para outras maneiras de se obter informações sobre um erro ou aviso, consulte [Erros do compilador do C#](#). Use [TreatWarningsAsErrors](#) para tratar todos os avisos como erros. Use [DisabledWarnings](#) para desabilitar determinados avisos.

## Nível de análise

Nível de análise	Significado
5	Exibe todos os <a href="#">avisos opcionais do ciclo de aviso 5</a> .
6	Exibe todos os <a href="#">avisos opcionais do ciclo de aviso 6</a> .
7	Exibe todos os <a href="#">avisos opcionais do ciclo de aviso 7</a> .
mais recente (padrão)	Exibe todos os avisos informativos até e incluindo a versão atual.
preview	Exibe todos os avisos informativos até e incluindo a versão prévia mais recente.
nenhum	Desativa todos os avisos informativos.

Para obter mais informações sobre avisos opcionais, confira [Ciclos de aviso](#).

Para obter informações sobre um erro ou aviso, você pode procurar o código de erro no [Índice da Ajuda](#). Para outras maneiras de se obter informações sobre um erro ou aviso,

consulte [Erros do compilador do C#](#). Use [TreatWarningsAsErrors](#) para tratar todos os avisos como erros. Use [NoWarn](#) para desabilitar determinados avisos.

## TreatWarningsAsErrors

A opção [TreatWarningsAsErrors](#) trata todos os avisos como erros. Você também pode usar o [TreatWarningsAsErrors](#) para definir apenas alguns avisos como erros. Se você ativar [TreatWarningsAsErrors](#), pode usar [WarningsNotAsErrors](#) para listar os avisos que não devem ser tratados como erros.

XML

```
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

Todas as mensagens de aviso são relatadas como erros. O processo de compilação é interrompido (nenhum arquivo de saída é criado). Por padrão, [TreatWarningsAsErrors](#) não foi implementado, o que significa que os avisos não impedem a geração de um arquivo de saída. Opcionalmente, se você deseja que apenas avisos específicos sejam tratados como erros, você pode especificar uma lista separada por vírgulas de números de aviso para serem tratados como erros. O conjunto de todos os avisos de nulidade pode ser especificado com a abreviação [Anulável](#). Use [WarningLevel](#) para especificar o nível de avisos que você deseja que o compilador exiba. Use [NoWarn](#) para desabilitar determinados avisos.

### ⓘ Importante

Há duas diferenças sutis entre usar o elemento `<TreatWarningsAsErrors>` no arquivo `csproj` e usar a opção de linha de comando do MSBuild `warnaserror`.

*TreatWarningsAsErrors*: afeta apenas o compilador do C#, e não outras tarefas do MSBuild no arquivo `csproj`. A opção de linha de comando `warnaserror` afeta todas as tarefas. Em segundo lugar, o compilador não produz a saída nos avisos quando o *TreatWarningsAsErrors* é usado. O compilador produz a saída quando a opção de linha de comando `warnaserror` é usada.

## WarningsAsErrors e WarningsNotAsErrors

As opções [WarningsAsErrors](#) e [WarningsNotAsErrors](#) substituem a opção [TreatWarningsAsErrors](#) para obter uma lista de avisos. Essa opção pode ser usada com todos os avisos CS. O prefixo "CS" é opcional. Você pode usar o número ou "CS"

seguido do erro ou número de aviso. Para outros elementos que afetam avisos, confira as [Propriedades comuns do MSBuild](#).

Habilite os avisos 0219 e 0168 como erros:

XML

```
<WarningsAsErrors>0219,CS0168</WarningsAsErrors>
```

Desabilite os mesmos avisos como erros:

XML

```
<WarningsNotAsErrors>0219,CS0168</WarningsNotAsErrors>
```

Você usará o **WarningsAsErrors** para configurar um conjunto de avisos como erros. Use o **WarningsNotAsErrors** para configurar um conjunto de avisos que não devem ser erros, ao definir todos os avisos como erros.

## NoWarn

A opção **NoWarn** permite suprimir a exibição de um ou mais avisos pelo compilador. Separe vários números de aviso com uma vírgula.

XML

```
<NoWarn>number1, number2</NoWarn>
```

`number1, number2` números de aviso que você deseja que o compilador suprima. Você especificará a parte numérica do identificador de aviso. Por exemplo, se quiser suprimir `CS0028`, você pode especificar `<NoWarn>28</NoWarn>`. O compilador ignora silenciosamente números de aviso passados para **NoWarn** que eram válidos nas versões anteriores, mas que foram removidos. Por exemplo, `CS0679` era válido no compilador no Visual Studio .NET 2002, mas foi removido posteriormente.

Os avisos a seguir não podem ser suprimidos pela opção **NoWarn**:

- Aviso do compilador (nível 1) CS2002
- Aviso do compilador (nível 1) CS2023
- Aviso do compilador (nível 1) CS2029

## CodeAnalysisRuleSet

Especifica um arquivo de conjunto de regras que configura diagnósticos específicos.

XML

```
<CodeAnalysisRuleSet>MyConfiguration.ruleset</CodeAnalysisRuleSet>
```

Onde `MyConfiguration.ruleset` é o caminho para o arquivo de conjunto de regras. Para obter mais informações sobre como usar conjuntos de regras, confira o artigo na [documentação do Visual Studio sobre conjuntos de regras](#).

## ErrorLog

Especifique um arquivo para registrar todos os diagnósticos do compilador e do analisador.

XML

```
<ErrorLog>compiler-diagnostics.sarif</ErrorLog>
```

A opção `ErrorLog` faz com que o compilador gere um [log SARIF \(Static Analysis Results Interchange Format\)](#). Os logs SARIF normalmente são lidos por ferramentas que analisam os resultados do diagnóstico do compilador e do analisador.

Você pode especificar o formato SARIF usando o argumento `version` para o elemento `ErrorLog`:

XML

```
<ErrorLog>logVersion21.json,version=2.1</ErrorLog>
```

O separador pode ser uma vírgula (,) ou um ponto e vírgula (;). Os valores válidos para versão são: "1", "2" e "2.1". O padrão é "1". "2" e "2.1" indicam SARIF versão 2.1.0.

## ReportAnalyzer

Reporte informações adicionais do analisador, como o tempo de execução.

XML

```
<ReportAnalyzer>true</ReportAnalyzer>
```

A opção **ReportAnalyzer** faz com que o compilador emita informações de log do MSBuild adicionais, que detalham as características de desempenho dos analisadores na compilação. Normalmente, é usada pelos autores do analisador como parte da validação do analisador.

 **Importante**

As informações de log adicionais geradas por esse sinalizador só são geradas quando a opção de linha de comando `-verbosity:detailed` é usada. Confira o artigo de [opções](#) na documentação do MSBuild, para obter mais informações.

# Opções do compilador C# que controlam a geração de código

Artigo • 16/03/2023

As opções a seguir controlam a geração de código pelo compilador. A nova sintaxe MSBuild é mostrada em **Negrito**. A sintaxe `csc.exe` mais antiga é mostrada em `code style`.

- **DebugType** / `-debug`: emitir (ou não emitir) informações de depuração.
- **Optimize** / `-optimize`: habilitar otimizações.
- **Deterministic** / `-deterministic`: produzir saída equivalente byte por byte da mesma fonte de entrada.
- **ProduceOnlyReferenceAssembly** / `-refonly`: produzir um assembly de referência, em vez de um assembly completo, como a saída primária.

## DebugType

A opção **DebugType** faz o compilador gerar informações de depuração e colocá-las nos arquivos de saída. As informações de depuração são adicionadas por padrão.

XML

```
<DebugType>pdbonly</DebugType>
```

Para todas as versões do compilador que começam com o C# 6.0, não há diferença entre *pdbonly* e *full*. Escolha *pdbonly*. Para alterar o local do arquivo *.pdb*, consulte [PdbFile](#).

Os seguintes valores são válidos:

Valor	Significado
<code>full</code>	Emita informações de depuração no arquivo <i>.pdb</i> usando o formato padrão para a plataforma atual: <b>Windows</b> : um arquivo pdb do Windows. <b>Linux/macOS</b> : um arquivo <a href="#">PDB portátil</a> .
<code>pdbonly</code>	Mesmo que <code>full</code> . Confira a observação abaixo para obter mais informações.
<code>portable</code>	Emita informações de depuração no arquivo <i>.pdb</i> usando o formato <a href="#">PDB portátil multiplataforma</a> .

Valor	Significado
embedded	Emita informações de depuração no <code>.dll/.exe</code> em si (o arquivo <code>.pdb</code> não é produzido) usando o formato <a href="#">PDB Portátil</a> .

### ⓘ Importante

As informações a seguir se aplicam somente a compiladores mais antigos que o C# 6.0. O valor desse elemento pode ser `full` ou `pdbonly`. O argumento `full`, que será aplicado se `pdbonly` não for especificado, permite anexar um depurador ao programa em execução. Especificar `pdbonly` habilita a depuração do código-fonte quando o programa é iniciado no depurador, mas exibirá somente o assembler quando o programa em execução for anexado ao depurador. Use essa opção para criar builds de depuração. Caso `Full` seja usado, lembre-se de que isso influenciará a velocidade e o tamanho do código otimizado JIT e haverá um pequeno impacto na qualidade do código com `full`. Recomenda-se `pdbonly` ou nenhum PDB para gerar código de versão. Uma diferença entre `pdbonly` e `full` é que, com `full`, o compilador emite uma [DebuggableAttribute](#), que é usada para avisar ao compilador JIT que as informações de depuração estão disponíveis. Portanto, haverá um erro se o código contiver [DebuggableAttribute](#) definido como `false`, caso `full` seja usado. Para obter informações sobre como configurar o desempenho de depuração de um aplicativo, consulte [Facilitando a Depuração de uma Imagem](#).

## Otimizar

A opção **Optimize** habilita ou desabilita otimizações executadas pelo compilador para tornar o arquivo de saída menor, mais rápido e mais eficiente. A opção `Optimize` está habilitada por padrão para uma configuração de build *Release*. Ele está desativado por padrão para uma configuração de build *Debug*.

XML

```
<Optimize>true</Optimize>
```

Defina a opção **Optimize** na página de propriedades da **Build** para seu projeto no Visual Studio.

**Optimize** também informa o Common Language Runtime para otimizar o código em tempo de execução. Por padrão, as otimizações estão desabilitadas. Especifique

**Optimize+** para habilitar otimizações. Ao criar um módulo a ser usado por um assembly, use as mesmas configurações **Optimize** usadas pelo assembly. É possível combinar as opções **Optimize** e **Debug**.

## Determinística

Faz com que o compilador produza um assembly cuja saída byte a byte é idêntica entre compilações para entradas idênticas.

XML

```
<Deterministic>true</Deterministic>
```

Por padrão, a saída do compilador de um determinado conjunto de entradas é exclusiva, pois o compilador adiciona um carimbo de data/hora e um MVID (um [Module.ModuleVersionId](#)). Basicamente, é um GUID que identifica exclusivamente o módulo e a versão.), que é gerado com base em números aleatórios. Use a opção `<Deterministic>` para produzir um *assembly determinística*, cujo conteúdo binário seja idêntico entre compilações, desde que a entrada permaneça a mesma. Nesse build, os campos de carimbo de data/hora e MVID serão substituídos por valores derivados de um hash de todas as entradas de compilação. O compilador considera as seguintes entradas que afetam o determinismo:

- A sequência de parâmetros de linha de comando.
- O conteúdo do arquivo de resposta .rsp do compilador.
- A versão precisa do compilador usado e seus assemblies referenciados.
- O caminho do diretório atual.
- O conteúdo binário de todos os arquivos passados explicitamente para o compilador direta ou indiretamente, incluindo:
  - Arquivos de origem
  - Assemblies referenciados
  - Módulos referenciados
  - Recursos
  - O arquivo de chave de nome forte
  - @ arquivos de resposta
  - Analisadores
  - Conjuntos de regras
  - Outros arquivos que podem ser usados por analisadores
- A cultura atual (para o idioma no qual as mensagens de diagnóstico e exceção são produzidas).

- A codificação padrão (ou a página de código atual) se a codificação não for especificada.
- A existência, a inexistência e o conteúdo dos arquivos em caminhos de pesquisa do compilador (especificados, por exemplo, por `-lib` ou `-recurse`).
- A plataforma CLR (Common Language Runtime) na qual o compilador é executado.
- O valor de `%LIBPATH%`, que pode afetar o carregamento de dependência do analisador.

A compilação determinística pode ser usada para estabelecer se um binário é compilado de uma fonte confiável. A saída determinística pode ser útil quando a origem está disponível publicamente. Ele também pode determinar se as etapas de build que dependem de alterações no binário usado no processo de build.

## ProduceOnlyReferenceAssembly

A opção **ProduceOnlyReferenceAssembly** indica que um assembly de referência deve ser gerado em vez de um assembly de implementação, como a saída primária. O parâmetro **ProduceOnlyReferenceAssembly** silenciosamente desabilita a geração de PDBs, uma vez que assemblies de referência não podem ser executados.

XML

```
<ProduceOnlyReferenceAssembly>true</ProduceOnlyReferenceAssembly>
```

Assemblies de referência são um tipo especial de assembly. Os assemblies contêm apenas a quantidade mínima de metadados necessários para representar a superfície de API pública da biblioteca. Eles incluem declarações para todos os membros que são significativas ao referenciar um assembly em ferramentas de build, mas excluem todas as implementações de membros e declarações de membros privados que não têm nenhum impacto observável em seu contrato de API. Para obter mais informações, consulte [Assemblies de referência](#).

As opções **ProduceOnlyReferenceAssembly** e [ProduceReferenceAssembly](#) são mutuamente exclusivas.

# Opções do compilador C# para segurança

Artigo • 10/05/2023

As opções a seguir controlam as opções de segurança do compilador. A nova sintaxe MSBuild é mostrada em **Negrito**. A sintaxe `csc.exe` mais antiga é mostrada em `code style`.

- **PublicSign** / `-publicsign`: assine publicamente o assembly.
- **DelaySign** / `-delaysign`: assine com atraso o assembly usando apenas a parte pública da chave de nome forte.
- **KeyFile** / `-keyfile`: especifique um arquivo de chave de nome forte.
- **KeyContainer** / `-keycontainer`: especifique um contêiner de chave de nome forte.
- **HighEntropyVA** / `-highentropyva`: habilitar a ASLR (Randomização de Layout de Espaço de Endereço) de alta entropia

## PublicSign

Essa opção faz com que o compilador aplique uma chave pública, mas, na verdade, não assina o assembly. A opção **PublicSign** também define um bit no assembly que informa ao runtime que o arquivo realmente está assinado.

XML

```
<PublicSign>true</PublicSign>
```

A opção **PublicSign** requer o uso das opções **KeyFile** ou **KeyContainer**. As opções **KeyFile** e **KeyContainer** especificam a chave pública. As opções **PublicSign** e **DelaySign** são mutuamente exclusivas. Às vezes chamado de "sinal falso" ou "sinal de OSS", a assinatura pública inclui a chave pública em um assembly de saída e define o sinalizador "assinado". A assinatura pública não assina o assembly com uma chave privada. Os desenvolvedores usam o sinal público para projetos de código aberto. As pessoas criam assemblies compatíveis com os assemblies "totalmente assinados" lançados quando não têm acesso à chave privada usada para assinar os assemblies. Como poucos consumidores precisam realmente verificar se o assembly está totalmente assinado, esses assemblies criados publicamente podem ser usados em quase todos os cenários nos quais o assembly totalmente assinado seria usado.

# DelaySign

Essa opção faz com que o compilador reserve espaço no arquivo de saída para que uma assinatura digital possa ser adicionada mais tarde.

XML

```
<DelaySign>true</DelaySign>
```

Use **DelaySign**- se você quiser um assembly totalmente assinado. Use **DelaySign** se você quiser apenas colocar a chave pública no assembly. A opção **DelaySign** não tem nenhum efeito a menos que seja usada com [KeyFile](#) ou [KeyContainer](#). As opções [KeyContainer](#) e [PublicSign](#) são mutuamente exclusivas. Quando você solicita um assembly totalmente assinado, o compilador usa o hash no arquivo que contém o manifesto (metadados de assembly) e sinaliza esse hash com a chave particular. Essa operação cria uma assinatura digital que é armazenada no arquivo que contém o manifesto. Quando um assembly é assinado com atraso, o compilador não computa e nem armazena a assinatura. Em vez disso, o compilador reserva espaço no arquivo para que a assinatura possa ser adicionada posteriormente.

Usar **DelaySign** permite que um testador coloque o assembly no cache global. Após o teste, é possível assinar completamente o assembly, colocando a chave particular no assembly com o utilitário [Assembly Linker](#). Para obter mais informações, consulte [Criando e usando assemblies de nomes fortes](#) e [Atraso na Assinatura de um Assembly](#).

# KeyFile

Especifica o nome de arquivo que contém a chave de criptografia.

XML

```
<KeyFile>filename</KeyFile>
```

`file` é o nome do arquivo que contém a chave de nome forte. Quando esta opção é usada, o compilador insere a chave pública da linha especificada no manifesto do assembly e, em seguida, assina o assembly definitivo com a chave privada. Para gerar um arquivo de chave, digite `sn -k file` na linha de comando. Se você compilar com [-target:module](#), o nome do arquivo de chave será mantido no módulo e incorporado ao assembly criado quando você compilar um assembly com [AddModules](#). Também é possível passar suas informações de criptografia para o compilador com [KeyContainer](#). Use [DelaySign](#) se quiser um assembly parcialmente assinado. Caso [KeyFile](#) e

**KeyContainer** sejam especificadas na mesma compilação, o compilador primeiro tentará o contêiner de chaves. Se isso ocorrer, o assembly será assinado com as informações no contêiner de chaves. Se o compilador não localizar o contêiner de chaves, ele tentará o arquivo especificado com [KeyFile](#). Se isso for bem-sucedido, o assembly será assinado com as informações no arquivo de chaves e as informações de chave serão instaladas no contêiner de chave. Na próxima compilação, o contêiner de chaves será válido. Um arquivo de chave pode conter apenas a chave pública. Para obter mais informações, consulte [Criando e usando assemblies de nomes fortes](#) e [Atraso na Assinatura de um Assembly](#).

## KeyContainer

Especifica o nome do contêiner da chave de criptografia.

XML

```
<KeyContainer>container</KeyContainer>
```

`container` é o nome do contêiner de chaves de nome forte. Quando a opção **KeyContainer** for usada, o compilador criará um componente compartilhável. O compilador insere uma chave pública do contêiner especificado no manifesto do assembly e assina o assembly final com a chave privada. Para gerar um arquivo de chave, digite `sn -k file` na linha de comando. `sn -i` instala o par de chaves no contêiner. Não há suporte para essa opção quando o compilador é executado no CoreCLR. Para assinar um assembly ao compilar no CoreCLR, use a opção [KeyFile](#). Se você compilar com [TargetType](#), o nome do arquivo de chave será mantido no módulo e incorporado no assembly quando você compilar este módulo em um assembly com [AddModules](#). Também é possível especificar essa opção como um atributo personalizado ([System.Reflection.AssemblyKeyNameAttribute](#)) no código-fonte de qualquer módulo MSIL (Microsoft Intermediate Language). Também é possível passar suas informações de criptografia para o compilador com [KeyFile](#). Use [DelaySign](#) para adicionar a chave pública ao manifesto do assembly, mas assinando o assembly até que ele seja testado. Para obter mais informações, consulte [Criando e usando assemblies de nomes fortes](#) e [Atraso na Assinatura de um Assembly](#).

## HighEntropyVA

A opção do compilador **HighEntropyVA** informa ao kernel do Windows se um determinado executável é compatível com ASLR (Address Space Layout Randomization) de alta entropia.

```
<HighEntropyVA>true</HighEntropyVA>
```

Essa opção especifica que um executável de 64 bits ou um executável marcado pela opção do compilador **PlatformTarget** dá suporte a um espaço de endereço virtual de alta entropia. A opção é habilitada por padrão para todas as versões do .NET Standard e do .NET Core e versões do .NET Framework começando com .NET Framework 4.5.

A opção **HighEntropyVA** permite que as versões compatíveis do kernel do Windows usem níveis mais altos de entropia ao randomizar o layout do espaço de endereço de um processo como parte da ASLR. O uso de níveis mais altos de entropia significa que um número maior de endereços pode ser alocado para regiões de memória como pilhas e heaps. Como resultado, fica mais difícil adivinhar a localização de uma região específica da memória. Quando a opção do compilador **HighEntropyVA** é especificada, o executável de destino e todos os módulos dos quais ele depende devem ser capazes de manipular valores de ponteiro que sejam maiores que 4 GB (gigabytes) quando eles estiverem em execução como um processo de 64 bits.

# Opções do compilador C# que especificam recursos

Artigo • 09/05/2023

As opções a seguir controlam como o compilador C# cria ou importa recursos do Win32. A nova sintaxe MSBuild é mostrada em **Negrito**. A sintaxe `csc.exe` mais antiga é mostrada em `code style`.

- **Win32Resource** / `-win32res`: especifique um arquivo de recurso Win32 (.res).
- **Win32Icon** / `-win32icon`: referece metadados do arquivo ou arquivos do assembly especificados.
- **Win32Manifest** / `-win32manifest`: especifique um arquivo de manifesto Win32 (.xml).
- **NoWin32Manifest** / `-nowin32manifest`: não inclua o manifesto padrão do Win32.
- **Recursos** / `-resource`: insira o recurso especificado (forma curta: /res).
- **LinkResources** / `-linkresources`: vincule o recurso especificado a esse assembly.

## Win32Resource

A opção **Win32Resource** insere um recurso do Win32 no arquivo de saída.

XML

```
<Win32Resource>filename</Win32Resource>
```

`filename` é o arquivo de recurso que você deseja adicionar ao seu arquivo de saída. Um recurso do Win32 pode conter informações de versão ou de bitmap (ícone) que ajudariam a identificar seu aplicativo no Explorador de Arquivos. Se você não especificar essa opção, o compilador gerará informações de versão com base na versão do assembly.

## Win32Icon

A opção **Win32Icon** insere um arquivo .ico no arquivo de saída, que fornece ao arquivo de saída a aparência desejada no Explorador de Arquivos.

XML

```
<Win32Icon>filename</Win32Icon>
```

`filename` é o arquivo `.ico` que você deseja adicionar ao seu arquivo de saída. Um arquivo `.ico` pode ser criado com o [Compilador de Recurso](#). O Compilador de Recurso é invocado quando você compila um programa do Visual C++; um arquivo `.ico` é criado com base no arquivo `.rc`.

## Win32Manifest

Use a opção **Win32Manifest** para especificar um arquivo de manifesto do aplicativo Win32 definido pelo usuário para ser inserido em um arquivo PE do projeto.

### XML

```
<Win32Manifest>filename</Win32Manifest>
```

`filename` é o nome e o local do arquivo de manifesto personalizado. Por padrão, o compilador C# insere um manifesto de aplicativo que especifica um nível de execução solicitado de "asInvoker". Ele cria o manifesto na mesma pasta na qual o executável é criado. Se você quiser fornecer um manifesto personalizado, por exemplo, para especificar um nível de execução solicitado de "highestAvailable" ou "requireAdministrator", use esta opção para especificar o nome do arquivo.

### ⓘ Observação

Essa opção e a opção **Win32Resources** são mutuamente exclusivas. Se você tentar usar ambas as opções na mesma linha de comando, você obterá um erro de build.

Um aplicativo que não tem nenhum manifesto do aplicativo que especifica um nível de execução solicitado estará sujeito à virtualização de arquivos e registro sob o recurso Controle de Conta de Usuário no Windows. Para obter mais informações, consulte [Controle de Conta de Usuário](#).

Seu aplicativo estará sujeito à virtualização se alguma dessas condições for verdadeira:

- Você usa a opção **NoWin32Manifest** e não fornece um manifesto em uma etapa de build posterior ou como parte de um arquivo de Recurso (.res) do Windows usando a opção **Win32Resource**.
- Você fornece um manifesto personalizado que não especifica um nível de execução solicitado.

O Visual Studio cria um arquivo `.manifest` padrão e o armazena nos diretórios de depuração e liberação juntamente com o arquivo executável. Você pode adicionar um manifesto personalizado criando um em qualquer editor de texto e, em seguida, adicionando o arquivo ao projeto. Como alternativa, você pode clicar com o botão direito do mouse no ícone Projeto no **Gerenciador de Soluções**, clicar em **Adicionar Novo Item** e selecionar **Arquivo de Manifesto do Aplicativo**. Depois de adicionar o arquivo de manifesto novo ou existente, ele aparecerá na lista suspensa **Manifesto**. Para obter mais informações, consulte [Página Aplicativo, Designer de Projeto \(C#\)](#).

Você pode fornecer o manifesto do aplicativo como uma etapa de pós-build personalizada ou como parte de um arquivo de recurso Win32 usando a opção **NoWin32Manifest**. Use essa mesma opção se quiser que o aplicativo seja sujeito à virtualização de arquivo ou Registro no Windows Vista.

## NoWin32Manifest

Use a opção **NoWin32Manifest** para instruir o compilador a não inserir nenhum manifesto do aplicativo no arquivo executável.

XML

```
<NoWin32Manifest />
```

Quando essa opção for usada, o aplicativo estará sujeito à virtualização no Windows Vista, a menos que você forneça um manifesto do aplicativo em um arquivo de recurso Win32 ou durante uma etapa de build posterior.

No Visual Studio, defina essa opção na página **Propriedade do Aplicativo** selecionando a opção **Criar aplicativo sem um manifesto** na lista suspensa **Manifesto**. Para obter mais informações, consulte [Página Aplicativo, Designer de Projeto \(C#\)](#).

## Recursos

Insere o recurso especificado no arquivo de saída.

XML

```
<Resources Include="filename">
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</Resources>
```

`filename` é o arquivo de recurso do .NET Framework que você deseja inserir no arquivo de saída. `identifier` (opcional) é nome lógico do recurso; o nome usado para carregar o recurso. O padrão é o nome do arquivo. `accessibility-modifier` é a acessibilidade do recurso: público ou privado. O padrão é público. Por padrão, recursos são públicos no assembly quando são criados usando o compilador C#. Para tornar os recursos privados, especifique `private` como o modificador de acessibilidade. Não é permitida nenhuma outra acessibilidade diferente de `public` ou `private`. Se `filename` for um arquivo de recurso do .NET Framework criado, por exemplo, por [Resgen.exe](#) ou no ambiente de desenvolvimento, ele poderá ser acessado com membros no namespace [System.Resources](#). Para obter mais informações, consulte [System.Resources.ResourceManager](#). Para todos os outros recursos, use os métodos `GetManifestResource` na classe [Assembly](#) para acessar o recurso em tempo de execução. A ordem dos recursos no arquivo de saída é determinada com base na ordem especificada no arquivo de projeto.

## LinkResources

Cria um link para um recurso do .NET Framework no arquivo de saída. O arquivo de recurso não é adicionado ao arquivo de saída. **LinkResources** é diferente da opção **Resource**, que insere um arquivo de recurso no arquivo de saída.

XML

```
<LinkResources Include=filename>
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</LinkResources>
```

`filename` é o arquivo de recurso do .NET Framework ao qual você deseja vincular a partir do assembly. `identifier` (opcional) é o nome lógico do recurso; o nome usado para carregar o recurso. O padrão é o nome do arquivo. `accessibility-modifier` é a acessibilidade do recurso: público ou privado. O padrão é público. Por padrão, os recursos vinculados são públicos no assembly quando são criados usando o compilador C#. Para tornar os recursos privados, especifique `private` como o modificador de acessibilidade. Não é permitido nenhum outro modificador diferente de `public` ou de `private`. Se `filename` for um arquivo de recurso do .NET Framework criado, por exemplo, por [Resgen.exe](#) ou no ambiente de desenvolvimento, ele poderá ser acessado com membros no namespace [System.Resources](#). Para obter mais informações, consulte [System.Resources.ResourceManager](#). Para todos os outros recursos, use os métodos `GetManifestResource` na classe [Assembly](#) para acessar o recurso em tempo de execução.

O arquivo especificado em `filename` pode ter qualquer formato. Por exemplo, crie uma parte DLL nativa do assembly de maneira que possa ser instalada no cache de assembly global e acessado no código gerenciado no assembly. É possível fazer a mesma coisa no Assembly Linker. Para obter mais informações, consulte [Al.exe \(Assembly Linker\)](#) e [Trabalhando com assemblies e o cache de assembly global](#).

# Opções diversas do compilador C#

Artigo • 02/06/2023

As opções a seguir controlam o comportamento diverso do compilador. A nova sintaxe MSBuild é mostrada em **Negrito**. A sintaxe de linha de comando `csc.exe` mais antiga é mostrada em `code style`.

- **ResponseFiles** / `@CustomOpts.RSP`: ler o arquivo de resposta especificado para conhecer mais opções.
- **NoLogo** / `-nologo` : Suprimir a mensagem de direitos autorais do compilador.
- **NoConfig** / `-noconfig`: não incluir automaticamente o arquivo `CSC.RSP`.

## ResponseFiles

A opção **ResponseFiles** possibilita especificar um arquivo que contém opções do compilador e arquivos de código-fonte a serem compilados.

XML

```
<ResponseFiles>response_file</ResponseFiles>
```

O `response_file` especifica o arquivo que lista as opções do compilador ou os arquivos de código-fonte a serem compilados. As opções do compilador e os arquivos de código-fonte serão processados pelo compilador se eles tiverem sido especificados na linha de comando. Para especificar mais de um arquivo de resposta em uma compilação, especifique várias opções de arquivo de resposta. Em um arquivo de resposta, várias opções de compilador e de arquivos de código-fonte podem ser exibidas em uma linha. Uma única especificação de opção do compilador deve ser exibida em uma linha (não é possível abranger várias linhas). Os arquivos de resposta podem ter comentários que começam com o símbolo `#`. Especificar opções do compilador de dentro de um arquivo de resposta é como emitir esses comandos na linha de comando. O compilador processa as opções de comando como elas são lidas. Os argumentos da linha de comando podem substituir opções listadas anteriormente em arquivos de resposta. Por outro lado, opções em um arquivo de resposta substituirão as opções listadas anteriormente na linha de comando ou em outros arquivos de resposta. O C# fornece o arquivo `csc.rsp`, localizado no mesmo diretório que o arquivo `csc.exe`. Para obter mais informações sobre o formato do arquivo de resposta, consulte [NoConfig](#). Essa opção do compilador não pode ser definida no ambiente de desenvolvimento do Visual Studio nem pode ser alterada por meio de programação. A seguir, há algumas linhas de um exemplo de arquivo de resposta:

## Console

```
# build the first output file  
-target:exe -out:MyExe.exe source1.cs source2.cs
```

## NoLogo

A opção **NoLogo** inibe a exibição da faixa de conexão quando o compilador é iniciado e inibe a exibição de mensagens informativas durante a compilação.

### XML

```
<NoLogo>true</NoLogo>
```

## NoConfig

A opção **NoConfig** instruirá o compilador a não compilar com o arquivo *csc.rsp*.

### XML

```
<NoConfig>true</NoConfig>
```

O arquivo *csc.rsp* referencia todos os assemblies que acompanham o .NET Framework. As referências reais que o ambiente de desenvolvimento do Visual Studio .NET inclui dependem do tipo de projeto. É possível modificar o arquivo *csc.rsp* e especificar opções do compilador adicionais que devem ser incluídas em cada compilação. Se você não desejar que o compilador procure e use as configurações no arquivo *csc.rsp*, especifique **NoConfig**. Essa opção do compilador não está disponível no Visual Studio e não pode ser alterada programaticamente.

# Opções avançadas do compilador C#

Artigo • 24/07/2023

As opções a seguir dão suporte a cenários avançados. A nova sintaxe MSBuild é mostrada em **Negrito**. A sintaxe mais antiga `csc.exe` é mostrada em `code style`.

- **MainEntryPoint**, **StartupObject** / `-main`: especifique o tipo que contém o ponto de entrada.
- **PdbFile** / `-pdb`: especifique o nome do arquivo de informações de depuração.
- **PathMap** / `-pathmap`: especifique um mapeamento para os nomes do caminho de origem gerados pelo compilador.
- **ApplicationConfiguration** / `-appconfig`: especifique um arquivo de configuração de aplicativo que contenha configurações de associação de assembly.
- **AdditionalLibPaths** / `-lib`: especifique diretórios adicionais para pesquisar por referências.
- **GenerateFullPaths** / `-fullpath`: o compilador gera caminhos totalmente qualificados.
- **PreferredUILang** / `-preferreduilang`: especifique o nome do idioma de saída preferencial.
- **BaseAddress** / `-baseaddress`: especifique o endereço básico para criação da biblioteca.
- **ChecksumAlgorithm** / `-checksumalgorithm`: especifique o algoritmo para calcular a soma de verificação do arquivo de origem armazenada no PDB.
- **CodePage** / `-codepage`: especifique a página de código a ser usada ao abrir arquivos de origem.
- **Utf8Output** / `-utf8output`: gere mensagens do compilador em codificação UTF-8.
- **FileAlignment** / `-filealign`: especifique o alinhamento usado em seções de arquivo de saída.
- **ErrorEndLocation** / `-errorendlocation`: coluna e linha de saída do local final de cada erro.
- **NoStandardLib** / `-nostdlib`: não faça referência à biblioteca padrão *mscorlib.dll*.
- **SubsystemVersion** / `-subsystemversion`: especifique a versão do subsistema deste assembly.
- **ModuleAssemblyName** / `-moduleassemblyname`: especifique o nome do assembly do qual esse módulo fará parte.
- **ReportIVTs** / `-reportivts`: produz informações adicionais para informações sobre [System.Runtime.CompilerServices.InternalVisibleToAttribute](#).

Você adiciona qualquer uma dessas opções em um elemento `<PropertyGroup>` em seu arquivo `*.csproj`:

XML

```
<PropertyGroup>
  <StartupObject>...</StartupObject>
  ...
</PropertyGroup>
```

## MainEntryPoint ou StartupObject

Esta opção especifica a classe que contém o ponto de entrada para o programa, se mais de uma classe contiver o método `Main`.

XML

```
<StartupObject>MyNamespace.Program</StartupObject>
```

ou

XML

```
<MainEntryPoint>MyNamespace.Program</MainEntryPoint>
```

Em que `Program` é o tipo que contém o método `Main`. O nome de classe informado deve ser totalmente qualificado. Ele deve incluir o namespace completo que contém a classe, seguido do nome de classe. Por exemplo, quando o método `Main` é localizado dentro da classe `Program` no namespace `MyApplication.Core`, a opção do compilador deve ser `-main:MyApplication.Core.Program`. Se a sua compilação incluir mais de um tipo com o método `Main`, você poderá especificar qual tipo contém o método `Main`.

### ⓘ Observação

Essa opção não pode ser usada para um projeto que inclua **instruções de nível superior**, mesmo que esse projeto contenha um ou mais métodos `Main`.

## PdbFile

A opção do compilador **PdbFile** especifica o nome e o local do arquivo de símbolos de depuração. O valor `filename` especifica o nome e o local do arquivo de símbolos de depuração.

#### XML

```
<PdbFile>filename</PdbFile>
```

Quando você especifica **DebugType**, o compilador cria um arquivo *.pdb* no mesmo diretório em que o compilador criará o arquivo de saída (.exe ou .dll). O arquivo *.pdb* tem o mesmo nome de arquivo de base que o arquivo de saída. O **PdbFile** permite que você especifique um nome de arquivo não padrão e um local para o arquivo *.pdb*. Essa opção do compilador não pode ser definida no ambiente de desenvolvimento do Visual Studio nem pode ser alterada por meio de programação.

## PathMap

A opção do compilador **PathMap** especifica como mapear caminhos físicos para os nomes de caminho de origem emitidos pelo compilador. Essa opção mapeia cada caminho físico no computador em que o compilador é executado para um caminho correspondente que deve ser gravado nos arquivos de saída. No exemplo a seguir, `path1` é o caminho completo para os arquivos de origem no ambiente atual e `sourcePath1` é o caminho de origem substituído para `path1` qualquer arquivo de saída. Para especificar vários caminhos de origem mapeados, separe-os com uma vírgula.

#### XML

```
<PathMap>path1=sourcePath1, path2=sourcePath2</PathMap>
```

O compilador grava o caminho de origem em sua saída pelos seguintes motivos:

1. O caminho de origem é substituído por um argumento quando o **CallerFilePathAttribute** é aplicado a um parâmetro opcional.
2. O caminho de origem é inserido em um arquivo PDB.
3. O caminho do arquivo PDB é inserido em um arquivo PE (executável portátil).

## ApplicationConfiguration

A opção do compilador **ApplicationConfiguration** permite que um aplicativo C# especifique o local de um arquivo (app.config) de configuração de aplicativo de

assembly para o CLR (Common Language Runtime) em tempo de associação do assembly.

XML

```
<ApplicationConfiguration>file</ApplicationConfiguration>
```

Em que `file` é o arquivo de configuração de aplicativo que contém as configurações de associação de assembly. Uma aplicação do **ApplicationConfiguration** é para cenários avançados em que um assembly precisa referenciar, ao mesmo tempo, a versão do .NET Framework e a versão do .NET Framework para Silverlight de um assembly de referência específico. Por exemplo, um designer XAML gravado no Windows Presentation Foundation (WPF) pode ter que referenciar a Área de Trabalho do WPF, para a interface do usuário do designer e o subconjunto do WPF incluído no Silverlight. O mesmo assembly do designer deve acessar ambos os assemblies. Por padrão, as referências separadas causam um erro do compilador, pois a associação de assembly considera os dois assemblies equivalentes. A opção do compilador **ApplicationConfiguration** permite a especificação do local de um arquivo app.config que desabilita o comportamento padrão por meio de uma marcação `<supportPortability>`, conforme mostrado no exemplo a seguir.

XML

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

O compilador passa o local do arquivo para a lógica de associação de assembly do CLR.

### ① Observação

Para usar o arquivo app.config que já está definido no projeto, adicione a marcação de propriedade `<UseAppConfigForCompiler>` ao arquivo `.csproj` e defina o valor dele como `true`. Para especificar um arquivo app.config diferente, adicione a marca de propriedade `<AppConfigForCompiler>` e defina seu valor para o local do arquivo.

O exemplo a seguir mostra um arquivo app.config que habilita um aplicativo a referenciar as implementações do .NET Framework e do .NET Framework para Silverlight de qualquer assembly do .NET Framework que exista em ambas as implementações. A opção do compilador **ApplicationConfiguration** especifica o local desse arquivo app.config.

XML

```
<configuration>
  <runtime>
    <assemblyBinding>
      <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
      <supportPortability PKT="31bf3856ad364e35" enable="false"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

## AdditionalLibPaths

A opção **AdditionalLibPaths** especifica o local dos assemblies referenciados com a opção [References](#).

XML

```
<AdditionalLibPaths>dir1[,dir2]</AdditionalLibPaths>
```

Em que `dir1` é um diretório para o compilador examinar se um assembly referenciado não for encontrado no diretório de trabalho atual (o diretório do qual você está invocando o compilador) ou no diretório de sistema do Common Language Runtime. `dir2` é um ou mais diretórios adicionais a serem pesquisados para as referências de assembly. Separe os nomes de diretório com vírgula e não use espaço em branco entre eles. O compilador pesquisa referências de assembly que não são totalmente qualificadas na seguinte ordem:

1. Diretório de trabalho atual.
2. O diretório de sistema do Common Language Runtime.
3. Diretórios especificados por **AdditionalLibPaths**.
4. Diretórios especificados pela variável de ambiente LIB.

Use **Reference** para especificar uma referência de assembly. **AdditionalLibPaths** é aditivo. Especificá-lo mais de uma vez o acrescenta aos valores anteriores. Como o caminho para o assembly dependente não é especificado no manifesto do assembly, o aplicativo vai encontrar e usar o assembly no cache de assembly global. O compilador que faz referência ao assembly não implica que o Common Language Runtime possa localizar e carregar o assembly em tempo de execução. Consulte [Como o tempo de execução localiza assemblies](#) para obter detalhes sobre como o runtime pesquisa assemblies referenciados.

## GenerateFullPaths

A opção **GenerateFullPath**s faz com que o compilador especifique o caminho completo para o arquivo ao listar erros de compilação e avisos.

Xml

```
<GenerateFullPath>true</GenerateFullPath>
```

Por padrão, erros e avisos oriundos da compilação especificam o nome do arquivo no qual o erro foi encontrado. A opção **GenerateFullPath**s faz com que o compilador especifique o caminho completo para o arquivo. Essa opção do compilador não está disponível no Visual Studio e não pode ser alterada programaticamente.

## PreferredUILang

Usando a opção do compilador **PreferredUILang**, é possível especificar o idioma em que o compilador C# exibe as saídas, como as mensagens de erro.

XML

```
<PreferredUILang>language</PreferredUILang>
```

Em que `language` é o [nome do idioma](#) a ser usado na saída do compilador. É possível usar a opção do compilador **PreferredUILang** para especificar o idioma que você deseja que o compilador C# use nas mensagens de erro e em outras saídas da linha de comando. Se o pacote de idiomas não estiver instalado, será usada a configuração de idioma do sistema operacional no lugar.

## BaseAddress

A opção **BaseAddress** permite especificar o endereço básico preferido para carregar a DLL. Para obter mais informações sobre quando e por que usar essa opção, consulte o [Blog do Larry Osterman](#).

XML

```
<BaseAddress>address</BaseAddress>
```

Em que `address` é o endereço básico da DLL. Esse endereço pode ser especificado como um número decimal, hexadecimal ou octal. O endereço básico padrão da DLL é definido pelo Common Language Runtime do .NET. A palavra de menor relevância neste endereço será abreviada. Por exemplo, se for especificado `0x11110001`, será

arredondado para `0x11110000`. Para concluir o processo de assinatura de uma DLL, use SN.EXE com a opção -R.

## ChecksumAlgorithm

Essa opção controla o algoritmo de soma de verificação que usamos para codificar os arquivos de origem no PDB.

XML

```
<ChecksumAlgorithm>algorithm</ChecksumAlgorithm>
```

`algorithm` deve ser `SHA1` (padrão) ou `SHA256`.

## CodePage

Esta opção especifica qual página de código deve ser usada durante a compilação caso a página necessária não seja a página de código padrão atual do sistema.

XML

```
<CodePage>id</CodePage>
```

Em que `id` é a ID da página de código a ser usada para todos os arquivos de código-fonte da compilação. O compilador tentará primeiro interpretar todos os arquivos de origem como UTF-8. Se os seus arquivos de código-fonte estiverem em uma codificação diferente de UTF-8 e usarem caracteres diferentes de ASCII de 7 bits, use a opção **CodePage** para especificar qual página de código deve ser usada. **CodePage** se aplica a todos os arquivos de código-fonte da sua compilação. Consulte [GetCPIInfo](#) para obter informações sobre como localizar quais páginas de código têm suporte do sistema.

## Utf8Output

A opção **Utf8Output** exibe a saída do compilador usando a codificação UTF-8.

XML

```
<Utf8Output>true</Utf8Output>
```

Em algumas configurações internacionais, a saída do compilador não pode ser exibida corretamente no console. Use **Utf8Output** e redirecione a saída do compilador para um arquivo.

## FileAlignment

A opção **FileAlignment** permite que você especifique o tamanho das seções em seu arquivo de saída. Os valores válidos são 512, 1024, 2048, 4096 e 8192. Esses valores estão em bytes.

XML

```
<FileAlignment>number</FileAlignment>
```

Defina a opção **FileAlignment** na página **Avançado** das propriedades de **Compilação** para seu projeto no Visual Studio. Cada seção será alinhada em um limite, múltiplo do valor de **FileAlignment**. Não há padrão fixo. Se **FileAlignment** não é especificado, o Common Language Runtime escolhe um padrão em tempo de compilação. Ao especificar o tamanho da seção, você afeta o tamanho do arquivo de saída. Modificar o tamanho da seção pode ser útil para programas que serão executados em dispositivos menores. Use [DUMPBIN](#) para ver informações sobre as seções em seu arquivo de saída.

## ErrorEndLocation

Instrui o compilador para emitir como saída a linha e a coluna do local final de cada erro.

XML

```
<ErrorEndLocation>true</ErrorEndLocation>
```

Por padrão, o compilador grava o local inicial na origem para todos os erros e avisos. Quando essa opção é definida como true, o compilador grava o local inicial e final de cada erro e aviso.

## NoStandardLib

**NoStandardLib** impede a importação de mscorlib.dll, que define todo o namespace System.

XML

```
<NoStandardLib>true</NoStandardLib>
```

Use essa opção se desejar definir ou criar seus próprios objetos e namespace System. Se você não especificar **NoStandardLib**, o mscorlib.dll será importado no programa (o mesmo que especificar `<NoStandardLib>false</NoStandardLib>`).

## SubsystemVersion

Especifica a versão mínima do subsistema em que é executado o arquivo executável. Normalmente, essa opção garante que o arquivo executável possa usar recursos de segurança que não estão disponíveis em versões mais antigas do Windows.

### ⓘ Observação

Para especificar o subsistema em si, use a opção do compilador **TargetType**.

#### XML

```
<SubsystemVersion>major.minor</SubsystemVersion>
```

`major.minor` especifica a versão mínima obrigatória do subsistema, conforme expresso em uma notação de ponto para versões principais e secundárias. Por exemplo, você pode especificar que um aplicativo não pode ser executado em um sistema operacional mais antigo que o Windows 7. Defina o valor dessa opção como 6.01, conforme descreverá a tabela mais adiante neste artigo. Você especifica os valores de `major` e `minor` como inteiros. Zeros à esquerda na versão `minor` não alteram a versão, mas zeros à direita alteram. Por exemplo, 6.1 e 6.01 se referem à mesma versão, mas 6.10 se refere a uma versão diferente. É recomendável expressar a versão secundária como dois dígitos para evitar confusão.

A seguinte tabela lista as versões de subsistema comuns do Windows.

Versão do Windows	Versão do subsistema
Windows Server 2003	5,02
Windows Vista	6,00
Windows 7	6,01
Windows Server 2008	6,01

Versão do Windows	Versão do subsistema
Windows 8	6.02

O valor padrão da opção do compilador **SubsystemVersion** depende das condições da seguinte lista:

- O valor padrão é 6.02 se qualquer opção do compilador na lista a seguir for definida:
  - [-target:appcontainerexe](#)
  - [-target:winmdobj](#)
  - [-platform:arm](#)
- O valor padrão será 6.00 se você estiver usando o MSBuild, se tiver como destino o .NET Framework 4.5 e se não definiu nenhuma das opções de compilador que foram especificadas anteriormente na lista.
- O valor padrão é 4.00 se nenhuma das condições anteriores for verdadeira.

## ModuleAssemblyName

Especifica o nome de um assembly cujos tipos não públicos podem ser acessados por um *.netmodule*.

XML

```
<ModuleAssemblyName>assembly_name</ModuleAssemblyName>
```

O **ModuleAssemblyName** deverá ser usado ao compilar um *.netmodule* e quando as seguintes condições forem true:

- O *.netmodule* precisa de acesso a tipos não públicos em um assembly existente.
- Você sabe o nome do assembly no qual o *.netmodule* será compilado.
- O assembly existente concedeu acesso de assembly amigável ao assembly em que o *.netmodule* será compilado.

Para obter mais informações sobre a compilação de um *.netmodule*, confira a opção **TargetType** do módulo. Para obter mais informações sobre assemblies amigos, consulte [Assemblies Amigáveis](#).

## ReportIVTs

Habilite ou desabilite informações de diagnóstico adicionais sobre [System.Runtime.CompilerServices.InternalsVisibleToAttribute](#) encontradas durante a

compilação:

XML

```
<ReportIVTs>true</ReportIVTs>
```

Os diagnósticos serão habilitados se o conteúdo do elemento for `true` e desabilitados se o conteúdo for `false` ou não estiver presente.

**ReportIVTs** relata as seguintes informações quando habilitado:

1. Qualquer diagnóstico de membro inacessível inclui seu assembly de origem, se diferente do assembly atual.
2. O compilador imprime a identidade do assembly do projeto que está sendo compilado, o nome do assembly e a chave pública.
3. Para cada referência passada para o compilador, ele imprime:
  - a. A identidade do assembly da referência
  - b. Se a referência concede `InternalsVisibleTo` ao projeto atual
  - c. O nome e todas as chaves públicas de todos os assemblies concedidos `InternalsVisibleTo` deste assembly

# Comentários da documentação XML

Artigo • 11/02/2023

Os arquivos de origem C# podem ter comentários estruturados que produzem a documentação da API para os tipos definidos nesses arquivos. O compilador C# produz um arquivo *XML* que contém dados estruturados que representam os comentários e as assinaturas de API. Outras ferramentas podem processar essa saída XML para criar uma documentação legível na forma de páginas da Web ou arquivos PDF, por exemplo.

Esse processo oferece muitas vantagens para você adicionar a documentação da API no código:

- O compilador C# combina a estrutura do código C# com o texto dos comentários em um único documento XML.
- O compilador C# verifica se os comentários correspondem às assinaturas da API para marcas relevantes.
- As ferramentas que processam os arquivos de documentação XML podem definir elementos XML e atributos específicos dessas ferramentas.

Ferramentas como o Visual Studio fornecem o IntelliSense para muitos elementos XML comuns usados em comentários de documentação.

Este artigo aborda estes tópicos:

- Comentários de documentação e geração de arquivos XML
- Marcas validadas pelo compilador C# e pelo Visual Studio
- Formato do arquivo XML gerado

## Criar saída de documentação XML

Crie a documentação do código gravando campos de comentário especiais indicados por barras triplas. Os campos de comentário incluem elementos XML que descrevem o bloco de código que segue os comentários. Por exemplo:

C#

```
/// <summary>
/// This class performs an important function.
/// </summary>
public class MyClass {}
```

Você define a opção [GenerateDocumentationFile](#) ou [DocumentationFile](#) e o compilador localizará todos os campos de comentário com marcas XML no código-fonte e criará um arquivo de documentação XML com base nesses comentários. Quando essa opção está habilitada, o compilador gera o aviso [CS1591](#) para qualquer membro publicamente visível declarado no projeto sem comentários da documentação XML.

## Formatos de comentário XML

O uso de comentário da documentação XML requer delimitadores, que indicam onde um comentário da documentação começa e termina. Você usa os seguintes delimitadores com as marcas de documentação XML:

- `///` Delimitador de linha única: os exemplos de documentação e modelos de projeto em C# usam esse formulário. Se houver um espaço em branco após o delimitador, ele não será incluído na saída XML.

### ➊ Observação

O Visual Studio insere automaticamente as marcas `<summary>` e `</summary>` e posiciona o cursor nessas marcas após você digitar o delimitador `///` no editor de código. Você pode ativar ou desativar esse recurso na [caixa de diálogo Opções](#).

- `/** */` Delimitadores multilinha: os delimitadores `/** */` têm as seguintes regras de formatação:
  - Na linha que contém o delimitador `/**`, se o resto da linha for um espaço em branco, a linha não será processada para comentários. Se o primeiro caractere após o delimitador `/**` for um espaço em branco, esse caractere de espaço em branco será ignorado e o restante da linha será processado. Caso contrário, todo o texto da linha após o delimitador `/**` é processado como parte do comentário.
  - Na linha que contém o delimitador `*/`, se houver apenas espaços em branco até o delimitador `*/`, essa linha será ignorada. Caso contrário, o texto na linha até o delimitador `*/` é processado como parte do comentário.
  - Para as linhas após a que começa com o delimitador `/**`, o compilador procura um padrão comum no início de cada linha. O padrão pode consistir de espaço em branco opcional e um asterisco (\*), seguido de mais espaço em branco opcional. Se o compilador encontrar um padrão comum no início de cada linha

que não começa com o delimitador `/**` ou termina com o delimitador `*/`, ele ignorará esse padrão para cada linha.

- A única parte do comentário a seguir que será processada é a linha que começa com `<summary>`. Os três formatos de marca produzem os mesmos comentários.

C#

```
/** <summary>text</summary> */  
  
/**  
<summary>text</summary>  
*/  
  
/**  
* <summary>text</summary>  
*/
```

- O compilador identifica um padrão comum de " \* " no início da segunda e terceira linhas. O padrão não é incluído na saída.

C#

```
/**  
* <summary>  
* text </summary>*/
```

- O compilador não encontra nenhum padrão comum no seguinte comentário porque o segundo caractere na terceira linha não é um asterisco. Todo o texto na segunda e terceira linhas é processado como parte do comentário.

C#

```
/**  
* <summary>  
text </summary>  
*/
```

- O compilador não encontra nenhum padrão no seguinte comentário por dois motivos. Primeiro, o número de espaços antes do asterisco não é consistente. Segundo, a quinta linha começa com uma guia, que não coincide com espaços. Todo o texto das linhas de dois a cinco é processado como parte do comentário.

C#

```
/**  
 * <summary>  
 * text  
 * text2  
 * </summary>  
 */
```

Para consultar elementos XML (por exemplo, sua função processa elementos XML específicos que você deseja descrever em um comentário da documentação XML), você pode usar o mecanismo de citação padrão (&lt; e &gt;). Para consultar identificadores genéricos em elementos de referência de código (`cref`), você pode usar os caracteres de escape (por exemplo, `cref="List&lt;T&gt;"`) ou chaves (`cref="List{T}"`). Como um caso especial, o compilador analisa as chaves como colchetes angulares para tornar o comentário da documentação menos incômodo para o autor ao fazer referência a identificadores genéricos.

#### (!) Observação

Os comentários da documentação XML não são metadados; eles não estão incluídos no assembly compilado e, portanto, não são acessíveis através de reflexão.

## Ferramentas que aceitam a entrada da documentação XML

As seguintes ferramentas criam a saída com base em comentários XML:

- [DocFX](#) : *DocFX* é um gerador de documentação de API para .NET, que atualmente dá suporte a C#, Visual Basic e F#. Ele também permite que você personalize a documentação de referência gerada. O DocFX cria um site HTML estático com base no código-fonte e nos arquivos Markdown. Além disso, o DocFX fornece a flexibilidade para personalizar o layout e o estilo do site por meio de modelos. Você também pode criar modelos personalizados.
- [Sandcastle](#) : as *ferramentas do Sandcastle* criam arquivos de ajuda para bibliotecas de classes gerenciadas que contêm páginas de referência conceituais e de API. As ferramentas do Sandcastle são baseadas em linha de comando e não possuem front-end de GUI, recursos de gerenciamento de projeto ou processo de build automatizado. O Construtor de Arquivo de Ajuda Sandcastle fornece GUI autônomo e ferramentas baseadas em linha de comando para criar um arquivo de ajuda de forma automatizada. Um pacote de integração do Visual Studio também

está disponível para que os projetos de ajuda possam ser criados e gerenciados inteiramente no Visual Studio.

- [Doxygen](#): o Doxygen gera um navegador de documentação online (em HTML) ou um manual de referência offline (no LaTeX) de um conjunto de arquivos de origem documentados. Também há suporte para gerar saída nas páginas de manual do RTF (MS Word), PostScript, PDF com hyperlink, HTML compactado, DocBook e Unix. Você pode configurar o Doxygen para extrair a estrutura de código de arquivos de origem não documentados.

## Cadeias de caracteres de ID

Cada tipo ou membro é armazenado em um elemento no arquivo XML de saída. Cada um desses elementos tem uma cadeia de caracteres de ID exclusiva que identifica o tipo ou membro. A cadeia de caracteres de ID deve responder por operadores, parâmetros, valores retornados, parâmetros de tipo genérico e parâmetros `ref`, `in` e `out`. Para codificar todos esses elementos potenciais, o compilador segue regras claramente definidas para gerar as cadeias de caracteres de ID. Programas que processam o arquivo XML usam a cadeia de identificação para identificar o item de metadados ou reflexão do .NET correspondente ao qual a documentação se aplica.

O compilador observa as seguintes regras quando gera as cadeias de identificação:

- Não há espaços em branco na cadeia de caracteres.
- A primeira parte da cadeia identifica o tipo de membro usando um único caractere seguido por dois-pontos. São usados os seguintes tipos de membro:

Caractere	Tipo de membro	Observações
N	namespace	Não é possível adicionar comentários de documentação a um namespace, mas será possível fazer referências cref a eles se houver suporte.
T	tipo	Um tipo é uma classe, interface, struct, enumeração ou delegado.
F	field	
P	propriedade	Inclui indexadores ou outras propriedades indexadas.
M	method	Inclui métodos especiais, como construtores e operadores.
E	event	

<b>Caractere</b>	<b>Tipo de membro</b>	<b>Observações</b>
!	cadeia de caracteres de erro	O restante da cadeia de caracteres fornece informações sobre o erro. O compilador C# gera informações de erro para links que não podem ser resolvidos.

- A segunda parte da cadeia de caracteres é o nome totalmente qualificado do item, iniciando na raiz do namespace. O nome do item, seus tipos delimitadores e o namespace são separados por pontos. Se o nome do próprio item tiver pontos, eles serão substituídos pelo sustenido ('#'). Supõe-se que nenhum item tem um sustenido diretamente em seu nome. Por exemplo, o nome totalmente qualificado do construtor de cadeia de caracteres é "System.String.#ctor".
- Para propriedades e métodos, segue a lista de parâmetros entre parênteses. Se não houver parâmetros, não haverá parênteses. Os parâmetros são separados por vírgulas. A codificação de cada parâmetro segue diretamente como ele é codificado em uma assinatura .NET (Consulte [Microsoft.VisualStudio.CorDebugInterop.CorElementType](#) para obter as definições de todos os elementos de caps na lista a seguir):
  - Tipos base. Tipos regulares (`ELEMENT_TYPE_CLASS` ou `ELEMENT_TYPE_VALUETYPE`) são representados como o nome totalmente qualificado do tipo.
  - Tipos intrínsecos (por exemplo, `ELEMENT_TYPE_I4`, `ELEMENT_TYPE_OBJECT`, `ELEMENT_TYPE_STRING`, `ELEMENT_TYPE_TYPEDBYREF` e `ELEMENT_TYPE_VOID`) são representados como o nome totalmente qualificado do tipo completo correspondente. Por exemplo, `System.Int32` ou `System.TypedReference`.
  - `ELEMENT_TYPE_PTR` é representado como um '\*' após o tipo modificado.
  - `ELEMENT_TYPE_BYREF` é representado como um '@' após o tipo modificado.
  - `ELEMENT_TYPE_CMOD_OPT` é representado como um '!' e o nome totalmente qualificado da classe do modificador, após o tipo modificado.
  - `ELEMENT_TYPE_SZARRAY` é representado como "[]" após o tipo de elemento da matriz.
  - `ELEMENT_TYPE_ARRAY` é representado como `[lowerbound:size,lowerbound:size]` em que o número de vírgulas é a classificação -1 e os limites e o tamanho inferiores de cada dimensão, se conhecidos, são representados no formato decimal. Se um limite inferior ou tamanho não for especificado, ele será omitido. Se o limite e o tamanho inferiores de uma determinada dimensão forem omitidos, o '!' será omitido também. Por exemplo, uma matriz bidimensional com 1 como limites inferiores e tamanhos não especificados é `[1:1:]`.

- Somente para operadores de conversão (`op_Implicit` e `op_Explicit`), o valor retornado do método é codificado como um ~ seguido pelo tipo de retorno. Por exemplo: `<member name="M:System.Decimal.op_Explicit(System.Decimal arg)~System.Int32">` é a marca do operador de conversão `public static explicit operator int (decimal value);` declarado na classe `System.Decimal`.
- Para tipos genéricos, o nome do tipo é seguido por um caractere de acento grave e, em seguida, por um número que indica o número de parâmetros de tipo genérico. Por exemplo: `<member name="T:SampleClass`^2">` é a marcação de um tipo definido como `public class SampleClass<T, U>`. Para métodos que aceitam tipos genéricos como parâmetros, os parâmetros de tipo genérico são especificados como números precedidos por caracteres de acento grave (por exemplo `0,`1). Cada número representa uma notação de matriz com base em zero para parâmetros genéricos do tipo.
  - `ELEMENT_TYPE_PINNED` é representado como um '^' após o tipo modificado. O compilador C# nunca gera essa codificação.
  - `ELEMENT_TYPE_CMOD_REQ` é representado como um '!' e o nome totalmente qualificado da classe do modificador, após o tipo modificado. O compilador C# nunca gera essa codificação.
  - `ELEMENT_TYPE_GENERICARRAY` é representado como "[?]" após o tipo de elemento da matriz. O compilador C# nunca gera essa codificação.
  - `ELEMENT_TYPE_FNPTR` é representado como "=FUNC:type(signature)", em que `type` é o tipo de retorno e `assinatura` são os argumentos do método. Se não houver nenhum argumento, os parênteses serão omitidos. O compilador C# nunca gera essa codificação.
  - Os seguintes componentes de assinatura não são representados, porque não são usadas para diferenciar métodos sobrecarregados:
    - convenção de chamada
    - tipo de retorno
    - `ELEMENT_TYPE_SENTINEL`

Os exemplos a seguir mostram como as cadeias de identificação de uma classe e seus membros são geradas:

C#

```
namespace MyNamespace
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:MyNamespace.MyClass".
    /// </summary>
```

```

public unsafe class MyClass
{
    /// <summary>
    /// Enter description here for the first constructor.
    /// ID string generated is "M:MyNamespace.MyClass.#ctor".
    /// </summary>
    public MyClass() { }

    /// <summary>
    /// Enter description here for the second constructor.
    /// ID string generated is
    "M:MyNamespace.MyClass.#ctor(System.Int32)".
    /// </summary>
    /// <param name="i">Describe parameter.</param>
    public MyClass(int i) { }

    /// <summary>
    /// Enter description here for field message.
    /// ID string generated is "F:MyNamespace.MyClass.message".
    /// </summary>
    public string? message;

    /// <summary>
    /// Enter description for constant PI.
    /// ID string generated is "F:MyNamespace.MyClass.PI".
    /// </summary>
    public const double PI = 3.14;

    /// <summary>
    /// Enter description for method func.
    /// ID string generated is "M:MyNamespace.MyClass.func".
    /// </summary>
    /// <returns>Describe return value.</returns>
    public int func() { return 1; }

    /// <summary>
    /// Enter description for method someMethod.
    /// ID string generated is
    "M:MyNamespace.MyClass.someMethod(System.String,System.Int32@,System.Void*)"
    .
    /// </summary>
    /// <param name="str">Describe parameter.</param>
    /// <param name="num">Describe parameter.</param>
    /// <param name="ptr">Describe parameter.</param>
    /// <returns>Describe return value.</returns>
    public int someMethod(string str, ref int nm, void* ptr) { return 1;
}

    /// <summary>
    /// Enter description for method anotherMethod.
    /// ID string generated is
    "M:MyNamespace.MyClass.anotherMethod(System.Int16[],System.Int32[0:,0:])".
    /// </summary>
    /// <param name="array1">Describe parameter.</param>
    /// <param name="array">Describe parameter.</param>

```

```

/// <returns>Describe return value.</returns>
public int anotherMethod(short[] array1, int[,] array) { return 0; }

/// <summary>
/// Enter description for operator.
/// ID string generated is
"M:MyNamespace.MyClass.op_Addition(MyNamespace.MyClass,MyNamespace.MyClass)"
.

/// </summary>
/// <param name="first">Describe parameter.</param>
/// <param name="second">Describe parameter.</param>
/// <returns>Describe return value.</returns>
public static MyClass operator +(MyClass first, MyClass second) {
return first; }

/// <summary>
/// Enter description for property.
/// ID string generated is "P:MyNamespace.MyClass.prop".
/// </summary>
public int prop { get { return 1; } set { } }

/// <summary>
/// Enter description for event.
/// ID string generated is "E:MyNamespace.MyClass.OnHappened".
/// </summary>
public event Del? OnHappened;

/// <summary>
/// Enter description for index.
/// ID string generated is
"P:MyNamespace.MyClass.Item(System.String)".
/// </summary>
/// <param name="str">Describe parameter.</param>
/// <returns></returns>
public int this[string s] { get { return 1; } }

/// <summary>
/// Enter description for class Nested.
/// ID string generated is "T:MyNamespace.MyClass.Nested".
/// </summary>
public class Nested { }

/// <summary>
/// Enter description for delegate.
/// ID string generated is "T:MyNamespace.MyClass.Del".
/// </summary>
/// <param name="i">Describe parameter.</param>
public delegate void Del(int i);

/// <summary>
/// Enter description for operator.
/// ID string generated is
"M:MyNamespace.MyClass.op_Explicit(MyNamespace.MyClass)~System.Int32".
/// </summary>
/// <param name="myParameter">Describe parameter.</param>

```

```
    /// <returns>Describe return value.</returns>
    public static explicit operator int(MyClass myParameter) { return 1;
}
}
}
```

## Especificação da linguagem C#

Para obter mais informações, consulte o anexo [Especificação da linguagem C#](#) nos comentários da documentação.

# Marcas XML recomendadas para comentários de documentação

Artigo • 09/05/2023

Os comentários da documentação do C# usam elementos XML para definir a estrutura da documentação de saída. Uma consequência desse recurso é que você pode adicionar qualquer XML válido em seus comentários de documentação. O compilador C# copia esses elementos no arquivo XML de saída. Embora você possa usar qualquer XML válido em seus comentários (incluindo qualquer elemento HTML válido), o código de documentação é recomendado por muitos motivos.

A seguir, temos algumas recomendações, cenários de caso de uso gerais e coisas que você precisa saber ao usar marcas de documentação XML em seu código C#. Embora você possa colocar marcas em seus comentários de documentação, este artigo descreve as marcas recomendadas para os constructos de idioma mais comuns. Em todos os casos, você deve seguir essas recomendações:

- Para fins de consistência, todos os tipos visíveis publicamente e seus membros públicos devem ser documentados.
- Membros particulares também podem ser documentados usando comentários XML. No entanto, isso expõe o funcionamento interno (potencialmente confidencial) da sua biblioteca.
- No mínimo, os tipos e seus membros devem ter uma marca `<summary>`, porque seu conteúdo é necessário para o IntelliSense.
- O texto da documentação deve ser escrito usando frases terminadas com ponto final.
- Classes parciais têm suporte total e informações da documentação serão concatenadas em uma única entrada para cada tipo.

A documentação XML começa com `///`. Quando você cria um novo projeto, os assistentes colocam algumas linhas iniciais `///` para você. O processamento desses comentários tem algumas restrições:

- A documentação deve ser em XML bem formado. Se o XML não estiver bem formado, o compilador gerará um aviso. O arquivo de documentação conterá um comentário que diz que um erro foi encontrado.
- Algumas das marcas recomendadas têm significado especial:
  - A marca `<param>` é usada para descrever parâmetros. Se ela é usada, o compilador verifica se o parâmetro existe e se todos os parâmetros são descritos na documentação. Se a verificação falha, o compilador emite um aviso.

- O atributo `cref` pode ser anexado a qualquer marca para fazer referência a um elemento de código. O compilador verifica se esse elemento de código existe. Se a verificação falha, o compilador emite um aviso. O compilador respeita qualquer instrução `using` quando procura por um tipo descrito no atributo `cref`.
- A marca `<summary>` é usada pelo IntelliSense no Visual Studio para exibir informações adicionais sobre um tipo ou membro.

### ① Observação

O arquivo XML não fornece informações completas sobre o tipo e os membros (por exemplo, ele não contém nenhuma informação de tipo). Para obter informações completas sobre um tipo ou membro, use o arquivo de documentação que reflete o membro ou tipo real.

- Os desenvolvedores são livres para criar seu próprio conjunto de marcas. O compilador as copiará para o arquivo de saída.

Algumas das marcas recomendadas podem ser usadas em qualquer elemento de linguagem. Outras têm uso mais especializado. Por fim, algumas das marcas são usadas para formatar texto em sua documentação. Este artigo descreve as marcas recomendadas organizadas por uso.

O compilador verifica a sintaxe dos elementos seguidos por um único \* na lista a seguir. O Visual Studio fornece o IntelliSense para as marcas verificadas pelo compilador e todas as marcas seguidas por \*\* na lista a seguir. Além das marcas listadas aqui, o compilador e o Visual Studio validam as marcas `<b>`, `<i>`, `<u>`, `<br/>` e `<a>`. O compilador também valida `<tt>`, que foi preterido em HTML.

- **Marcas gerais usadas** para vários elementos. Essas marcas são o conjunto mínimo de qualquer API.
  - `<summary>`: o valor desse elemento é exibido no IntelliSense no Visual Studio.
  - `<remarks>` \*\*
- **Marcas usadas para membros**. Essas marcas são usadas ao documentar métodos e propriedades.
  - `<returns>`: o valor desse elemento é exibido no IntelliSense no Visual Studio.
  - `<param>` \*: o valor desse elemento é exibido no IntelliSense no Visual Studio.
  - `<paramref>`
  - `<exception>` \*
  - `<value>`: o valor desse elemento é exibido no IntelliSense no Visual Studio.
- **Formatar a saída da documentação**. Essas marcas fornecem instruções de formatação para ferramentas que geram documentação.

- <para>
- <list>
- <c>
- <code>
- <example> \*\*
- **Reutilizar texto da documentação.** Essas marcas fornecem ferramentas que facilitam a reutilização de comentários XML.
  - <inheritdoc> \*\*
  - <include> \*
- **Gerar links e referências.** Essas marcas geram links para outra documentação.
  - <see> \*
  - <seealso> \*
  - cref
  - href
- **Marcas para tipos e métodos genéricos.** Essas marcas são usadas somente em tipos e métodos genéricos
  - <typeparam> \*: o valor desse elemento é exibido no IntelliSense no Visual Studio.
  - <typeparamref>

#### ① Observação

Os comentários de documentação não podem ser aplicados a um namespace.

Se você quiser que os colchetes angulares sejam exibidos no texto de um comentário de documentação, use a codificação HTML de < e > que são &lt; e &gt;; respectivamente. Essa codificação é mostrada no exemplo a seguir.

C#

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

## Marcas gerais

### <summary>

XML

```
<summary>description</summary>
```

A marca `<summary>` deve ser usada para descrever um tipo ou um membro de tipo. Use `<remarks>` para adicionar mais informações a uma descrição de tipo. Use o atributo `cref` para habilitar ferramentas de documentação como o [DocFX](#) e o [Sandcastle](#) para criar hiperlinks internos para páginas de documentação em elementos de código. O texto da marca `<summary>` é a única fonte de informações sobre o tipo no IntelliSense e também é exibido na janela Pesquisador de objetos.

## <remarks>

XML

```
<remarks>
description
</remarks>
```

A marca `<remarks>` é usada para adicionar informações sobre o tipo, complementando as informações especificadas com `<summary>`. Essas informações são exibidas na janela do Pesquisador de Objetos. Essa marca pode incluir explicações mais longas. Você pode achar que usar seções `CDATA` para markdown torna a gravação mais conveniente.

Ferramentas como [docfx](#) processam o texto de markdown em seções `CDATA`.

## Membros do documento

### <returns>

XML

```
<returns>description</returns>
```

A marca `<returns>` deve ser usada no comentário para uma declaração de método descrever o valor retornado.

### <param>

XML

```
<param name="name">description</param>
```

- `name`: o nome do parâmetro de um método. Coloque o nome entre aspas duplas (""). Os nomes dos parâmetros devem corresponder à assinatura da API. Se um ou mais parâmetros não forem abordados, o compilador emitirá um aviso. O compilador também emitirá um aviso se o valor `name` não corresponder a um parâmetro formal na declaração do método.

A marca `<param>` deve ser usada no comentário para uma declaração de método para descrever um dos parâmetros do método. Para documentar vários parâmetros, use várias marcas `<param>`. O texto da marca `<param>` é exibido no IntelliSense, o Navegador de objetos e o Relatório Web de comentários de código.

## <paramref>

XML

```
<paramref name="name"/>
```

- `name`: o nome do parâmetro ao qual você deseja fazer referência. Coloque o nome entre aspas duplas ("").

A marca `<paramref>` fornece uma maneira de indicar que uma palavra nos comentários do código, por exemplo, em um bloco `<summary>` ou `<remarks>` refere-se a um parâmetro. O arquivo XML pode ser processado para formatar essa palavra de alguma forma distinta, como com uma fonte em negrito ou itálico.

## <exception>

XML

```
<exception cref="member">description</exception>
```

- `cref = "member"`: uma referência a uma exceção que está disponível no ambiente de compilação atual. O compilador verifica se a exceção apresentada existe e move o `member` para o nome de elemento canônico no XML de saída. `member` deve ser exibido entre aspas duplas ("").

A marca `<exception>` permite que você especifique quais exceções podem ser lançadas. Essa marca pode ser aplicada às definições de métodos, propriedades, eventos e indexadores.

## <value>

XML

```
<value>property-description</value>
```

A marca `<value>` permite descrever o valor que uma propriedade representa. Observe que quando você adiciona uma propriedade por meio do assistente de código no ambiente de desenvolvimento do Visual Studio .NET, ele adicionará uma marca `<summary>` à nova propriedade. Adicione manualmente uma marca `<value>` para descrever o valor que a propriedade representa.

## Formatar a saída da documentação

### <para>

XML

```
<remarks>
  <para>
    This is an introductory paragraph.
  </para>
  <para>
    This paragraph contains more details.
  </para>
</remarks>
```

A marca `<para>` é para ser usada dentro de uma marca, como `<summary>`, `<remarks>` ou `<returns>`, e permite adicionar estrutura ao texto. A `<para>` marca cria um parágrafo espaçado duplo. Use a `<br/>` marca se quiser um único parágrafo espaçado.

### <list>

XML

```
<list type="bullet|number|table">
  <listheader>
    <term>term</term>
    <description>description</description>
  </listheader>
  <item>
    <term>Assembly</term>
    <description>The library or executable built from a compilation.
  </description>
```

```
</item>
</list>
```

O bloco `<listheader>` é usado para definir a linha de cabeçalho de uma tabela ou lista de definição. Ao definir uma tabela, é necessário fornecer uma entrada para `term` no título. Cada item na lista é especificado com um bloco `<item>`. Ao criar uma lista de definições, será necessário especificar `term` e `description`. No entanto, para uma tabela, lista com marcadores ou lista numerada, será necessário fornecer apenas uma entrada para `description`. Uma lista ou tabela pode ter quantos blocos `<item>` forem necessários.

## <C>

XML

```
<c>text</c>
```

A marca `<c>` oferece uma forma de indicar que o texto em uma descrição deve ser marcado como código. Use `<code>` para indicar várias linhas como código.

## <code>

XML

```
<code>
    var index = 5;
    index++;
</code>
```

A marca `<code>` é usada para indicar várias linhas de código. Use `<c>` para indicar que o texto dentro uma descrição deve ser marcado como código.

## <example>

XML

```
<example>
This shows how to increment an integer.
<code>
    var index = 5;
    index++;
</code>
</example>
```

A marca `<example>` permite especificar um exemplo de como usar um método ou outro membro da biblioteca. Normalmente, isso envolve o uso da marca `<code>`.

## Reutilizar texto da documentação

### `<inheritdoc>`

XML

```
<inheritdoc [cref=""] [path=""]/>
```

Herdar comentários XML de classes base, interfaces e métodos semelhantes. Usar `inheritdoc` elimina a cópia e a colagem indesejadas de comentários XML duplicados e mantém automaticamente os comentários XML sincronizados. Observe que quando você adicionar a marca `<inheritdoc>` a um tipo, todos os membros herdarão os comentários também.

- `cref`: especifique o membro do qual herdar a documentação. As marcas já definidas no membro atual não são substituídas pelas herdadas.
- `path`: a consulta de expressão XPath que resultará em um nó definido para mostrar. Você pode usar esse atributo para filtrar as marcas para incluir ou excluir da documentação herdada.

Adicione seus comentários XML em classes ou interfaces base e deixe herdado copiar os comentários para implementar classes. Adicione seus comentários XML aos métodos síncronos e deixe herdado copiar os comentários para suas versões assíncronas dos mesmos métodos. Se você quiser copiar os comentários de um membro específico, use o atributo `cref` para especificar o membro.

### `<include>`

XML

```
<include file='filename' path='tagpath[@name="id"]' />
```

- `filename`: o nome do arquivo XML que contém a documentação. O nome do arquivo pode ser qualificado com um caminho relativo ao arquivo de código-fonte. Coloque `filename` entre aspas simples (' ').

- `tagpath`: O caminho das marcas em `filename` que leva à marca `name`. Coloque o caminho entre aspas simples (' ').
- `name`: o especificador de nome na marca que precede os comentários; `name` terá um `id`.
- `id`: a ID da marca que precede os comentários. Coloque a ID entre aspas duplas ("").

A marca `<include>` permite consultar comentários em outro arquivo que descrevem os tipos e membros em seu código-fonte. Incluir um arquivo externo é uma alternativa para inserir comentários de documentação diretamente em seu arquivo de código-fonte. Colocando a documentação em um arquivo separado, é possível aplicar o controle do código-fonte à documentação separadamente do código-fonte. Uma pessoa pode fazer o check-out do arquivo de código-fonte e outra pessoa pode fazer o check-out do arquivo de documentação. A marca `<include>` usa a sintaxe XML XPath. Consulte a documentação do XPath para obter maneiras de personalizar o uso de `<include>`.

## Gerar links e referências

### `<see>`

XML

```

<see cref="member"/>
<!-- or -->
<see cref="member">Link text</see>
<!-- or -->
<see href="link">Link Text</see>
<!-- or -->
<see langword="keyword"/>

```

- `cref="member"`:Uma referência a um membro ou campo disponível para ser chamado do ambiente de compilação atual. O compilador verifica se o elemento de código fornecido existe e passa `member` para o nome de elemento no XML de saída. Coloque `member` entre aspas duplas (" "). Você pode fornecer texto de link diferente para um "cref", usando uma marca de fechamento separada.
- `href="link"`:um link clicável para uma determinada URL. Por exemplo, `<see href="https://github.com">GitHub</see>` produz um link clicável com texto GitHub que é vinculado a `https://github.com`.
- `langword="keyword"`: uma palavra-chave de idioma, como `true` ou uma das outras palavras-chave válidas.

A marca `<see>` permite especificar um link de dentro do texto. Use `<seealso>` para indicar que o texto deve ser colocado em uma seção Confira também. Use o [atributo cref](#) para criar hyperlinks internos para páginas de documentação para elementos de código. Você inclui os parâmetros de tipo para especificar uma referência a um tipo ou método genérico, como `cref="IDictionary{T, U}"`. Além disso, `href` é um atributo válido que funcionará como um hyperlink.

## <seealso>

XML

```
<seealso cref="member"/>
<!-- or --&gt;
&lt;seealso href="link"&gt;Link Text&lt;/seealso&gt;</pre>
```

- `cref="member"`:Uma referência a um membro ou campo disponível para ser chamado do ambiente de compilação atual. O compilador verifica se o elemento de código fornecido existe e passa `member` para o nome de elemento no XML de saída. `member` deve ser exibido entre aspas duplas (" ").
- `href="link"`:um link clicável para uma determinada URL. Por exemplo, `<seealso href="https://github.com">GitHub</seealso>` produz um link clicável com texto GitHub que é vinculado a `https://github.com`.

A marca `<seealso>` permite especificar o texto que você quer que seja exibido na seção Confira também Use `<see>` para especificar um link de dentro do texto. Não é possível aninhar a marca `seealso` dentro da marca `summary`.

## atributo cref

O atributo `cref` em uma marca de documentação XML significa "referência de código". Ele especifica que o texto interno da marca é um elemento de código, como um tipo, método ou propriedade. Ferramentas de documentação, como o [DocFX](#) e o [Sandcastle](#), usam os atributos `cref` para gerar automaticamente os hyperlinks para a página em que o tipo ou o membro está documentado.

## Atributo href

O atributo `href` significa uma referência a uma página da Web. Você pode usá-lo para fazer referência direta à documentação online sobre sua API ou biblioteca.

# Tipos e métodos genéricos

## <typeparam>

XML

```
<typeparam name="TResult">The type returned from this method</typeparam>
```

- `TResult`: o nome do parâmetro de tipo. Coloque o nome entre aspas duplas (" ").

A marca `<typeparam>` deve ser usada no comentário para um tipo genérico ou para uma declaração de método descrever um dos parâmetros de tipo. Adicione uma marca para cada parâmetro de tipo do tipo ou do método genérico. O texto da marca `<typeparam>` será exibido no IntelliSense.

## <typeparamref>

XML

```
<typeparamref name=" TKey" />
```

- `TKey`: o nome do parâmetro de tipo. Coloque o nome entre aspas duplas (" ").

Use essa marca para habilitar os consumidores do arquivo de documentação a formatar a palavra de alguma forma distinta, por exemplo, em itálico.

## Marcas definidas pelo usuário

Todas as marcas descritas acima representam as marcas que são reconhecidas pelo compilador C#. No entanto, o usuário é livre para definir suas próprias marcas.

Ferramentas como o Sandcastle dão suporte para marcas extras, como `<event>` ↗ e `<note>` ↗, e até mesmo à [documentação de namespaces](#) ↗. Ferramentas de geração de documentação internas ou personalizadas também podem ser usadas com as marcas padrão e vários formatos de saída, de HTML a PDF, podem ter suporte.

# Exemplo de comentários de documentação XML

Artigo • 28/03/2023

Este artigo contém três exemplos para adicionar comentários de documentação XML à maioria dos elementos de linguagem C#. O primeiro exemplo mostra como você documenta uma classe com membros diferentes. O segundo mostra como você reutilizaria explicações para uma hierarquia de classes ou interfaces. O terceiro mostra marcas a serem usadas para classes e membros genéricos. O segundo e o terceiro exemplos usam conceitos abordados no primeiro exemplo.

## Documentar uma classe, struct ou interface

O exemplo a seguir mostra elementos de linguagem comuns e as marcas que você provavelmente usará para descrever esses elementos. Os comentários da documentação descrevem o uso das marcas, em vez da própria classe.

C#

```
/// <summary>
/// Every class and member should have a one sentence
/// summary describing its purpose.
/// </summary>
/// <remarks>
/// You can expand on that one sentence summary to
/// provide more information for readers. In this case,
/// the <c>ExampleClass</c> provides different C#
/// elements to show how you would add documentation
/// comments for most elements in a typical class.
/// <para>
/// The remarks can add multiple paragraphs, so you can
/// write detailed information for developers that use
/// your work. You should add everything needed for
/// readers to be successful. This class contains
/// examples for the following:
/// </para>
/// <list type="table">
/// <item>
/// <term>Summary</term>
/// <description>
/// This should provide a one sentence summary of the class or member.
/// </description>
/// </item>
/// <item>
/// <term>Remarks</term>
/// <description>
```

```
/// This is typically a more detailed description of the class or member
/// </description>
/// </item>
/// <item>
/// <term>para</term>
/// <description>
/// The para tag separates a section into multiple paragraphs
/// </description>
/// </item>
/// <item>
/// <term>list</term>
/// <description>
/// Provides a list of terms or elements
/// </description>
/// </item>
/// <item>
/// <term>returns, param</term>
/// <description>
/// Used to describe parameters and return values
/// </description>
/// </item>
/// <item>
/// <term>value</term>
/// <description>Used to describe properties</description>
/// </item>
/// <item>
/// <term>exception</term>
/// <description>
/// Used to describe exceptions that may be thrown
/// </description>
/// </item>
/// <item>
/// <term>c, cref, see,seealso</term>
/// <description>
/// These provide code style and links to other
/// documentation elements
/// </description>
/// </item>
/// <item>
/// <term>example, code</term>
/// <description>
/// These are used for code examples
/// </description>
/// </item>
/// </list>
/// <para>
/// The list above uses the "table" style. You could
/// also use the "bullet" or "number" style. Neither
/// would typically use the "term" element.
/// <br/>
/// Note: paragraphs are double spaced. Use the *br*
/// tag for single spaced lines.
/// </para>
/// </remarks>
public class ExampleClass
```

```
{  
    /// <value>  
    /// The <c>Label</c> property represents a label  
    /// for this instance.  
    /// </value>  
    /// <remarks>  
    /// The <see cref="Label"/> is a <see langword="string"/>  
    /// that you use for a label.  
    /// <para>  
    /// Note that there isn't a way to provide a "cref" to  
    /// each accessor, only to the property itself.  
    /// </para>  
    /// </remarks>  
    public string? Label  
    {  
        get;  
        set;  
    }  
  
    /// <summary>  
    /// Adds two integers and returns the result.  
    /// </summary>  
    /// <returns>  
    /// The sum of two integers.  
    /// </returns>  
    /// <param name="left">  
    /// The left operand of the addition.  
    /// </param>  
    /// <param name="right">  
    /// The right operand of the addition.  
    /// </param>  
    /// <example>  
    /// <code>  
    /// int c = Math.Add(4, 5);  
    /// if (c > 10)  
    /// {  
    ///     Console.WriteLine(c);  
    /// }  
    /// </code>  
    /// </example>  
    /// <exception cref="System.OverflowException">  
    /// Thrown when one parameter is  
    /// <see cref="Int32.MaxValue">MaxValue</see> and the other is  
    /// greater than 0.  
    /// Note that here you can also use  
    /// <see  
    href="https://learn.microsoft.com/dotnet/api/system.int32.maxvalue"/>  
        /// to point a web page instead.  
    /// </exception>  
    /// <see cref="ExampleClass"/> for a list of all  
    /// the tags in these examples.  
    /// <seealso cref="ExampleClass.Label"/>  
    public static int Add(int left, int right)  
    {  
        if ((left == int.MaxValue && right > 0) || (right ==
```

```

        int.MaxValue && left > 0))
            throw new System.OverflowException();

        return left + right;
    }
}

/// <summary>
/// This is an example of a positional record.
/// </summary>
/// <remarks>
/// There isn't a way to add XML comments for properties
/// created for positional records, yet. The language
/// design team is still considering what tags should
/// be supported, and where. Currently, you can use
/// the "param" tag to describe the parameters to the
/// primary constructor.
/// </remarks>
/// <param name="FirstName">
/// This tag will apply to the primary constructor parameter.
/// </param>
/// <param name="LastName">
/// This tag will apply to the primary constructor parameter.
/// </param>
public record Person(string FirstName, string LastName);
}

```

Adicionar documentação pode desordenar o código-fonte com grandes conjuntos de comentários destinados aos usuários da biblioteca. Use a marca `<Include>` para separar seus comentários XML da fonte. Seu código-fonte faz referência a um arquivo XML com a marca `<Include>`:

C#

```

/// <include file='xml_include_tag.xml'
path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.xml'
path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
    }
}

```

O segundo arquivo, `xml_include_tag.xml`, contém os comentários de documentação a seguir:

```
XML

<MyDocs>
  <MyMembers name="test">
    <summary>
      The summary for this type.
    </summary>
  </MyMembers>
  <MyMembers name="test2">
    <summary>
      The summary for this other type.
    </summary>
  </MyMembers>
</MyDocs>
```

## Documentar uma hierarquia de classes e interfaces

O elemento `<inheritdoc>` significa que um tipo ou membro *herda* comentários de documentação de uma classe base ou interface. Você também pode usar o elemento `<inheritdoc>` com o atributo `cref` para herdar comentários de um membro do mesmo tipo. O exemplo a seguir mostra formas de usar essa marca. Observe que quando você adiciona o atributo `inheritdoc` a um tipo, os comentários de membro são herdados. Você pode impedir o uso de comentários herdados escrevendo comentários sobre os membros no tipo derivado. Eles serão escolhidos em vez dos comentários herdados.

```
C#

/// <summary>
/// A summary about this class.
/// </summary>
/// <remarks>
/// These remarks would explain more about this class.
/// In this example, these comments also explain the
/// general information about the derived class.
/// </remarks>
public class MainClass
{
}

///<inheritdoc/>
public class DerivedClass : MainClass
{}
```

```
/// <summary>
/// This interface would describe all the methods in
/// its contract.
/// </summary>
/// <remarks>
/// While elided for brevity, each method or property
/// in this interface would contain docs that you want
/// to duplicate in each implementing class.
/// </remarks>
public interface ITestInterface
{
    /// <summary>
    /// This method is part of the test interface.
    /// </summary>
    /// <remarks>
    /// This content would be inherited by classes
    /// that implement this interface when the
    /// implementing class uses "inheritdoc"
    /// </remarks>
    /// <returns>The value of <paramref name="arg" /> </returns>
    /// <param name="arg">The argument to the method</param>
    int Method(int arg);
}

///<inheritdoc cref="ITestInterface"/>
public class ImplementingClass : ITestInterface
{
    // doc comments are inherited here.
    public int Method(int arg) => arg;
}

/// <summary>
/// This class shows hows you can "inherit" the doc
/// comments from one method in another method.
/// </summary>
/// <remarks>
/// You can inherit all comments, or only a specific tag,
/// represented by an xpath expression.
/// </remarks>
public class InheritOnlyReturns
{
    /// <summary>
    /// In this example, this summary is only visible for this method.
    /// </summary>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="/returns"/>
    public static bool MyChildMethod() { return false; }
}

/// <Summary>
/// This class shows an example ofsharing comments across methods.
/// </Summary>
```

```

public class InheritAllButRemarks
{
    /// <summary>
    /// In this example, this summary is visible on all the methods.
    /// </summary>
    /// <remarks>
    /// The remarks can be inherited by other methods
    /// using the xpath expression.
    /// </remarks>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="/*[not(self::remarks)]"/>
    public static bool MyChildMethod() { return false; }
}

```

## Tipos genéricos

Use a marca `<typeparam>` para descrever parâmetros de tipo em tipos e métodos genéricos. O valor do atributo `cref` requer uma nova sintaxe para fazer referência a um método ou classe genérico:

C#

```

/// <summary>
/// This is a generic class.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}" />
/// type as a cref attribute.
/// In generic classes and methods, you'll often want to reference the
/// generic type, or the type parameter.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

/// <Summary>
/// This shows examples of typeparamref and typeparam tags
/// </Summary>
public class ParamsAndParamRefs
{
    /// <summary>
    /// The GetGenericValue method.
    /// </summary>
    /// <remarks>
    /// This sample shows how to specify the <see cref="GetGenericValue" />
    /// method as a cref attribute.
    /// The parameter and return value are both of an arbitrary type,
    /// <typeparamref name="T" />
}

```

```
/// </remarks>
public static T GetGenericValue<T>(T para)
{
    return para;
}
```

## Exemplo de classe matemática

O código a seguir mostra um exemplo realista de adição de comentários de documentos a uma biblioteca de matemática.

C#

```
namespace TaggedLibrary
{
    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
    /// <summary>
    /// The main <c>Math</c> class.
    /// Contains all methods for performing basic math functions.
    /// <list type="bullet">
    /// <item>
    /// <term>Add</term>
    /// <description>Addition Operation</description>
    /// </item>
    /// <item>
    /// <term>Subtract</term>
    /// <description>Subtraction Operation</description>
    /// </item>
    /// <item>
    /// <term>Multiply</term>
    /// <description>Multiplication Operation</description>
    /// </item>
    /// <item>
    /// <term>Divide</term>
    /// <description>Division Operation</description>
    /// </item>
    /// </list>
    /// </summary>
    /// <remarks>
    /// <para>
    /// This class can add, subtract, multiply and divide.
    /// </para>
    /// <para>
    /// These operations can be performed on both
    /// integers and doubles.
    /// </para>
    /// </remarks>
```

```
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers <paramref name="a"/> and <paramref name="b"/>
    /// and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">
    /// Thrown when one parameter is <see cref="Int32.MaxValue"/> and
the other
    /// is greater than 0.
    /// </exception>
    /// See <see cref="Math.Add(double, double)"> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)">
    /// <seealso cref="Math.Multiply(int, int)">
    /// <seealso cref="Math.Divide(int, int)">
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) ||
            (b == int.MaxValue && a > 0))
        {
            throw new System.OverflowException();
        }
        return a + b;
    }

    // Adds two doubles and returns the result
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/>
    /// and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Add(4.5, 5.4);
    /// if (c > 10)
    /// {
```

```
///      Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// <exception cref="System.OverflowException">
/// Thrown when one parameter is max and the other
/// is greater than 0.</exception>
/// See <see cref="Math.Add(int, int)" /> to add integers.
/// <seealso cref="Math.Subtract(double, double)" />
/// <seealso cref="Math.Multiply(double, double)" />
/// <seealso cref="Math.Divide(double, double)" />
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Add(double a, double b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == double.MaxValue && b > 0)
        || (b == double.MaxValue && a > 0))
    {
        throw new System.OverflowException();
    }

    return a + b;
}

// Subtracts an integer from another and returns the result
/// <summary>
/// Subtracts <paramref name="b"/> from <paramref name="a"/>
/// and returns the result.
/// </summary>
/// <returns>
/// The difference between two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Subtract(4, 5);
/// if (c > 1)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Subtract(double, double)" /> to subtract
doubles.
/// <seealso cref="Math.Add(int, int)" />
/// <seealso cref="Math.Multiply(int, int)" />
/// <seealso cref="Math.Divide(int, int)" />
/// <param name="a">An integer.</param>
/// <param name="b">An integer.</param>
public static int Subtract(int a, int b)
{
    return a - b;
}
```

```
// Subtracts a double from another and returns the result
/// <summary>
/// Subtracts a double <paramref name="b"/> from another
/// double <paramref name="a"/> and returns the result.
/// </summary>
/// <returns>
/// The difference between two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Subtract(4.5, 5.4);
/// if (c > 1)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Subtract(int, int)"> to subtract integers.
/// <seealso cref="Math.Add(double, double)">
/// <seealso cref="Math.Multiply(double, double)">
/// <seealso cref="Math.Divide(double, double)">
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Subtract(double a, double b)
{
    return a - b;
}

// Multiplies two integers and returns the result
/// <summary>
/// Multiplies two integers <paramref name="a"/>
/// and <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The product of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Multiply(4, 5);
/// if (c > 100)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Multiply(double, double)"> to multiply
doubles.
/// <seealso cref="Math.Add(int, int)">
/// <seealso cref="Math.Subtract(int, int)">
/// <seealso cref="Math.Divide(int, int)">
/// <param name="a">An integer.</param>
/// <param name="b">An integer.</param>
public static int Multiply(int a, int b)
{
    return a * b;
}
```

```
}

// Multiplies two doubles and returns the result
/// <summary>
/// Multiplies two doubles <paramref name="a"/> and
/// <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The product of two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Multiply(4.5, 5.4);
/// if (c > 100.0)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Multiply(int, int)"> to multiply integers.
/// <seealso cref="Math.Add(double, double)">
/// <seealso cref="Math.Subtract(double, double)">
/// <seealso cref="Math.Divide(double, double)">
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Multiply(double a, double b)
{
    return a * b;
}

// Divides an integer by another and returns the result
/// <summary>
/// Divides an integer <paramref name="a"/> by another
/// integer <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The quotient of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Divide(4, 5);
/// if (c > 1)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// <exception cref="System.DivideByZeroException">
/// Thrown when <paramref name="b"/> is equal to 0.
/// </exception>
/// See <see cref="Math.Divide(double, double)"> to divide doubles.
/// <seealso cref="Math.Add(int, int)">
/// <seealso cref="Math.Subtract(int, int)">
/// <seealso cref="Math.Multiply(int, int)">
/// <param name="a">An integer dividend.</param>
```

```

    ///<param name="b">An integer divisor.</param>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    ///<summary>
    /// Divides a double <paramref name="a"/> by another double
    /// <paramref name="b"/> and returns the result.
    ///</summary>
    ///<returns>
    /// The quotient of two doubles.
    ///</returns>
    ///<example>
    ///<code>
    /// double c = Math.Divide(4.5, 5.4);
    /// if (c > 1.0)
    /// {
    ///     Console.WriteLine(c);
    /// }
    ///</code>
    ///</example>
    ///<exception cref="System.DivideByZeroException">
    /// Thrown when <paramref name="b"/> is equal to 0.
    ///</exception>
    /// See <see cref="Math.Divide(int, int)"> to divide integers.
    ///<seealso cref="Math.Add(double, double)">
    ///<seealso cref="Math.Subtract(double, double)">
    ///<seealso cref="Math.Multiply(double, double)">
    ///<param name="a">A double precision dividend.</param>
    ///<param name="b">A double precision divisor.</param>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}

```

Você pode descobrir que o código está obscurecido por todos os comentários. O exemplo final mostra como você adaptaria essa biblioteca para usar a marca `include`. Você move toda a documentação para um arquivo XML:

#### XML

```

<docs>
<members name="math">
<Math>
<summary>
The main <c>Math</c> class.
Contains all methods for performing basic math functions.
</summary>

```

```
<remarks>
<para>This class can add, subtract, multiply and divide.</para>
<para>These operations can be performed on both integers and doubles.
</para>
</remarks>
</Math>
<AddInt>
<summary>
Adds two integers <paramref name="a"/> and <paramref name="b"/>
and returns the result.
</summary>
<returns>
The sum of two integers.
</returns>
<example>
<code>
int c = Math.Add(4, 5);
if (c > 10)
{
    Console.WriteLine(c);
}
</code>
</example>
<exception cref="System.OverflowException">Thrown when one
parameter is max
and the other is greater than 0.</exception>
See <see cref="Math.Add(double, double)"> to add doubles.
<seealso cref="Math.Subtract(int, int)">
<seealso cref="Math.Multiply(int, int)">
<seealso cref="Math.Divide(int, int)">
<param name="a">An integer.</param>
<param name="b">An integer.</param>
</AddInt>
<AddDouble>
<summary>
Adds two doubles <paramref name="a"/> and <paramref name="b"/>
and returns the result.
</summary>
<returns>
The sum of two doubles.
</returns>
<example>
<code>
double c = Math.Add(4.5, 5.4);
if (c > 10)
{
    Console.WriteLine(c);
}
</code>
</example>
<exception cref="System.OverflowException">Thrown when one parameter
is max
and the other is greater than 0.</exception>
See <see cref="Math.Add(int, int)"> to add integers.
<seealso cref="Math.Subtract(double, double)">
```

```
<seealso cref="Math.Multiply(double, double)" />
<seealso cref="Math.Divide(double, double)" />
<param name="a">A double precision number.</param>
<param name="b">A double precision number.</param>
</AddDouble>
<SubtractInt>
    <summary>
    Subtracts <paramref name="b"/> from <paramref name="a"/> and
    returns the result.
    </summary>
    <returns>
    The difference between two integers.
    </returns>
    <example>
        <code>
int c = Math.Subtract(4, 5);
if (c > 1)
{
    Console.WriteLine(c);
}
</code>
    </example>
    See <see cref="Math.Subtract(double, double)" /> to subtract doubles.
    <seealso cref="Math.Add(int, int)" />
    <seealso cref="Math.Multiply(int, int)" />
    <seealso cref="Math.Divide(int, int)" />
    <param name="a">An integer.</param>
    <param name="b">An integer.</param>
</SubtractInt>
<SubtractDouble>
    <summary>
    Subtracts a double <paramref name="b"/> from another
    double <paramref name="a"/> and returns the result.
    </summary>
    <returns>
    The difference between two doubles.
    </returns>
    <example>
        <code>
double c = Math.Subtract(4.5, 5.4);
if (c > 1)
{
    Console.WriteLine(c);
}
</code>
    </example>
    See <see cref="Math.Subtract(int, int)" /> to subtract integers.
    <seealso cref="Math.Add(double, double)" />
    <seealso cref="Math.Multiply(double, double)" />
    <seealso cref="Math.Divide(double, double)" />
    <param name="a">A double precision number.</param>
    <param name="b">A double precision number.</param>
</SubtractDouble>
<MultiplyInt>
    <summary>
```

```
Multiplies two integers <paramref name="a"/> and
<paramref name="b"/> and returns the result.
</summary>
<returns>
The product of two integers.
</returns>
<example>
<code>
int c = Math.Multiply(4, 5);
if (c > 100)
{
    Console.WriteLine(c);
}
</code>
</example>
See <see cref="Math.Multiply(double, double)"> to multiply doubles.
<seealso cref="Math.Add(int, int)">
<seealso cref="Math.Subtract(int, int)">
<seealso cref="Math.Divide(int, int)">
<param name="a">An integer.</param>
<param name="b">An integer.</param>
</MultiplyInt>
<MultiplyDouble>
<summary>
Multiplies two doubles <paramref name="a"/> and
<paramref name="b"/> and returns the result.
</summary>
<returns>
The product of two doubles.
</returns>
<example>
<code>
double c = Math.Multiply(4.5, 5.4);
if (c > 100.0)
{
    Console.WriteLine(c);
}
</code>
</example>
See <see cref="Math.Multiply(int, int)"> to multiply integers.
<seealso cref="Math.Add(double, double)">
<seealso cref="Math.Subtract(double, double)">
<seealso cref="Math.Divide(double, double)">
<param name="a">A double precision number.</param>
<param name="b">A double precision number.</param>
</MultiplyDouble>
<DivideInt>
<summary>
Divides an integer <paramref name="a"/> by another integer
<paramref name="b"/> and returns the result.
</summary>
<returns>
The quotient of two integers.
</returns>
<example>
```

```

<code>
int c = Math.Divide(4, 5);
if (c > 1)
{
    Console.WriteLine(c);
}
</code>
</example>
<exception cref="System.DivideByZeroException">
Thrown when <paramref name="b"/> is equal to 0.
</exception>
See <see cref="Math.Divide(double, double)" /> to divide doubles.
<seealso cref="Math.Add(int, int)" />
<seealso cref="Math.Subtract(int, int)" />
<seealso cref="Math.Multiply(int, int)" />
<param name="a">An integer dividend.</param>
<param name="b">An integer divisor.</param>
</DivideInt>
<DivideDouble>
<summary>
Divides a double <paramref name="a"/> by another
double <paramref name="b"/> and returns the result.
</summary>
<returns>
The quotient of two doubles.
</returns>
<example>
<code>
double c = Math.Divide(4.5, 5.4);
if (c > 1.0)
{
    Console.WriteLine(c);
}
</code>
</example>
<exception cref="System.DivideByZeroException">Thrown when <paramref
name="b"/> is equal to 0.</exception>
See <see cref="Math.Divide(int, int)" /> to divide integers.
<seealso cref="Math.Add(double, double)" />
<seealso cref="Math.Subtract(double, double)" />
<seealso cref="Math.Multiply(double, double)" />
<param name="a">A double precision dividend.</param>
<param name="b">A double precision divisor.</param>
</DivideDouble>
</members>
</docs>

```

No XML acima, os comentários de documentação de cada membro aparecem diretamente dentro de uma marca cujo nome corresponde ao que eles fazem. Você pode escolher sua própria estratégia. O código usa a marca `<include>` para fazer referência ao elemento apropriado no arquivo XML:

C#

```
namespace IncludeTag
{
    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
    /// <include file='include.xml'
    path='docs/members[@name="math"]/Math/*'>
    public class Math
    {
        // Adds two integers and returns the result
        /// <include file='include.xml'
        path='docs/members[@name="math"]/AddInt/*'>
        public static int Add(int a, int b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a >
0))
                throw new System.OverflowException();

            return a + b;
        }

        // Adds two doubles and returns the result
        /// <include file='include.xml'
        path='docs/members[@name="math"]/AddDouble/*'>
        public static double Add(double a, double b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == double.MaxValue && b > 0) || (b == double.MaxValue &&
a > 0))
                throw new System.OverflowException();

            return a + b;
        }

        // Subtracts an integer from another and returns the result
        /// <include file='include.xml'
        path='docs/members[@name="math"]/SubtractInt/*'>
        public static int Subtract(int a, int b)
        {
            return a - b;
        }

        // Subtracts a double from another and returns the result
        /// <include file='include.xml'
        path='docs/members[@name="math"]/SubtractDouble/*'>
        public static double Subtract(double a, double b)
        {
```

```

        return a - b;
    }

    // Multiplies two integers and returns the result
    /// <include file='include.xml'
path='docs/members[@name="math"]/MultiplyInt/*'>
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    /// <include file='include.xml'
path='docs/members[@name="math"]/MultiplyDouble/*'>
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    /// <include file='include.xml'
path='docs/members[@name="math"]/DivideInt/*'>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    /// <include file='include.xml'
path='docs/members[@name="math"]/DivideDouble/*'>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}

```

- O atributo `file` representa o nome do arquivo XML que contém a documentação.
- O atributo `path` representa uma consulta `XPath` para o `tag name` presente no `file` especificado.
- O atributo `name` representa o especificador de nome na marca que precede os comentários.
- O atributo `id`, que pode ser usado no lugar de `name`, representa a ID da marca que precede os comentários.

# Erros do compilador de C#

Artigo • 07/04/2023

Alguns erros do compilador do C# têm tópicos correspondentes que explicam por que o erro é gerado e, em alguns casos, como corrigir o erro. Use uma das etapas a seguir para ver se a ajuda está disponível para uma mensagem de erro específica.

- Se você estiver usando o Visual Studio, escolha o número do erro (por exemplo, CS0029) na [Janela de Saída](#) e, em seguida, pressione a tecla F1.
- Digite o número de erro na caixa *Filtrar por título* no sumário.

Se nenhuma dessas etapas levar a informações sobre o erro, vá para o fim desta página e envie comentários que incluem o número ou o texto do erro.

Para obter informações sobre como configurar opções de aviso e erro no C#, consulte [Opções do compilador de C#](#) ou [Página de Build, Designer de Projeto \(C#\)](#) do Visual Studio.

## ⓘ Observação

Seu computador pode mostrar diferentes nomes ou locais para alguns dos elementos de interface do usuário do Visual Studio nas instruções a seguir. A edição do Visual Studio que você possui e as configurações que você usa determinam esses elementos. Para obter mais informações, consulte [Personalizando o IDE](#).

## Confira também

- [Opções do compilador de C#](#)
- [Página de Build, Designer de Projeto \(C#\)](#)
- [WarningLevel \(opções do compilador do C#\)](#)
- [NoWarn \(opções do compilador do C#\)](#)

# Detailed table of contents

Article • 06/08/2023

- [Foreword](#)
- [Introduction](#)
- [§1 Scope](#)
- [§2 Normative references](#)
- [§3 Terms and definitions](#)
- [§4 General description](#)
- [§5 Conformance](#)
- [§6 Lexical structure](#)
  - [§6.1 Programs](#)
  - [§6.2 Grammars](#)
    - [§6.2.1 General](#)
    - [§6.2.2 Grammar notation](#)
    - [§6.2.3 Lexical grammar](#)
    - [§6.2.4 Syntactic grammar](#)
    - [§6.2.5 Grammar ambiguities](#)
  - [§6.3 Lexical analysis](#)
    - [§6.3.1 General](#)
    - [§6.3.2 Line terminators](#)
    - [§6.3.3 Comments](#)
    - [§6.3.4 White space](#)
  - [§6.4 Tokens](#)
    - [§6.4.1 General](#)
    - [§6.4.2 Unicode character escape sequences](#)
    - [§6.4.3 Identifiers](#)
    - [§6.4.4 Keywords](#)
    - [§6.4.5 Literals](#)
      - [§6.4.5.1 General](#)
      - [§6.4.5.2 Boolean literals](#)
      - [§6.4.5.3 Integer literals](#)
      - [§6.4.5.4 Real literals](#)
      - [§6.4.5.5 Character literals](#)
      - [§6.4.5.6 String literals](#)
      - [§6.4.5.7 The null literal](#)
    - [§6.4.6 Operators and punctuators](#)
  - [§6.5 Pre-processing directives](#)
    - [§6.5.1 General](#)
    - [§6.5.2 Conditional compilation symbols](#)

- [§6.5.3](#) Pre-processing expressions
- [§6.5.4](#) Definition directives
- [§6.5.5](#) Conditional compilation directives
- [§6.5.6](#) Diagnostic directives
- [§6.5.7](#) Region directives
- [§6.5.8](#) Line directives
- [§6.5.9](#) Pragma directives
- [§7](#) Basic concepts
  - [§7.1](#) Application startup
  - [§7.2](#) Application termination
  - [§7.3](#) Declarations
  - [§7.4](#) Members
    - [§7.4.1](#) General
    - [§7.4.2](#) Namespace members
    - [§7.4.3](#) Struct members
    - [§7.4.4](#) Enumeration members
    - [§7.4.5](#) Class members
    - [§7.4.6](#) Interface members
    - [§7.4.7](#) Array members
    - [§7.4.8](#) Delegate members
  - [§7.5](#) Member access
    - [§7.5.1](#) General
    - [§7.5.2](#) Declared accessibility
    - [§7.5.3](#) Accessibility domains
    - [§7.5.4](#) Protected access
    - [§7.5.5](#) Accessibility constraints
  - [§7.6](#) Signatures and overloading
  - [§7.7](#) Scopes
    - [§7.7.1](#) General
    - [§7.7.2](#) Name hiding
      - [§7.7.2.1](#) General
      - [§7.7.2.2](#) Hiding through nesting
      - [§7.7.2.3](#) Hiding through inheritance
  - [§7.8](#) Namespace and type names
    - [§7.8.1](#) General
    - [§7.8.2](#) Unqualified names
    - [§7.8.3](#) Fully qualified names
  - [§7.9](#) Automatic memory management
  - [§7.10](#) Execution order
- [§8](#) Types

- [§8.1 General](#)
- [§8.2 Reference types](#)
  - [§8.2.1 General](#)
  - [§8.2.2 Class types](#)
  - [§8.2.3 The object type](#)
  - [§8.2.4 The dynamic type](#)
  - [§8.2.5 The string type](#)
  - [§8.2.6 Interface types](#)
  - [§8.2.7 Array types](#)
  - [§8.2.8 Delegate types](#)
- [§8.3 Value types](#)
  - [§8.3.1 General](#)
  - [§8.3.2 The System.ValueType type](#)
  - [§8.3.3 Default constructors](#)
  - [§8.3.4 Struct types](#)
  - [§8.3.5 Simple types](#)
  - [§8.3.6 Integral types](#)
  - [§8.3.7 Floating-point types](#)
  - [§8.3.8 The Decimal type](#)
  - [§8.3.9 The Bool type](#)
  - [§8.3.10 Enumeration types](#)
  - [§8.3.11 Tuple types](#)
  - [§8.3.12 Nullable value types](#)
  - [§8.3.13 Boxing and unboxing](#)
- [§8.4 Constructed types](#)
  - [§8.4.1 General](#)
  - [§8.4.2 Type arguments](#)
  - [§8.4.3 Open and closed types](#)
  - [§8.4.4 Bound and unbound types](#)
  - [§8.4.5 Satisfying constraints](#)
- [§8.5 Type parameters](#)
- [§8.6 Expression tree types](#)
- [§8.7 The dynamic type](#)
- [§8.8 Unmanaged types](#)
- [§9 Variables](#)
  - [§9.1 General](#)
  - [§9.2 Variable categories](#)
    - [§9.2.1 General](#)
    - [§9.2.2 Static variables](#)
    - [§9.2.3 Instance variables](#)

- [§9.2.3.1 General](#)
- [§9.2.3.2 Instance variables in classes](#)
- [§9.2.3.3 Instance variables in structs](#)
- [§9.2.4 Array elements](#)
- [§9.2.5 Value parameters](#)
- [§9.2.6 Reference parameters](#)
- [§9.2.7 Output parameters](#)
- [§9.2.8 Input parameters](#)
- [§9.2.9 Local variables](#)
  - [§9.2.9.1 Discards](#)
- [§9.3 Default values](#)
- [§9.4 Definite assignment](#)
  - [§9.4.1 General](#)
  - [§9.4.2 Initially assigned variables](#)
  - [§9.4.3 Initially unassigned variables](#)
  - [§9.4.4 Precise rules for determining definite assignment](#)
    - [§9.4.4.1 General](#)
    - [§9.4.4.2 General rules for statements](#)
    - [§9.4.4.3 Block statements, checked, and unchecked statements](#)
    - [§9.4.4.4 Expression statements](#)
    - [§9.4.4.5 Declaration statements](#)
    - [§9.4.4.6 If statements](#)
    - [§9.4.4.7 Switch statements](#)
    - [§9.4.4.8 While statements](#)
    - [§9.4.4.9 Do statements](#)
    - [§9.4.4.10 For statements](#)
    - [§9.4.4.11 Break, continue, and goto statements](#)
    - [§9.4.4.12 Throw statements](#)
    - [§9.4.4.13 Return statements](#)
    - [§9.4.4.14 Try-catch statements](#)
    - [§9.4.4.15 Try-finally statements](#)
    - [§9.4.4.16 Try-catch-finally statements](#)
    - [§9.4.4.17 Foreach statements](#)
    - [§9.4.4.18 Using statements](#)
    - [§9.4.4.19 Lock statements](#)
    - [§9.4.4.20 Yield statements](#)
    - [§9.4.4.21 General rules for constant expressions](#)
    - [§9.4.4.22 General rules for simple expressions](#)
    - [§9.4.4.23 General rules for expressions with embedded expressions](#)
    - [§9.4.4.24 Invocation expressions and object creation expressions](#)

- [§9.4.4.25](#) Simple assignment expressions
- [§9.4.4.26](#) && expressions
- [§9.4.4.27](#) || expressions
- [§9.4.4.28](#) ! expressions
- [§9.4.4.29](#) ?? expressions
- [§9.4.4.30](#) ?: expressions
- [§9.4.4.31](#) Anonymous functions
- [§9.4.4.32](#) Throw expressions
- [§9.4.4.33](#) Rules for variables in local functions
- [§9.4.4.34](#) is-pattern expressions
- [§9.5](#) Variable references
- [§9.6](#) Atomicity of variable references
- [§9.7](#) Reference variables and returns
  - [§9.7.1](#) General
  - [§9.7.2](#) Ref safe contexts
    - [§9.7.2.1](#) General
    - [§9.7.2.2](#) Local variable ref safe context
    - [§9.7.2.3](#) Parameter ref safe context
    - [§9.7.2.4](#) Field ref safe context
    - [§9.7.2.5](#) Operators
    - [§9.7.2.6](#) Function invocation
    - [§9.7.2.7](#) Values
    - [§9.7.2.8](#) Constructor invocations
    - [§9.7.2.9](#) Limitations on reference variables
- [§10](#) Conversions
  - [§10.1](#) General
  - [§10.2](#) Implicit conversions
    - [§10.2.1](#) General
    - [§10.2.2](#) Identity conversion
    - [§10.2.3](#) Implicit numeric conversions
    - [§10.2.4](#) Implicit enumeration conversions
    - [§10.2.5](#) Implicit interpolated string conversions
    - [§10.2.6](#) Implicit nullable conversions
    - [§10.2.7](#) Null literal conversions
    - [§10.2.8](#) Implicit reference conversions
    - [§10.2.9](#) Boxing conversions
    - [§10.2.10](#) Implicit dynamic conversions
    - [§10.2.11](#) Implicit constant expression conversions
    - [§10.2.12](#) Implicit conversions involving type parameters
    - [§10.2.13](#) Implicit tuple conversions

- [§10.2.14 User-defined implicit conversions](#)
- [§10.2.15 Anonymous function conversions and method group conversions](#)
- [§10.2.16 Default literal conversions](#)
- [§10.2.17 Implicit throw conversions](#)
- [§10.3 Explicit conversions](#)
  - [§10.3.1 General](#)
  - [§10.3.2 Explicit numeric conversions](#)
  - [§10.3.3 Explicit enumeration conversions](#)
  - [§10.3.4 Explicit nullable conversions](#)
  - [§10.3.5 Explicit reference conversions](#)
  - [§10.3.6 Explicit tuple conversions](#)
  - [§10.3.7 Unboxing conversions](#)
  - [§10.3.8 Explicit dynamic conversions](#)
  - [§10.3.9 Explicit conversions involving type parameters](#)
  - [§10.3.10 User-defined explicit conversions](#)
- [§10.4 Standard conversions](#)
  - [§10.4.1 General](#)
  - [§10.4.2 Standard implicit conversions](#)
  - [§10.4.3 Standard explicit conversions](#)
- [§10.5 User-defined conversions](#)
  - [§10.5.1 General](#)
  - [§10.5.2 Permitted user-defined conversions](#)
  - [§10.5.3 Evaluation of user-defined conversions](#)
  - [§10.5.4 User-defined implicit conversions](#)
  - [§10.5.5 User-defined explicit conversions](#)
- [§10.6 Conversions involving nullable types](#)
  - [§10.6.1 Nullable Conversions](#)
  - [§10.6.2 Lifted conversions](#)
- [§10.7 Anonymous function conversions](#)
  - [§10.7.1 General](#)
  - [§10.7.2 Evaluation of anonymous function conversions to delegate types](#)
  - [§10.7.3 Evaluation of lambda expression conversions to expression tree types](#)
- [§10.8 Method group conversions](#)
- [§11 Patterns and pattern matching](#)
  - [§11.1 General](#)
  - [§11.2 Pattern forms](#)
    - [§11.2.1 General](#)
    - [§11.2.2 Declaration pattern](#)
    - [§11.2.3 Constant pattern](#)
    - [§11.2.4 Var pattern](#)

- [§11.3 Pattern subsumption](#)
- [§11.4 Pattern exhaustiveness](#)
- [§12 Expressions](#)
  - [§12.1 General](#)
  - [§12.2 Expression classifications](#)
    - [§12.2.1 General](#)
    - [§12.2.2 Values of expressions](#)
  - [§12.3 Static and Dynamic Binding](#)
    - [§12.3.1 General](#)
    - [§12.3.2 Binding-time](#)
    - [§12.3.3 Dynamic binding](#)
    - [§12.3.4 Types of subexpressions](#)
  - [§12.4 Operators](#)
    - [§12.4.1 General](#)
    - [§12.4.2 Operator precedence and associativity](#)
    - [§12.4.3 Operator overloading](#)
    - [§12.4.4 Unary operator overload resolution](#)
    - [§12.4.5 Binary operator overload resolution](#)
    - [§12.4.6 Candidate user-defined operators](#)
    - [§12.4.7 Numeric promotions](#)
      - [§12.4.7.1 General](#)
      - [§12.4.7.2 Unary numeric promotions](#)
      - [§12.4.7.3 Binary numeric promotions](#)
    - [§12.4.8 Lifted operators](#)
  - [§12.5 Member lookup](#)
    - [§12.5.1 General](#)
    - [§12.5.2 Base types](#)
  - [§12.6 Function members](#)
    - [§12.6.1 General](#)
    - [§12.6.2 Argument lists](#)
      - [§12.6.2.1 General](#)
      - [§12.6.2.2 Corresponding parameters](#)
      - [§12.6.2.3 Run-time evaluation of argument lists](#)
    - [§12.6.3 Type inference](#)
      - [§12.6.3.1 General](#)
      - [§12.6.3.2 The first phase](#)
      - [§12.6.3.3 The second phase](#)
      - [§12.6.3.4 Input types](#)
      - [§12.6.3.5 Output types](#)
      - [§12.6.3.6 Dependence](#)

- [§12.6.3.7](#) Output type inferences
- [§12.6.3.8](#) Explicit parameter type inferences
- [§12.6.3.9](#) Exact inferences
- [§12.6.3.10](#) Lower-bound inferences
- [§12.6.3.11](#) Upper-bound inferences
- [§12.6.3.12](#) Fixing
- [§12.6.3.13](#) Inferred return type
- [§12.6.3.14](#) Type inference for conversion of method groups
- [§12.6.3.15](#) Finding the best common type of a set of expressions
- [§12.6.4](#) Overload resolution
  - [§12.6.4.1](#) General
  - [§12.6.4.2](#) Applicable function member
  - [§12.6.4.3](#) Better function member
  - [§12.6.4.4](#) Better parameter-passing mode
  - [§12.6.4.5](#) Better conversion from expression
  - [§12.6.4.6](#) Exactly matching expression
  - [§12.6.4.7](#) Better conversion target
  - [§12.6.4.8](#) Overloading in generic classes
- [§12.6.5](#) Compile-time checking of dynamic member invocation
- [§12.6.6](#) Function member invocation
  - [§12.6.6.1](#) General
  - [§12.6.6.2](#) Invocations on boxed instances
- [§12.7](#) Deconstruction
- [§12.8](#) Primary expressions
  - [§12.8.1](#) General
  - [§12.8.2](#) Literals
  - [§12.8.3](#) Interpolated string expressions
  - [§12.8.4](#) Simple names
  - [§12.8.5](#) Parenthesized expressions
  - [§12.8.6](#) Tuple expressions
  - [§12.8.7](#) Member access
    - [§12.8.7.1](#) General
    - [§12.8.7.2](#) Identical simple names and type names
  - [§12.8.8](#) Null Conditional Member Access
  - [§12.8.9](#) Invocation expressions
    - [§12.8.9.1](#) General
    - [§12.8.9.2](#) Method invocations
    - [§12.8.9.3](#) Extension method invocations
    - [§12.8.9.4](#) Delegate invocations
  - [§12.8.10](#) Null Conditional Invocation Expression

- [§12.8.11 Element access](#)
  - [§12.8.11.1 General](#)
  - [§12.8.11.2 Array access](#)
  - [§12.8.11.3 Indexer access](#)
- [§12.8.12 Null Conditional Element Access](#)
- [§12.8.13 This access](#)
- [§12.8.14 Base access](#)
- [§12.8.15 Postfix increment and decrement operators](#)
- [§12.8.16 The new operator](#)
  - [§12.8.16.1 General](#)
  - [§12.8.16.2 Object creation expressions](#)
  - [§12.8.16.3 Object initializers](#)
  - [§12.8.16.4 Collection initializers](#)
  - [§12.8.16.5 Array creation expressions](#)
  - [§12.8.16.6 Delegate creation expressions](#)
  - [§12.8.16.7 Anonymous object creation expressions](#)
- [§12.8.17 The typeof operator](#)
- [§12.8.18 The sizeof operator](#)
- [§12.8.19 The checked and unchecked operators](#)
- [§12.8.20 Default value expressions](#)
- [§12.8.21 Stack allocation](#)
- [§12.8.22 nameof expressions](#)
- [§12.8.23 Anonymous method expressions](#)
- [§12.9 Unary operators](#)
  - [§12.9.1 General](#)
  - [§12.9.2 Unary plus operator](#)
  - [§12.9.3 Unary minus operator](#)
  - [§12.9.4 Logical negation operator](#)
  - [§12.9.5 Bitwise complement operator](#)
  - [§12.9.6 Prefix increment and decrement operators](#)
  - [§12.9.7 Cast expressions](#)
  - [§12.9.8 Await expressions](#)
    - [§12.9.8.1 General](#)
    - [§12.9.8.2 Awaitable expressions](#)
    - [§12.9.8.3 Classification of await expressions](#)
    - [§12.9.8.4 Run-time evaluation of await expressions](#)
- [§12.10 Arithmetic operators](#)
  - [§12.10.1 General](#)
  - [§12.10.2 Multiplication operator](#)
  - [§12.10.3 Division operator](#)

- [§12.10.4 Remainder operator](#)
- [§12.10.5 Addition operator](#)
- [§12.10.6 Subtraction operator](#)
- [§12.11 Shift operators](#)
- [§12.12 Relational and type-testing operators](#)
  - [§12.12.1 General](#)
  - [§12.12.2 Integer comparison operators](#)
  - [§12.12.3 Floating-point comparison operators](#)
  - [§12.12.4 Decimal comparison operators](#)
  - [§12.12.5 Boolean equality operators](#)
  - [§12.12.6 Enumeration comparison operators](#)
  - [§12.12.7 Reference type equality operators](#)
  - [§12.12.8 String equality operators](#)
  - [§12.12.9 Delegate equality operators](#)
  - [§12.12.10 Equality operators between nullable value types and the null literal](#)
  - [§12.12.11 Tuple equality operators](#)
  - [§12.12.12 The is operator](#)
    - [§12.12.12.1 The is-type operator](#)
    - [§12.12.12.2 The is-pattern operator](#)
  - [§12.12.13 The as operator](#)
- [§12.13 Logical operators](#)
  - [§12.13.1 General](#)
  - [§12.13.2 Integer logical operators](#)
  - [§12.13.3 Enumeration logical operators](#)
  - [§12.13.4 Boolean logical operators](#)
  - [§12.13.5 Nullable Boolean & and | operators](#)
- [§12.14 Conditional logical operators](#)
  - [§12.14.1 General](#)
  - [§12.14.2 Boolean conditional logical operators](#)
  - [§12.14.3 User-defined conditional logical operators](#)
- [§12.15 The null coalescing operator](#)
- [§12.16 The throw expression operator](#)
- [§12.17 Declaration expressions](#)
- [§12.18 Conditional operator](#)
- [§12.19 Anonymous function expressions](#)
  - [§12.19.1 General](#)
  - [§12.19.2 Anonymous function signatures](#)
  - [§12.19.3 Anonymous function bodies](#)
  - [§12.19.4 Overload resolution](#)
  - [§12.19.5 Anonymous functions and dynamic binding](#)

- [§12.19.6 Outer variables](#)
  - [§12.19.6.1 General](#)
  - [§12.19.6.2 Captured outer variables](#)
  - [§12.19.6.3 Instantiation of local variables](#)
- [§12.19.7 Evaluation of anonymous function expressions](#)
- [§12.19.8 Implementation Example](#)
- [§12.20 Query expressions](#)
  - [§12.20.1 General](#)
  - [§12.20.2 Ambiguities in query expressions](#)
  - [§12.20.3 Query expression translation](#)
    - [§12.20.3.1 General](#)
    - [§12.20.3.2 Query expressions with continuations](#)
    - [§12.20.3.3 Explicit range variable types](#)
    - [§12.20.3.4 Degenerate query expressions](#)
    - [§12.20.3.5 From, let, where, join and orderby clauses](#)
    - [§12.20.3.6 Select clauses](#)
    - [§12.20.3.7 Group clauses](#)
    - [§12.20.3.8 Transparent identifiers](#)
  - [§12.20.4 The query-expression pattern](#)
- [§12.21 Assignment operators](#)
  - [§12.21.1 General](#)
  - [§12.21.2 Simple assignment](#)
  - [§12.21.3 Ref assignment](#)
  - [§12.21.4 Compound assignment](#)
  - [§12.21.5 Event assignment](#)
- [§12.22 Expression](#)
- [§12.23 Constant expressions](#)
- [§12.24 Boolean expressions](#)
- [§13 Statements](#)
  - [§13.1 General](#)
  - [§13.2 End points and reachability](#)
  - [§13.3 Blocks](#)
    - [§13.3.1 General](#)
    - [§13.3.2 Statement lists](#)
  - [§13.4 The empty statement](#)
  - [§13.5 Labeled statements](#)
  - [§13.6 Declaration statements](#)
    - [§13.6.1 General](#)
    - [§13.6.2 Local variable declarations](#)
    - [§13.6.3 Local constant declarations](#)

- [§13.6.4 Local function declarations](#)
- [§13.7 Expression statements](#)
- [§13.8 Selection statements](#)
  - [§13.8.1 General](#)
  - [§13.8.2 The if statement](#)
  - [§13.8.3 The switch statement](#)
- [§13.9 Iteration statements](#)
  - [§13.9.1 General](#)
  - [§13.9.2 The while statement](#)
  - [§13.9.3 The do statement](#)
  - [§13.9.4 The for statement](#)
  - [§13.9.5 The foreach statement](#)
- [§13.10 Jump statements](#)
  - [§13.10.1 General](#)
  - [§13.10.2 The break statement](#)
  - [§13.10.3 The continue statement](#)
  - [§13.10.4 The goto statement](#)
  - [§13.10.5 The return statement](#)
  - [§13.10.6 The throw statement](#)
- [§13.11 The try statement](#)
- [§13.12 The checked and unchecked statements](#)
- [§13.13 The lock statement](#)
- [§13.14 The using statement](#)
- [§13.15 The yield statement](#)
- [§14 Namespaces](#)
  - [§14.1 General](#)
  - [§14.2 Compilation units](#)
  - [§14.3 Namespace declarations](#)
  - [§14.4 Extern alias directives](#)
  - [§14.5 Using directives](#)
    - [§14.5.1 General](#)
    - [§14.5.2 Using alias directives](#)
    - [§14.5.3 Using namespace directives](#)
    - [§14.5.4 Using static directives](#)
  - [§14.6 Namespace member declarations](#)
  - [§14.7 Type declarations](#)
  - [§14.8 Qualified alias member](#)
    - [§14.8.1 General](#)
    - [§14.8.2 Uniqueness of aliases](#)
- [§15 Classes](#)

- [§15.1 General](#)
- [§15.2 Class declarations](#)
  - [§15.2.1 General](#)
  - [§15.2.2 Class modifiers](#)
    - [§15.2.2.1 General](#)
    - [§15.2.2.2 Abstract classes](#)
    - [§15.2.2.3 Sealed classes](#)
    - [§15.2.2.4 Static classes](#)
      - [§15.2.2.4.1 General](#)
      - [§15.2.2.4.2 Referencing static class types](#)
  - [§15.2.3 Type parameters](#)
  - [§15.2.4 Class base specification](#)
    - [§15.2.4.1 General](#)
    - [§15.2.4.2 Base classes](#)
    - [§15.2.4.3 Interface implementations](#)
  - [§15.2.5 Type parameter constraints](#)
  - [§15.2.6 Class body](#)
  - [§15.2.7 Partial declarations](#)
  - [§15.3 Class members](#)
    - [§15.3.1 General](#)
    - [§15.3.2 The instance type](#)
    - [§15.3.3 Members of constructed types](#)
    - [§15.3.4 Inheritance](#)
    - [§15.3.5 The new modifier](#)
    - [§15.3.6 Access modifiers](#)
    - [§15.3.7 Constituent types](#)
    - [§15.3.8 Static and instance members](#)
    - [§15.3.9 Nested types](#)
      - [§15.3.9.1 General](#)
      - [§15.3.9.2 Fully qualified name](#)
      - [§15.3.9.3 Declared accessibility](#)
      - [§15.3.9.4 Hiding](#)
      - [§15.3.9.5 this access](#)
      - [§15.3.9.6 Access to private and protected members of the containing type](#)
      - [§15.3.9.7 Nested types in generic classes](#)
    - [§15.3.10 Reserved member names](#)
      - [§15.3.10.1 General](#)
      - [§15.3.10.2 Member names reserved for properties](#)
      - [§15.3.10.3 Member names reserved for events](#)
      - [§15.3.10.4 Member names reserved for indexers](#)

- [§15.3.10.5](#) Member names reserved for finalizers
- [§15.4](#) Constants
- [§15.5](#) Fields
  - [§15.5.1](#) General
  - [§15.5.2](#) Static and instance fields
  - [§15.5.3](#) Readonly fields
    - [§15.5.3.1](#) General
    - [§15.5.3.2](#) Using static readonly fields for constants
    - [§15.5.3.3](#) Versioning of constants and static readonly fields
  - [§15.5.4](#) Volatile fields
  - [§15.5.5](#) Field initialization
  - [§15.5.6](#) Variable initializers
    - [§15.5.6.1](#) General
    - [§15.5.6.2](#) Static field initialization
    - [§15.5.6.3](#) Instance field initialization
- [§15.6](#) Methods
  - [§15.6.1](#) General
  - [§15.6.2](#) Method parameters
    - [§15.6.2.1](#) General
    - [§15.6.2.2](#) Value parameters
    - [§15.6.2.3](#) Input parameters
    - [§15.6.2.4](#) Reference parameters
    - [§15.6.2.5](#) Output parameters
    - [§15.6.2.6](#) Parameter arrays
  - [§15.6.3](#) Static and instance methods
  - [§15.6.4](#) Virtual methods
  - [§15.6.5](#) Override methods
  - [§15.6.6](#) Sealed methods
  - [§15.6.7](#) Abstract methods
  - [§15.6.8](#) External methods
  - [§15.6.9](#) Partial methods
  - [§15.6.10](#) Extension methods
  - [§15.6.11](#) Method body
- [§15.7](#) Properties
  - [§15.7.1](#) General
  - [§15.7.2](#) Static and instance properties
  - [§15.7.3](#) Accessors
  - [§15.7.4](#) Automatically implemented properties
  - [§15.7.5](#) Accessibility
  - [§15.7.6](#) Virtual, sealed, override, and abstract accessors

- [§15.8 Events](#)
  - [§15.8.1 General](#)
  - [§15.8.2 Field-like events](#)
  - [§15.8.3 Event accessors](#)
  - [§15.8.4 Static and instance events](#)
  - [§15.8.5 Virtual, sealed, override, and abstract accessors](#)
- [§15.9 Indexers](#)
  - [§15.9.1 General](#)
  - [§15.9.2 Indexer and Property Differences](#)
- [§15.10 Operators](#)
  - [§15.10.1 General](#)
  - [§15.10.2 Unary operators](#)
  - [§15.10.3 Binary operators](#)
  - [§15.10.4 Conversion operators](#)
- [§15.11 Instance constructors](#)
  - [§15.11.1 General](#)
  - [§15.11.2 Constructor initializers](#)
  - [§15.11.3 Instance variable initializers](#)
  - [§15.11.4 Constructor execution](#)
  - [§15.11.5 Default constructors](#)
- [§15.12 Static constructors](#)
- [§15.13 Finalizers](#)
- [§15.14 Iterators](#)
  - [§15.14.1 General](#)
  - [§15.14.2 Enumerator interfaces](#)
  - [§15.14.3 Enumerable interfaces](#)
  - [§15.14.4 Yield type](#)
  - [§15.14.5 Enumerator objects](#)
    - [§15.14.5.1 General](#)
    - [§15.14.5.2 The MoveNext method](#)
    - [§15.14.5.3 The Current property](#)
    - [§15.14.5.4 The Dispose method](#)
  - [§15.14.6 Enumerable objects](#)
    - [§15.14.6.1 General](#)
    - [§15.14.6.2 The GetEnumerator method](#)
- [§15.15 Async Functions](#)
  - [§15.15.1 General](#)
  - [§15.15.2 Task-type builder pattern](#)
  - [§15.15.3 Evaluation of a task-returning async function](#)
  - [§15.15.4 Evaluation of a void-returning async function](#)

- [§16 Structs](#)
  - [§16.1 General](#)
  - [§16.2 Struct declarations](#)
    - [§16.2.1 General](#)
    - [§16.2.2 Struct modifiers](#)
    - [§16.2.3 Ref modifier](#)
    - [§16.2.4 Partial modifier](#)
    - [§16.2.5 Struct interfaces](#)
    - [§16.2.6 Struct body](#)
  - [§16.3 Struct members](#)
  - [§16.4 Class and struct differences](#)
    - [§16.4.1 General](#)
    - [§16.4.2 Value semantics](#)
    - [§16.4.3 Inheritance](#)
    - [§16.4.4 Assignment](#)
    - [§16.4.5 Default values](#)
    - [§16.4.6 Boxing and unboxing](#)
    - [§16.4.7 Meaning of this](#)
    - [§16.4.8 Field initializers](#)
    - [§16.4.9 Constructors](#)
    - [§16.4.10 Static constructors](#)
    - [§16.4.11 Automatically implemented properties](#)
    - [§16.4.12 Safe context constraint](#)
      - [§16.4.12.1 General](#)
      - [§16.4.12.2 Parameter safe context](#)
      - [§16.4.12.3 Local variable safe context](#)
      - [§16.4.12.4 Field safe context](#)
      - [§16.4.12.5 Operators](#)
      - [§16.4.12.6 Method and property invocation](#)
      - [§16.4.12.7 stackalloc](#)
      - [§16.4.12.8 Constructor invocations](#)
- [§17 Arrays](#)
  - [§17.1 General](#)
  - [§17.2 Array types](#)
    - [§17.2.1 General](#)
    - [§17.2.2 The System.Array type](#)
    - [§17.2.3 Arrays and the generic collection interfaces](#)
  - [§17.3 Array creation](#)
  - [§17.4 Array element access](#)
  - [§17.5 Array members](#)

- [§17.6](#) Array covariance
- [§17.7](#) Array initializers
- [§18](#) Interfaces
  - [§18.1](#) General
  - [§18.2](#) Interface declarations
    - [§18.2.1](#) General
    - [§18.2.2](#) Interface modifiers
    - [§18.2.3](#) Variant type parameter lists
      - [§18.2.3.1](#) General
      - [§18.2.3.2](#) Variance safety
      - [§18.2.3.3](#) Variance conversion
    - [§18.2.4](#) Base interfaces
  - [§18.3](#) Interface body
  - [§18.4](#) Interface members
    - [§18.4.1](#) General
    - [§18.4.2](#) Interface methods
    - [§18.4.3](#) Interface properties
    - [§18.4.4](#) Interface events
    - [§18.4.5](#) Interface indexers
    - [§18.4.6](#) Interface member access
  - [§18.5](#) Qualified interface member names
  - [§18.6](#) Interface implementations
    - [§18.6.1](#) General
    - [§18.6.2](#) Explicit interface member implementations
    - [§18.6.3](#) Uniqueness of implemented interfaces
    - [§18.6.4](#) Implementation of generic methods
    - [§18.6.5](#) Interface mapping
    - [§18.6.6](#) Interface implementation inheritance
    - [§18.6.7](#) Interface re-implementation
    - [§18.6.8](#) Abstract classes and interfaces
- [§19](#) Enums
  - [§19.1](#) General
  - [§19.2](#) Enum declarations
  - [§19.3](#) Enum modifiers
  - [§19.4](#) Enum members
  - [§19.5](#) The System.Enum type
  - [§19.6](#) Enum values and operations
- [§20](#) Delegates
  - [§20.1](#) General
  - [§20.2](#) Delegate declarations

- [§20.3 Delegate members](#)
- [§20.4 Delegate compatibility](#)
- [§20.5 Delegate instantiation](#)
- [§20.6 Delegate invocation](#)
- [§21 Exceptions](#)
  - [§21.1 General](#)
  - [§21.2 Causes of exceptions](#)
  - [§21.3 The System.Exception class](#)
  - [§21.4 How exceptions are handled](#)
  - [§21.5 Common exception classes](#)
- [§22 Attributes](#)
  - [§22.1 General](#)
  - [§22.2 Attribute classes](#)
    - [§22.2.1 General](#)
    - [§22.2.2 Attribute usage](#)
    - [§22.2.3 Positional and named parameters](#)
    - [§22.2.4 Attribute parameter types](#)
  - [§22.3 Attribute specification](#)
  - [§22.4 Attribute instances](#)
    - [§22.4.1 General](#)
    - [§22.4.2 Compilation of an attribute](#)
    - [§22.4.3 Run-time retrieval of an attribute instance](#)
  - [§22.5 Reserved attributes](#)
    - [§22.5.1 General](#)
    - [§22.5.2 The AttributeUsage attribute](#)
    - [§22.5.3 The Conditional attribute](#)
      - [§22.5.3.1 General](#)
      - [§22.5.3.2 Conditional methods](#)
      - [§22.5.3.3 Conditional attribute classes](#)
    - [§22.5.4 The Obsolete attribute](#)
    - [§22.5.5 Caller-info attributes](#)
      - [§22.5.5.1 General](#)
      - [§22.5.5.2 The CallerLineNumber attribute](#)
      - [§22.5.5.3 The CallerFilePath attribute](#)
      - [§22.5.5.4 The CallerMemberName attribute](#)
  - [§22.6 Attributes for interoperation](#)
- [§23 Unsafe code](#)
  - [§23.1 General](#)
  - [§23.2 Unsafe contexts](#)
  - [§23.3 Pointer types](#)

- [§23.4](#) Fixed and moveable variables
- [§23.5](#) Pointer conversions
  - [§23.5.1](#) General
  - [§23.5.2](#) Pointer arrays
- [§23.6](#) Pointers in expressions
  - [§23.6.1](#) General
  - [§23.6.2](#) Pointer indirection
  - [§23.6.3](#) Pointer member access
  - [§23.6.4](#) Pointer element access
  - [§23.6.5](#) The address-of operator
  - [§23.6.6](#) Pointer increment and decrement
  - [§23.6.7](#) Pointer arithmetic
  - [§23.6.8](#) Pointer comparison
  - [§23.6.9](#) The sizeof operator
- [§23.7](#) The fixed statement
- [§23.8](#) Fixed-size buffers
  - [§23.8.1](#) General
  - [§23.8.2](#) Fixed-size buffer declarations
  - [§23.8.3](#) Fixed-size buffers in expressions
  - [§23.8.4](#) Definite assignment checking
- [§23.9](#) Stack allocation
- [§A](#) Grammar
  - [§A.1](#) General
  - [§A.2](#) Lexical grammar
  - [§A.3](#) Syntactic grammar
  - [§A.4](#) Grammar extensions for unsafe code
- [§B](#) Portability issues
  - [§B.1](#) General
  - [§B.2](#) Undefined behavior
  - [§B.3](#) Implementation-defined behavior
  - [§B.4](#) Unspecified behavior
  - [§B.5](#) Other issues
- [§C](#) Standard library
  - [§C.1](#) General
  - [§C.2](#) Standard Library Types defined in ISO/IEC 23271
  - [§C.3](#) Standard Library Types not defined in ISO/IEC 23271
  - [§C.4](#) Format Specifications
  - [§C.5](#) Library Type Abbreviations
- [§D](#) Documentation comments
  - [§D.1](#) General

- [§D.2](#) Introduction
- [§D.3](#) Recommended tags
  - [§D.3.1](#) General
  - [§D.3.2](#) <c>
  - [§D.3.3](#) <code>
  - [§D.3.4](#) <example>
  - [§D.3.5](#) <exception>
  - [§D.3.6](#) <include>
  - [§D.3.7](#) <list>
  - [§D.3.8](#) <para>
  - [§D.3.9](#) <param>
  - [§D.3.10](#) <paramref>
  - [§D.3.11](#) <permission>
  - [§D.3.12](#) <remarks>
  - [§D.3.13](#) <returns>
  - [§D.3.14](#) <see>
  - [§D.3.15](#) <seealso>
  - [§D.3.16](#) <summary>
  - [§D.3.17](#) <typeparam>
  - [§D.3.18](#) <typeparamref>
  - [§D.3.19](#) <value>
- [§D.4](#) Processing the documentation file
  - [§D.4.1](#) General
  - [§D.4.2](#) ID string format
  - [§D.4.3](#) ID string examples
- [§D.5](#) An example
  - [§D.5.1](#) C# source code
  - [§D.5.2](#) Resulting XML
- [§E](#) Bibliography

# Foreword

Article • 03/23/2023

This specification replaces ECMA-334:2022. Changes from the previous edition include the addition of the following:

- Binary integer literals
- Embedded digit separators in numeric literals
- Leading-digit separators in binary and hexadecimal integer literals
- `out` variables
- Discards
- Tuple types
- Pattern Matching
- `ref` locals and returns, conditional `ref` expressions, `ref` with `this` in extension methods, and reassignment of `ref` local variables
- Local Functions
- More expression-bodied members
- `throw` Expressions
- Generalized `async` return types
- `async Main` method
- `default` literal expressions
- Non-trailing named arguments
- `private protected` access modifier
- `in` parameter modifier
- `readonly` structs
- `ref` structs
- Indexing movable fixed buffer without pinning
- Initializers on `stackalloc` arrays
- Pattern-based `fixed` statements
- `System.Delegate` and `System.Enum` as *class\_type* constraints.
- Additional generic constraints
- Allow expression variables in more locations
- Attach attributes to the backing field of auto-implemented properties
- Reduce ambiguity of overload resolution

# Introdução

Artigo • 16/09/2021

O C# (pronuncia-se "C Sharp") é uma linguagem de programação simples, moderna, orientada a objeto e fortemente tipada. O C# tem suas raízes na família C de idiomas e estará imediatamente familiarizado com os programadores C, C++ e Java. O C# é padronizado pela ECMA International como o \*ECMA-334 \_ Standard e por ISO/IEC como o padrão \_ \*iso/IEC 23270\*\*. O compilador C# da Microsoft para o .NET Framework é uma implementação em conformidade desses dois padrões.

O C# é uma linguagem orientada a objeto, mas inclui ainda suporte para programação **orientada a componentes**. O design de software atual depende cada vez mais dos componentes de software na forma de pacotes independentes e autodescritivos de funcionalidade. O principal é que esses componentes apresentam um modelo de programação com propriedades, métodos e eventos; eles têm atributos que fornecem informações declarativas sobre o componente; e incorporam sua própria documentação. O c# fornece construções de linguagem para dar suporte direto a esses conceitos, tornando o C# uma linguagem muito natural para criar e usar componentes de software.

Vários recursos do C# auxiliam na construção de aplicativos robustos e duráveis: **coleta de lixo** \_ recupera automaticamente a memória ocupada por objetos não utilizados; a \_ manipulação de exceção\_ fornece uma abordagem estruturada e extensível para detecção e recuperação de erros; e o **design tipo-seguro** da linguagem torna impossível a leitura de variáveis não inicializadas, para indexar matrizes além dos limites ou para executar conversões de tipo desmarcadas.

C# tem um **sistema de tipo unificado**. Todos os tipos do C#, incluindo tipos primitivos, como `int` e `double`, herdam de um único tipo de `object` raiz. Assim, todos os tipos compartilham um conjunto de operações comuns, e valores de qualquer tipo podem ser armazenados, transportados e operados de maneira consistente. Além disso, C# oferece suporte a tipos de referência e tipos de valor definidos pelo usuário, permitindo a alocação dinâmica de objetos, bem como o armazenamento em linha de estruturas leves.

Para garantir que os programas e bibliotecas C# possam evoluir ao longo do tempo de maneira compatível, muito ênfase foi colocado no **controle de versão** no design do C#. Muitas linguagens de programação prestam pouca atenção a esse problema e, como resultado, programas escritos nessas linguagens quebram com mais frequência do que o necessário quando versões mais recentes das bibliotecas dependentes são introduzidas. Aspectos do design do C# que foram influenciados diretamente pelas

considerações de controle de versão incluem os `virtual` `override` modificadores and separados, as regras para resolução de sobrecarga de método e suporte para declarações de membro de interface explícitas.

O restante deste capítulo descreve os recursos essenciais da linguagem C#. Embora os capítulos posteriores descrevam as regras e exceções em uma maneira mais detalhada e, às vezes, matemática, este capítulo busca clareza e brevidade às custas da conclusão. A intenção é fornecer ao leitor uma introdução à linguagem que facilitará a gravação de programas antigos e a leitura de capítulos posteriores.

## Hello world

O programa "Hello, World" é usado tradicionalmente para introduzir uma linguagem de programação. Este é para C#:

```
C#  
  
using System;  
  
class Hello  
{  
    static void Main()  
    {  
        Console.WriteLine("Hello, World");  
    }  
}
```

Os arquivos de origem em C# normalmente têm a extensão de arquivo `.cs`. Supondo que o programa "Olá, mundo" esteja armazenado no arquivo `hello.cs`, o programa pode ser compilado com o compilador do Microsoft C# usando a linha de comando

```
Console  
  
csc hello.cs
```

que produz um assembly executável chamado `hello.exe`. A saída produzida por este aplicativo quando é executada é

```
Console  
  
Hello, World
```

O programa "Hello, World" começa com uma diretiva `using` que faz referência ao namespace `System`. Namespaces fornecem um meio hierárquico de organizar

bibliotecas e programas em C#. Os namespaces contêm tipos e outros namespaces — por exemplo, o namespace `System` contém uma quantidade de tipos, como a classe `Console` referenciada no programa e diversos outros namespaces, como `IO` e `Collections`. A diretiva `using` que faz referência a um determinado namespace permite o uso não qualificado dos tipos que são membros desse namespace. Devido à diretiva `using`, o programa pode usar `Console.WriteLine` como um atalho para `System.Console.WriteLine`.

A classe `Hello` declarada pelo programa "Hello, World" tem um único membro, o método chamado `Main`. O `Main` método é declarado com o `static` modificador. Embora os métodos de instância possam fazer referência a uma determinada instância de objeto delimitador usando a palavra-chave `this`, métodos estáticos operam sem referência a um objeto específico. Por convenção, um método estático denominado `Main` serve como ponto de entrada de um programa.

A saída do programa é produzida pelo método `WriteLine` da classe `Console` no namespace `System`. Essa classe é fornecida pelo .NET Framework bibliotecas de classe, que, por padrão, são automaticamente referenciadas pelo compilador do Microsoft C#. Observe que o C# em si não tem uma biblioteca de tempo de execução separada. Em vez disso, o .NET Framework é a biblioteca de tempo de execução do C#.

## Estrutura do programa

Os principais conceitos organizacionais em C# são *programas* , *namespaces* , *tipos* , *Membros* e *assemblies* ., *types* , *members* , and *\*assemblies\** . Os programas C# consistem em um ou mais arquivos de origem. Os programas declaram tipos que contêm membros e podem ser organizados em namespaces. Classes e interfaces são exemplos de tipos. Campos, métodos, propriedades e eventos são exemplos de membros. Quando os programas em C# são compilados, eles são empacotados fisicamente em assemblies. Normalmente, os assemblies têm a extensão de arquivo `.exe` ou `.dll` , dependendo se eles implementam *aplicativos* ou *\_bibliotecas* \*.

O exemplo

```
C#  
  
using System;  
  
namespace Acme.Collections  
{  
    public class Stack  
    {  
        Entry top;
```

```

public void Push(object data) {
    top = new Entry(top, data);
}

public object Pop() {
    if (top == null) throw new InvalidOperationException();
    object result = top.data;
    top = top.next;
    return result;
}

class Entry
{
    public Entry next;
    public object data;

    public Entry(Entry next, object data) {
        this.next = next;
        this.data = data;
    }
}
}

```

declara uma classe chamada `Stack` em um namespace chamado `Acme.Collections`. O nome totalmente qualificado dessa classe é `Acme.Collections.Stack`. A classe contém vários membros: um campo chamado `top`, dois métodos chamados `Push` e `Pop` e uma classe aninhada chamada `Entry`. A classe `Entry` ainda contém três membros: um campo chamado `next`, um campo chamado `data` e um construtor. Supondo que o código-fonte do exemplo seja armazenado no arquivo `acme.cs`, a linha de comando

Console

```
csc /t:library acme.cs
```

compila o exemplo como uma biblioteca (o código sem um ponto de entrada `Main`) e produz um assembly denominado `acme.dll`.

Os assemblies contêm código executável na forma de \* instruções de IL (**Intermediate Language**) e informações simbólicas na forma de *\_ Metadata* \*. Antes de ser executado, o código de IL em um assembly é automaticamente convertido em código específico do processador pelo compilador JIT (Just-In-Time) do .NET Common Language Runtime.

Como um assembly é uma unidade autodescritiva da funcionalidade que contém o código e os metadados, não é necessário de diretivas `#include` e arquivos de cabeçalho

no C#. Os tipos públicos e os membros contidos em um assembly específico são disponibilizados em um programa C# simplesmente fazendo referência a esse assembly ao compilar o programa. Por exemplo, esse programa usa a classe `Acme.Collections.Stack` do assembly `acme.dll`:

```
C#  
  
using System;  
using Acme.Collections;  
  
class Test  
{  
    static void Main()  
    {  
        Stack s = new Stack();  
        s.Push(1);  
        s.Push(10);  
        s.Push(100);  
        Console.WriteLine(s.Pop());  
        Console.WriteLine(s.Pop());  
        Console.WriteLine(s.Pop());  
    }  
}
```

Se o programa estiver armazenado no arquivo `test.cs`, quando `test.cs` for compilado, o `acme.dll` assembly poderá ser referenciado usando a opção do compilador `/r`:

```
Console  
  
csc /r:acme.dll test.cs
```

Isso cria um assembly executável denominado `test.exe`, que, quando executado, produz a saída:

```
Console  
  
100  
10  
1
```

O C# permite que o texto de origem de um programa seja armazenado em vários arquivos de origem. Quando um programa em C# com vários arquivo é compilado, todos os arquivos de origem são processados juntos e os arquivos de origem podem referenciar livremente uns aos outros. Conceitualmente, é como se todos os arquivos de origem fossem concatenados em um arquivo grande antes de serem processados.

Declarações de encaminhamento nunca são necessárias em C#, porque, com poucas exceções, a ordem de declaração é insignificante. O C# não limita um arquivo de origem para declarar somente um tipo público nem requer o nome do arquivo de origem para corresponder a um tipo declarado no arquivo de origem.

## Tipos e variáveis

Há dois tipos de tipo em C#: \*tipos de valor \_ e \_ tipos de referência \*. As variáveis de tipos de valor contêm diretamente seus dados enquanto variáveis de tipos de referência armazenam referências a seus dados, o último sendo conhecido como objetos. Com tipos de referência, é possível que duas variáveis referenciem o mesmo objeto e, portanto, é possível que operações em uma variável afetem o objeto referenciado por outra variável. Com tipos de valor, cada variável tem sua própria cópia dos dados e não é possível que operações em uma variável afetem a outra (exceto no caso de variáveis de parâmetros `ref` e `out`).

Os tipos de valor do C# são divididos em *tipos simples* , *tipos de enumeração* , *tipos de struct* e *tipos anuláveis* , e os tipos de referência do C# são divididos em *tipos de classe* , *tipos de interface* , *tipos de matriz\** e *tipos delegados* \*.

A tabela a seguir fornece uma visão geral do sistema de tipos do C#.

Categoria	Descrição	
Tipos de valor	Tipos simples	Integral com sinal: <code>sbyte</code> , <code>short</code> , <code>int</code> , <code>long</code>
		Integral sem sinal: <code>byte</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code>
		Caracteres Unicode: <code>char</code>
		Ponto flutuante IEEE: <code>float</code> , <code>double</code>
		Decimal de alta precisão: <code>decimal</code>
		Booliano: <code>bool</code>
Tipos enum	Tipos definidos pelo usuário do formulário <code>enum E { ... }</code>	
Tipos struct	Tipos definidos pelo usuário do formulário <code>struct S { ... }</code>	
Tipos anuláveis	Extensões de todos os outros tipos de valor com um valor <code>null</code>	
Tipos de referência	Tipos de aula	Classe base definitiva de todos os outros tipos: <code>object</code>

Categoria	Descrição
	Cadeia de caracteres Unicode: <code>string</code>
	Tipos definidos pelo usuário do formulário <code>class C {...}</code>
Tipos de interface	Tipos definidos pelo usuário do formulário <code>interface I {...}</code>
Tipos de matriz	Unidimensional e multidimensional, por exemplo, <code>int[]</code> e <code>int[,]</code>
Tipos delegados	Tipos definidos pelo usuário do formulário, por exemplo, <code>delegate int D(...)</code>

Os tipos integrais oito dão suporte a valores de 8 bits, 16 bits, 32 bits e 64 bits no formulário com ou sem sinal.

Os dois tipos de ponto flutuante, `float` e `double`, são representados usando os formatos IEEE 754 de precisão única de 32 bits e de 64 bits de precisão dupla.

O tipo `decimal` é um tipo de dados de 128 bits adequado para cálculos financeiros e monetários.

`bool` O tipo do C# é usado para representar valores Booleanos — valores que são `true` ou `false`.

O processamento de cadeia de caracteres e caracteres em C# usa codificação Unicode. O tipo `char` representa uma unidade de código UTF-16 e o tipo `string` representa uma sequência de unidades de código UTF-16.

A tabela a seguir resume os tipos numéricos do C#.

Categoria	Bits	Tipo	Intervalo/precisão
Integral assinada	8	<code>sbyte</code>	-128... 127
	16	<code>short</code>	-32768... 32,767
	32	<code>int</code>	-2147483648... 2,147,483,647
	64	<code>long</code>	-9.223.372.036.854.775.808... 9,223,372,036,854,775,807
Integral sem sinal	8	<code>byte</code>	0... 255
	16	<code>ushort</code>	0... 65,535
	32	<code>uint</code>	0... 4,294,967,295

Categoria	Bits	Tipo	Intervalo/precisão
	64	<code>ulong</code>	0... 18, 446, 744, 073, 709, 551, 615
Ponto flutuante	32	<code>float</code>	$1,5 \times 10^{-45}$ a $3,4 \times 10^{38}$ , precisão de 7 dígitos
	64	<code>double</code>	$5,0 \times 10^{-324}$ a $1,7 \times 10^{308}$ , precisão de 15 dígitos
Decimal	128	<code>decimal</code>	$1,0 \times 10^{-28}$ a $7,9 \times 10^{28}$ , precisão de 28 dígitos

Os programas em C# usam **declarações de tipos** para criar novos tipos. Uma declaração de tipo especifica o nome e os membros do novo tipo. Cinco das categorias de tipos do C# são definíveis pelo usuário: tipos de classe, tipos de struct, tipos de interface, tipos de enumeração e tipos delegados.

Um tipo de classe define uma estrutura de dados que contém membros de dados (campos) e membros de função (métodos, propriedades e outros). Os tipos de classe dão suporte à herança única e ao polimorfismo, mecanismos nos quais as classes derivadas podem estender e especializar as classes base.

Um tipo struct é semelhante a um tipo de classe, pois representa uma estrutura com membros de dados e membros de função. No entanto, ao contrário das classes, as structs são tipos de valor e não exigem alocação de heap. Os tipos de estrutura não dão suporte à herança especificada pelo usuário, e todos os tipos de structs são herdados implicitamente do tipo `object`.

Um tipo de interface define um contrato como um conjunto nomeado de membros da função pública. Uma classe ou struct que implementa uma interface deve fornecer implementações dos membros da função da interface. Uma interface pode herdar de várias interfaces base e uma classe ou estrutura pode implementar várias interfaces.

Um delegado é um tipo que representa referências aos métodos com uma lista de parâmetros e tipo de retorno específicos. Delegados possibilitam o tratamento de métodos como entidades que podem ser atribuídos a variáveis e passadas como parâmetros. Os delegados são parecidos com o conceito de ponteiros de função em outras linguagens, mas ao contrário dos ponteiros de função, os delegados são orientados a objetos e fortemente tipados.

Tipos de classe, struct, interface e delegado oferecem suporte a genéricos, no qual eles podem ser parametrizados com outros tipos.

Um tipo de enumeração é um tipo distinto com constantes nomeadas. Cada tipo de enumeração tem um tipo subjacente, que deve ser um dos oito tipos integrais. O

conjunto de valores de um tipo enum é o mesmo que o conjunto de valores do tipo subjacente.

O C# dá suporte a matrizes uni e multidimensionais de qualquer tipo. Ao contrário dos tipos listados acima, os tipos de matriz não precisam ser declarados antes de serem usados. Em vez disso, os tipos de matriz são construídos seguindo um nome de tipo entre colchetes. Por exemplo, `int[]` é uma matriz unidimensional de `int`, `int[,]` é uma matriz bidimensional de `int` e `int[][]` é uma matriz unidimensional de matrizes unidimensionais do `int`.

Os tipos anuláveis também não precisam ser declarados antes que possam ser usados. Para cada tipo de valor não anulável `T`, há um tipo anulável correspondente `T?`, que pode conter um valor adicional `null`. Por exemplo, `int?` é um tipo que pode conter qualquer número inteiro de 32 bits ou o valor `null`.

O sistema de tipos do C# é unificado, de modo que um valor de qualquer tipo pode ser tratado como um objeto. Cada tipo no C#, direta ou indiretamente, deriva do tipo de classe `object`, e `object` é a classe base definitiva de todos os tipos. Os valores de tipos de referência são tratados como objetos simplesmente exibindo os valores como tipo `object`. Os valores dos tipos de valor são tratados como objetos executando \* operações **Boxing** \_ e \_ \*unboxing\*\*. No exemplo a seguir, um valor `int` é convertido em `object` e volta novamente ao `int`.

```
C#  
  
using System;  
  
class Test  
{  
    static void Main()  
    {  
        int i = 123;  
        object o = i;           // Boxing  
        int j = (int)o;         // Unboxing  
    }  
}
```

Quando um valor de um tipo de valor é convertido em tipo `object`, uma instância de objeto, também chamada de "caixa", é alocada para conter o valor e o valor é copiado para essa caixa. Por outro lado, quando uma `object` referência é convertida em um tipo de valor, é feita uma verificação de que o objeto referenciado é uma caixa do tipo de valor correto e, se a verificação for bem sucedido, o valor na caixa será copiado.

O sistema de tipos unificados do C# significa efetivamente que os tipos de valor podem se tornar objetos "sob demanda". Devido à unificação, as bibliotecas de finalidade geral

que usam o tipo `object` podem ser usadas com os tipos de referência e os tipos de valor.

Existem vários tipos de **variáveis** no C#, incluindo campos, elementos de matriz, variáveis locais e parâmetros. As variáveis representam locais de armazenamento e cada variável tem um tipo que determina quais valores podem ser armazenados na variável, conforme mostrado na tabela a seguir.

Tipo de variável	Conteúdo possível
Tipo de valor não nulo	Um valor de tipo exato
Tipos de valor anulável	Um valor nulo ou um valor desse tipo exato
<code>object</code>	Uma referência nula, uma referência a um objeto de qualquer tipo de referência ou uma referência a um valor em caixa de qualquer tipo de valor
Tipo de classe	Uma referência nula, uma referência a uma instância desse tipo de classe ou uma referência a uma instância de uma classe derivada desse tipo de classe
Tipo de interface	Uma referência nula, uma referência a uma instância de um tipo de classe que implementa esse tipo de interface, ou uma referência a um valor em caixa de um tipo de valor que implementa esse tipo de interface
Tipo de matriz	Uma referência nula, uma referência a uma instância desse tipo de matriz ou uma referência a uma instância de um tipo de matriz compatível
Tipo delegado	Uma referência nula ou uma referência a uma instância desse tipo delegado

## Expressões

As **expressões** são construídas a partir de **operandos** e **operadores**\*. and \*operators\*. Os operadores de uma expressão indicam quais operações devem ser aplicadas aos operandos. Exemplos de operadores incluem `+`, `-`, `*`, `/` e `new`. Exemplos de operandos incluem literais, campos, variáveis locais e expressões.

Quando uma expressão contém vários operadores, a \*precedência\* dos operadores controla a ordem na qual os operadores individuais são avaliados. Por exemplo, a expressão `x + y - z` é avaliada como `x + (y * z)` porque o operador `*` tem precedência maior do que o operador `+`.

A maioria dos operadores pode ser *sobre carregada*. A sobrecarga de operador permite que implementações de operador definidas pelo usuário sejam especificadas para operações em que um ou ambos os operandos são de um tipo struct ou de classe definida pelo usuário.

A tabela a seguir resume os operadores do C#, listando as categorias de operador em ordem de precedência, da mais alta para a mais baixa. Os operadores em uma mesma categoria têm precedência igual.

Categoria	Expression	Descrição
Primário	<code>x.m</code>	Acesso de membros
	<code>x(...)</code>	Invocação de método e delegado
	<code>x[...]</code>	Acesso de matriz e indexador
	<code>x++</code>	Pós-incremento
	<code>x--</code>	Pós-decremento
	<code>new T(...)</code>	Criação de objeto e delegado
	<code>new T(...){...}</code>	Criação de objeto com inicializador
	<code>new {...}</code>	Inicializador de objeto anônimo
	<code>new T[...]</code>	Criação de matriz
	<code>typeof(T)</code>	Obter objeto <code>System.Type</code> para <code>T</code>
Unário	<code>checked(x)</code>	Avalia expressão no contexto selecionado
	<code>unchecked(x)</code>	Avalia expressão no contexto desmarcado
	<code>default(T)</code>	Obter valor padrão do tipo <code>T</code>
	<code>delegate {...}</code>	Função anônima (método anônimo)
	<code>+x</code>	Identidade
	<code>-x</code>	Negação
	<code>!x</code>	Negação lógica
	<code>~x</code>	Negação bit a bit
	<code>++x</code>	Pré-incremento

Categoria	Expression	Descrição
	<code>--x</code>	Pré-decremento
	<code>(T)x</code>	Converter explicitamente <code>x</code> no tipo <code>T</code>
	<code>await x</code>	Aguardar assincronamente <code>x</code> para concluir
Multiplicativo	<code>x * y</code>	Multiplicação
	<code>x / y</code>	Divisão
	<code>x % y</code>	Resto
Aditiva	<code>x + y</code>	Adição, concatenação de cadeia de caracteres, combinação de delegados
	<code>x - y</code>	Subtração, remoção de delegado
Shift	<code>x &lt;&lt; y</code>	Shift esquerdo
	<code>x &gt;&gt; y</code>	Shift direito
Teste de tipo e relacional	<code>x &lt; y</code>	Menor que
	<code>x &gt; y</code>	Maior que
	<code>x &lt;= y</code>	Menor ou igual
	<code>x &gt;= y</code>	Maior ou igual
	<code>x is T</code>	Retorna <code>true</code> se <code>x</code> for um <code>T</code> , caso contrário, <code>false</code>
	<code>x as T</code>	Retorna <code>x</code> digitado como <code>T</code> ou <code>null</code> , se <code>x</code> não for um <code>T</code>
Igualitário	<code>x == y</code>	Igual
	<code>x != y</code>	Diferente
AND lógico	<code>x &amp; y</code>	AND bit a bit inteiro, AND lógico booleano
XOR lógico	<code>x ^ y</code>	XOR bit a bit inteiro, XOR lógico booleano
OR lógico	<code>x   y</code>	OR bit a bit inteiro, OR lógico booleano
AND condicional	<code>x &amp;&amp; y</code>	Avalia <code>y</code> somente se <code>x</code> for <code>true</code>
OR condicional	<code>x    y</code>	Avalia <code>y</code> somente se <code>x</code> for <code>false</code>
Coalescência nula	<code>x ?? y</code>	Avalia como <code>y</code> se <code>x</code> é <code>null</code> , para de <code>x</code> outra forma

Categoria	Expression	Descrição
Condisional	<code>x ? y : z</code>	Avalia <code>y</code> se <code>x</code> for <code>true</code> , <code>z</code> se <code>x</code> for <code>false</code>
Atribuição ou função anônima	<code>x = y</code>	Atribuição
	<code>x op= y</code>	Atribuição composta; os operadores com <code>*=</code> <code>/=</code> <code>%=</code> <code>+=</code> <code>-=</code> <code>&lt;&lt;=</code> suporte <code>&gt;&gt;=</code> são <code>&amp;=</code> <code>^=</code> <code> =</code>
	<code>(T x) =&gt; y</code>	Função anônima (expressão lambda)

## Instruções

As ações de um programa são expressas usando **instruções**. O C# oferece suporte a vários tipos diferentes de instruções, algumas delas definidas em termos de instruções inseridas.

Um **bloco** permite a produção de várias instruções em contextos nos quais uma única instrução é permitida. Um bloco é composto por uma lista de instruções escritas entre os delimitadores `{` e `}`.

**Instruções de declaração** são usadas para declarar constantes e variáveis locais.

**Instruções de expressão** são usadas para avaliar expressões. As expressões que podem ser usadas como instruções incluem invocações de método, alocações de objeto usando o `new` operador, atribuições usando `=` e os operadores de atribuição compostos, operações de incremento e decréscimo usando os `++` `--` operadores e e as expressões `Await`.

**Instruções de seleção** são usadas para selecionar uma dentre várias instruções possíveis para execução com base no valor de alguma expressão. Neste grupo estão as instruções `if` e `switch`.

**Instruções de iteração** são usadas para executar repetidamente uma instrução inserida. Neste grupo estão as instruções `while`, `do`, `for` e `foreach`.

**Instruções de salto** são usadas para transferir o controle. Neste grupo estão as instruções `break`, `continue`, `goto`, `throw`, `return` e `yield`.

A instrução `try... catch` é usada para capturar exceções que ocorrem durante a execução de um bloco, e a instrução `try... finally` é usada para especificar o código de finalização que é executado sempre, se uma exceção ocorrer ou não.

As `checked` `unchecked` instruções e são usadas para controlar o contexto de verificação de estouro para operações aritméticas de tipo integral e conversões.

A instrução `lock` é usada para obter o bloqueio de exclusão mútua para um determinado objeto, executar uma instrução e, em seguida, liberar o bloqueio.

A instrução `using` é usada para obter um recurso, executar uma instrução e, em seguida, descartar esse recurso.

Veja abaixo exemplos de cada tipo de instrução

### Declarações de variáveis locais

C#

```
static void Main() {
    int a;
    int b = 2, c = 3;
    a = 1;
    Console.WriteLine(a + b + c);
}
```

### Declaração de constante local

C#

```
static void Main() {
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

### Instrução de expressão

C#

```
static void Main() {
    int i;
    i = 123;           // Expression statement
    Console.WriteLine(i); // Expression statement
    i++;              // Expression statement
    Console.WriteLine(i); // Expression statement
}
```

### if privacidade

C#

```
static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("No arguments");
    }
    else {
        Console.WriteLine("One or more arguments");
    }
}
```

## switch privacidade

C#

```
static void Main(string[] args) {
    int n = args.Length;
    switch (n) {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine("{0} arguments", n);
            break;
    }
}
```

## while privacidade

C#

```
static void Main(string[] args) {
    int i = 0;
    while (i < args.Length) {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

## do privacidade

C#

```
static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    }
```

```
    } while (s != null);  
}
```

## for privacidade

C#

```
static void Main(string[] args) {  
    for (int i = 0; i < args.Length; i++) {  
        Console.WriteLine(args[i]);  
    }  
}
```

## foreach privacidade

C#

```
static void Main(string[] args) {  
    foreach (string s in args) {  
        Console.WriteLine(s);  
    }  
}
```

## break privacidade

C#

```
static void Main() {  
    while (true) {  
        string s = Console.ReadLine();  
        if (s == null) break;  
        Console.WriteLine(s);  
    }  
}
```

## continue privacidade

C#

```
static void Main(string[] args) {  
    for (int i = 0; i < args.Length; i++) {  
        if (args[i].StartsWith("/")) continue;  
        Console.WriteLine(args[i]);  
    }  
}
```

## goto privacidade

C#

```
static void Main(string[] args) {
    int i = 0;
    goto check;
loop:
    Console.WriteLine(args[i++]);
check:
    if (i < args.Length) goto loop;
}
```

return privacidade

C#

```
static int Add(int a, int b) {
    return a + b;
}

static void Main() {
    Console.WriteLine(Add(1, 2));
    return;
}
```

yield privacidade

C#

```
static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}

static void Main() {
    foreach (int x in Range(-10,10)) {
        Console.WriteLine(x);
    }
}
```

throw``try instruções e

C#

```
static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}
```

```
static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
    finally {
        Console.WriteLine("Good bye!");
    }
}
```

## checked``unchecked instruções e

C#

```
static void Main() {
    int i = int.MaxValue;
    checked {
        Console.WriteLine(i + 1);          // Exception
    }
    unchecked {
        Console.WriteLine(i + 1);          // Overflow
    }
}
```

## lock privacidade

C#

```
class Account
{
    decimal balance;
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

## using privacidade

C#

```
static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

## Classes e objetos

\*As **classes** são as mais fundamentais para os tipos do C#. Uma classe é uma estrutura de dados que combina ações (métodos e outros membros da função) e estado (campos) em uma única unidade. Uma classe fornece uma definição para *instâncias* criadas dinamicamente da classe, também conhecida como *objetos*. Classes dão suporte a *herança* e *polimorfismo*, mecanismos pelos quais *classes derivadas* podem estender e especializar *classes base* \*.

Novas classes são criadas usando declarações de classe. Uma declaração de classe inicia com um cabeçalho que especifica os atributos e modificadores de classe, o nome da classe, a classe base (se fornecida) e as interfaces implementadas pela classe. O cabeçalho é seguido pelo corpo da classe, que consiste em uma lista de declarações de membro escrita entre os delimitadores { e }.

A seguir está uma declaração de uma classe simples chamada `Point`:

C#

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Instâncias de classes são criadas usando o operador `new`, que aloca memória para uma nova instância, chama um construtor para inicializar a instância e retorna uma referência à instância. As instruções a seguir criam dois `Point` objetos e armazenam referências a esses objetos em duas variáveis:

C#

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

A memória ocupada por um objeto é recuperada automaticamente quando o objeto não está mais em uso. Não é necessário nem possível desalocar explicitamente os objetos em C#.

## Membros

Os membros de uma classe são **\*membros estáticos \_ ou \_ membros da instância \***. Os membros estáticos pertencem às classes e os membros de instância pertencem aos objetos (instâncias de classes).

A tabela a seguir fornece uma visão geral dos tipos de membros que uma classe pode conter.

Membro	Descrição
Constantes	Valores constantes associados à classe
Campos	Variáveis de classe
Métodos	Os cálculos e as ações que podem ser executados pela classe
Propriedades	Ações associadas à leitura e à gravação de propriedades nomeadas da classe
Indexadores	Ações associadas a instâncias de indexação da classe como uma matriz
Eventos	Notificações que podem ser geradas pela classe
Operadores	Operadores de conversões e expressão com suporte da classe
Construtores	Ações necessárias para inicializar instâncias da classe ou a própria classe
Destruidores	Ações a serem executadas antes de as instâncias da classe serem descartadas permanentemente
Tipos	Tipos aninhados declarados pela classe

## Acessibilidade

Cada membro de uma classe tem uma acessibilidade associada, que controla as regiões de texto do programa que são capazes de acessar o membro. Há cinco formas possíveis de acessibilidade. Esses métodos estão resumidos na tabela a seguir.

Acessibilidade	Significado
<code>public</code>	Acesso não limitado
<code>protected</code>	Acesso limitado a essa classe ou classes derivadas dessa classe
<code>internal</code>	Acesso limitado a este programa
<code>protected internal</code>	Acesso limitado a esse programa ou a classes derivadas dessa classe
<code>private</code>	Acesso limitado a essa classe

## Parâmetros de tipo

Uma definição de classe pode especificar um conjunto de parâmetros de tipo seguindo o nome da classe com colchetes angulares com uma lista de nomes de parâmetro de tipo. Em seguida, os parâmetros de tipo podem ser usados no corpo das declarações de classe para definir os membros da classe. No exemplo a seguir, os parâmetros de tipo de `Pair` são `TFirst` e `TSecond`:

```
C#
public class Pair<TFirst, TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

Um tipo de classe que é declarado para pegar parâmetros de tipo é chamado de tipo de classe genérica. Tipos de struct, de interface e de delegado também podem ser genéricos.

Quando a classe genérica é usada, os argumentos de tipo devem ser fornecidos para cada um dos parâmetros de tipo:

```
C#
Pair<int, string> pair = new Pair<int, string> { First = 1, Second = "two" };
int i = pair.First;      // TFirst is int
string s = pair.Second; // TSecond is string
```

Um tipo genérico com argumentos de tipo fornecidos, como `Pair<int, string>` acima, é chamado de tipo construído.

## Classes base

Uma declaração de classe pode especificar uma classe base, seguindo os parâmetros de nome da classe e tipo com dois-pontos e o nome da classe base. Omitir uma especificação de classe base é o mesmo que derivar do `object` de tipo. No exemplo a seguir, a classe base de `Point3D` é `Point` e a classe base de `Point` é `object`:

C#

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Point3D: Point
{
    public int z;

    public Point3D(int x, int y, int z): base(x, y) {
        this.z = z;
    }
}
```

Uma classe herda os membros de sua classe base. Herança significa que uma classe implicitamente contém todos os membros de sua classe base, exceto para os construtores estáticos e de instância, e os destruidores da classe base. Uma classe derivada pode adicionar novos membros aos que ela herda, mas ela não pode remover a definição de um membro herdado. No exemplo anterior, `Point3D` herda os campos `x` e `y` de `Point` e cada instância `Point3D` contém três campos: `x`, `y` e `z`.

Existe uma conversão implícita de um tipo de classe para qualquer um de seus tipos de classe base. Portanto, uma variável de um tipo de classe pode referenciar uma instância dessa classe ou uma instância de qualquer classe derivada. Por exemplo, dadas as declarações de classe anteriores, uma variável do tipo `Point` podem referenciar um `Point` ou um `Point3D`:

C#

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

## Campos

Um campo é uma variável que está associada a uma classe ou a uma instância de uma classe.

Um campo declarado com o `static` modificador define um ***campo estático***. Um campo estático identifica exatamente um local de armazenamento. Não importa quantas instâncias de uma classe são criadas, há sempre apenas uma cópia de um campo estático.

Um campo declarado sem o `static` modificador define um ***campo de instância***. Cada instância de uma classe contém uma cópia separada de todos os campos de instância dessa classe.

No exemplo a seguir, cada instância da classe `Color` tem uma cópia separada dos campos de instância `r`, `g` e `b`, mas há apenas uma cópia dos campos estáticos `Black`, `White`, `Red`, `Green` e `Blue`:

```
C#  
  
public class Color  
{  
    public static readonly Color Black = new Color(0, 0, 0);  
    public static readonly Color White = new Color(255, 255, 255);  
    public static readonly Color Red = new Color(255, 0, 0);  
    public static readonly Color Green = new Color(0, 255, 0);  
    public static readonly Color Blue = new Color(0, 0, 255);  
    private byte r, g, b;  
  
    public Color(byte r, byte g, byte b) {  
        this.r = r;  
        this.g = g;  
        this.b = b;  
    }  
}
```

Conforme mostrado no exemplo anterior, os ***campos somente leitura*** podem ser declarados com um modificador `readonly`. A atribuição a um `readonly` campo só pode ocorrer como parte da declaração do campo ou em um construtor na mesma classe.

## Métodos

Um **\*método\*** é um membro que implementa uma computação ou ação que pode ser executada por um objeto ou uma classe. Os **métodos estáticos** são acessados por meio da classe. Os **métodos da instância\*** são acessados por meio de instâncias da classe.

Os métodos têm uma lista (possivelmente vazia) de **\*parâmetros** \_, que representam valores ou referências variáveis passadas para o método e um \_ *tipo de retorno* \*, que especifica o tipo do valor calculado e retornado pelo método. O tipo de retorno de um método é **void** se ele não retornar um valor.

Como os tipos, os métodos também podem ter um conjunto de parâmetros de tipo, para que os quais os argumentos de tipo devem ser especificados quando o método é chamado. Ao contrário dos tipos, os argumentos de tipo geralmente podem ser inferidos de argumentos de uma chamada de método e não precisam ser fornecidos explicitamente.

A **assinatura** de um método deve ser exclusiva na classe na qual o método é declarado. A assinatura de um método consiste no nome do método, número de parâmetros de tipo e número, modificadores e tipos de seus parâmetros. A assinatura de um método não inclui o tipo de retorno.

## Parâmetros

Os parâmetros são usados para passar valores ou referências de variável aos métodos. Os parâmetros de um método obtêm seus valores reais de **argumentos** que são especificados quando o método é invocado. Há quatro tipos de parâmetros: parâmetros de valor, parâmetros de referência, parâmetros de saída e matrizes de parâmetros.

Um **parâmetro de valor** é usado para passar o parâmetro de entrada. Um parâmetro de valor corresponde a uma variável local que obtém seu valor inicial do argumento passado para o parâmetro. As modificações em um parâmetro de valor não afetam o argumento passado para o parâmetro.

Os parâmetros de valor podem ser opcionais, especificando um valor padrão para que os argumentos correspondentes possam ser omitidos.

Um **parâmetro de referência** é usado para passar os parâmetros de entrada e saída. O argumento passado para um parâmetro de referência deve ser uma variável e, durante a execução do método, o parâmetro de referência representa o mesmo local de armazenamento como a variável de argumento. Um parâmetro de referência é declarado com o modificador **ref**. O exemplo a seguir mostra o uso de parâmetros **ref**.

```
C#
```

```
using System;  
  
class Test  
{
```

```

    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);           // Outputs "2 1"
    }
}

```

Um **parâmetro de saída** é usado para passar o parâmetro de entrada. Um parâmetro de saída é semelhante a um parâmetro de referência, exceto que o valor inicial do argumento fornecido não é importante. Um parâmetro de saída é declarado com o modificador `out`. O exemplo a seguir mostra o uso de parâmetros `out`.

C#

```

using System;

class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem);      // Outputs "3 1"
    }
}

```

Uma **matriz de parâmetros** permite que um número variável de argumentos sejam passados para um método. Uma matriz de parâmetro é declarada com o modificador `params`. Somente o último parâmetro de um método pode ser uma matriz de parâmetros e o tipo de uma matriz de parâmetros deve ser um tipo de matriz unidimensional. Os `Write` `WriteLine` métodos e da `System.Console` classe são bons exemplos de uso de matriz de parâmetros. Eles são declarados como segue.

C#

```

public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
}

```

```
    ...  
}
```

Dentro de um método que usa uma matriz de parâmetros, a matriz de parâmetros se comporta exatamente como um parâmetro regular de um tipo de matriz. No entanto, em uma invocação de um método com uma matriz de parâmetros, é possível passar um único argumento do tipo da matriz de parâmetro ou qualquer número de argumentos do tipo de elemento da matriz de parâmetros. No último caso, uma instância de matriz é automaticamente criada e inicializada com os argumentos determinados. Esse exemplo

```
C#
```

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

é equivalente ao escrito a seguir.

```
C#
```

```
string s = "x={0} y={1} z={2}";  
object[] args = new object[3];  
args[0] = x;  
args[1] = y;  
args[2] = z;  
Console.WriteLine(s, args);
```

## Corpo do método e variáveis locais

O corpo de um método especifica as instruções a serem executadas quando o método é invocado.

Um corpo de método pode declarar variáveis que são específicas para a invocação do método. Essas variáveis são chamadas de **variáveis locais**. Uma declaração de variável local especifica um nome de tipo, um nome de variável e, possivelmente, um valor inicial. O exemplo a seguir declara uma variável local `i` com um valor inicial de zero e uma variável local `j` sem valor inicial.

```
C#
```

```
using System;  
  
class Squares  
{  
    static void Main() {  
        int i = 0;
```

```
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

O C# requer que uma variável local seja **atribuída definitivamente** antes de seu valor poder ser obtido. Por exemplo, se a declaração do `i` anterior não incluisse um valor inicial, o compilador relataria um erro para usos subsequentes de `i` porque `i` não seria definitivamente atribuído a esses pontos do programa.

Um método pode usar instruções `return` para retornar o controle é pelo chamador. Em um método que retorna `void`, as instruções `return` não podem especificar uma expressão. Em um método que retorna instruções não-`void`, `return` deve incluir uma expressão que computa o valor de retorno.

## Métodos estáticos e de instância

Um método declarado com um `static` modificador é um **método estático**. Um método estático não funciona em uma instância específica e pode acessar diretamente apenas membros estáticos.

Um método declarado sem um `static` modificador é um **método de instância**. Um método de instância opera em uma instância específica e pode acessar membros estáticos e de instância. A instância em que um método de instância foi invocado pode ser explicitamente acessada como `this`. É um erro se referir a `this` em um método estático.

A seguinte classe `Entity` tem membros estáticos e de instância.

```
C#
class Entity
{
    static int nextSerialNo;
    int serialNo;

    public Entity() {
        serialNo = nextSerialNo++;
    }

    public int GetSerialNo() {
        return serialNo;
    }
}
```

```
}

public static int GetNextSerialNo() {
    return nextSerialNo;
}

public static void SetNextSerialNo(int value) {
    nextSerialNo = value;
}
}
```

Cada instância `Entity` contém um número de série (e, possivelmente, outras informações que não são mostradas aqui). O construtor `Entity` (que é como um método de instância) inicializa a nova instância com o próximo número de série disponível. Como o construtor é um membro de instância, ele tem permissão para acessar tanto o campo de instância `serialNo` e o campo estático `nextSerialNo`.

Os métodos estáticos `GetNextSerialNo` e `SetNextSerialNo` podem acessar o campo estático `nextSerialNo`, mas seria um erro para eles acessar diretamente o campo de instância `serialNo`.

O exemplo a seguir mostra o uso da `Entity` classe.

C#

```
using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}
```

Observe que os métodos estáticos `SetNextSerialNo` e `GetNextSerialNo` são invocados na classe enquanto o método de instância `GetSerialNo` é chamado em instâncias da classe.

## Métodos abstratos, virtuais e de substituição

Quando uma declaração de método de instância inclui um `virtual` modificador, o método é considerado um *\*método virtual\**. Quando nenhum `virtual` modificador está presente, o método é considerado um *\_método não virtual\_\**.

Quando um método virtual é invocado, o **tipo de tempo de execução**\* da instância para a qual a invocação ocorre determina a implementação do método real a ser invocado. Em uma invocação de método não virtual, o *tipo de tempo de compilação*\* da instância é o fator determinante.

Um método virtual pode ser **substituído** em uma classe derivada. Quando uma declaração de método de instância inclui um `override` modificador, o método substitui um método virtual herdado pela mesma assinatura. Enquanto uma declaração de método virtual apresenta um novo método, uma declaração de método de substituição restringe um método virtual herdado existente fornecendo uma nova implementação do método.

Um método **abstrato** é um método virtual sem implementação. Um método abstract é declarado com o `abstract` modificador e é permitido somente em uma classe que também é declarada `abstract`. Um método abstrato deve ser substituído em cada classe derivada não abstrata.

O exemplo a seguir declara uma classe abstrata, `Expression`, que representa um nó de árvore de expressão e três classes derivadas, `Constant`, `VariableReference` e `Operation`, que implementam nós de árvore de expressão para operações aritméticas, referências de variável e constantes. (Isso é semelhante a, mas não deve ser confundido com os tipos de árvore de expressão introduzidos em [tipos de árvore de expressão](#)).

C#

```
using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}

public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }

    public override double Evaluate(Hashtable vars) {
        return value;
    }
}
```

```

    }

}

public class VariableReference: Expression
{
    string name;

    public VariableReference(string name) {
        this.name = name;
    }

    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}

```

As quatro classes anteriores podem ser usadas para modelar expressões aritméticas. Por exemplo, usando instâncias dessas classes, a expressão `x + 3` pode ser representada da seguinte maneira.

C#

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

O método `Evaluate` de uma instância `Expression` é chamado para avaliar a expressão especificada e produzir um valor `double`. O método usa como um argumento a `Hashtable` que contém nomes de variáveis (como chaves das entradas) e valores (como valores das entradas). O `Evaluate` método é um método abstrato virtual, o que significa que classes derivadas não abstratas devem substituí-lo para fornecer uma implementação real.

Uma implementação de `Evaluate` do `Constant` retorna apenas a constante armazenada. A `VariableReference` implementação de pesquisa o nome da variável na tabela de hash e retorna o valor resultante. Uma implementação de `Operation` primeiro avalia os operandos esquerdo e direito (chamando recursivamente seus métodos `Evaluate`) e, em seguida, executa a operação aritmética determinada.

O seguinte programa usa as classes `Expression` para avaliar a expressão `x * (y + 2)` para valores diferentes de `x` e `y`.

C#

```
using System;
using System.Collections;

class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));           // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));           // Outputs "16.5"
    }
}
```

## Sobrecarga de método

O método **sobrecarregamento** permite que vários métodos na mesma classe tenham o mesmo nome desde que tenham assinaturas exclusivas. Ao compilar uma invocação de um método sobrecarregado, o compilador usa a resolução **sobrecarga** para determinar o método específico a ser invocado. A resolução de sobrecarga localizará o método que melhor corresponde aos argumentos ou relatará um erro se nenhuma correspondência for encontrada. O exemplo a seguir mostra a resolução de sobrecarga em vigor. O comentário para cada invocação no método `Main` mostra qual método é realmente chamado.

C#

```
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }

    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    static void Main() {
        F();                      // Invokes F()
        F(1);                     // Invokes F(int)
        F(1.0);                   // Invokes F(double)
        F("abc");                 // Invokes F(object)
        F((double)1);              // Invokes F(double)
        F((object)1);              // Invokes F(object)
        F<int>(1);                // Invokes F<T>(T)
        F(1, 1);                  // Invokes F(double, double)
    }
}
```

```
    }  
}
```

Conforme mostrado no exemplo, um determinado método sempre pode ser selecionado ao converter explicitamente os argumentos para os tipos de parâmetro exatos e/ou fornecendo explicitamente os argumentos de tipo.

## Outros membros da função

Os membros que contêm código executável são conhecidos coletivamente como **membros de função** de uma classe. A seção anterior descreve os métodos, que são o tipo principal de membros da função. Esta seção descreve os outros tipos de membros de função com suporte em C#: construtores, propriedades, indexadores, eventos, operadores e destruidores.

O código a seguir mostra uma classe genérica chamada `List<T>`, que implementa uma lista de objetos que aumenta. A classe contém vários exemplos dos tipos mais comuns de membros da função.

C#

```
public class List<T> {  
    // Constant...  
    const int defaultCapacity = 4;  
  
    // Fields...  
    T[] items;  
    int count;  
  
    // Constructors...  
    public List(int capacity = defaultCapacity) {  
        items = new T[capacity];  
    }  
  
    // Properties...  
    public int Count {  
        get { return count; }  
    }  
    public int Capacity {  
        get {  
            return items.Length;  
        }  
        set {  
            if (value < count) value = count;  
            if (value != items.Length) {  
                T[] newItems = new T[value];  
                Array.Copy(items, 0, newItems, 0, count);  
                items = newItems;  
            }  
        }  
    }  
}
```

```

        }

    }

    // Indexer...
    public T this[int index] {
        get {
            return items[index];
        }
        set {
            items[index] = value;
            OnChanged();
        }
    }

    // Methods...
    public void Add(T item) {
        if (count == Capacity) Capacity = count * 2;
        items[count] = item;
        count++;
        OnChanged();
    }
    protected virtual void OnChanged() {
        if (Changed != null) Changed(this, EventArgs.Empty);
    }
    public override bool Equals(object other) {
        return Equals(this, other as List<T>);
    }
    static bool Equals(List<T> a, List<T> b) {
        if (a == null) return b == null;
        if (b == null || a.count != b.count) return false;
        for (int i = 0; i < a.count; i++) {
            if (!object.Equals(a.items[i], b.items[i])) {
                return false;
            }
        }
        return true;
    }

    // Event...
    public event EventHandler Changed;

    // Operators...
    public static bool operator ==(List<T> a, List<T> b) {
        return Equals(a, b);
    }
    public static bool operator !=(List<T> a, List<T> b) {
        return !Equals(a, b);
    }
}

```

## Construtores

O C# dá suporte aos construtores estáticos e de instância. Um **\*Construtor de instância** é um membro que implementa as ações necessárias para inicializar uma instância de uma classe. Um **\_construtor estático\*** é um membro que implementa as ações necessárias para inicializar uma classe em si quando ela é carregada pela primeira vez.

Um construtor é declarado como um método sem nenhum tipo de retorno e o mesmo nome que a classe contém. Se uma declaração de Construtor incluir um `static` modificador, ele declara um construtor estático. Caso contrário, ela declara um construtor de instância.

Construtores de instância podem ser sobrecarregados. Por exemplo, a classe `List<T>` declara dois construtores de instância, um sem parâmetros e um que utiliza um parâmetro `int`. Os construtores de instância são invocados usando o operador `new`. As instruções a seguir alocam duas `List<string>` instâncias usando cada um dos construtores da `List` classe.

C#

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

Diferentemente de outros membros, construtores de instância não são herdados e uma classe não tem nenhum construtor de instância que não os que são realmente declarados na classe. Se nenhum construtor de instância for fornecido para uma classe, então um construtor vazio sem parâmetros será fornecido automaticamente.

## Propriedades

**\*As propriedades** \_ são uma extensão natural dos campos. Elas são denominadas membros com tipos associados, e a sintaxe para acessar os campos e as propriedades é a mesma. No entanto, diferentemente dos campos, as propriedades não denotam locais de armazenamento. Em vez disso, as propriedades têm **\_acessadores\*** que especificam as instruções a serem executadas quando seus valores são lidos ou gravados.

Uma propriedade é declarada como um campo, exceto que a declaração termina com um `get` acessador e/ou um `set` acessador escrito entre os delimitadores `{` e `}` em vez de terminar em um ponto e vírgula. Uma propriedade que tem um `get` acessador e um `set` acessador é uma propriedade **Read-Write** \_, uma propriedade que tem apenas um `get` acessador é uma \_propriedade somente leitura\*\_ e uma propriedade que tem apenas um `set` acessador é uma propriedade \_somente gravação \*.

Um `get` acessador corresponde a um método sem parâmetros com um valor de retorno do tipo de propriedade. Exceto como o destino de uma atribuição, quando uma propriedade é referenciada em uma expressão, o `get` acessador da propriedade é invocado para calcular o valor da propriedade.

Um `set` acessador corresponde a um método com um único parâmetro chamado `value` e nenhum tipo de retorno. Quando uma propriedade é referenciada como o destino de uma atribuição ou como o operando de `++` ou `--`, o `set` acessador é invocado com um argumento que fornece o novo valor.

A classe `List<T>` declara duas propriedades, `Count` e `Capacity`, que são somente leitura e leitura/gravação, respectivamente. A seguir está um exemplo de uso dessas propriedades.

C#

```
List<string> names = new List<string>();
names.Capacity = 100;           // Invokes set accessor
int i = names.Count;          // Invokes get accessor
int j = names.Capacity;        // Invokes get accessor
```

Como nos campos e métodos, o C# dá suporte a propriedades de instância e a propriedades estáticas. As propriedades estáticas são declaradas com o `static` modificador e as propriedades de instância são declaradas sem ele.

Os acessadores de uma propriedade podem ser virtuais. Quando uma declaração de propriedade inclui um modificador `virtual`, `abstract` ou `override`, ela se aplica aos acessadores da propriedade.

## Indexadores

Um **indexador** é um membro que permite que objetos sejam indexados da mesma forma que uma matriz. Um indexador é declarado como uma propriedade, exceto se o nome do membro for `this` seguido por uma lista de parâmetros escrita entre os delimitadores `[` e `]`. Os parâmetros estão disponíveis nos acessadores do indexador. Semelhante às propriedades, os indexadores podem ser de leitura-gravação, somente leitura e somente gravação, e os acessadores de um indexador pode ser virtuais.

A classe `List` declara um indexador único de leitura-gravação que usa um parâmetro `int`. O indexador possibilita indexar instâncias `List` com valores `int`. Por exemplo,

C#

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Os indexadores podem ser sobre carregados, o que significa que uma classe pode declarar vários indexadores, desde que o número ou os tipos de seus parâmetros sejam diferentes.

## Eventos

Um **evento** é um membro que permite que uma classe ou objeto forneça notificações. Um evento é declarado como um campo, exceto que a declaração inclui uma `event` palavra-chave e o tipo deve ser um tipo delegado.

Em uma classe que declara um membro de evento, o evento se comporta exatamente como um campo de um tipo delegado (desde que o evento não seja abstrato e não declare acessadores). O campo armazena uma referência a um delegado que representa os manipuladores de eventos que foram adicionados ao evento. Se nenhum manipulador de eventos estiver presente, o campo será `null`.

A classe `List<T>` declara um membro único de evento chamado `Changed`, que indica que um novo item foi adicionado à lista. O `Changed` evento é gerado pelo `OnChanged` método virtual, que primeiro verifica se o evento é `null` (ou seja, se não há manipuladores presentes). A noção de gerar um evento é precisamente equivalente a invocar o delegado representado pelo evento — assim, não há constructos de linguagem especial para gerar eventos.

Os clientes reagem a eventos por meio de **manipuladores de eventos**. Os manipuladores de eventos são conectados usando o operador `+=` e removidos usando o operador `-=`. O exemplo a seguir anexa um manipulador de eventos para o evento `Changed` de um `List<string>`.

C#

```
using System;

class Test
{
    static int changeCount;
```

```

    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }

    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);           // Outputs "3"
    }
}

```

Para cenários avançados nos quais o controle do armazenamento subjacente de um evento é desejado, uma declaração de evento pode fornecer explicitamente acessadores `add` e `remove`, que são um pouco semelhantes ao acessador `set` de uma propriedade.

## Operadores

Um *operador* é um membro que define o significado da aplicação de um operador de expressão específico para instâncias de uma classe. Três tipos de operadores podem ser definidos: operadores unários, operadores binários e operadores de conversão. Todos os operadores devem ser declarados como `public` e `static`.

A classe `List<T>` declara dois operadores, `operator==` e `operator!=`, e, portanto, dá um novo significado para as expressões que aplicam esses operadores a instâncias `List`. Especificamente, os operadores definem a igualdade de duas `List<T>` instâncias como comparar cada um dos objetos contidos usando seus `Equals` métodos. O exemplo a seguir usa o operador `==` para comparar duas instâncias `List<int>`.

C#

```

using System;

class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b);           // Outputs "True"
        b.Add(3);
        Console.WriteLine(a == b);           // Outputs "False"
    }
}

```

```
    }  
}
```

O primeiro `Console.WriteLine` gera `True` porque as duas listas contêm o mesmo número de objetos com os mesmos valores na mesma ordem. Como `List<T>` não definiu `operator==`, o primeiro `Console.WriteLine` geraria `False` porque `a` e `b` referenciam diferentes instâncias `List<int>`.

## Destruidores

Um **destruidor** é um membro que implementa as ações necessárias para destruir uma instância de uma classe. Os destruidores não podem ter parâmetros, eles não podem ter modificadores de acessibilidade e não podem ser invocados explicitamente. O destruidor de uma instância é invocado automaticamente durante a coleta de lixo.

O coletor de lixo tem permissão de uma ampla latitude para decidir quando coletar objetos e executar destruidores. Especificamente, o tempo das invocações do destruidor não é determinístico e os destruidores podem ser executados em qualquer thread. Por esses e outros motivos, as classes devem implementar destruidores somente quando nenhuma outra solução for viável.

A instrução `using` fornece uma abordagem melhor para a destruição de objetos.

## Estruturas

Como classes, os **structs** são estruturas de dados que podem conter membros de dados e os membros da função, mas, ao contrário das classes, as estruturas são tipos de valor e não precisam de alocação de heap. Uma variável de um tipo struct armazena diretamente os dados do struct, enquanto que uma variável de um tipo de classe armazena uma referência a um objeto alocado dinamicamente. Os tipos de estrutura não dão suporte à herança especificada pelo usuário, e todos os tipos de structs são herdados implicitamente do tipo `object`.

Os structs são particularmente úteis para estruturas de dados pequenas que têm semântica de valor. Números complexos, pontos em um sistema de coordenadas ou pares chave-valor em um dicionário são exemplos de structs. O uso de structs, em vez de classes para estruturas de dados pequenas, pode fazer uma grande diferença no número de alocações de memória que um aplicativo executa. Por exemplo, o programa a seguir cria e inicializa uma matriz de 100 pontos. Com `Point` implementado como uma classe, 101 objetos separados são instanciados — um para a matriz e um para os elementos de 100.

C#

```
class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}
```

Uma alternativa é criar `Point` uma struct.

C#

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Agora, somente um objeto é instanciado — um para a matriz — e as instâncias `Point` são armazenadas em linha na matriz.

Construtores de struct são invocados com o operador `new`, mas isso não significa que a memória está sendo alocada. Em vez de alocar dinamicamente um objeto e retornar uma referência a ele, um construtor de struct simplesmente retorna o valor do struct (normalmente em um local temporário na pilha), e esse valor é, então, copiado conforme necessário.

Com classes, é possível que duas variáveis referenciem o mesmo objeto e, portanto, é possível que operações em uma variável afetem o objeto referenciado por outra variável. Com structs, as variáveis têm sua própria cópia dos dados e não é possível que as operações em um afetem o outro. Por exemplo, a saída produzida pelo fragmento de código a seguir depende de se `Point` ser uma classe ou uma estrutura.

C#

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

Se `Point` é uma classe, a saída é `20` porque `a` e `b` faz referência ao mesmo objeto. Se `Point` for uma struct, a saída será `10` porque a atribuição de `a` para `b` cria uma cópia do valor, e essa cópia não será afetada pela atribuição subsequente para `a.x`.

O exemplo anterior destaca duas das limitações dos structs. Primeiro, copiar um struct inteiro é, geralmente, menos eficiente do que copiar uma referência de objeto, então a passagem de atribuição e de valor do parâmetro pode ser mais custosa com structs que com tipos de referência. Segundo, com exceção para parâmetros `ref` e `out`, não é possível criar referências para structs, o que rege o uso em diversas situações.

## Matrizes

Uma `*matriz_` é uma estrutura de dados que contém um número de variáveis que são acessadas por meio de índices computados. As variáveis contidas em uma matriz, também chamadas de *elementos* da matriz, são do mesmo tipo, e esse tipo é chamado de *tipo de elemento \_ \* da matriz*.

Os tipos de matriz são tipos de referência, e a declaração de uma variável de matriz simplesmente reserva espaço para uma referência a uma instância de matriz. As instâncias de matriz reais são criadas dinamicamente em tempo de execução usando o `new` operador. A `new` operação especifica o **comprimento** da nova instância de matriz, que é então corrigida para o tempo de vida da instância. Os índices dos elementos de uma matriz variam de `0` a `Length - 1`. O operador `new` inicializa automaticamente os elementos de uma matriz usando o valor padrão, que, por exemplo, é zero para todos os tipos numéricos e `null` para todos os tipos de referência.

O exemplo a seguir cria uma matriz de elementos `int`, inicializa a matriz e imprime o conteúdo da matriz.

C#

```
using System;

class Test
{
    static void Main() {
```

```

int[] a = new int[10];
for (int i = 0; i < a.Length; i++) {
    a[i] = i * i;
}
for (int i = 0; i < a.Length; i++) {
    Console.WriteLine("a[{0}] = {1}", i, a[i]);
}
}
}

```

Este exemplo cria e opera em uma única matriz de dimensão unidimensional. O C# também dá suporte a *matrizes multidimensionais*. O número de dimensões de um tipo de matriz, também conhecido como *Rank*\* do tipo de matriz, é um além do número de vírgulas gravados entre os colchetes do tipo de matriz. O exemplo a seguir aloca uma matriz unidimensional, bidimensional e tridimensional.

C#

```

int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];

```

A matriz `a1` contém 10 elementos, a matriz `a2` contém 50 ( $10 \times 5$ ) elementos e a matriz `a3` contém 100 ( $10 \times 5 \times 2$ ) elementos.

O tipo do elemento de uma matriz pode ser qualquer tipo, incluindo um tipo de matriz. Uma matriz com elementos de um tipo de matriz é chamada às vezes de **matriz denteada**, pois os tamanhos das matrizes do elemento nem sempre precisam ser iguais. O exemplo a seguir aloca uma matriz de matrizes de `int`:

C#

```

int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];

```

A primeira linha cria uma matriz com três elementos, cada um do tipo `int[]`, e cada um com um valor inicial de `null`. As linhas subsequentes inicializam os três elementos com referências às instâncias individuais da matriz de tamanhos variados.

O `new` operador permite que os valores iniciais dos elementos da matriz sejam especificados usando um ***inicializador de matriz***, que é uma lista de expressões gravadas entre os delimitadores `{` e `}`. O exemplo a seguir aloca e inicializa um `int[]` com três elementos.

C#

```
int[] a = new int[] {1, 2, 3};
```

Observe que o comprimento da matriz é inferido do número de expressões entre `{` e `}`. A variável local e declarações de campo podem ser reduzidas ainda mais, de modo que o tipo de matriz não precise ser redefinido.

C#

```
int[] a = {1, 2, 3};
```

Os dois exemplos anteriores são equivalentes ao seguinte:

C#

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

## Interfaces

Uma **interface** define um contrato que pode ser implementado por classes e estruturas. Uma interface pode conter métodos, propriedades, eventos e indexadores. Uma interface não fornece implementações dos membros que define — ela simplesmente especifica os membros que devem ser fornecidos por classes ou estruturas que implementam a interface.

As interfaces podem empregar a **herança múltipla**. No exemplo a seguir, a interface `IComboBox` herda de `ITextBox` e `IListBox`.

C#

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}
```

```
interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

Classes e structs podem implementar várias interfaces. No exemplo a seguir, a classe `EditBox` implementa `IControl` e `IDataBound`.

```
C#

interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox: IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```

Quando uma classe ou struct implementa uma interface específica, as instâncias dessa classe ou struct podem ser convertidas implicitamente para esse tipo de interface. Por exemplo,

```
C#

EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

Em casos nos quais uma instância não é conhecida por ser estática para implementar uma interface específica, é possível usar conversões de tipo dinâmico. Por exemplo, as instruções a seguir usam conversões dinâmicas de tipo para obter as implementações de um objeto `IControl` e de `IDataBound` interface. Como o tipo real do objeto é `EditBox`, as conversões são realizadas com sucesso.

```
C#

object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

Na classe anterior `EditBox`, o `Paint` método da `IControl` interface e o `Bind` método da `IDataBound` interface são implementados usando `public` Membros. O C# também dá suporte a ***implementações explícitas de membros de interface***, usando o que a classe ou struct pode evitar fazer os membros `public`. Uma implementação de membro de interface explícita é escrita usando o nome do membro de interface totalmente qualificado. Por exemplo, a classe `EditBox` pode implementar os métodos `IControl.Paint` e `IDataBound.Bind` usando implementações de membros de interface explícita da seguinte maneira.

C#

```
public class EditBox: IControl, IDataBound
{
    void IControl.Paint() {...}
    void IDataBound.Bind(Binder b) {...}
}
```

Os membros de interface explícita só podem ser acessados por meio do tipo de interface. Por exemplo, a implementação de `IControl.Paint` fornecida pela classe `Previous EditBox` só pode ser invocada pela primeira vez pela conversão da `EditBox` referência para o `IControl` tipo de interface.

C#

```
EditBox editBox = new EditBox();
editBox.Paint();                                // Error, no such method
IControl control = editBox;
control.Paint();                                 // Ok
```

## Enumerações

Um ***tipo enum*** é um tipo de valor diferente com um conjunto de constantes nomeadas. O exemplo a seguir declara e usa um tipo enum chamado `Color` com três valores constantes, `Red`, `Green` e `Blue`.

C#

```
using System;

enum Color
{
    Red,
    Green,
    Blue
```

```

}

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}

```

Cada tipo de enumeração tem um tipo integral correspondente chamado de *tipo subjacente* do tipo de enumeração. Um tipo de enumeração que não declara explicitamente um tipo subjacente tem um tipo subjacente de `int`. O formato de armazenamento de um tipo de enumeração e o intervalo de valores possíveis são determinados pelo seu tipo subjacente. O conjunto de valores que um tipo de enumeração pode assumir não é limitado por seus membros enum. Em particular, qualquer valor do tipo subjacente de uma enumeração pode ser convertido para o tipo enum e é um valor válido distinto desse tipo enum.

O exemplo a seguir declara um tipo enum chamado `Alignment` com um tipo subjacente de `sbyte`.

C#

```

enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}

```

Conforme mostrado no exemplo anterior, uma declaração de membro de enumeração pode incluir uma expressão constante que especifica o valor do membro. O valor constante para cada membro de enumeração deve estar no intervalo do tipo subjacente da enumeração. Quando uma declaração de membro de enumeração não especifica explicitamente um valor, o membro recebe o valor zero (se for o primeiro membro no tipo enum) ou o valor do membro enum textualmente anterior mais um.

Os valores de enumeração podem ser convertidos em valores inteiros e vice-versa usando conversões de tipo. Por exemplo,

C#

```
int i = (int)Color.Blue;           // int i = 2;
Color c = (Color)2;                // Color c = Color.Blue;
```

O valor padrão de qualquer tipo de enumeração é o valor integral zero convertido para o tipo de enumeração. Nos casos em que as variáveis são inicializadas automaticamente para um valor padrão, esse é o valor dado a variáveis de tipos enum. Para que o valor padrão de um tipo de enumeração seja facilmente disponível, o literal `0` converte implicitamente em qualquer tipo de enumeração. Dessa forma, o seguinte é permitido.

C#

```
Color c = 0;
```

## Delegados

Um **delegado** é um tipo que representa referências aos métodos com uma lista de parâmetros e tipo de retorno específicos. Delegados possibilitam o tratamento de métodos como entidades que podem ser atribuídos a variáveis e passadas como parâmetros. Os delegados são parecidos com o conceito de ponteiros de função em outras linguagens, mas ao contrário dos ponteiros de função, os delegados são orientados a objetos e fortemente tipados.

O exemplo a seguir declara e usa um tipo delegado chamado `Function`.

C#

```
using System;

delegate double Function(double x);

class Multiplier
```

```

{
    double factor;

    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}

```

Uma instância do tipo delegado `Function` pode fazer referência a qualquer método que usa um argumento `double` e retorna um valor `double`. O `Apply` método aplica um dado `Function` aos elementos de a `double[]`, retornando um `double[]` com os resultados. No método `Main`, `Apply` é usado para aplicar três funções diferentes para um `double[]`.

Um delegado pode referenciar um método estático (como `Square` ou `Math.Sin` no exemplo anterior) ou um método de instância (como `m.Multiply` no exemplo anterior). Um delegado que referencia um método de instância também referencia um objeto específico, e quando o método de instância é invocado por meio do delegado, esse objeto se torna `this` na invocação.

Os delegados podem ser criados usando funções anônimas, que são "métodos embutidos" criados dinamicamente. As funções anônimas podem ver as variáveis locais dos métodos ao redor. Assim, o exemplo multiplicador acima pode ser escrito mais facilmente sem usar uma `Multiplier` classe:

C#

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

Uma propriedade interessante e útil de um delegado é que ele não sabe ou se importa com a classe do método que referencia; o que importa é que o método referenciado tem os mesmos parâmetros e o tipo de retorno do delegado.

## Atributos

Tipos, membros e outras entidades em um programa C# dão suporte a modificadores que controlam determinados aspectos de seu comportamento. Por exemplo, a acessibilidade de um método é controlada usando os modificadores `public`, `protected`, `internal` e `private`. O C# generaliza essa funcionalidade, de modo que os tipos definidos pelo usuário de informações declarativas podem ser anexados a entidades de programa e recuperados no tempo de execução. Os programas especificam essas informações declarativas adicionais, definindo e usando os *atributos*.

O exemplo a seguir declara um atributo `HelpAttribute` que pode ser colocado em entidades de programa para fornecem links para a documentação associada.

C#

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

Todas as classes de atributo derivam da `System.Attribute` classe base fornecida pelo .NET Framework. Os atributos podem ser aplicados, fornecendo seu nome, junto com

quaisquer argumentos, dentro dos colchetes pouco antes da declaração associada. Se o nome de um atributo terminar em `Attribute`, essa parte do nome poderá ser omitida quando o atributo for referenciado. Por exemplo, o atributo `HelpAttribute` pode ser usado da seguinte maneira.

C#

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) {}
}
```

Este exemplo anexa um `HelpAttribute` à `Widget` classe e outro `HelpAttribute` ao `Display` método na classe. Os construtores públicos de uma classe de atributo controlam as informações que devem ser fornecidas quando o atributo é anexado a uma entidade de programa. As informações adicionais podem ser fornecidas ao referenciar propriedades públicas de leitura-gravação da classe de atributo (como a referência anterior à propriedade `Topic`).

O exemplo a seguir mostra como as informações de atributo para uma determinada entidade de programa podem ser recuperadas em tempo de execução usando reflexão.

C#

```
using System;
using System.Reflection;

class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine(" Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }

    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```

Quando um atributo específico for solicitado por meio de reflexão, o construtor para a classe de atributo será invocado com as informações fornecidas na origem do programa e a instância do atributo resultante será retornada. Se forem fornecidas informações adicionais por meio de propriedades, essas propriedades serão definidas para os valores fornecidos antes que a instância do atributo seja retornada.

# 1 Scope

Article • 01/13/2022

This specification describes the form and establishes the interpretation of programs written in the C# programming language. It describes

- The representation of C# programs;
- The syntax and constraints of the C# language;
- The semantic rules for interpreting C# programs;
- The restrictions and limits imposed by a conforming implementation of C#.

This specification does not describe

- The mechanism by which C# programs are transformed for use by a data-processing system;
- The mechanism by which C# applications are invoked for use by a data-processing system;
- The mechanism by which input data are transformed for use by a C# application;
- The mechanism by which output data are transformed after being produced by a C# application;
- The size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;
- All minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

## 2 Normative references

Article • 03/18/2022

The following normative documents contain provisions, which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this specification are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid specifications.

ISO/IEC 23271:2012, *Common Language Infrastructure (CLI), Partition IV: Base Class Library (BCL), Extended Numerics Library, and Extended Array Library*.

ISO 80000-2, *Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*.

ISO/IEC 2382, *Information technology — Vocabulary*.

ISO/IEC 60559:2020, *Information technology — Microprocessor Systems — Floating-Point arithmetic*

The Unicode Consortium. The Unicode Standard,

<https://www.unicode.org/standard/standard.html> ↗

# 3 Terms and definitions

Article • 06/14/2022

For the purposes of this specification, the following definitions apply. Other terms are defined where they appear in *italic* type or on the left side of a syntax rule. Terms explicitly defined in this specification are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this specification are to be interpreted according to ISO/IEC 2382.1. Mathematical symbols not defined in this specification are to be interpreted according to ISO 80000-2.

- **application**
  - assembly with an entry point
- **application domain**
  - entity that enables application isolation by acting as a container for application state
- **argument**
  - expression in the comma-separated list bounded by the parentheses in a method or instance constructor call expression or bounded by the square brackets in an element access expression
- **assembly**
  - one or more files output by the compiler as a result of program compilation
- **behavior**
  - external appearance or action
- **behavior, implementation-defined**
  - unspecified behavior where each implementation documents how the choice is made
- **behavior, undefined**
  - behavior, upon use of a non-portable or erroneous construct or of erroneous data, for which this specification imposes no requirements
- **behavior, unspecified**
  - behavior where this specification provides two or more possibilities and imposes no further requirements on which is chosen in any instance
- **character** (when used without a qualifier)
  - In the context of a non-Unicode encoding, the meaning of character in that encoding; or
  - In the context of a character literal or a value of type char, a Unicode code point in the range U+0000 to U+FFFF (including surrogate code points), that is a UTF-16 code unit; or
  - Otherwise, a Unicode code point
- **class library**

- assembly that can be used by other assemblies
- **compilation unit**
  - ordered sequence of Unicode characters that is input to a compiler
- **diagnostic message**
  - message belonging to an implementation-defined subset of the implementation's output messages
- **error, compile-time**
  - error reported during program translation
- **exception**
  - exceptional condition reported during program execution
- **implementation**
  - particular set of software (running in a particular translation environment under particular control options) that performs translation of programs for, and supports execution of methods in, a particular execution environment
- **module**
  - the contents of an assembly produced by a compiler. Some implementations may have facilities to produce assemblies that contain more than one module. The behavior in such situations is outside the scope of this specification
- **namespace**
  - logical organizational system grouping related program elements
- **parameter**
  - variable declared as part of a method, instance constructor, operator, or indexer definition, which acquires a value on entry to that function member
- **program**
  - one or more compilation units that are presented to the compiler and are run or executed by an execution environment
- **unsafe code**
  - code that is permitted to perform such lower-level operations as declaring and operating on pointers, performing conversions between pointers and integral types, and taking the address of variables
- **warning, compile-time**
  - informational message reported during program translation, which is intended to identify a potentially questionable usage of a program element

# 4 General description

Article • 04/08/2022

## This text is informative.

This specification is intended to be used by implementers, academics, and application programmers. As such, it contains a considerable amount of explanatory material that, strictly speaking, is not necessary in a formal language specification.

This specification is divided into the following subdivisions: front matter; language syntax, constraints, and semantics; and annexes.

Examples are provided to illustrate possible forms of the constructions described. References are used to refer to related clauses. Notes are provided to give advice or guidance to implementers or programmers. Annexes provide additional information and summarize the information contained in this specification.

## End of informative text.

Informative text is indicated in the following ways:

1. Whole or partial clauses or annexes delimited by “**This clause/text is informative**” and “**End of informative text**”.
2. *Example*: The following example ... code fragment, possibly with some narrative ... *end example* The *Example*: and *end example* markers are in the same paragraph for single paragraph examples. If an example spans multiple paragraphs, the *end example* marker should be its own paragraph.
3. *Note*: narrative ... *end note* The *Note*: and *end note* markers are in the same paragraph for single paragraph notes. If a note spans multiple paragraphs, the *end note* marker should be its own paragraph.

All text not marked as being informative is normative.

# 5 Conformance

Article • 03/18/2022

Conformance is of interest to the following audiences:

- Those designing, implementing, or maintaining C# implementations.
- Governmental or commercial entities wishing to procure C# implementations.
- Testing organizations wishing to provide a C# conformance test suite.
- Programmers wishing to port code from one C# implementation to another.
- Educators wishing to teach Standard C#.
- Authors wanting to write about Standard C#.

As such, conformance is most important, and the bulk of this specification is aimed at specifying the characteristics that make C# implementations and C# programs conforming ones.

The text in this specification that specifies requirements is considered *normative*. All other text in this specification is *informative*; that is, for information purposes only. Unless stated otherwise, all text is normative. Normative text is further broken into *required* and *conditional* categories. *Conditionally normative* text specifies a feature and its requirements where the feature is optional. However, if that feature is provided, its syntax and semantics shall be exactly as specified.

Undefined behavior is indicated in this specification only by the words 'undefined behavior.'

A *strictly conforming program* shall use only those features of the language specified in this specification as being required. (This means that a strictly conforming program cannot use any conditionally normative feature.) It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior.

A *conforming implementation* of C# shall accept any strictly conforming program.

A conforming implementation of C# shall provide and support all the types, values, objects, properties, methods, and program syntax and semantics described in the normative (but not the conditionally normative) parts in this specification.

A conforming implementation of C# shall interpret characters in conformance with the Unicode Standard. Conforming implementations shall accept compilation units encoded with the UTF-8 encoding form.

A conforming implementation of C# shall not successfully translate source containing a #error preprocessing directive unless it is part of a group skipped by conditional

compilation.

A conforming implementation of C# shall produce at least one diagnostic message if the source program violates any rule of syntax, or any negative requirement (defined as a “shall” or “shall not” or “error” or “warning” requirement), unless that requirement is marked with the words “no diagnostic is required”.

A conforming implementation of C# is permitted to provide additional types, values, objects, properties, and methods beyond those described in this specification, provided they do not alter the behavior of any strictly conforming program. Conforming implementations are required to diagnose programs that use extensions that are ill formed according to this specification. Having done so, however, they can compile and execute such programs. (The ability to have extensions implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this specification.)

A conforming implementation of C# shall be accompanied by a document that defines all implementation-defined characteristics, and all extensions.

A conforming implementation of C# shall support the class library documented in Annex C. This library is included by reference in this specification.

A ***conforming program*** is one that is acceptable to a conforming implementation. (Such a program is permitted to contain extensions or conditionally normative features.)

# Estrutura lexical

Artigo • 16/09/2021

## Programas

Um **programa** em C# \* é composto por um ou mais *arquivos de origem*, conhecidos formalmente como \_ unidades de *compilação*\* (unidades de **compilação**). Um arquivo de origem é uma sequência ordenada de caracteres Unicode. Normalmente, os arquivos de origem têm uma correspondência um-para-um com arquivos em um sistema de arquivos, mas essa correspondência não é necessária. Para portabilidade máxima, é recomendável que os arquivos em um sistema de arquivos sejam codificados com a codificação UTF-8.

Em termos conceituais, um programa é compilado usando três etapas:

1. Transformação, que converte um arquivo de um repertório de caractere específico e esquema de codificação em uma sequência de caracteres Unicode.
2. Análise lexical, que traduz um fluxo de caracteres de entrada Unicode em um fluxo de tokens.
3. Análise sintática, que traduz o fluxo de tokens no código executável.

## Gramática

Essa especificação apresenta a sintaxe da linguagem de programação C# usando duas gramáticas. A \***gramática lexical** \_ (**gramática lexical**) define como os caracteres Unicode são combinados em terminadores de linha de formulário, espaço em branco, comentários, tokens e diretivas de pré-processamento. A *sintaxe sintática*\* (**gramática sintática**) define como os tokens resultantes da gramática léxica são combinados para formar programas em C#.

## Notação gramatical

As gramáticas léxicas e sintáticas são apresentadas no formulário Backus-Naur usando a notação da ferramenta gramática ANTLR.

## Gramática lexical

A gramática lexical do C# é apresentada em [análise lexical](#), [tokens](#) e [diretivas de pré-processamento](#). Os símbolos de terminal da gramática lexical são os caracteres do

conjunto de caracteres Unicode, e a gramática léxica especifica como os caracteres são combinados para tokens de formulário ([tokens](#)), espaços em branco ([espaço em branco](#)), comentários ([comentários](#)) e diretivas de pré-processamento ([diretivas de pré-processamento](#)).

Cada arquivo de origem em um programa C# deve estar de acordo com a produção de *entrada* da gramática léxica ([análise lexical](#)).

## Gramática sintática

A gramática sintática do C# é apresentada nos capítulos e apêndices que seguem este capítulo. Os símbolos de terminal da gramática sintática são os tokens definidos pela gramática lexical e a gramática sintática especifica como os tokens são combinados para formar programas em C#.

Cada arquivo de origem em um programa C# deve estar de acordo com o *compilation\_unit* produção da gramática sintática ([unidades de compilação](#)).

## Análise léxica

A produção de *entrada* define a estrutura lexical de um arquivo de origem C#. Cada arquivo de origem em um programa C# deve estar de acordo com esta produção de gramática lexical.

```
antlr

input
: input_section?
;

input_section
: input_section_part+
;

input_section_part
: input_element* new_line
| pp_directive
;

input_element
: whitespace
| comment
| token
;
```

Cinco elementos básicos compõem a estrutura lexical de um arquivo de origem C#: terminadores de linha ([terminadores de linha](#)), espaço em branco ([espaço em branco](#)), comentários ([comentários](#)), tokens ([tokens](#)) e diretivas de pré-processamento ([diretivas de pré-processamento](#)). Desses elementos básicos, somente os tokens são significativos na gramática sintática de um programa C# ([gramática sintática](#)).

O processamento lexical de um arquivo de origem C# consiste em reduzir o arquivo em uma sequência de tokens que se torna a entrada para a análise sintática. Terminadores de linha, espaços em branco e comentários podem servir a tokens separados, e diretivas de pré-processamento podem fazer com que as seções do arquivo de origem sejam ignoradas, mas, caso contrário, esses elementos léxicos não têm impacto sobre a estrutura sintática de um programa em C#.

No caso de literais de cadeia de caracteres interpolados ([literais de cadeia de caracteres interpolados](#)), um único token é produzido inicialmente pela análise lexical, mas é dividido em vários elementos de entrada que estão sujeitos repetidamente à análise lexical até que todos os literais de cadeia de caracteres interpolados tenham sido resolvidos. Os tokens resultantes servem como entrada para a análise sintática.

Quando várias produções de gramática léxicas correspondem a uma sequência de caracteres em um arquivo de origem, o processamento lexical sempre forma o mais longo possível elemento léxico. Por exemplo, a sequência de caracteres `//` é processada como o início de um comentário de linha única porque esse elemento léxico é maior do que um único `/` token.

## Terminadores de linha

Terminadores de linha dividem os caracteres de um arquivo de origem C# em linhas.

```
antlr

new_line
: '<Carriage return character (U+000D)>'
| '<Line feed character (U+000A)>'
| '<Carriage return character (U+000D) followed by line feed character
(U+000A)>'
| '<Next line character (U+0085)>'
| '<Line separator character (U+2028)>'
| '<Paragraph separator character (U+2029)>'
;
```

Para compatibilidade com ferramentas de edição de código-fonte que adicionam marcadores de fim de arquivo e para permitir que um arquivo de origem seja exibido

como uma sequência de linhas corretamente encerradas, as seguintes transformações são aplicadas, em ordem, a cada arquivo de origem em um programa em C#:

- Se o último caractere do arquivo de origem for um caractere de controle Z ( U+001A ), esse caractere será excluído.
- Um caractere de retorno de carro ( U+000D ) é adicionado ao final do arquivo de origem se esse arquivo de origem não estiver vazio e se o último caractere do arquivo de origem não for um retorno de carro ( U+000D ), um feed de linha ( U+000A ), um separador de linha ( U+2028 ) ou um separador de parágrafo ( U+2029 ).

## Comentários

Dois formulários de comentários são compatíveis: comentários de linha única e comentários delimitados.\ Os **comentários de linha única** \_ começam com os caracteres // e se estendem até o final da linha de origem. Os \_comentários delimitados\_ começam com os caracteres /\* e terminam com os caracteres \*/ . Os comentários delimitados podem abranger várias linhas.

```
antlr

comment
    : single_line_comment
    | delimited_comment
    ;

single_line_comment
    : '//' input_character*
    ;

input_character
    : '<Any Unicode character except a new_line_character>'
    ;

new_line_character
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;

delimited_comment
    : '/*' delimited_comment_section* asterisk+ '/'
    ;

delimited_comment_section
    : '/'
```

```

| asterisk* not_slash_or_asterisk
;

asterisk
: '*'
;

not_slash_or_asterisk
: '<Any Unicode character except / or *>'
;

```

Os comentários não podem ser aninhados. As sequências de caracteres /\* e \*/ não têm um significado especial dentro de um // comentário e as sequências de caracteres // e /\* não têm nenhum significado especial dentro de um comentário delimitado.

Os comentários não são processados em literais de caractere e de cadeia de caracteres.

O exemplo

```
C#
/* Hello, world program
   This program writes "hello, world" to the console
*/
class Hello
{
    static void Main() {
        System.Console.WriteLine("hello, world");
    }
}
```

inclui um comentário delimitado.

O exemplo

```
C#
// Hello, world program
// This program writes "hello, world" to the console
//
class Hello // any name will do for this class
{
    static void Main() { // this method must be named "Main"
        System.Console.WriteLine("hello, world");
    }
}
```

mostra vários comentários de linha única.

## Espaço em branco

O espaço em branco é definido como qualquer caractere com a classe Unicode ZS (que inclui o caractere de espaço), bem como o caractere de tabulação horizontal, o caractere de tabulação vertical e o caractere de feed de formulário.

```
antlr

whitespace
: '<Any character with Unicode class Zs>'
| '<Horizontal tab character (U+0009)>'
| '<Vertical tab character (U+000B)>'
| '<Form feed character (U+000C)>'
;
```

## Tokens

Há vários tipos de tokens: identificadores, palavras-chave, literais, operadores e pontuadores. Os comentários e o espaço em branco não são tokens, embora atuem como separadores para tokens.

```
antlr

token
: identifier
| keyword
| integer_literal
| real_literal
| character_literal
| string_literal
| interpolated_string_literal
| operator_or_punctuator
;
```

## Sequências de escape de caractere Unicode

Uma sequência de escape de caractere Unicode representa um caractere Unicode. As sequências de escape de caractere Unicode são processadas em identificadores ([identificadores](#)), literais de caracteres ([literais de caracteres](#)) e literais de cadeia de caracteres regulares ([literais de cadeia de caracteres](#)). Um escape de caractere Unicode não é processado em nenhum outro local (por exemplo, para formar um operador, um pontuador ou uma palavra-chave).

```
antlr
```

```
unicode_escape_sequence
: '\u' hex_digit hex_digit hex_digit
| '\U' hex_digit hex_digit hex_digit hex_digit hex_digit
hex_digit hex_digit
;
```

Uma sequência de escape Unicode representa o caractere Unicode único formado pelo número hexadecimal após os `\u` caracteres "" ou "`\u`". Como o C# usa uma codificação de 16 bits de pontos de código Unicode em caracteres e valores de cadeia, um caractere Unicode no intervalo de U + 10000 a U + 10FFFF não é permitido em um literal de caractere e é representado usando um par substituto Unicode em um literal de cadeia de caracteres. Não há suporte para caracteres Unicode com pontos de código acima de 0x10FFFF.

Várias traduções não são executadas. Por exemplo, a cadeia de caracteres literal "`\u005Cu005C`" é equivalente a "`\u005C`" em vez de "`\`". O valor Unicode `\u005C` é o caractere "`\`".

O exemplo

```
C#
class Class1
{
    static void Test(bool \u0066) {
        char c = '\u0066';
        if (\u0066)
            System.Console.WriteLine(c.ToString());
    }
}
```

mostra vários usos de `\u0066`, que é a sequência de escape para a letra "`f`". O programa é equivalente a

```
C#
class Class1
{
    static void Test(bool f) {
        char c = 'f';
        if (f)
            System.Console.WriteLine(c.ToString());
    }
}
```

# Identificadores

As regras para identificadores fornecidos nesta seção correspondem exatamente às recomendadas pelo anexo 31 padrão Unicode, exceto que o sublinhado é permitido como um caractere inicial (como é tradicional na linguagem de programação C), as sequências de escape Unicode são permitidas em identificadores e o '@' caractere "" é permitido como um prefixo para habilitar as palavras-chave a serem usadas como identificadores.

```
antlr

identifier
: available_identifier
| '@' identifier_or_keyword
;

available_identifier
: '<An identifier_or_keyword that is not a keyword>'
;

identifier_or_keyword
: identifier_start_character identifier_part_character*
;

identifier_start_character
: letter_character
| '_'
;

identifier_part_character
: letter_character
| decimal_digit_character
| connecting_character
| combining_character
| formatting_character
;

letter_character
: '<A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl>'
| '<A unicode_escape_sequence representing a character of classes Lu,
Ll, Lt, Lm, Lo, or Nl>'
;

combining_character
: '<A Unicode character of classes Mn or Mc>'
| '<A unicode_escape_sequence representing a character of classes Mn or
Mc>'
;

decimal_digit_character
: '<A Unicode character of the class Nd>'
| '<A unicode_escape_sequence representing a character of the class Nd>'
```

```

;

connecting_character
: '<A Unicode character of the class Pc>'
| '<A unicode_escape_sequence representing a character of the class Pc>'
;

formatting_character
: '<A Unicode character of the class Cf>'
| '<A unicode_escape_sequence representing a character of the class Cf>'
;

```

Para obter informações sobre as classes de caracteres Unicode mencionadas acima, consulte o padrão Unicode, versão 3,0, seção 4,5.

Exemplos de identificadores válidos incluem "`identifier1`", "`_identifier2`" e "`@if`".

Um identificador em um programa de conformidade deve estar no formato canônico definido pelo formulário de normalização Unicode C, conforme definido pelo anexo 15 padrão Unicode. O comportamento ao encontrar um identificador que não está no formato de normalização C é definido pela implementação; no entanto, um diagnóstico não é necessário.

O prefixo "`@`" permite o uso de palavras-chave como identificadores, o que é útil ao fazer a interface com outras linguagens de programação. O caractere `@` não é realmente parte do identificador, portanto, o identificador pode ser visto em outros idiomas como um identificador normal, sem o prefixo. Um identificador com um `@` prefixo é chamado de *identificador textual*. O uso do `@` prefixo para identificadores que não são palavras-chave é permitido, mas altamente desencorajado como uma questão de estilo.

O exemplo:

```
C#

class @class
{
    public static void @static(bool @bool) {
        if (@bool)
            System.Console.WriteLine("true");
        else
            System.Console.WriteLine("false");
    }
}

class Class1
{
    static void M() {
        cl\u0061ss.st\u0061tic(true);
```

```
    }  
}
```

define uma classe denominada " `class` " com um método estático chamado " `static` " que usa um parâmetro denominado " `bool` ". Observe que, como os escapes Unicode não são permitidos em palavras-chave, o token " `c1\u0061ss` " é um identificador e é o mesmo identificador de " `@class` ".

Dois identificadores são considerados iguais se forem idênticos depois que as transformações a seguir forem aplicadas, na ordem:

- O prefixo " `@` ", se usado, será removido.
- Cada *unicode\_escape\_sequence* é transformada em seu caractere Unicode correspondente.
- Todas as *formatting\_character*s são removidas.

Os identificadores que contêm dois caracteres de sublinhado consecutivos ( `U+005F` ) são reservados para uso pela implementação. Por exemplo, uma implementação pode fornecer palavras-chave estendidas que começam com dois sublinhados.

## Palavras-chave

Uma *palavra-chave* é uma sequência semelhante a identificador de caracteres que é reservada e não pode ser usada como um identificador, exceto quando precedida pelo `@` caractere.

```
antlr
```

```
keyword  
: 'abstract' | 'as' | 'base' | 'bool' | 'break'  
| 'byte' | 'case' | 'catch' | 'char' | 'checked'  
| 'class' | 'const' | 'continue' | 'decimal' | 'default'  
| 'delegate' | 'do' | 'double' | 'else' | 'enum'  
| 'event' | 'explicit' | 'extern' | 'false' | 'finally'  
| 'fixed' | 'float' | 'for' | 'foreach' | 'goto'  
| 'if' | 'implicit' | 'in' | 'int' | 'interface'  
| 'internal' | 'is' | 'lock' | 'long' | 'namespace'  
| 'new' | 'null' | 'object' | 'operator' | 'out'  
| 'override' | 'params' | 'private' | 'protected' | 'public'  
| 'readonly' | 'ref' | 'return' | 'sbyte' | 'sealed'  
| 'short' | 'sizeof' | 'stackalloc' | 'static' | 'string'  
| 'struct' | 'switch' | 'this' | 'throw' | 'true'  
| 'try' | 'typeof' | 'uint' | 'ulong' | 'unchecked'  
| 'unsafe' | 'ushort' | 'using' | 'virtual' | 'void'  
| 'volatile' | 'while'  
;
```

Em alguns lugares da gramática, os identificadores específicos têm um significado especial, mas não são palavras-chave. Esses identificadores são, às vezes, chamados de "palavras-chave contextuais". Por exemplo, dentro de uma declaração de propriedade, os `get` identificadores "" e " `set` " têm significado especial ([acessadores](#)). Um identificador diferente de `get` ou `set` nunca é permitido nesses locais, portanto, esse uso não entra em conflito com o uso dessas palavras como identificadores. Em outros casos, como com o identificador " `var` " em declarações de variáveis locais digitadas implicitamente ([declarações de variáveis locais](#)), uma palavra-chave contextual pode entrar em conflito com nomes declarados. Nesses casos, o nome declarado tem precedência sobre o uso do identificador como uma palavra-chave contextual.

## Literais

Um *literal* é uma representação de código-fonte de um valor.

```
antlr
```

```
literal
: boolean_literal
| integer_literal
| real_literal
| character_literal
| string_literal
| null_literal
;
```

## Literais booleanos

Há dois valores literais booleanos: `true` e `false`.

```
antlr
```

```
boolean_literal
: 'true'
| 'false'
;
```

O tipo de um *boolean\_literal* é `bool`.

## Literais inteiros

Literais inteiros são usados para gravar valores de tipos `int`, `uint`, `long` e `ulong`. Os literais inteiros têm duas formas possíveis: decimal e hexadecimal.

antlr

```
integer_literal
: decimal_integer_literal
| hexadecimal_integer_literal
;

decimal_integer_literal
: decimal_digit+ integer_type_suffix?
;

decimal_digit
: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
;

integer_type_suffix
: 'U' | 'u' | 'L' | 'l' | 'UL' | 'U1' | 'uL' | 'u1' | 'LU' | 'Lu' | '1U'
| '1u'
;

hexadecimal_integer_literal
: '0x' hex_digit+ integer_type_suffix?
| '0X' hex_digit+ integer_type_suffix?
;

hex_digit
: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
| 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
;
```

O tipo de um literal inteiro é determinado da seguinte maneira:

- Se o literal não tiver nenhum sufixo, ele terá o primeiro desses tipos em que seu valor pode ser representado: `int` , `uint` , `long` , `ulong` .
- Se o literal for sufixado pelo `U` ou `u` , ele terá o primeiro desses tipos em que seu valor pode ser representado: `uint` , `ulong` .
- Se o literal for sufixado pelo `L` ou `l` , ele terá o primeiro desses tipos em que seu valor pode ser representado: `long` , `ulong` .
- Se o literal for sufixado por `UL` , `U1` , `uL` , `u1` , `LU` , `Lu` , `1U` ou `1u` , será do tipo `ulong` .

Se o valor representado por um literal inteiro estiver fora do intervalo do `ulong` tipo, ocorrerá um erro em tempo de compilação.

Como uma questão de estilo, é recomendável que " `L` " seja usado em vez de " `l` " ao escrever literais do tipo `long` , já que é fácil confundir a letra " `l` " com o dígito " `1` ".

Para permitir que os menores `int` valores e possíveis `long` sejam gravados como literais inteiros decimais, existem as duas seguintes regras:

- Quando um *decimal\_integer\_literal* com o valor 2147483648 ( $2^{31}$ ) e nenhum *integer\_type\_suffix* aparece como o token imediatamente após um token do operador menos unário ([operador menos unário](#)), o resultado é uma constante do tipo `int` com o valor -2147483648 ( $-2^{31}$ ). Em todas as outras situações, tal *decimal\_integer\_literal* é do tipo `uint`.
- Quando um *decimal\_integer\_literal* com o valor 9.223.372.036.854.775.808 ( $2^{63}$ ) e nenhum *integer\_type\_suffix* ou o *integer\_type\_suffix* `L` ou `l` aparece como o token imediatamente após um token do operador de subtração unário ([operador menos unário](#)), o resultado é uma constante do tipo `long` com o valor -9.223.372.036.854.775.808 ( $-2^{63}$ ). Em todas as outras situações, tal *decimal\_integer\_literal* é do tipo `ulong`.

## Literais reais

Os literais reais são usados para gravar valores de tipos `float`, `double` e `decimal`.

```
antlr

real_literal
: decimal_digit+ '.' decimal_digit+ exponent_part? real_type_SUFFIX?
| '.' decimal_digit+ exponent_part? real_type_SUFFIX?
| decimal_digit+ exponent_part real_type_SUFFIX?
| decimal_digit+ real_type_SUFFIX
;

exponent_part
: 'e' sign? decimal_digit+
| 'E' sign? decimal_digit+
;

sign
: '+'
| '-'
;

real_TYPE_SUFFIX
: 'F' | 'f' | 'D' | 'd' | 'M' | 'm'
;
```

Se nenhum *real\_type\_SUFFIX* for especificado, o tipo do literal real será `double`. Caso contrário, o sufixo de tipo real determina o tipo do literal real, da seguinte maneira:

- Um literal real sufixado por `F` ou `f` é do tipo `float`. Por exemplo, os literais `1f`, `1.5f`, `1e10f` e `123.456F` são todos do tipo `float`.

- Um literal real sufixado por `D` ou `d` é do tipo `double`. Por exemplo, os literais `1d`, `1.5d`, `1e10d` e `123.456D` são todos do tipo `double`.
- Um literal real sufixado por `M` ou `m` é do tipo `decimal`. Por exemplo, os literais `1m`, `1.5m`, `1e10m` e `123.456M` são todos do tipo `decimal`. Esse literal é convertido em um `decimal` valor usando o valor exato e, se necessário, arredondando para o valor representável mais próximo usando o arredondamento do banco ([o tipo decimal](#)). Qualquer escala aparente no literal é preservada, a menos que o valor seja arredondado ou o valor seja zero (nesse último caso, o sinal e a escala serão 0). Portanto, o literal `2.900m` será analisado para formar o decimal com sinal `0`, coeficiente `2900` e escala `3`.

Se o literal especificado não puder ser representado no tipo indicado, ocorrerá um erro em tempo de compilação.

O valor de um literal real do tipo `float` ou `double` é determinado usando o modo IEEE "Round to mais próximo".

Observe que, em um literal real, os dígitos decimais sempre são necessários após o ponto decimal. Por exemplo, `1.3F` é um literal real, mas `1.F` não é.

## Literais de caracteres

Um literal de caractere representa um único caractere e geralmente consiste em um caractere entre aspas, como em `'a'`.

Observação: a notação gramatical ANTLR torna o seguinte confuso! No ANTLR, quando você escreve, `\'` significa uma aspa simples `'`. E quando você escreve `\\"` -lo significa uma barra invertida única `\`. Portanto, a primeira regra para um literal de caractere significa que ela começa com uma aspa simples, depois um caractere e uma aspa simples. E as onze sequências de escape simples possíveis são,,,,,,,,,, `\'` `\"` `\\"` `\0` `\a` `\b` `\f` `\n` `\r` `\t` , `\v` .

```
antlr

character_literal
: '\' character '\''
;

character
: single_character
| simple_escape_sequence
| hexadecimal_escape_sequence
| unicode_escape_sequence
;

```

```

single_character
: '<Any character except \' (U+0027), \\" (U+005C), and
new_line_character>'
;

simple_escape_sequence
: '\\\\' | '\\"' | '\\\\\\' | '\\0' | '\\a' | '\\b' | '\\f' | '\\n' |
'\\r' | '\\t' | '\\v'
;

hexadecimal_escape_sequence
: '\\x' hex_digit hex_digit? hex_digit? hex_digit?;

```

Um caractere que segue um caractere de barra invertida ( \ ) em um *caractere* deve ser um dos seguintes caracteres: ' " \ 0 a b f n r t u U x , v . Caso contrário, ocorrerá um erro de tempo de compilação.

Uma sequência de escape hexadecimal representa um único caractere Unicode, com o valor formado pelo número hexadecimal após " \x ".

Se o valor representado por um literal de caractere for maior que U+FFFF , ocorrerá um erro em tempo de compilação.

Uma sequência de escape de caractere Unicode ([sequências de escape de caractere Unicode](#)) em um literal de caractere deve estar no intervalo U+0000 para U+FFFF .

Uma sequência de escape simples representa uma codificação de caractere Unicode, conforme descrito na tabela a seguir.

Sequência de escape	Nome do caractere	Codificação Unicode
\'	Aspas simples	0x0027
\"	Aspas duplas	0x0022
\\\	Barra invertida	0x005C
\0	Nulo	0x0000
\a	Alerta	0x0007
\b	Backspace	0x0008
\f	Avanço de formulário	0x000C
\n	Nova linha	0x000A
\r	Retorno de carro	0x000D

Sequência de escape	Nome do caractere	Codificação Unicode
\t	Guia horizontal	0x0009
\v	Guia vertical	0x000B

O tipo de um *character\_literal* é `char`.

## Literais de cadeia de caracteres

O C# dá suporte a duas formas de literais de cadeia de caracteres: \* literais de **cadeia de caracteres regulares** \_ e \_ *literais de cadeia de caracteres textuais* \*.

Um literal de cadeia de caracteres regular consiste em zero ou mais caracteres entre aspas duplas, como em `"hello"`, e pode incluir sequências de escape simples (como `\t` para o caractere de tabulação) e sequências de escape hexadecimal e Unicode.

Um literal de cadeia de caracteres textual consiste em um @ caractere seguido por um caractere de aspas duplas, zero ou mais caracteres e um caractere de aspas duplas de fechamento. Um exemplo simples é `@"hello"`. Em um literal de cadeia de caracteres textual, os caracteres entre os delimitadores são interpretados literalmente, a única exceção é um *quote\_escape\_sequence*. Em particular, sequências de escape simples e sequências de escape hexadecimal e Unicode não são processadas em literais de cadeia de caracteres textuais. Um literal de cadeia de caracteres textual pode abranger várias linhas.

```
antlr

string_literal
: regular_string_literal
| verbatim_string_literal
;

regular_string_literal
: '"' regular_string_literal_character* '"'
;

regular_string_literal_character
: single_regular_string_literal_character
| simple_escape_sequence
| hexadecimal_escape_sequence
| unicode_escape_sequence
;

single_regular_string_literal_character
: '<Any character except " (U+0022), \\ (U+005C), and
new_line_character>'
```

```

;

verbatim_string_literal
: '@' verbatim_string_literal_character* ''
;

verbatim_string_literal_character
: single_verbatim_string_literal_character
| quote_escape_sequence
;

single_verbatim_string_literal_character
: '<any character except ">"'
;

quote_escape_sequence
: """
;

```

Um caractere que segue um caractere de barra invertida (`\`) em um *regular\_string\_literal\_character* deve ser um dos seguintes caracteres: `,`, `'`, `"`, `\`, `0`, `a`, `b`, `f`, `n`, `r`, `t`, `u`, `U`, `x`, `v`. Caso contrário, ocorrerá um erro de tempo de compilação.

## O exemplo

C#

```

string a = "hello, world";                                // hello, world
string b = @"hello, world";                               // hello, world

string c = "hello \t world";                            // hello      world
string d = @"hello \t world";                           // hello \t world

string e = "Joe said \"Hello\" to me";                // Joe said "Hello" to me
string f = @"Joe said ""Hello"" to me";               // Joe said "Hello" to me

string g = @"\server\share\file.txt";                  // \server\share\file.txt
string h = @"\\server\share\file.txt";                 // \\server\share\file.txt

string i = "one\r\n\two\r\n\tthree";
string j = @"one
two
three";

```

mostra uma variedade de literais de cadeia de caracteres. O último literal de cadeia de caracteres, `j`, é um literal de cadeia de caracteres textual que abrange várias linhas. Os caracteres entre as aspas, incluindo espaços em branco, como novos caracteres de linha, são preservados literalmente.

Como uma sequência de escape hexadecimal pode ter um número variável de dígitos hexadecimais, a cadeia de caracteres literal `"\x123"` contém um único caractere com o valor hexadecimal 123. Para criar uma cadeia de caracteres que contém o caractere com o valor hexadecimal 12 seguido pelo caractere 3, é possível escrever `"\x00123"` ou `"\x12" + "3"` em vez disso.

O tipo de um *string\_literal* é `string`.

Cada literal de cadeia de caracteres não resulta necessariamente em uma nova instância de cadeia de caracteres. Quando dois ou mais literais de cadeia de caracteres equivalentes de acordo com o operador de igualdade de cadeia de caracteres ([operadores de igualdade de cadeia de caracteres](#)) aparecem no mesmo programa, esses literais de cadeia de caracteres fazem referência à mesma instância de cadeia de caracteres. Por exemplo, a saída produzida por

```
C#  
  
class Test  
{  
    static void Main() {  
        object a = "hello";  
        object b = "hello";  
        System.Console.WriteLine(a == b);  
    }  
}
```

ocorre `True` porque os dois literais se referem à mesma instância de cadeia de caracteres.

## Literais de cadeia de caracteres interpolados

Literais de cadeia de caracteres interpolados são semelhantes a literais de cadeia de caracteres, mas contêm buracos delimitados por `{` e `}`, onde as expressões podem ocorrer. Em tempo de execução, as expressões são avaliadas com a finalidade de ter seus formulários textuais substituídos na cadeia de caracteres no local onde o orifício ocorre. A sintaxe e a semântica da interpolação de cadeias de caracteres são descritas na seção ([cadeias de caracteres interpoladas](#)).

Como literais de cadeia de caracteres, literais de cadeia de caracteres interpolados podem ser regulares ou textuais. Os literais de cadeia de caracteres regulares interpolados são delimitados por `$"` e `"`, e os literais de cadeia de caracteres textuais interpolados são delimitados por `$@"` e `"`.

Assim como outros literais, a análise léxica de um literal de cadeia de caracteres interpolado inicialmente resulta em um único token, de acordo com a gramática abaixo. No entanto, antes da análise sintática, o único token de um literal de cadeia de caracteres interpolado é dividido em vários tokens para as partes da cadeia de caracteres que envolvem os orifícios, e os elementos de entrada que ocorrem nos orifícios são analisados lexicalmente de novo. Isso pode, por sua vez, produzir mais literais de cadeia de caracteres interpolados a serem processados, mas, se for lexicalmente correto, eventualmente levará a uma sequência de tokens para que a análise sintática seja processada.

```
antlr
```

```
interpolated_string_literal
: '$' interpolated_regular_string_literal
| '$' interpolated_verbatim_string_literal
;

interpolated_regular_string_literal
: interpolated_regular_string_whole
| interpolated_regular_string_start
interpolated_regular_string_literal_body interpolated_regular_string_end
;

interpolated_regular_string_literal_body
: regular_balanced_text
| interpolated_regular_string_literal_body
interpolated_regular_string_mid regular_balanced_text
;

interpolated_regular_string_whole
: """ interpolated_regular_string_character* """
;

interpolated_regular_string_start
: """ interpolated_regular_string_character* '{'
;

interpolated_regular_string_mid
: interpolation_format? '}'
interpolated_regular_string_characters_after_brace? '{'
;

interpolated_regular_string_end
: interpolation_format? '}'
interpolated_regular_string_characters_after_brace? """
;

interpolated_regular_string_characters_after_brace
: interpolated_regular_string_character_no_brace
| interpolated_regular_string_characters_after_brace
interpolated_regular_string_character
```

```

;

interpolated_regular_string_character
: single_interpolated_regular_string_character
| simple_escape_sequence
| hexadecimal_escape_sequence
| unicode_escape_sequence
| open_brace_escape_sequence
| close_brace_escape_sequence
;

interpolated_regular_string_character_no_brace
: '<Any interpolated_regular_string_character except
close_brace_escape_sequence and any hexadecimal_escape_sequence or
unicode_escape_sequence designating } (U+007D)>'
;

single_interpolated_regular_string_character
: '<Any character except \" (U+0022), \\ (U+005C), { (U+007B), }
(U+007D), and new_line_character>'
;

open_brace_escape_sequence
: '{{'
;

close_brace_escape_sequence
: '}}'
;

regular_balanced_text
: regular_balanced_text_part+
;

regular_balanced_text_part
: single_regular_balanced_text_character
| delimited_comment
| '@' identifier_or_keyword
| string_literal
| interpolated_string_literal
| '(' regular_balanced_text ')'
| '[' regular_balanced_text ']'
| '{' regular_balanced_text '}'
;

single_regular_balanced_text_character
: '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $
(U+0024), ( (U+0028), ) (U+0029), [ (U+005B), ] (U+005D), { (U+007B), }
(U+007D) and new_line_character>'
| '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
;

interpolation_format
: ':' interpolation_format_character+
;

```

```
interpolation_format_character
: '<Any character except \' (U+0022), : (U+003A), { (U+007B) and } (U+007D)>'
;

interpolated_verbatim_string_literal
: interpolated_verbatim_string_whole
| interpolated_verbatim_string_start
interpolated_verbatim_string_literal_body interpolated_verbatim_string_end
;

interpolated_verbatim_string_literal_body
: verbatim_balanced_text
| interpolated_verbatim_string_literal_body
interpolated_verbatim_string_mid verbatim_balanced_text
;

interpolated_verbatim_string_whole
: '@"' interpolated_verbatim_string_character* '"'
;

interpolated_verbatim_string_start
: '@"' interpolated_verbatim_string_character* '{'
;

interpolated_verbatim_string_mid
: interpolation_format? '}'
interpolated_verbatim_string_characters_after_brace? '{'
;

interpolated_verbatim_string_end
: interpolation_format? '}'
interpolated_verbatim_string_characters_after_brace? '"'
;

interpolated_verbatim_string_characters_after_brace
: interpolated_verbatim_string_character_no_brace
| interpolated_verbatim_string_characters_after_brace
interpolated_verbatim_string_character
;

interpolated_verbatim_string_character
: single_interpolated_verbatim_string_character
| quote_escape_sequence
| open_brace_escape_sequence
| close_brace_escape_sequence
;

interpolated_verbatim_string_character_no_brace
: '<Any interpolated_verbatim_string_character except close_brace_escape_sequence>'
;

single_interpolated_verbatim_string_character
```

```

    : '<Any character except \" (U+0022), { (U+007B) and } (U+007D)>'
    ;

verbatim_balanced_text
: verbatim_balanced_text_part+
;

verbatim_balanced_text_part
: single_verbatim_balanced_text_character
| comment
| '@' identifier_or_keyword
| string_literal
| interpolated_string_literal
| '(' verbatim_balanced_text ')'
| '[' verbatim_balanced_text ']'
| '{' verbatim_balanced_text '}'
;
;

single_verbatim_balanced_text_character
: '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $ (U+0024), ( (U+0028), ) (U+0029), [ (U+005B), ] (U+005D), { (U+007B) and } (U+007D)>'
| '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
;
;
```

Um token de *interpolated\_string\_literal* é reinterpretado como vários tokens e outros elementos de entrada da seguinte maneira, em ordem de ocorrência no *interpolated\_string\_literal*:

- As ocorrências dos seguintes são reinterpretadas como tokens individuais separados: o `$` sinal à esquerda, *interpolated\_regular\_string\_whole*, *interpolated\_regular\_string\_start*, *interpolated\_regular\_string\_mid*, *interpolated\_regular\_string\_end*, *interpolated\_verbatim\_string\_whole*, *interpolated\_verbatim\_string\_start*, *interpolated\_verbatim\_string\_mid* e *interpolated\_verbatim\_string\_end*.
- Ocorrências de *regular\_balanced\_text* e *verbatim\_balanced\_text* entre elas são reprocessadas como um *input\_section* ([análise lexical](#)) e são reinterpretadas como a sequência resultante de elementos de entrada. Isso pode, por sua vez, incluir tokens literais de cadeia de caracteres interpolados a serem reinterpretados.

A análise sintática recombinará os tokens em um *interpolated\_string\_expression* ([cadeias de caracteres interpoladas](#)).

Exemplos de TODO

## O literal nulo

```
antlr
```

```
null_literal
: 'null'
;
```

O *null\_literal* pode ser convertido implicitamente em um tipo de referência ou um tipo anulável.

## Operadores e pontuadores

Há vários tipos de operadores e pontuadores. Os operadores são usados em expressões para descrever as operações que envolvem um ou mais operandos. Por exemplo, a expressão `a + b` usa o operador `+` para adicionar os dois operandos `a` e `b`. Os pontuadores servem para agrupamento e separação.

```
antlr
```

```
operator_or_punctuator
: '{' | '}' | '[' | ']' | '(' | ')' | '.' | ',' | ':' | ';'
| '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '!' | '~'
| '=' | '<' | '>' | '?' | '??' | '::' | '++' | '--' | '&&' | '||'
| '>>' | '==' | '!= | '<=' | '>=' | '+=' | '-=' | '*=' | '/=' | '%='
| '&=' | '|=' | '^=' | '<<' | '<<=' | '=>'
;

right_shift
: '>>'

right_shift_assignment
: '>>='
;
```

A barra vertical nas produções *right\_shift* e *right\_shift\_assignment* são usadas para indicar que, ao contrário de outras produções na gramática sintática, nenhum caractere de qualquer tipo (nem mesmo espaço em branco) é permitido entre os tokens. Essas produções são tratadas especialmente para habilitar a manipulação correta de *type\_parameter\_list*s ([parâmetros de tipo](#)).

## Pré-processando diretivas

As diretivas de pré-processamento fornecem a capacidade de ignorar condicionalmente seções de arquivos de origem, relatar condições de erro e aviso e delinear regiões distintas de código-fonte. O termo "diretivas de pré-processamento" é usado apenas

para consistência com as linguagens de programação C e C++. No C#, não há uma etapa de pré-processamento separada; as diretivas de pré-processamento são processadas como parte da fase de análise lexical.

```
antlr

pp_directive
    : pp_declaraction
    | pp_conditional
    | pp_line
    | pp_diagnostic
    | pp_region
    | pp_pragma
    ;
```

As seguintes diretivas de pré-processamento estão disponíveis:

- `#define` e `#undef`, que são usados para definir e desdefinir, respectivamente, símbolos de compilação condicional([diretivas de declaração](#)).
- `#if`, `#elif`, `#else` e `#endif`, que são usados para ignorar seções de código-fonte ([diretivas de compilação condicional](#)) condicionalmente.
- `#line`, que é usado para controlar os números de linha emitidos para erros e avisos ([diretivas de linha](#)).
- `#error` e `#warning`, que são usados para emitir erros e avisos, respectivamente ([diretivas de diagnóstico](#)).
- `#region` e `#endregion`, que são usados para marcar explicitamente as seções do código-fonte ([diretivas de região](#)).
- `#pragma`, que é usado para especificar informações contextuais opcionais para o compilador ([diretivas pragma](#)).

Uma diretiva de pré-processamento sempre ocupa uma linha separada do código-fonte e sempre começa com um `#` caractere e um nome de diretiva de pré-processamento. Pode ocorrer espaço em branco antes do `#` caractere e entre o `#` caractere e o nome da diretiva.

Uma linha de origem contendo `#define` uma `#undef` diretiva,,, `#if` `#elif` `#else` `#endif` `#line` ou `#endregion` pode terminar com um comentário de linha única.

Comentários delimitados (o `/* */` estilo de comentários) não são permitidos em linhas de origem que contêm diretivas de pré-processamento.

As diretivas de pré-processamento não são tokens e não fazem parte da gramática sintática do C#. No entanto, as diretivas de pré-processamento podem ser usadas para

incluir ou excluir sequências de tokens e podem, dessa forma, afetar o significado de um programa em C#. Por exemplo, quando compilado, o programa:

```
C#  
  
#define A  
#undef B  
  
class C  
{  
#if A  
    void F() {}  
#else  
    void G() {}  
#endif  
  
#if B  
    void H() {}  
#else  
    void I() {}  
#endif  
}
```

resulta na mesma sequência de tokens que o programa:

```
C#  
  
class C  
{  
    void F() {}  
    void I() {}  
}
```

Portanto, embora lexicalmente, os dois programas são bem diferentes, sintaticamente, são idênticos.

## Símbolos de compilação condicional

A funcionalidade de compilação condicional fornecida pelas `#if` diretivas,, `#elif` `#else` e `#endif` é controlada por expressões de pré-processamento ([expressões de pré-processamento](#)) e símbolos de compilação condicional.

```
antlr  
  
conditional_symbol  
: '<Any identifier_or_keyword except true or false>'  
;
```

Um símbolo de compilação condicional tem dois Estados possíveis: `*defined` \_ ou \_ `undefined` \*. No início do processamento lexical de um arquivo de origem, um símbolo de compilação condicional é indefinido, a menos que tenha sido explicitamente definido por um mecanismo externo (como uma opção de compilador de linha de comando). Quando uma `#define` diretiva é processada, o símbolo de compilação condicional chamado nessa diretiva é definido nesse arquivo de origem. O símbolo permanece definido até que uma `#undef` diretiva para o mesmo símbolo seja processada ou até que o final do arquivo de origem seja atingido. Uma implicação disso é que `#define` as `#undef` diretivas e em um arquivo de origem não têm nenhum efeito sobre outros arquivos de origem no mesmo programa.

Quando referenciado em uma expressão de pré-processamento, um símbolo de compilação condicional definido tem o valor booleano `true` e um símbolo de compilação condicional indefinido tem o valor booleano `false`. Não há nenhum requisito de que os símbolos de compilação condicional sejam declarados explicitamente antes de serem referenciados em expressões de pré-processamento. Em vez disso, os símbolos não declarados são simplesmente indefinidos e, portanto, têm o valor `false`.

O espaço de nome para símbolos de compilação condicional é distinto e separado de todas as outras entidades nomeadas em um programa em C#. Símbolos de compilação condicional só podem ser referenciados em `#define` `#undef` diretivas e em expressões de pré-processamento.

## Pré-processando expressões

Expressões de pré-processamento podem ocorrer em `#if` diretivas e `#elif`. Os operadores `!`, `==`, `!=`, `&&` e `||` são permitidos em expressões de pré-processamento, e os parênteses podem ser usados para Agrupamento.

```
antlr

pp_expression
: whitespace? pp_or_expression whitespace?
;

pp_or_expression
: pp_and_expression
| pp_or_expression whitespace? '||' whitespace? pp_and_expression
;

pp_and_expression
: pp_equality_expression
| pp_and_expression whitespace? '&&' whitespace? pp_equality_expression
```

```

;

pp_equality_expression
: pp_unary_expression
| pp_equality_expression whitespace? '==' whitespace?
pp_unary_expression
| pp_equality_expression whitespace? '!= whitespace?
pp_unary_expression
;

pp_unary_expression
: pp_primary_expression
| '!' whitespace? pp_unary_expression
;

pp_primary_expression
: 'true'
| 'false'
| conditional_symbol
| '(' whitespace? pp_expression whitespace? ')'
;

```

Quando referenciado em uma expressão de pré-processamento, um símbolo de compilação condicional definido tem o valor booleano `true` e um símbolo de compilação condicional indefinido tem o valor booleano `false`.

A avaliação de uma expressão de pré-processamento sempre gera um valor booleano. As regras de avaliação para uma expressão de pré-processamento são as mesmas que aquelas para uma expressão constante ([expressões constantes](#)), exceto que as únicas entidades definidas pelo usuário que podem ser referenciadas são símbolos de compilação condicional.

## Diretivas de declaração

As diretivas de declaração são usadas para definir ou não definir símbolos de compilação condicional.

```

antlr

pp_declaration
: whitespace? '#' whitespace? 'define' whitespace conditional_symbol
pp_new_line
| whitespace? '#' whitespace? 'undef' whitespace conditional_symbol
pp_new_line
;

pp_new_line
: whitespace? single_line_comment? new_line
;
```

O processamento de uma `#define` diretiva faz com que o símbolo de compilação condicional fornecido se torne definido, começando com a linha de origem que segue a diretiva. Da mesma forma, o processamento de uma `#undef` diretiva faz com que o símbolo de compilação condicional fornecido se torne indefinido, começando pela linha de origem que segue a diretiva.

As `#define` `#undef` diretivas e em um arquivo de origem devem ocorrer antes do primeiro *token* ([tokens](#)) no arquivo de origem; caso contrário, ocorrerá um erro em tempo de compilação. Em termos intuitivos `#define`, `#undef` as diretivas devem preceder qualquer "código real" no arquivo de origem.

O exemplo:

```
C#  
  
#define Enterprise  
  
#if Professional || Enterprise  
    #define Advanced  
#endif  
  
namespace Megacorp.Data  
{  
    #if Advanced  
        class PivotTable {...}  
    #endif  
}
```

é válido porque as `#define` diretivas precedem o primeiro token (a `namespace` palavra-chave) no arquivo de origem.

O exemplo a seguir resulta em um erro de tempo de compilação porque um `#define` código real a seguir:

```
C#  
  
#define A  
namespace N  
{  
    #define B  
    #if B  
        class Class1 {}  
    #endif  
}
```

Um `#define` pode definir um símbolo de compilação condicional que já está definido, sem que haja nenhum intervir `#undef` para esse símbolo. O exemplo a seguir define um símbolo de compilação condicional `A` e, em seguida, define-o novamente.

```
C#
```

```
#define A  
#define A
```

O `#undef` pode "desdefinir" um símbolo de compilação condicional que não está definido. O exemplo a seguir define um símbolo de compilação condicional `A` e, em seguida, o define duas vezes; embora o segundo `#undef` não tenha efeito, ele ainda é válido.

```
C#
```

```
#define A  
#undef A  
#undef A
```

## Diretivas de compilação condicional

As diretivas de compilação condicional são usadas para incluir ou excluir de forma condicional partes de um arquivo de origem.

```
antlr
```

```
pp_conditional  
    : pp_if_section pp_elif_section* pp_else_section? pp_endif  
    ;  
  
pp_if_section  
    : whitespace? '#' whitespace? 'if' whitespace pp_expression pp_new_line  
conditional_section?  
    ;  
  
pp_elif_section  
    : whitespace? '#' whitespace? 'elif' whitespace pp_expression  
pp_new_line conditional_section?  
    ;  
  
pp_else_section:  
    | whitespace? '#' whitespace? 'else' pp_new_line conditional_section?  
    ;  
  
pp_endif  
    : whitespace? '#' whitespace? 'endif' pp_new_line
```

```

;

conditional_section
: input_section
| skipped_section
;

skipped_section
: skipped_section_part+
;

skipped_section_part
: skipped_characters? new_line
| pp_directive
;

skipped_characters
: whitespace? not_number_sign input_character*
;

not_number_sign
: '<Any input_character except #>'
;

```

Conforme indicado pela sintaxe, as diretivas de compilação condicional devem ser escritas como conjuntos que consistem em, em ordem, em uma `#if` diretiva, zero ou mais `#elif` diretivas, zero ou uma `#else` diretiva e uma `#endif` diretiva. Entre as diretivas estão seções condicionais do código-fonte. Cada seção é controlada pela diretiva imediatamente anterior. Uma seção condicional pode conter, por si só, diretivas de compilação condicional aninhadas desde que essas diretivas sejam conjuntos completos.

Um *pp\_conditional* seleciona no máximo um dos *conditional\_section*s contidos para o processamento lexical normal:

- As *pp\_expression*s das `#if` diretivas e `#elif` são avaliadas na ordem até que um resulte `true`. Se uma expressão produz `true`, a *conditional\_section* da diretiva correspondente é selecionada.
- Se todos os *pp\_expression*s produzem `false` e se uma `#else` diretiva estiver presente, a *conditional\_section* da `#else` diretiva será selecionada.
- Caso contrário, nenhum *conditional\_section* será selecionado.

O *conditional\_section* selecionado, se houver, é processado como um *input\_section* normal: o código-fonte contido na seção deve aderir à gramática léxica; os tokens são gerados a partir do código-fonte na seção; e as diretivas de pré-processamento na seção têm os efeitos prescritos.

Os *conditional\_section*s restantes, se houver, são processados como *skipped\_section*s: exceto pelas diretivas de pré-processamento, o código-fonte na seção não precisa aderir à gramática léxica; nenhum token é gerado a partir do código-fonte na seção; e as diretivas de pré-processamento na seção devem estar lexicalmente corretas, mas não são processadas de outra forma. Em um *conditional\_section* que está sendo processado como um *skipped\_section*, quaisquer *conditional\_section*s aninhados (contidos em `#if` construções aninhadas... `#endif` e `#region ... #endregion`) também são processados como *skipped\_section*s.

O exemplo a seguir ilustra como as diretivas de compilação condicional podem ser aninhadas:

```
C#  
  
#define Debug      // Debugging on  
#undef Trace     // Tracing off  
  
class PurchaseTransaction  
{  
    void Commit() {  
        #if Debug  
            CheckConsistency();  
        #if Trace  
            WriteToLog(this.ToString());  
        #endif  
        #endif  
        CommitHelper();  
    }  
}
```

Com exceção das diretivas de pré-processamento, o código-fonte ignorado não está sujeito à análise léxica. Por exemplo, o seguinte é válido apesar do comentário não finalizado na `#else` seção:

```
C#  
  
#define Debug      // Debugging on  
  
class PurchaseTransaction  
{  
    void Commit() {  
        #if Debug  
            CheckConsistency();  
        #else  
            /* Do something else  
        #endif  
    }  
}
```

No entanto, observe que as diretivas de pré-processamento precisam ser lexicalmente corretas mesmo em seções ignoradas do código-fonte.

As diretivas de pré-processamento não são processadas quando aparecem dentro de elementos de entrada de várias linhas. Por exemplo, o programa:

```
C#  
  
class Hello  
{  
    static void Main() {  
        System.Console.WriteLine(@"hello,  
#if Debug  
        world  
#else  
        Nebraska  
#endif  
        ");  
    }  
}
```

resulta na saída:

```
Console  
  
hello,  
#if Debug  
    world  
#else  
    Nebraska  
#endif
```

Em casos peculiares, o conjunto de diretivas de pré-processamento processadas pode depender da avaliação do *pp\_expression*. O exemplo:

```
C#  
  
#if X  
/*  
#else  
/* */ class Q { }  
#endif
```

sempre produz o mesmo fluxo de token ( `class Q {}` ), independentemente de `X` ser definido ou não. Se `X` for definido, as únicas diretivas processadas serão `#if` e `#endif`,

devido ao comentário de várias linhas. Se `x` for indefinido, três diretivas (`#if`, `#else`, `#endif`) fazem parte do conjunto de diretivas.

## Diretivas de diagnóstico

As diretivas de diagnóstico são usadas para gerar explicitamente mensagens de erro e de aviso que são relatadas da mesma maneira que outros erros e avisos em tempo de compilação.

antlr

```
pp_diagnostic
: whitespace? '#' whitespace? 'error' pp_message
| whitespace? '#' whitespace? 'warning' pp_message
;

pp_message
: new_line
| whitespace input_character* new_line
;
```

O exemplo:

C#

```
#warning Code review needed before check-in

#if Debug && Retail
    #error A build can't be both debug and retail
#endif

class Test {...}
```

sempre produz um aviso ("revisão de código necessária antes do check-in") e produz um erro de tempo de compilação ("uma compilação não pode ser depuração e varejo") se os símbolos condicionais `Debug` e `Retail` forem ambos definidos. Observe que um `pp_message` pode conter texto arbitrário; especificamente, ele não precisa conter tokens bem formados, conforme mostrado pelas aspas simples na palavra `can't`.

## Diretivas de região

As diretivas de região são usadas para marcar explicitamente regiões do código-fonte.

antlr

```
pp_region
: pp_start_region conditional_section? pp_end_region
;

pp_start_region
: whitespace? '#' whitespace? 'region' pp_message
;

pp_end_region
: whitespace? '#' whitespace? 'endregion' pp_message
;
```

Nenhum significado semântico é anexado a uma região; as regiões são destinadas ao uso pelo programador ou por ferramentas automatizadas para marcar uma seção do código-fonte. A mensagem especificada em uma `#region` `#endregion` diretiva ou, da mesma forma, não tem significado semântico; ela simplesmente serve para identificar a região. A correspondência `#region` e as `#endregion` diretivas podem ter diferentes `pp_message`s.

O processamento lexical de uma região:

```
C#
#region
...
#endregion
```

corresponde exatamente ao processamento lexical de uma diretiva de compilação condicional do formulário:

```
C#
#if true
...
#endif
```

## Diretivas de linha

As diretivas de linha podem ser usadas para alterar os números de linha e os nomes de arquivo de origem que são relatados pelo compilador em saída, como avisos e erros, e que são usados por atributos de informações do chamador ([atributos de informações do chamador](#)).

As diretivas de linha são usadas com mais frequência em ferramentas de metaprogramação que geram código-fonte C# de alguma outra entrada de texto.

```
antlr

pp_line
    : whitespace? '#' whitespace? 'line' whitespace line_indicator
pp_new_line
;

line_indicator
    : decimal_digit+ whitespace file_name
    | decimal_digit+
    | 'default'
    | 'hidden'
;

file_name
    : '"' file_name_character+ '"'
;

file_name_character
    : '<Any input_character except ">'

;
```

Quando não há `#line` diretivas presentes, o compilador relata os números de linha verdadeiros e os nomes de arquivo de origem em sua saída. Ao processar uma `#line` diretiva que inclui um *line\_indicator* que não é `default`, o compilador trata a linha após a diretiva como tendo o número de linha fornecido (e o nome do arquivo, se especificado).

Uma `#line default` diretiva reverte o efeito de todas as diretivas de `#line` anteriores. O compilador relata informações de linha verdadeiras para linhas subsequentes, precisamente como se nenhuma das `#line` diretivas tivesse sido processada.

Uma `#line hidden` diretiva não tem efeito sobre o arquivo e os números de linha relatados em mensagens de erro, mas afeta a depuração de nível de origem. Durante a depuração, todas as linhas entre uma `#line hidden` diretiva e a `#line` diretiva subsequente (que não é `#line hidden`) não têm nenhuma informação de número de linha. Ao percorrer o código no depurador, essas linhas serão ignoradas inteiramente.

Observe que uma *file\_name* difere de um literal de cadeia de caracteres regular, pois os caracteres de escape não são processados; o \ caractere "" simplesmente designa um caractere de barra invertida comum dentro de um *file\_name*.

## Diretivas pragma

A `#pragma` diretiva de pré-processamento é usada para especificar informações contextuais opcionais para o compilador. As informações fornecidas em uma `#pragma` diretiva nunca alterarão a semântica do programa.

```
antlr

pp_pragma
    : whitespace? '#' whitespace? 'pragma' whitespace pragma_body
pp_new_line
    ;

pragma_body
    : pragma_warning_body
    ;
```

O C# fornece `#pragma` diretivas para controlar os avisos do compilador. As versões futuras do idioma podem incluir `#pragma` diretivas adicionais. Para garantir a interoperabilidade com outros compiladores C#, o compilador do Microsoft C# não emite erros de compilação para diretivas desconhecidas `#pragma`; no entanto, essas diretivas geram avisos.

## Aviso de pragma

A `#pragma warning` diretiva é usada para desabilitar ou restaurar todos ou um determinado conjunto de mensagens de aviso durante a compilação do texto do programa subsequente.

```
antlr

pragma_warning_body
    : 'warning' whitespace warning_action
    | 'warning' whitespace warning_action whitespace warning_list
    ;

warning_action
    : 'disable'
    | 'restore'
    ;

warning_list
    : decimal_digit+ (whitespace? ',' whitespace? decimal_digit+)*
    ;
```

Uma `#pragma warning` diretiva que omite a lista de avisos afeta todos os avisos. Uma `#pragma warning` diretiva que inclui uma lista de avisos afeta apenas os avisos

especificados na lista.

Uma `#pragma warning disable` diretiva desabilita todos ou o conjunto de avisos fornecido.

Uma `#pragma warning restore` diretiva restaura todos ou o conjunto determinado de avisos para o estado que estava em vigor no início da unidade de compilação. Observe que, se um aviso específico tiver sido desabilitado externamente, um `#pragma warning restore` (seja para todos ou o aviso específico) não reabilitará esse aviso.

O exemplo a seguir mostra o uso de `#pragma warning` para desabilitar temporariamente o aviso relatado quando membros obsoletos são referenciados, usando o número de aviso do compilador do Microsoft C#.

C#

```
using System;

class Program
{
    [Obsolete]
    static void Foo() {}

    static void Main() {
#pragma warning disable 612
        Foo();
#pragma warning restore 612
    }
}
```

# Conceitos básicos

Artigo • 16/09/2021

## Inicialização de aplicativos

Um assembly que tem um *Entry Point* é chamado de \_aplicativo\*. Quando um aplicativo é executado, um novo *domínio\_ aplicativo\** é criado. Várias instanciações diferentes de um aplicativo podem existir no mesmo computador ao mesmo tempo, e cada uma tem seu próprio domínio de aplicativo.

Um domínio de aplicativo habilita o isolamento de aplicativos agindo como um contêiner para o estado do aplicativo. Um domínio de aplicativo atua como um contêiner e limite para os tipos definidos no aplicativo e as bibliotecas de classes que ele usa. Os tipos carregados em um domínio de aplicativo são diferentes do mesmo tipo carregado em outro domínio de aplicativo, e as instâncias de objetos não são diretamente compartilhadas entre domínios de aplicativo. Por exemplo, cada domínio de aplicativo tem sua própria cópia de variáveis estáticas para esses tipos, e um construtor estático para um tipo é executado no máximo uma vez por domínio de aplicativo. As implementações são gratuitas para fornecer políticas ou mecanismos específicos de implementação para a criação e destruição de domínios de aplicativo.

A *inicialização do aplicativo* ocorre quando o ambiente de execução chama um método designado, que é chamado de ponto de entrada do aplicativo. Esse método de ponto de entrada sempre é nomeado `Main` e pode ter uma das seguintes assinaturas:

C#

```
static void Main() {...}  
  
static void Main(string[] args) {...}  
  
static int Main() {...}  
  
static int Main(string[] args) {...}
```

Como mostrado, o ponto de entrada pode, opcionalmente, retornar um `int` valor. Esse valor de retorno é usado na terminação do aplicativo ([terminação do aplicativo](#)).

O ponto de entrada pode, opcionalmente, ter um parâmetro formal. O parâmetro pode ter qualquer nome, mas o tipo do parâmetro deve ser `string[]`. Se o parâmetro formal estiver presente, o ambiente de execução criará e passará um `string[]` argumento contendo os argumentos de linha de comando que foram especificados quando o

aplicativo foi iniciado. O `string[]` argumento nunca é nulo, mas pode ter um comprimento zero se nenhum argumento de linha de comando foi especificado.

Como o C# dá suporte à sobrecarga de método, uma classe ou struct pode conter várias definições de algum método, desde que cada uma tenha uma assinatura diferente. No entanto, em um único programa, nenhuma classe ou estrutura pode conter mais de um método chamado `Main` cuja definição o qualifica para ser usado como um ponto de entrada de aplicativo. Outras versões sobrecarregadas do `Main` são permitidas, no entanto, desde que tenham mais de um parâmetro, ou seu único parâmetro seja diferente do tipo `string[]`.

Um aplicativo pode ser composto por várias classes ou structs. É possível que mais de uma dessas classes ou structs contenham um método chamado `Main` cuja definição a qualifica para ser usada como um ponto de entrada de aplicativo. Nesses casos, um mecanismo externo (como uma opção de compilador de linha de comando) deve ser usado para selecionar um desses `Main` métodos como o ponto de entrada.

No C#, cada método deve ser definido como um membro de uma classe ou estrutura. Normalmente, a acessibilidade declarada ([acessibilidade declarada](#)) de um método é determinada pelos modificadores de acesso ([modificadores de acesso](#)) especificados em sua declaração e, da mesma forma, a acessibilidade declarada de um tipo é determinada pelos modificadores de acesso especificados em sua declaração. Para que um determinado método de um determinado tipo seja chamado, o tipo e o membro devem estar acessíveis. No entanto, o ponto de entrada do aplicativo é um caso especial. Especificamente, o ambiente de execução pode acessar o ponto de entrada do aplicativo independentemente de sua acessibilidade declarada e independentemente da acessibilidade declarada de suas declarações de tipo delimitador.

O método de ponto de entrada de aplicativo pode não estar em uma declaração de classe genérica.

Em todos os outros aspectos, os métodos de ponto de entrada se comportam como aqueles que não são pontos de entrada.

## Terminação de aplicativos

A [\*terminação de aplicativo\*](#) retorna o controle para o ambiente de execução.

Se o tipo de retorno do método do **ponto de entrada\*** do aplicativo for `int`, o valor retornado funcionará como o código de *status de encerramento* \_ \* do aplicativo \* \*. A finalidade desse código é permitir a comunicação de êxito ou falha no ambiente de execução.

Se o tipo de retorno do método de ponto de entrada for `void`, atingir a chave direita (`}`) que encerra esse método, ou executar uma `return` instrução que não tenha nenhuma expressão, resultará em um código de status de encerramento de `0`.

Antes do encerramento de um aplicativo, os destruidores para todos os seus objetos que ainda não foram coletados pelo lixo são chamados, a menos que essa limpeza tenha sido suprimida (por uma chamada para o método de biblioteca `GC.SuppressFinalize`, por exemplo).

## Declarações

As declarações em um programa C# definem os elementos constituintes do programa. Os programas em C# são organizados usando Namespaces ([namespaces](#)), que podem conter declarações de tipo e declarações de namespace aninhadas. Declarações de tipo ([declarações de tipo](#)) são usadas para definir classes ([classes](#)), estruturas([structs](#)), interfaces ([interfaces](#)), enumerações ([enums](#)) e delegados ([delegados](#)). Os tipos de membros permitidos em uma declaração de tipo dependem da forma da declaração de tipo. Por exemplo, declarações de classe podem conter declarações para constantes ([constantes](#)), campos([campos](#)), métodos ([métodos](#)), Propriedades ([Propriedades](#)), eventos ([eventos](#)), indexadores ([indexadores](#)), operadores ([operadores](#)), construtores de instância ([construtores deinstância](#)), construtores estáticos ([construtores estáticos](#)), destruidores ([destruidores](#)) e tipos aninhados ([tipos aninhados](#)).

Uma declaração define um nome no *espaço de declaração* ao qual a declaração pertence. Exceto para Membros sobrecarregados ([assinaturas e sobrecarga](#)), é um erro de tempo de compilação ter duas ou mais declarações que apresentam Membros com o mesmo nome em um espaço de declaração. Nunca é possível que um espaço de declaração contenha diferentes tipos de membros com o mesmo nome. Por exemplo, um espaço de declaração nunca pode conter um campo e um método com o mesmo nome.

Há vários tipos diferentes de espaços de declaração, conforme descrito a seguir.

- Em todos os arquivos de origem de um programa, *namespace\_member\_declaration*s sem *namespace\_declaration* delimitadores são membros de um único espaço de declaração combinado chamado de *espaço de declaração global*.
- Em todos os arquivos de origem de um programa, *namespace\_member\_declaration*s em *namespace\_declaration*s que têm o mesmo nome de namespace totalmente qualificado são membros de um único espaço de declaração combinado.
- Cada declaração de classe, struct ou interface cria um novo espaço de declaração. Os nomes são introduzidos nesse espaço de declaração por meio de

*class\_member\_declaration s, struct\_member\_declaration s, interface\_member\_declaration s ou type\_parameter s.* Exceto para declarações de construtor de instância sobrecarregadas e declarações de Construtor estáticos, uma classe ou struct não pode conter uma declaração de membro com o mesmo nome que a classe ou struct. Uma classe, estrutura ou interface permite a declaração de métodos e indexadores sobrecarregados. Além disso, uma classe ou struct permite a declaração de operadores e construtores de instância sobrecarregados. Por exemplo, uma classe, struct ou interface pode conter várias declarações de método com o mesmo nome, desde que essas declarações de método difiram em sua assinatura ([assinaturas e sobrecarga](#)). Observe que as classes base não contribuem para o espaço de declaração de uma classe, e as interfaces base não contribuem para o espaço de declaração de uma interface. Assim, uma classe ou interface derivada tem permissão para declarar um membro com o mesmo nome de um membro herdado. Esse membro é dito para **ocultar** o membro herdado.

- Cada declaração de delegado cria um novo espaço de declaração. Os nomes são introduzidos nesse espaço de declaração por meio de parâmetros formais (*fixed\_parameter s* e *parameter\_array s*) e *type\_parameter s*.
- Cada declaração de enumeração cria um novo espaço de declaração. Os nomes são introduzidos nesse espaço de declaração por meio de *enum\_member\_declarations*.
- Cada declaração de método, declaração de indexador, declaração de operador, declaração de construtor de instância e função anônima cria um novo espaço de declaração chamado **\*espaço de declaração de variável local \_**. Os nomes são introduzidos nesse espaço de declaração por meio de parâmetros formais (*\_fixed\_parameter \* s* e *parameter\_array s*) e *type\_parameter s*. O corpo do membro da função ou da função anônima, se houver, será considerado aninhado dentro do espaço de declaração da variável local. É um erro para um espaço de declaração de variável local e um espaço de declaração de variável local aninhada para conter elementos com o mesmo nome. Portanto, dentro de um espaço de declaração aninhada não é possível declarar uma variável local ou constante com o mesmo nome de uma variável local ou constante em um espaço de declaração de delimitador. É possível que dois espaços de declaração contenham elementos com o mesmo nome, desde que nenhum espaço de declaração contenha o outro.
- Cada *bloco* ou *switch\_block*, bem como uma instrução *for*, *foreach* e *using*, cria um espaço de declaração de variável local para variáveis locais e constantes locais. Os nomes são introduzidos nesse espaço de declaração por meio de *local\_variable\_declaration s* e *local\_constant\_declaration s*. Observe que os blocos que ocorrem como ou dentro do corpo de um membro de função ou função anônima são aninhados dentro do espaço de declaração de variável local

declarado por essas funções para seus parâmetros. Portanto, é um erro ter, por exemplo, um método com uma variável local e um parâmetro de mesmo nome.

- Cada *bloco* ou *switch\_block* cria um espaço de declaração separado para rótulos. Os nomes são introduzidos nesse espaço de declaração por meio de *labeled\_statement*s e os nomes são referenciados por meio de *goto\_statement*s. O **espaço de declaração de rótulo** de um bloco inclui todos os blocos aninhados. Portanto, em um bloco aninhado não é possível declarar um rótulo com o mesmo nome de um rótulo em um bloco delimitador.

Em geral, a ordem textual na qual os nomes são declarados não é significado. Em particular, a ordem textual não é significativa para a declaração e o uso de namespaces, constantes, métodos, propriedades, eventos, indexadores, operadores, construtores de instância, destruidores, construtores estáticos e tipos. A ordem de declaração é significativa das seguintes maneiras:

- A ordem de declaração para declarações de campo e declarações de variáveis locais determina a ordem na qual seus inicializadores (se houver) são executados.
- As variáveis locais devem ser definidas antes de serem usadas ([escopos](#)).
- A ordem de declaração para declarações de membro de enumeração ([membros de enum](#)) é significativa quando valores de *constant\_expression* são omitidos.

O espaço de declaração de um namespace é "Open finalizado" e duas declarações de namespace com o mesmo nome totalmente qualificado contribuem para o mesmo espaço de declaração. Por exemplo,

C#

```
namespace Megacorp.Data
{
    class Customer
    {
        ...
    }

    namespace Megacorp.Data
    {
        class Order
        {
            ...
        }
    }
}
```

As duas declarações de namespace acima contribuem para o mesmo espaço de declaração, neste caso, declarando duas classes com os nomes totalmente qualificados `Megacorp.Data.Customer` e `Megacorp.Data.Order`. Como as duas declarações contribuem

para o mesmo espaço de declaração, ela causaria um erro de tempo de compilação se cada uma continha uma declaração de uma classe com o mesmo nome.

Conforme especificado acima, o espaço de declaração de um bloco inclui todos os blocos aninhados. Portanto, no exemplo a seguir, os `F` `G` métodos e resultam em um erro de tempo de compilação porque o nome `i` é declarado no bloco externo e não pode ser declarado novamente no bloco interno. No entanto, os `H` `I` métodos e são válidos, já que os dois `i` são declarados em blocos não aninhados separados.

C#

```
class A
{
    void F() {
        int i = 0;
        if (true) {
            int i = 1;
        }
    }

    void G() {
        if (true) {
            int i = 0;
        }
        int i = 1;
    }

    void H() {
        if (true) {
            int i = 0;
        }
        if (true) {
            int i = 1;
        }
    }

    void I() {
        for (int i = 0; i < 10; i++)
            H();
        for (int i = 0; i < 10; i++)
            H();
    }
}
```

## Membros

Namespaces e tipos têm **Membros**. Os membros de uma entidade estão geralmente disponíveis por meio do uso de um nome qualificado que começa com uma referência à

entidade, seguida por um `.` token "", seguido pelo nome do membro.

Os membros de um tipo são declarados na declaração de tipo ou *herdados* da classe base do tipo. Quando um tipo é herdado de uma classe base, todos os membros da classe base, exceto construtores de instância, destruidores e construtores estáticos, se tornam membros do tipo derivado. A acessibilidade declarada de um membro de classe base não controla se o membro é herdado — a herança se estende a qualquer membro que não seja um construtor de instância, construtor estático ou destruidor. No entanto, um membro herdado pode não estar acessível em um tipo derivado, devido à sua acessibilidade declarada ([acessibilidade declarada](#)) ou porque está ocultada por uma declaração no próprio tipo ([ocultando por herança](#)).

## Membros de namespace

Namespaces e tipos que não têm namespace delimitador são membros do *namespace global*. Isso corresponde diretamente aos nomes declarados no espaço de declaração global.

Namespaces e tipos declarados em um namespace são membros desse namespace. Isso corresponde diretamente aos nomes declarados no espaço de declaração do namespace.

Namespaces não têm nenhuma restrição de acesso. Não é possível declarar namespaces privados, protegidos ou internos, e os nomes de namespace são sempre acessíveis publicamente.

## Membros de struct

Os membros de uma struct são os membros declarados na struct e os membros herdados da classe base direta da struct `System.ValueType` e a classe base indireta `object`.

Os membros de um tipo simples correspondem diretamente aos membros do tipo struct com o alias do tipo simples:

- Os membros de `sbyte` são os membros da `System.SByte` estrutura.
- Os membros de `byte` são os membros da `System.Byte` estrutura.
- Os membros de `short` são os membros da `System.Int16` estrutura.
- Os membros de `ushort` são os membros da `System.UInt16` estrutura.
- Os membros de `int` são os membros da `System.Int32` estrutura.
- Os membros de `uint` são os membros da `System.UInt32` estrutura.

- Os membros de `long` são os membros da `System.Int64` estrutura.
- Os membros de `ulong` são os membros da `System.UInt64` estrutura.
- Os membros de `char` são os membros da `System.Char` estrutura.
- Os membros de `float` são os membros da `System.Single` estrutura.
- Os membros de `double` são os membros da `System.Double` estrutura.
- Os membros de `decimal` são os membros da `System.Decimal` estrutura.
- Os membros de `bool` são os membros da `System.Boolean` estrutura.

## Membros de enumeração

Os membros de uma enumeração são as constantes declaradas na enumeração e os membros herdados da classe base direta da enumeração `System.Enum` e as classes base indiretas `System.ValueType` e `object`.

## Membros de classe

Os membros de uma classe são os membros declarados na classe e os membros herdados da classe base (exceto para `object` a classe que não tem nenhuma classe base). Os membros herdados da classe base incluem as constantes, os campos, os métodos, as propriedades, os eventos, os indexadores, os operadores e os tipos da classe base, mas não os construtores de instância, destruidores e construtores estáticos da classe base. Os membros da classe base são herdados sem considerar sua acessibilidade.

Uma declaração de classe pode conter declarações de constantes, campos, métodos, propriedades, eventos, indexadores, operadores, construtores de instância, destruidores, construtores e tipos estáticos.

Os membros de `object` e `string` correspondem diretamente aos membros dos tipos de classe que eles alias:

- Os membros de `object` são os membros da `System.Object` classe.
- Os membros de `string` são os membros da `System.String` classe.

## Membros da interface

Os membros de uma interface são os membros declarados na interface e em todas as interfaces base da interface. Os membros da classe `object` não são, estritamente falando, membros de qualquer interface ([membros da interface](#)). No entanto, os

membros da classe `object` estão disponíveis por meio de pesquisa de membros em qualquer tipo de interface ([pesquisa de membros](#)).

## Membros de matriz

Os membros de uma matriz são os membros herdados da classe `System.Array`.

## Delegar Membros

Os membros de um delegado são os membros herdados da classe `System.Delegate`.

## Acesso de membros

Declarações de Membros permitem o controle sobre o acesso de membro. A acessibilidade de um membro é estabelecida pela acessibilidade declarada ([acessibilidade declarada](#)) do membro combinado com a acessibilidade do tipo que o contém imediatamente, se houver.

Quando o acesso a um membro específico é permitido, o membro é considerado *\*acessível* .. Por outro lado, quando o acesso a um determinado membro não é permitido, o membro é considerado *\_inacessível* \*. O acesso a um membro é permitido quando o local textual no qual o acesso ocorre está incluído no domínio de acessibilidade ([domínios de acessibilidade](#)) do membro.

## Acessibilidade declarada

A *acessibilidade declarada* de um membro pode ser uma das seguintes:

- Public, que é selecionado incluindo um `public` modificador na declaração de membro. O significado intuitivo de `public` é "acesso não limitado".
- Protegido, que é selecionado incluindo um `protected` modificador na declaração de membro. O significado intuitivo de `protected` é "acesso limitado à classe que a contém ou tipos derivados da classe que a contém".
- Interno, que é selecionado incluindo um `internal` modificador na declaração de membro. O significado intuitivo de `internal` é "acesso limitado a este programa".
- Protegido interno (ou seja, protegido ou interno), que é selecionado incluindo um `protected` e um `internal` modificador na declaração de membro. O significado intuitivo de `protected internal` é "acesso limitado a este programa ou tipos derivados da classe que a contém".

- Privado, que é selecionado incluindo um `private` modificador na declaração de membro. O significado intuitivo de `private` é "acesso limitado ao tipo recipiente".

Dependendo do contexto no qual uma declaração de membro ocorre, somente determinados tipos de acessibilidade declarada são permitidos. Além disso, quando uma declaração de membro não inclui nenhum modificador de acesso, o contexto no qual a declaração ocorre determina a acessibilidade declarada padrão.

- Os namespaces implicitamente `public` declararam acessibilidade. Nenhum modificador de acesso é permitido em declarações de namespace.
- Tipos declarados em unidades de compilação ou namespaces podem ter `public` ou `internal` declarar acessibilidade e padrão para a `internal` acessibilidade declarada.
- Os membros de classe podem ter qualquer um dos cinco tipos de acessibilidade declarada e o padrão para a `private` acessibilidade declarada. (Observe que um tipo declarado como membro de uma classe pode ter qualquer um dos cinco tipos de acessibilidade declarada, enquanto um tipo declarado como membro de um namespace pode ter apenas `public` ou `internal` declarado acessibilidade.)
- Os membros de struct podem ter `public`, `internal` ou `private` a acessibilidade declarada e o padrão para a `private` acessibilidade declarada porque as estruturas são lacradas implicitamente. Membros de struct introduzidos em uma struct (ou seja, não herdados por essa struct) não podem ter `protected` ou `protected` `internal` declarar acessibilidade. (Observe que um tipo declarado como membro de uma struct pode ter `public`, `internal` ou `private` a acessibilidade declarada, enquanto um tipo declarado como membro de um namespace pode ter apenas `public` ou `internal` declarado a acessibilidade.)
- Os membros da interface implicitamente `public` declararam acessibilidade. Nenhum modificador de acesso é permitido em declarações de membro de interface.
- Membros de enumeração implicitamente `public` declararam acessibilidade. Nenhum modificador de acesso é permitido em declarações de membro de enumeração.

## Domínios de acessibilidade

O \*domínio de acessibilidade \_ de um membro consiste nas seções (possivelmente disjunção) do texto do programa no qual o acesso ao membro é permitido. Para fins de definição do domínio de acessibilidade de um membro, um membro será considerado de *nível superior* se ele não for declarado dentro de um tipo, e um membro será considerado *aninhado* se for declarado dentro de outro tipo. Além disso, o *texto do*

*programa* de um programa é definido como todo o texto do programa contido em todos os arquivos de origem do programa, e o texto do programa de um tipo é definido como todo o texto do programa contido no `_type_declaration` \* s desse tipo (incluindo, possivelmente, tipos aninhados dentro do tipo).

O domínio de acessibilidade de um tipo predefinido (como `object`, `int` ou `double`) é ilimitado.

O domínio de acessibilidade de um tipo não associado de nível superior `T` ([tipos vinculados e desvinculados](#)) que é declarado em um programa `P` é definido da seguinte maneira:

- Se a acessibilidade declarada do `T` for `public`, o domínio de acessibilidade do `T` é o texto do programa `P` e qualquer programa que faça referência `P` a ele.
- Se a acessibilidade declarada de `T` for `internal`, o domínio de acessibilidade de `T` será o texto de programa de `P`.

A partir dessas definições, a seguir, o domínio de acessibilidade de um tipo não associado de nível superior é sempre pelo menos o texto do programa do programa no qual esse tipo é declarado.

O domínio de acessibilidade para um tipo construído `T<A1, ..., An>` é a interseção do domínio de acessibilidade do tipo genérico não associado `T` e os domínios de acessibilidade dos argumentos de tipo `A1, ..., An`.

O domínio de acessibilidade de um membro aninhado `M` declarado em um tipo `T` dentro de um programa `P` é definido da seguinte maneira (observando que `M` ele pode possivelmente ser um tipo):

- Se a acessibilidade declarada de `M` for `public`, o domínio de acessibilidade de `M` será o domínio de acessibilidade de `T`.
- Se a acessibilidade declarada de `M` for `protected internal`, permita que `D` seja a União do texto do programa `P` e o texto do programa de qualquer tipo derivado de `T`, que é declarado fora do `P`. O domínio de acessibilidade do `M` é a interseção do domínio de acessibilidade do `T` com o `D`.
- Se a acessibilidade declarada `M` for `protected`, deixe que `D` seja a União do texto do programa `T` e o texto do programa de qualquer tipo derivado de `T`. O domínio de acessibilidade do `M` é a interseção do domínio de acessibilidade do `T` com o `D`.
- Se a acessibilidade declarada de `M` for `internal`, o domínio de acessibilidade de `M` será a interseção do domínio de acessibilidade de `T` com o texto de programa de

P.

- Se a acessibilidade declarada de `M` for `private`, o domínio de acessibilidade de `M` será o texto de programa de `T`.

A partir dessas definições, a seguir, o domínio de acessibilidade de um membro aninhado é sempre pelo menos o texto do programa do tipo no qual o membro é declarado. Além disso, ele segue que o domínio de acessibilidade de um membro nunca é mais inclusivo do que o domínio de acessibilidade do tipo no qual o membro é declarado.

Em termos intuitivos, quando um tipo ou membro `M` é acessado, as etapas a seguir são avaliadas para garantir que o acesso seja permitido:

- Primeiro, se `M` for declarado dentro de um tipo (em oposição a uma unidade de compilação ou um namespace), ocorrerá um erro de tempo de compilação se esse tipo não estiver acessível.
- Em seguida, se `M` for `public`, o acesso será permitido.
- Caso contrário, se `M` for `protected internal`, o acesso será permitido se ocorrer dentro do programa em que `M` é declarado ou se ocorrer dentro de uma classe derivada da classe em que `M` é declarada e ocorre por meio do tipo de classe derivada ([acesso protegido para membros da instância](#)).
- Caso contrário, se `M` for `protected`, o acesso será permitido se ocorrer dentro da classe em que `M` é declarado, ou se ocorrer dentro de uma classe derivada da classe em que `M` é declarada e ocorre por meio do tipo de classe derivada ([acesso protegido para membros da instância](#)).
- Caso contrário, se `M` for `internal`, o acesso será permitido se ocorrer dentro do programa em que `M` é declarado.
- Caso contrário, se `M` for `private`, o acesso será permitido se ocorrer dentro do tipo em que `M` é declarado.
- Caso contrário, o tipo ou o membro será inacessível e ocorrerá um erro em tempo de compilação.

No exemplo

C#

```
public class A
{
    public static int X;
    internal static int Y;
    private static int Z;
}
```

```

internal class B
{
    public static int X;
    internal static int Y;
    private static int Z;

    public class C
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }

    private class D
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }
}

```

as classes e os membros têm os seguintes domínios de acessibilidade:

- O domínio de acessibilidade de `A` e `A.X` é ilimitado.
- O domínio de acessibilidade do `A.Y`, `B.B.X`, `B.Y`, `B.C`, `B.C.X` e `B.C.Y` é o texto do programa do programa que o contém.
- O domínio de acessibilidade do `A.Z` é o texto do programa do `A`.
- O domínio de acessibilidade do `B.Z` e `B.D` é o texto do programa do `B`, incluindo o texto do programa de `B.C` e `B.D`.
- O domínio de acessibilidade do `B.C.Z` é o texto do programa do `B.C`.
- O domínio de acessibilidade do `B.D.X` e `B.D.Y` é o texto do programa do `B`, incluindo o texto do programa de `B.C` e `B.D`.
- O domínio de acessibilidade do `B.D.Z` é o texto do programa do `B.D`.

Como ilustra o exemplo, o domínio de acessibilidade de um membro nunca é maior que o de um tipo recipiente. Por exemplo, embora todos os `X` Membros tenham acessibilidade declarada pública, todos, mas `A.X` têm domínios de acessibilidade restritos por um tipo recipiente.

Conforme descrito em [Membros](#), todos os membros de uma classe base, exceto para construtores de instância, destruidores e construtores estáticos, são herdados por tipos derivados. Isso inclui até membros privados de uma classe base. No entanto, o domínio de acessibilidade de um membro privado inclui apenas o texto do programa do tipo no qual o membro é declarado. No exemplo

```

class A
{
    int x;

    static void F(B b) {
        b.x = 1;           // Ok
    }
}

class B: A
{
    static void F(B b) {
        b.x = 1;           // Error, x not accessible
    }
}

```

a `B` classe herda o membro privado `x` da `A` classe. Como o membro é privado, ele só pode ser acessado dentro do *class\_body* de `A`. Assim, o acesso a é `b.x` bem-sucedida no `A.F` método, mas falha no `B.F` método.

## Acesso protegido para membros de instância

Quando um `protected` membro de instância é acessado fora do texto do programa da classe na qual ele é declarado e quando um `protected internal` membro da instância é acessado fora do texto do programa no qual ele é declarado, o acesso deve ocorrer dentro de uma declaração de classe que deriva da classe na qual ele é declarado. Além disso, o acesso é necessário para ocorrer por meio de uma instância desse tipo de classe derivada ou um tipo de classe construído a partir dele. Essa restrição impede que uma classe derivada acesse membros protegidos de outras classes derivadas, mesmo quando os membros são herdados da mesma classe base.

Permita que `B` seja uma classe base que declare um membro de instância protegida `M` e que `D` seja uma classe derivada de `B`. Dentro do *class\_body* de `D`, o acesso ao `M` pode ter um dos seguintes formatos:

- Um *type\_name* ou *primary\_expression* não qualificado do formulário `M`.
- Uma *primary\_expression* do formulário `E.M`, desde que o tipo `E` seja `T` ou uma classe derivada de `T`, em que `T` é o tipo de classe `D`, ou um tipo de classe construído a partir de `D`
- Uma *primary\_expression* do formulário `base.M`.

Além dessas formas de acesso, uma classe derivada pode acessar um construtor de instância protegida de uma classe base em um *constructor\_initializer* ([inicializadores de Construtor](#)).

## No exemplo

C#

```
public class A
{
    protected int x;

    static void F(A a, B b) {
        a.x = 1;          // Ok
        b.x = 1;          // Ok
    }
}

public class B: A
{
    static void F(A a, B b) {
        a.x = 1;          // Error, must access through instance of B
        b.x = 1;          // Ok
    }
}
```

no `A`, é possível acessar `x` por meio de instâncias do `A` e do `B`, já que, em ambos os casos, o acesso ocorre por meio de uma instância do `A` ou de uma classe derivada de `A`. No entanto, no `B`, não é possível acessar `x` por meio de uma instância do `A`, pois `A` não deriva de `B`.

## No exemplo

C#

```
class C<T>
{
    protected T x;
}

class D<T>: C<T>
{
    static void F() {
        D<T> dt = new D<T>();
        D<int> di = new D<int>();
        D<string> ds = new D<string>();
        dt.x = default(T);
        di.x = 123;
        ds.x = "test";
    }
}
```

as três atribuições a `x` são permitidas porque todas elas ocorrem por meio de instâncias de tipos de classe construídas a partir do tipo genérico.

## Restrições de acessibilidade

Várias construções na linguagem C# exigem que um tipo seja *pelo menos como acessível como* um membro ou outro tipo. Um tipo `T` é considerado, pelo menos, como acessível como membro ou tipo `M` se o domínio de acessibilidade do `T` for um superconjunto do domínio de acessibilidade de `M`. Em outras palavras, `T` é pelo menos tão acessível quanto `M` se `T` estiver acessível em todos os contextos em que o `M` está acessível.

Existem as seguintes restrições de acessibilidade:

- A classe base direta de um tipo de classe deve ser, pelo menos, tão acessível quanto o próprio tipo de classe.
- As interfaces base explícitas de um tipo de interface devem ser, pelo menos, tão acessíveis quanto o próprio tipo de interface.
- O tipo de retorno e os tipos de parâmetro de um tipo delegado devem ser, pelo menos, tão acessíveis quanto o próprio tipo delegado.
- O tipo de uma constante deve ser, pelo menos, tão acessível quanto a própria constante.
- O tipo de um campo deve ser, pelo menos, tão acessível quanto o próprio campo.
- O tipo de retorno e os tipos de parâmetro de um método devem ser, pelo menos, tão acessíveis quanto o próprio método.
- O tipo de uma propriedade deve ser, pelo menos, tão acessível quanto a propriedade em si.
- O tipo de um evento deve ser, pelo menos, tão acessível quanto o próprio evento.
- O tipo e os tipos de parâmetro de um indexador devem ser, pelo menos, tão acessíveis quanto o próprio indexador.
- O tipo de retorno e os tipos de parâmetro de um operador devem ser, pelo menos, tão acessíveis quanto o próprio operador.
- Os tipos de parâmetro de um construtor de instância devem ser pelo menos tão acessíveis quanto o próprio construtor de instância.

No exemplo

C#

```
class A {...}

public class B: A {...}
```

a `B` classe resulta em um erro de tempo de compilação porque `A` não é pelo menos tão acessível quanto `B`.

Da mesma forma, no exemplo

```
C#  
  
class A {...}  
  
public class B  
{  
    A F() {...}  
  
    internal A G() {...}  
  
    public A H() {...}  
}
```

o `H` método em `B` resulta em um erro de tempo de compilação porque o tipo de retorno `A` não é pelo menos tão acessível quanto o método.

## Assinaturas e sobrecarga

Os métodos, construtores de instância, indexadores e operadores são caracterizados por suas *assinaturas*:

- A assinatura de um método consiste no nome do método, no número de parâmetros de tipo e no tipo e tipo (valor, referência ou saída) de cada um de seus parâmetros formais, considerados na ordem da esquerda para a direita. Para essas finalidades, qualquer parâmetro de tipo do método que ocorre no tipo de um parâmetro formal é identificado não por seu nome, mas por sua posição ordinal na lista de argumentos de tipo do método. A assinatura de um método especificamente não inclui o tipo de retorno, o `params` modificador que pode ser especificado para o parâmetro mais à direita, nem as restrições de parâmetro de tipo opcional.
- A assinatura de um construtor de instância consiste no tipo e tipo (valor, referência ou saída) de cada um de seus parâmetros formais, considerados na ordem da esquerda para a direita. A assinatura de um construtor de instância especificamente não inclui o `params` modificador que pode ser especificado para o parâmetro mais à direita.
- A assinatura de um indexador consiste no tipo de cada um de seus parâmetros formais, considerados na ordem da esquerda para a direita. A assinatura de um

indexador especificamente não inclui o tipo de elemento, nem inclui o `params` modificador que pode ser especificado para o parâmetro mais à direita.

- A assinatura de um operador consiste no nome do operador e no tipo de cada um de seus parâmetros formais, considerados na ordem da esquerda para a direita. A assinatura de um operador especificamente não inclui o tipo de resultado.

As assinaturas são o mecanismo de habilitação para **sobrecarga** de membros em classes, estruturas e interfaces:

- A sobrecarga de métodos permite que uma classe, estrutura ou interface declare vários métodos com o mesmo nome, desde que suas assinaturas sejam exclusivas nessa classe, struct ou interface.
- A sobrecarga de construtores de instância permite que uma classe ou estrutura declare vários construtores de instância, desde que suas assinaturas sejam exclusivas dentro dessa classe ou estrutura.
- A sobrecarga de indexadores permite que uma classe, estrutura ou interface declare vários indexadores, desde que suas assinaturas sejam exclusivas nessa classe, struct ou interface.
- A sobrecarga de operadores permite que uma classe ou estrutura declare vários operadores com o mesmo nome, desde que suas assinaturas sejam exclusivas dentro dessa classe ou estrutura.

Embora `out` e `ref` os modificadores de parâmetro sejam considerados parte de uma assinatura, os membros declarados em um único tipo não podem diferir na assinatura exclusivamente por `ref` e `out`. Ocorrerá um erro de tempo de compilação se dois membros forem declarados no mesmo tipo com assinaturas que seriam as mesmas se todos os parâmetros nos dois métodos com `out` modificadores fossem alterados para `ref` modificadores. Para outras finalidades de correspondência de assinatura (por exemplo, ocultar ou substituir) `ref` e `out` são consideradas parte da assinatura e não coincidem entre si. (Essa restrição é permitir que programas em C# sejam facilmente convertidos para serem executados no Common Language Infrastructure (CLI), que não fornece uma maneira de definir métodos que diferem somente em `ref` e `out`.)

Para os fins de assinaturas, os tipos `object` e `dynamic` são considerados iguais. Os membros declarados em um único tipo podem, portanto, não ser diferentes na assinatura exclusivamente pelo `object` e `dynamic`.

O exemplo a seguir mostra um conjunto de declarações de método sobrecarregadas junto com suas assinaturas.

```

interface ITest
{
    void F();                                // F()

    void F(int x);                           // F(int)

    void F(ref int x);                      // F(ref int)

    void F(out int x);                      // F(out int)      error

    void F(int x, int y);                  // F(int, int)

    int F(string s);                        // F(string)

    int F(int x);                          // F(int)          error

    void F(string[] a);                   // F(string[])

    void F(params string[] a);           // F(string[])     error
}

```

Observe que quaisquer `ref` `out` modificadores de parâmetro e ([parâmetros de método](#)) fazem parte de uma assinatura. Portanto, `F(int)` e `F(ref int)` são assinaturas exclusivas. No entanto, `F(ref int)` e `F(out int)` não podem ser declarados na mesma interface porque suas assinaturas diferem exclusivamente por `ref` e `out`. Além disso, observe que o tipo de retorno e o `params` modificador não fazem parte de uma assinatura, portanto, não é possível sobrecarregar unicamente com base no tipo de retorno ou na inclusão ou exclusão do `params` modificador. Como tal, as declarações dos métodos `F(int)` e `F(params string[])` identificadas acima resultam em um erro de tempo de compilação.

## Escopos

O *\*escopo \_* de um nome é a região do texto do programa no qual é possível fazer referência à entidade declarada pelo nome sem qualificação do nome. Os escopos podem ser *aninhados*, e um escopo interno pode redeclarar o significado de um nome a partir de um escopo externo (isso não, no entanto, remove a restrição imposta por [declarações](#) que dentro de um bloco aninhado não é possível declarar uma variável local com o mesmo nome que uma variável local em um bloco delimitador). O nome do escopo externo, em seguida, é dito como *\_ Hidden\** na região do texto do programa coberto pelo escopo interno, e o acesso ao nome externo só é possível qualificando o nome.

- O escopo de um membro de namespace declarado por um *namespace\_member\_declaration* ([membros de namespace](#)) sem delimitador *namespace\_declaration* é o texto completo do programa.
- O escopo de um membro de namespace declarado por um *namespace\_member\_declaration* dentro de um *namespace\_declaration* cujo nome totalmente qualificado é `N` a *namespace\_body* de cada *namespace\_declaration* cujo nome totalmente qualificado é `N` ou começa com `N`, seguido por um ponto.
- O escopo do nome definido por uma *extern\_alias\_directive* se estende sobre os *using\_directives*, *global\_attributes* e *namespace\_member\_declarations* de seu logo contendo a unidade de compilação ou o corpo do namespace. Um *extern\_alias\_directive* não contribui com nenhum novo membro para o espaço de declaração subjacente. Em outras palavras, um *extern\_alias\_directive* não é transitiva, mas, em vez disso, afeta apenas a unidade de compilação ou o corpo do namespace no qual ele ocorre.
- O escopo de um nome definido ou importado por um *using\_directive* ([usando diretivas](#)) se estende sobre os *namespace\_member\_declarations* do *compilation\_unit* ou *namespace\_body* em que a *using\_directive* ocorre. Uma *using\_directive* pode tornar zero ou mais nomes de namespace, tipo ou membro disponíveis em um *compilation\_unit* ou *namespace\_body* específico, mas não contribui com novos membros para o espaço de declaração subjacente. Em outras palavras, um *using\_directive* não é transitiva, mas, em vez disso, afeta apenas o *compilation\_unit* ou *namespace\_body* em que ele ocorre.
- O escopo de um parâmetro de tipo declarado por um *type\_parameter\_list* em um *class\_declaration* ([declarações de classe](#)) é o *class\_base*, *type\_parameter\_constraints\_clauses* e *class\_body* do *class\_declaration*.
- O escopo de um parâmetro de tipo declarado por um *type\_parameter\_list* em um *struct\_declaration* ([declarações struct](#)) é o *struct\_interfaces*, *type\_parameter\_constraints\_clauses* e *struct\_body* do *struct\_declaration*.
- O escopo de um parâmetro de tipo declarado por um *type\_parameter\_list* em um *interface\_declaration* ([declarações de interface](#)) é o *interface\_base*, *type\_parameter\_constraints\_clauses* e *interface\_body* do *interface\_declaration*.
- O escopo de um parâmetro de tipo declarado por um *type\_parameter\_list* em um *delegate\_declaration* ([declarações de delegado](#)) é o *return\_type*, *formal\_parameter\_list* e *type\_parameter\_constraints\_clauses* do *delegate\_declaration*.
- O escopo de um membro declarado por um *class\_member\_declaration* ([corpo de classe](#)) é a *class\_body* na qual a declaração ocorre. Além disso, o escopo de um membro de classe se estende à *class\_body* dessas classes derivadas que são incluídas no domínio de acessibilidade ([domínios de acessibilidade](#)) do membro.

- O escopo de um membro declarado por um *struct\_member\_declaration* ([membros de struct](#)) é a *struct\_body* na qual a declaração ocorre.
- O escopo de um membro declarado por um *enum\_member\_declaration* ([membros de enumeração](#)) é o *enum\_body* no qual a declaração ocorre.
- O escopo de um parâmetro declarado em um *method\_declaration* ([métodos](#)) é a *method\_body* do *method\_declaration*.
- O escopo de um parâmetro declarado em um *indexer\_declaration* ([indexadores](#)) é o *accessor\_declarations* do *indexer\_declaration*.
- O escopo de um parâmetro declarado em um *operator\_declaration* ([Operators](#)) é o *bloco* desse *operator\_declaration*.
- O escopo de um parâmetro declarado em um *constructor\_declaration* ([construtores de instância](#)) é o *constructor\_initializer* e o *bloco* desse *constructor\_declaration*.
- O escopo de um parâmetro declarado em uma *lambda\_expression* ([expressões de função anônimas](#)) é o *anonymous\_function\_body* do *lambda\_expression*.
- O escopo de um parâmetro declarado em uma *anonymous\_method\_expression* ([expressões de função anônimas](#)) é o *bloco* desse *anonymous\_method\_expression*.
- O escopo de um rótulo declarado em um *labeled\_statement* ([instruções rotuladas](#)) é o *bloco* no qual a declaração ocorre.
- O escopo de uma variável local declarada em uma *local\_variable\_declaration* ([declarações de variável local](#)) é o bloco no qual a declaração ocorre.
- O escopo de uma variável local declarada em uma *switch\_block* de uma *switch* instrução ([a instrução switch](#)) é a *switch\_block*.
- O escopo de uma variável local declarada em uma *for\_initializer* de uma *for* instrução ([a instrução for](#)) é a *for\_initializer*, a *for\_condition*, a *for\_iterator* e a *instrução* contida da *for* instrução.
- O escopo de uma constante local declarada em uma *local\_constant\_declaration* ([declarações de constantes locais](#)) é o bloco no qual a declaração ocorre. É um erro de tempo de compilação para se referir a uma constante local em uma posição textual que precede seu *constant\_declarator*.
- O escopo de uma variável declarada como parte de um *foreach\_statement*, *using\_statement*, *lock\_statement* ou *query\_expression* é determinado pela expansão da construção fornecida.

Dentro do escopo de um namespace, classe, struct ou membro de enumeração, é possível fazer referência ao membro em uma posição textual que precede a declaração do membro. Por exemplo,

C#

```
class A
{
    void F() {
```

```
i = 1;  
}  
  
int i = 0;  
}
```

Aqui, é válido para `F` que o faça referência `i` antes de ser declarado.

Dentro do escopo de uma variável local, é um erro de tempo de compilação para se referir à variável local em uma posição textual que precede a *local\_variable\_declarator* da variável local. Por exemplo,

C#

```
class A  
{  
    int i = 0;  
  
    void F() {  
        i = 1; // Error, use precedes declaration  
        int i;  
        i = 2;  
    }  
  
    void G() {  
        int j = (j = 1); // Valid  
    }  
  
    void H() {  
        int a = 1, b = ++a; // Valid  
    }  
}
```

No `F` método acima, a primeira atribuição para `i` especificamente não se refere ao campo declarado no escopo externo. Em vez disso, ele se refere à variável local e resulta em um erro de tempo de compilação porque ele precede textualmente a declaração da variável. No `G` método, o uso de `j` no inicializador para a declaração de `j` é válido porque o uso não precede a *local\_variable\_declarator*. No `H` método, um *local\_variable\_declarator* subsequentemente se refere a uma variável local declarada em um *local\_variable\_declarator* anterior dentro do mesmo *local\_variable\_declaration*.

As regras de escopo para variáveis locais são projetadas para garantir que o significado de um nome usado em um contexto de expressão seja sempre o mesmo dentro de um bloco. Se o escopo de uma variável local fosse estender apenas de sua declaração para o final do bloco, no exemplo acima, a primeira atribuição seria atribuída à variável de instância e a segunda atribuição seria atribuída à variável local, possivelmente levando a

erros de tempo de compilação se as instruções do bloco fossem posteriormente reorganizadas.

O significado de um nome dentro de um bloco pode ser diferente com base no contexto no qual o nome é usado. No exemplo

```
C#  
  
using System;  
  
class A {}  
  
class Test  
{  
    static void Main() {  
        string A = "hello, world";  
        string s = A; // expression context  
  
        Type t = typeof(A); // type context  
  
        Console.WriteLine(s); // writes "hello, world"  
        Console.WriteLine(t); // writes "A"  
    }  
}
```

o nome `A` é usado em um contexto de expressão para se referir à variável local `A` e em um contexto de tipo para fazer referência à classe `A`.

## Ocultação de nomes

O escopo de uma entidade normalmente abrange mais texto de programa do que o espaço de declaração da entidade. Em particular, o escopo de uma entidade pode incluir declarações que apresentam novos espaços de declaração contendo entidades de mesmo nome. Tais declarações fazem com que a entidade original se torne **\*Hidden\***. Por outro lado, uma entidade é considerada **\_ visível\*** quando não está oculta.

A ocultação de nome ocorre quando os escopos se sobrepõem no aninhamento e quando os escopos se sobrepõem pela herança. As características dos dois tipos de ocultação são descritas nas seções a seguir.

### Ocultando por meio de aninhamento

O nome que oculta o aninhamento pode ocorrer como resultado de aninhar namespaces ou tipos dentro de namespaces, como resultado de aninhar tipos dentro de classes ou estruturas, e como resultado de declarações de parâmetro e variável local.

No exemplo

```
C#  
  
class A  
{  
    int i = 0;  
  
    void F() {  
        int i = 1;  
    }  
  
    void G() {  
        i = 1;  
    }  
}
```

dentro do `F` método, a variável de instância `i` é ocultada pela variável local `i`, mas dentro do `G` método, `i` ainda se refere à variável de instância.

Quando um nome em um escopo interno oculta um nome em um escopo externo, ele oculta todas as ocorrências sobreporadas desse nome. No exemplo

```
C#  
  
class Outer  
{  
    static void F(int i) {}  
  
    static void F(string s) {}  
  
    class Inner  
    {  
        void G() {  
            F(1);           // Invokes Outer.Inner.F  
            F("Hello");    // Error  
        }  
  
        static void F(long l) {}  
    }  
}
```

a chamada `F(1)` invoca o `F` declarado em `Inner` porque todas as ocorrências externas de `F` são ocultadas pela declaração interna. Pelo mesmo motivo, a chamada `F("Hello")` resulta em um erro de tempo de compilação.

## Ocultando por meio de herança

O nome que se oculta pela herança ocorre quando as classes ou estruturas redeclararam nomes que foram herdados de classes base. Esse tipo de ocultação de nome usa uma das seguintes formas:

- Uma constante, campo, propriedade, evento ou tipo introduzido em uma classe ou struct oculta todos os membros da classe base com o mesmo nome.
- Um método introduzido em uma classe ou struct oculta todos os membros da classe base que não são de método com o mesmo nome e todos os métodos da classe base com a mesma assinatura (nome do método e contagem de parâmetros, modificadores e tipos).
- Um indexador introduzido em uma classe ou estrutura oculta todos os indexadores de classe base com a mesma assinatura (contagem de parâmetros e tipos).

As regras que regem as declarações de operador ([operadores](#)) tornam impossível para uma classe derivada declarar um operador com a mesma assinatura de um operador em uma classe base. Portanto, os operadores nunca ocultam um ao outro.

Ao contrário de ocultar um nome de um escopo externo, ocultar um nome acessível de um escopo herdado faz com que um aviso seja relatado. No exemplo

```
C#  
  
class Base  
{  
    public void F() {}  
}  
  
class Derived: Base  
{  
    public void F() {}          // Warning, hiding an inherited name  
}
```

a declaração de `F` no `Derived` faz com que um aviso seja relatado. Ocultar um nome herdado não é especificamente um erro, já que isso impediria uma evolução separada das classes base. Por exemplo, a situação acima pode ter surgido porque uma versão posterior do `Base` introduziu um `F` método que não estava presente em uma versão anterior da classe. Se houvesse um erro na situação acima, qualquer alteração feita em uma classe base em uma biblioteca de classes com controle de versão separada poderia potencialmente fazer com que as classes derivadas se tornassem inválidas.

O aviso causado pela ocultação de um nome herdado pode ser eliminado por meio do uso do `new` modificador:

C#

```
class Base
{
    public void F() {}
}

class Derived: Base
{
    new public void F() {}
}
```

O `new` modificador indica que o `F` no `Derived` é "New", e que realmente se destina a ocultar o membro herdado.

Uma declaração de um novo membro oculta um membro herdado somente dentro do escopo do novo membro.

C#

```
class Base
{
    public static void F() {}
}

class Derived: Base
{
    new private static void F() {}      // Hides Base.F in Derived only
}

class MoreDerived: Derived
{
    static void G() { F(); }           // Invokes Base.F
}
```

No exemplo acima, a declaração de `F` em `Derived` oculta o `F` que foi herdado de `Base`, mas como o novo `F` no `Derived` tem acesso privado, seu escopo não se estende para `MoreDerived`. Portanto, a chamada `F()` em `MoreDerived.G` é válida e será invocada `Base.F`.

## Namespace e nomes de tipos

Vários contextos em um programa C# exigem um *Namespace\_Name* ou um *type\_name* ser especificado.

```

namespace_name
: namespace_or_type_name
;

type_name
: namespace_or_type_name
;

namespace_or_type_name
: identifier type_argument_list?
| namespace_or_type_name '.' identifier type_argument_list?
| qualified_alias_member
;

```

Uma *Namespace\_Name* é uma *namespace\_or\_type\_name* que se refere a um namespace. Após a resolução, conforme descrito abaixo, a *namespace\_or\_type\_name* de um *Namespace\_Name* deve se referir a um namespace ou, caso contrário, ocorrerá um erro de tempo de compilação. Nenhum argumento de tipo ([argumentos de tipo](#)) pode estar presente em um *Namespace\_Name* (somente tipos podem ter argumentos de tipo).

Uma *type\_name* é uma *namespace\_or\_type\_name* que se refere a um tipo. Após a resolução, conforme descrito abaixo, a *namespace\_or\_type\_name* de um *type\_name* deve se referir a um tipo ou, caso contrário, ocorrerá um erro de tempo de compilação.

Se o *namespace\_or\_type\_name* for um membro de alias qualificado, seu significado será conforme descrito em [qualificadores de alias de namespace](#). Caso contrário, um *namespace\_or\_type\_name* tem uma das quatro formas:

- `I`
- `I<A1, ..., Ak>`
- `N.I`
- `N.I<A1, ..., Ak>`

onde `I` é um único identificador, `N` é um *namespace\_or\_type\_name* e `<A1, ..., Ak>` é um *type\_argument\_list* opcional. Quando nenhum *type\_argument\_list* for especificado, considere `k` ser zero.

O significado de um *namespace\_or\_type\_name* é determinado da seguinte maneira:

- Se a *namespace\_or\_type\_name* estiver no formato `I` ou no formato `I<A1, ..., Ak>` :
  - Se `k` for zero e o *namespace\_or\_type\_name* aparecer dentro de uma declaração de método genérico ([métodos](#)) e se essa declaração incluir um parâmetro de

tipo ([parâmetros de tipo](#)) com `I` o nome, o *namespace\_or\_type\_name* se referirá a esse parâmetro de tipo.

- Caso contrário, se o *namespace\_or\_type\_name* aparecer dentro de uma declaração de tipo, para cada tipo de instância `T` ([o tipo de instância](#)), começando com o tipo de instância dessa declaração de tipo e continuando com o tipo de instância de cada classe de delimitadora ou declaração struct (se houver):
  - Se `K` for zero e a declaração de `T` incluir um parâmetro de tipo com `I` o nome, o *namespace\_or\_type\_name* se referirá a esse parâmetro de tipo.
  - Caso contrário, se o *namespace\_or\_type\_name* aparecer dentro do corpo da declaração de tipo, e `T` ou qualquer um de seus tipos base contiver um tipo acessível aninhado com `I` parâmetros de nome e `K` tipo, então o *namespace\_or\_type\_name* se referirá a esse tipo construído com os argumentos de tipo fornecidos. Se houver mais de um tipo, o tipo declarado dentro do tipo mais derivado será selecionado. Observe que os membros não tipo (constantes, campos, métodos, propriedades, indexadores, operadores, construtores de instância, destruidores e construtores estáticos) e membros de tipo com um número diferente de parâmetros de tipo são ignorados ao determinar o significado do *namespace\_or\_type\_name*.
- Se as etapas anteriores não tiverem sido bem-sucedidas, para cada namespace `N`, começando com o namespace no qual o *namespace\_or\_type\_name* ocorre, continuando com cada namespace delimitador (se houver) e terminando com o namespace global, as etapas a seguir são avaliadas até que uma entidade esteja localizada:
  - Se `K` for zero e `I` for o nome de um namespace em `N`, então:
    - Se o local onde a *namespace\_or\_type\_name* ocorre estiver entre uma declaração de namespace para `N` e a declaração de namespace contiver um *extern\_alias\_directive* ou *using\_alias\_directive* que associa o nome a `I` um namespace ou tipo, o *namespace\_or\_type\_name* será ambíguo e ocorrerá um erro em tempo de compilação.
    - Caso contrário, o *namespace\_or\_type\_name* se refere ao namespace chamado `I` em `N`.
  - Caso contrário, se `N` contiver um tipo acessível com `I` parâmetros `Name` e `K` `Type`, então:
    - Se `K` for zero e o local em que o *namespace\_or\_type\_name* ocorrer é incluído por uma declaração de namespace para `N` e a declaração de namespace contém um *extern\_alias\_directive* ou *using\_alias\_directive* que associa o nome a `I` um namespace ou tipo, o *namespace\_or\_type\_name* é ambíguo e ocorre um erro em tempo de compilação.

- Caso contrário, o *namespace\_or\_type\_name* se refere ao tipo construído com os argumentos de tipo fornecidos.
- Caso contrário, se o local em que o *namespace\_or\_type\_name* ocorrer for incluído por uma declaração de namespace para *N* :
  - Se *K* for zero e a declaração de namespace contiver um *extern\_alias\_directive* ou *using\_alias\_directive* que associa o nome a *I* um namespace ou tipo importado, a *namespace\_or\_type\_name* se referirá a esse namespace ou tipo.
  - Caso contrário, se os namespaces e as declarações de tipo importados pelo *using\_namespace\_directive*s e *using\_alias\_directive*s da declaração de namespace contiverem exatamente um tipo acessível *I* com parâmetros de nome e *K* tipo, a *namespace\_or\_type\_name* se referirá a esse tipo construído com os argumentos de tipo fornecidos.
  - Caso contrário, se os namespaces e as declarações de tipo importados pelo *using\_namespace\_directive*s e *using\_alias\_directive*s da declaração de namespace contiverem mais de um tipo acessível com *I* parâmetros de nome e *K* tipo, a *namespace\_or\_type\_name* será ambígua e ocorrerá um erro.
- Caso contrário, o *namespace\_or\_type\_name* será indefinido e ocorrerá um erro em tempo de compilação.
- Caso contrário, a *namespace\_or\_type\_name* será da forma *N.I* ou do formulário *N.I<A1, ..., Ak>* . *N* é resolvido primeiro como um *namespace\_or\_type\_name*. Se a resolução de *N* não for bem-sucedida, ocorrerá um erro em tempo de compilação. Caso contrário, *N.I* ou *N.I<A1, ..., Ak>* é resolvido da seguinte maneira:
  - Se *K* for zero e *N* se referir a um namespace e *N* contiver um namespace aninhado com name *I*, o *namespace\_or\_type\_name* se referirá a esse namespace aninhado.
  - Caso contrário, se se *N* referir a um namespace e *N* contiver um tipo acessível com *I* parâmetros Name e *K* Type, o *namespace\_or\_type\_name* se referirá a esse tipo construído com os argumentos de tipo fornecidos.
  - Caso contrário, se *N* se referir a uma classe (possivelmente construída) ou tipo struct e *N* ou qualquer uma de suas classes base contiver um tipo acessível aninhado com *I* parâmetros Name e *K* Type, o *namespace\_or\_type\_name* se referirá a esse tipo construído com os argumentos de tipo fornecidos. Se houver mais de um tipo, o tipo declarado dentro do tipo mais derivado será selecionado. Observe que, se o significado de *N.I* estiver sendo determinado como parte da resolução da especificação da classe base de *N*, a classe base direta de *N* será considerada objeto ([classes base](#)).

- o Caso contrário, `N.I` é um *namespace\_or\_type\_name* inválido e ocorre um erro em tempo de compilação.

Um *namespace\_or\_type\_name* tem permissão para fazer referência a uma classe estática ([classes estáticas](#)) somente se

- O *namespace\_or\_type\_name* é o `T` em uma *namespace\_or\_type\_name* do formulário `T.I` ou
- O *namespace\_or\_type\_name* é o `T` em uma *typeof\_expression* ([lista de argumentos](#)<sup>1</sup>) do formulário `typeof(T)`.

## Nomes totalmente qualificados

Cada namespace e tipo tem um **nome totalmente qualificado**, que identifica exclusivamente o namespace ou o tipo entre todos os outros. O nome totalmente qualificado de um namespace ou tipo `N` é determinado da seguinte maneira:

- Se `N` for um membro do namespace global, seu nome totalmente qualificado será `N`.
- Caso contrário, seu nome totalmente qualificado é `S.N`, em que `S` é o nome totalmente qualificado do namespace ou tipo no qual `N` é declarado.

Em outras palavras, o nome totalmente qualificado de `N` é o caminho hierárquico completo dos identificadores que resultam `N`, começando do namespace global. Como cada membro de um namespace ou tipo deve ter um nome exclusivo, ele segue que o nome totalmente qualificado de um namespace ou tipo é sempre exclusivo.

O exemplo a seguir mostra várias declarações de namespace e tipo junto com seus nomes totalmente qualificados associados.

```
C#
class A {} // A
namespace X // X
{
    class B // X.B
    {
        class C {} // X.B.C
    }

    namespace Y // X.Y
    {
        class D {} // X.Y.D
    }
}
```

```
}
```

```
namespace X.Y          // X.Y
```

```
{
```

```
    class E {}         // X.Y.E
```

```
}
```

## Gerenciamento automático de memória

O C# emprega o gerenciamento automático de memória, que libera os desenvolvedores de alocar e liberar manualmente a memória ocupada por objetos. As políticas de gerenciamento automático de memória são implementadas por um *coletor de lixo*. O ciclo de vida de gerenciamento de memória de um objeto é o seguinte:

1. Quando o objeto é criado, a memória é alocada para ele, o construtor é executado e o objeto é considerado dinâmico.
2. Se o objeto, ou qualquer parte dele, não puder ser acessado por nenhuma possível continuação de execução, além da execução de destruidores, o objeto será considerado não em uso e se tornará elegível para destruição. O compilador C# e o coletor de lixo podem optar por analisar o código para determinar quais referências a um objeto podem ser usadas no futuro. Por exemplo, se uma variável local que está no escopo for a única referência existente a um objeto, mas essa variável local nunca for referida em qualquer continuação possível de execução do ponto de execução atual no procedimento, o coletor de lixo pode (mas não é necessário) tratar o objeto como não está mais em uso.
3. Depois que o objeto estiver qualificado para destruição, em algum momento não especificado, o destruidor ([destruidores](#)) (se houver) para o objeto será executado. Em circunstâncias normais, o destruidor para o objeto é executado apenas uma vez, embora APIs específicas de implementação possam permitir que esse comportamento seja substituído.
4. Depois que o destruidor de um objeto for executado, se esse objeto ou qualquer parte dele, não puder ser acessado por qualquer continuação possível de execução, incluindo a execução de destruidores, o objeto será considerado inacessível e o objeto se tornará elegível para a coleta.
5. Finalmente, em algum momento depois que o objeto se tornar qualificado para a coleta, o coletor de lixo liberará a memória associada a esse objeto.

O coletor de lixo mantém informações sobre o uso do objeto e usa essas informações para tomar decisões de gerenciamento de memória, como o local na memória para localizar um objeto recém-criado, quando realocar um objeto e quando um objeto não está mais em uso ou inacessível.

Como outras linguagens que assumem a existência de um coletor de lixo, o C# é projetado para que o coletor de lixo possa implementar uma ampla gama de políticas de gerenciamento de memória. Por exemplo, o C# não exige que os destruidores sejam executados ou que os objetos sejam coletados assim que estiverem qualificados ou que os destruidores sejam executados em qualquer ordem específica ou em qualquer thread específico.

O comportamento do coletor de lixo pode ser controlado, em algum grau, por meio de métodos estáticos na classe `System.GC`. Essa classe pode ser usada para solicitar que uma coleção ocorra, destruidores sejam executados (ou não executados) e assim por diante.

Como o coletor de lixo tem permissão de uma ampla latitude ao decidir quando coletar objetos e executar destruidores, uma implementação de conformidade pode produzir uma saída diferente da mostrada pelo código a seguir. O programa

C#

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }
}

class B
{
    object Ref;

    public B(object o) {
        Ref = o;
    }

    ~B() {
        Console.WriteLine("Destruct instance of B");
    }
}

class Test
{
    static void Main() {
        B b = new B(new A());
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

Cria uma instância da classe A e uma instância da classe B. Esses objetos se tornam qualificados para a coleta de lixo quando a variável b é atribuída ao valor null, já que, após esse tempo, é impossível para qualquer código escrito pelo usuário acessá-los. A saída pode ser

Console

```
Destruct instance of A  
Destruct instance of B
```

ou

Console

```
Destruct instance of B  
Destruct instance of A
```

Porque o idioma impõe nenhuma restrição na ordem em que os objetos são coletados como lixo.

Em casos sutis, a distinção entre "qualificado para destruição" e "elegível para coleta" pode ser importante. Por exemplo,

C#

```
using System;  
  
class A  
{  
    ~A() {  
        Console.WriteLine("Destruct instance of A");  
    }  
  
    public void F() {  
        Console.WriteLine("A.F");  
        Test.RefA = this;  
    }  
}  
  
class B  
{  
    public A Ref;  
  
    ~B() {  
        Console.WriteLine("Destruct instance of B");  
        Ref.F();  
    }  
}
```

```

class Test
{
    public static A RefA;
    public static B RefB;

    static void Main()
    {
        RefB = new B();
        RefA = new A();
        RefB.Ref = RefA;
        RefB = null;
        RefA = null;

        // A and B now eligible for destruction
        GC.Collect();
        GC.WaitForPendingFinalizers();

        // B now eligible for collection, but A is not
        if (RefA != null)
            Console.WriteLine("RefA is not null");
    }
}

```

No programa acima, se o coletor de lixo optar por executar o destruidor de `A` antes do destruidor de `B`, a saída desse programa poderá ser:

```

Console

Destruct instance of A
Destruct instance of B
A.F
RefA is not null

```

Observe que, embora a instância do `A` não esteja em uso e o `A` destruidor fosse executado, ainda é possível que os métodos de `A` (neste caso, `F`) sejam chamados de outro destruidor. Além disso, observe que a execução de um destruidor pode fazer com que um objeto se torne utilizável do programa principal novamente. Nesse caso, a execução do `B` destruidor causou uma instância do `A` que não estava em uso anteriormente para tornar-se acessível da referência dinâmica `Test.RefA`. Após a chamada para `WaitForPendingFinalizers`, a instância do `B` é elegível para a coleção, mas a instância do `A` não é, devido à referência `Test.RefA`.

Para evitar confusão e comportamento inesperado, geralmente é uma boa ideia para destruidores apenas realizar a limpeza nos dados armazenados nos próprios campos de seu objeto e não executar ações em objetos referenciados ou em campos estáticos.

Uma alternativa ao uso de destruidores é deixar uma classe implementar a `System.IDisposable` interface. Isso permite que o cliente do objeto determine quando

liberar os recursos do objeto, normalmente acessando o objeto como um recurso em uma `using` instrução ([a instrução using](#)).

## Ordem de execução

A execução de um programa em C# prossegue de forma que os efeitos colaterais de cada thread em execução sejam preservados em pontos de execução críticos. Um *efeito colateral* é definido como uma leitura ou gravação de um campo volátil, uma gravação em uma variável não volátil, uma gravação em um recurso externo e o lançamento de uma exceção. Os pontos de execução críticos em que a ordem desses efeitos colaterais devem ser preservados são referências a campos voláteis ([campos voláteis](#)), `lock` instruções ([a instrução de bloqueio](#)) e criação e encerramento de threads. O ambiente de execução é livre para alterar a ordem de execução de um programa em C#, sujeito às seguintes restrições:

- A dependência de dados é preservada dentro de um thread de execução. Ou seja, o valor de cada variável é calculado como se todas as instruções no thread fossem executadas na ordem do programa original.
- As regras de ordenação de inicialização são preservadas ([inicialização de campo](#) e [inicializadores de variável](#)).
- A ordenação de efeitos colaterais é preservada em relação às leituras e gravações voláteis ([campos voláteis](#)). Além disso, o ambiente de execução não precisa avaliar parte de uma expressão se puder deduzir que o valor da expressão não é usado e que nenhum efeito colateral necessário é produzido (incluindo qualquer causado pela chamada de um método ou acesso a um campo volátil). Quando a execução do programa é interrompida por um evento assíncrono (como uma exceção lançada por outro thread), não há garantia de que os efeitos laterais observáveis estão visíveis na ordem do programa original.

# Tipos

Artigo • 16/09/2021

Os tipos da linguagem C# são divididos em duas categorias principais: *tipos de valor* \_ e \_tipos de referência\*. Os tipos de valor e de referência podem ser *tipos genéricos*, que usam um ou mais parâmetros \_ Type \*. Os parâmetros de tipo podem designar tipos de valor e de referência.

antlr

```
type
: value_type
| reference_type
| type_parameter
| type_unsafe
;
```

A categoria final dos tipos, ponteiros, está disponível somente em código não seguro. Isso é abordado mais detalhadamente em [tipos de ponteiro](#).

Tipos de valor diferem dos tipos de referência, pois as variáveis dos tipos de valor contêm diretamente seus dados, enquanto as variáveis dos tipos de referência armazenam \*referências \_ aos seus dados, o último é conhecido como \_ objetos \*. Com os tipos de referência, é possível que duas variáveis referenciem o mesmo objeto e, assim, possíveis para operações em uma variável afetem o objeto referenciado pela outra variável. Com tipos de valor, as variáveis têm sua própria cópia dos dados e não é possível que as operações em um afetem a outra.

O sistema de tipos do C# é unificado, de modo que um valor de qualquer tipo pode ser tratado como um objeto. Cada tipo no C#, direta ou indiretamente, deriva do tipo de classe `object`, e `object` é a classe base definitiva de todos os tipos. Os valores de tipos de referência são tratados como objetos simplesmente exibindo os valores como tipo `object`. Os valores dos tipos de valor são tratados como objetos executando as operações boxing e unboxing ([boxing e unboxing](#)).

## Tipos de valor

Um tipo de valor é um tipo de struct ou um tipo de enumeração. O C# fornece um conjunto de tipos de struct predefinidos chamados de *tipos simples*. Os tipos simples são identificados por meio de palavras reservadas.

antlr

```
value_type
: struct_type
| enum_type
;

struct_type
: type_name
| simple_type
| nullable_type
;

simple_type
: numeric_type
| 'bool'
;

numeric_type
: integral_type
| floating_point_type
| 'decimal'
;

integral_type
: 'sbyte'
| 'byte'
| 'short'
| 'ushort'
| 'int'
| 'uint'
| 'long'
| 'ulong'
| 'char'
;

floating_point_type
: 'float'
| 'double'
;

nullable_type
: non_nullable_value_type '?'
;

non_nullable_value_type
: type
;

enum_type
: type_name
;
```

Ao contrário de uma variável de um tipo de referência, uma variável de um tipo de valor pode conter o valor `null` somente se o tipo de valor for um tipo anulável. Para cada tipo de valor não anulável, há um tipo de valor anulável correspondente, indicando o mesmo conjunto de valores, mas o valor `null`.

A atribuição a uma variável de um tipo de valor cria uma cópia do valor que está sendo atribuído. Isso difere da atribuição a uma variável de um tipo de referência, que copia a referência, mas não o objeto identificado pela referência.

## O tipo System. ValueType

Todos os tipos de valor herdam implicitamente da classe `System.ValueType`, que, por sua vez, herda da classe `object`. Não é possível que qualquer tipo seja derivado de um tipo de valor, e tipos de valor são, portanto, lacrados implicitamente ([classes lacradas](#)).

Observe que `System.ValueType` não é um *value\_type*. Em vez disso, é uma *class\_type* da qual todas as *value\_type*s são derivadas automaticamente.

## Construtores padrão

Todos os tipos de valor declaram implicitamente um construtor de instância sem parâmetros públicos chamado de **\*construtor padrão\***. O construtor padrão retorna uma instância inicializada em zero conhecida como *\_valor padrão\** para o tipo de valor:

- Para todos os *Simple\_Type*s, o valor padrão é o valor produzido por um padrão de bit de todos os zeros:
  - Para `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` e `ulong`, o valor padrão é `0`.
  - Para `char`, o valor padrão é `'\x0000'`.
  - Para `float`, o valor padrão é `0.0f`.
  - Para `double`, o valor padrão é `0.0d`.
  - Para `decimal`, o valor padrão é `0.0m`.
  - Para `bool`, o valor padrão é `false`.
- Para um *enum\_type* `E`, o valor padrão é `0`, convertido para o tipo `E`.
- Para um *struct\_type*, o valor padrão é o valor produzido definindo todos os campos de tipo de valor como seu valor padrão e todos os campos de tipo de referência como `null`.
- Para um *nullable\_type* o valor padrão é uma instância para a qual a `HasValue` propriedade é `false` e a `Value` propriedade é indefinida. O valor padrão também é conhecido como o *valor nulo* do tipo anulável.

Como qualquer outro construtor de instância, o construtor padrão de um tipo de valor é invocado usando o `new` operador. Por motivos de eficiência, esse requisito não tem como objetivo realmente fazer com que a implementação gere uma chamada de construtor. No exemplo a seguir, `i` as variáveis e `j` são ambas inicializadas como zero.

C#

```
class A
{
    void F() {
        int i = 0;
        int j = new int();
    }
}
```

Como cada tipo de valor implicitamente tem um construtor de instância sem parâmetros público, não é possível que um tipo de struct contenha uma declaração explícita de um construtor sem parâmetros. No entanto, um tipo struct é permitido para declarar construtores de instância com parâmetros ([construtores](#)).

## Tipos struct

Um tipo de struct é um tipo de valor que pode declarar constantes, campos, métodos, propriedades, indexadores, operadores, construtores de instância, construtores estáticos e tipos aninhados. A declaração de tipos de struct é descrita em [declarações de struct](#).

## Tipos simples

O C# fornece um conjunto de tipos de struct predefinidos chamados de *tipos simples*. Os tipos simples são identificados por meio de palavras reservadas, mas essas palavras reservadas são simplesmente aliases para tipos de struct predefinidos no `System` namespace, conforme descrito na tabela a seguir.

Palavra reservada	Tipo com alias
<code>sbyte</code>	<code>System.SByte</code>
<code>byte</code>	<code>System.Byte</code>
<code>short</code>	<code>System.Int16</code>
<code>ushort</code>	<code>System.UInt16</code>
<code>int</code>	<code>System.Int32</code>

Palavra reservada	Tipo com alias
<code>uint</code>	<code>System.UInt32</code>
<code>long</code>	<code>System.Int64</code>
<code>ulong</code>	<code>System.UInt64</code>
<code>char</code>	<code>System.Char</code>
<code>float</code>	<code>System.Single</code>
<code>double</code>	<code>System.Double</code>
<code>bool</code>	<code>System.Boolean</code>
<code>decimal</code>	<code>System.Decimal</code>

Como um tipo simples é alias de um tipo struct, cada tipo simples tem membros. Por exemplo, `int` tem os membros declarados em `System.Int32` e os membros herdados de `System.Object`, e as instruções a seguir são permitidas:

C#

```
int i = int.MaxValue;           // System.Int32.MaxValue constant
string s = i.ToString();       // System.Int32.ToString() instance method
string t = 123.ToString();     // System.Int32.ToString() instance method
```

Os tipos simples diferem de outros tipos de struct, pois permitem determinadas operações adicionais:

- Os tipos mais simples permitem que os valores sejam criados escrevendo *literais* ([literais](#)). Por exemplo, `123` é um literal do tipo `int` e `'a'` é um literal do tipo `char`. O C# não faz provisão para literais de tipos struct em geral, e valores não padrão de outros tipos struct são, por fim, sempre criados por meio de construtores de instância desses tipos struct.
- Quando os operandos de uma expressão são todas constantes de tipo simples, é possível que o compilador avalie a expressão em tempo de compilação. Essa expressão é conhecida como um *constant\_expression* ([expressões constantes](#)). As expressões que envolvem operadores definidos por outros tipos de struct não são consideradas expressões constantes.
- Por meio `const` de declarações, é possível declarar constantes dos tipos simples ([constantes](#)). Não é possível ter constantes de outros tipos de struct, mas um efeito semelhante é fornecido por `static readonly` campos.

- As conversões que envolvem tipos simples podem participar da avaliação de operadores de conversão definidos por outros tipos de struct, mas um operador de conversão definido pelo usuário nunca pode participar da avaliação de outro operador definido pelo usuário ([avaliação de conversões definidas pelo usuário](#)).

## Tipos integrais

O C# dá suporte a nove tipos integrais: `sbyte` `byte` `short` `ushort` `int` `uint` `long` `ulong` e `char`. Os tipos integrais têm os seguintes tamanhos e intervalos de valores:

- O `sbyte` tipo representa os inteiros de 8 bits assinados com valores entre -128 e 127.
- O `byte` tipo representa inteiros de 8 bits não assinados com valores entre 0 e 255.
- O `short` tipo representa os inteiros de 16 bits assinados com valores entre -32768 e 32767.
- O `ushort` tipo representa inteiros de 16 bits não assinados com valores entre 0 e 65535.
- O `int` tipo representa os inteiros de 32 bits assinados com valores entre -2147483648 e 2147483647.
- O `uint` tipo representa inteiros de 32 bits sem sinal com valores entre 0 e 4294967295.
- O `long` tipo representa os inteiros de 64 bits assinados com valores entre -9.223.372.036.854.775.808 e 9223372036854775807.
- O `ulong` tipo representa inteiros de 64 bits sem sinal com valores entre 0 e 18446744073709551615.
- O `char` tipo representa inteiros de 16 bits não assinados com valores entre 0 e 65535. O conjunto de valores possíveis para o tipo `char` corresponde ao conjunto de caracteres Unicode. Embora `char` o tenha a mesma representação que `ushort`, nem todas as operações permitidas em um tipo são permitidas no outro.

Os operadores unários e binários de tipo integral sempre operam com precisão de 32 bits assinada, precisão de 32 bits sem sinal, precisão de 64 bits assinada ou precisão de 64 bits não assinado:

- Para unários `+` e `~` operadores, o operando é convertido em tipo `T`, em que `T` é o primeiro de `int`, `uint`, `long`, e `ulong` que pode representar totalmente todos os valores possíveis do operando. Em seguida, a operação é executada usando a precisão do tipo `T` e o tipo do resultado é `T`.
- Para o operador unário `-`, o operando é convertido em tipo `T`, em que `T` é o primeiro de `int` e `long` que pode representar totalmente todos os valores

possíveis do operando. Em seguida, a operação é executada usando a precisão do tipo `T` e o tipo do resultado é `T`. O operador unário `-` não pode ser aplicado a operandos do tipo `ulong`.

- Para os operadores Binary`..... + - * .. / % & ^ | == != > < >= &lt;=`, os operandos são convertidos em tipo `T`, em que `T` é o primeiro de `int`, `uint`, `long` e `ulong` que pode representar totalmente todos os valores possíveis de ambos os operandos. Em seguida, a operação é executada usando a precisão do tipo `T` e o tipo do resultado é `T` (ou `bool` para os operadores relacionais). Não é permitido que um operando seja do tipo `long` e o outro seja do tipo `ulong` com os operadores binários.
- Para o binário `<<` e os `>>` operadores, o operando esquerdo é convertido em tipo `T`, em que `T` é o primeiro de `int`, `uint`, `long` e `ulong` que pode representar totalmente todos os valores possíveis do operando. Em seguida, a operação é executada usando a precisão do tipo `T` e o tipo do resultado é `T`.

O `char` tipo é classificado como um tipo integral, mas difere dos outros tipos integrais de duas maneiras:

- Não há conversões implícitas de outros tipos para o `char` tipo. Em particular, embora os `sbyte` tipos, `byte` e `ushort` tenham intervalos de valores que são totalmente representáveis usando o `char` tipo, conversões implícitas de `sbyte`, `byte` ou `ushort` para `char` não existir.
- As constantes do `char` tipo devem ser gravadas como *character\_literal*s ou como *integer\_literal*s em combinação com uma conversão para tipo `char`. Por exemplo, `(char)10` é o mesmo que `'\x000A'`.

Os `checked` operadores e as `unchecked` instruções são usados para controlar a verificação de estouro para operações aritméticas de tipo integral e conversões ([os operadores marcados e desmarcados](#)). Em um `checked` contexto, um estouro produz um erro de tempo de compilação ou faz com que um seja `System.OverflowException` gerado. Em um `unchecked` contexto, os estouros são ignorados e quaisquer bits de ordem superior que não couberem no tipo de destino serão descartados.

## Tipos de ponto flutuante

O C# dá suporte a dois tipos de ponto flutuante: `float` e `double`. Os `float` `double` tipos e são representados usando os formatos IEEE 754 de precisão única de 32 bits e de precisão dupla de 64 bits, que fornecem os seguintes conjuntos de valores:

- Zero positivo e zero negativo. Na maioria das situações, zero positivo e zero negativo se comportam de forma idêntica ao valor zero simples, mas determinadas operações distinguem entre os dois ([operador de divisão](#)).
- Infinity positivo e infinito negativo. Os infinitos são produzidos por operações como a divisão de um número diferente de zero por zero. Por exemplo, `1.0 / 0.0` produz infinito positivo e `-1.0 / 0.0` produz infinito negativo.
- O valor *não-a-numérico*, geralmente abreviado como NaN. NaNs são produzidos por operações de ponto flutuante inválidas, como a divisão de zero por zero.
- O conjunto finito de valores diferentes de zero do formulário `s * m * 2^e`, em que `s` é 1 ou -1, e `m` e `e` são determinados pelo tipo de ponto flutuante específico: para `float`,  $0 < m < 2^{24}$  e  $-149 \leq e \leq 104$  para `double`,  $0 < m < 2^{53}$  e  $-1075 \leq e \leq 970$ . Números de ponto flutuante desnormalizados são considerados valores não zero válidos.

O `float` tipo pode representar valores que variam de aproximadamente `1.5 * 10^-45` a `3.4 * 10^38` com uma precisão de 7 dígitos.

O `double` tipo pode representar valores que variam de aproximadamente `5.0 * 10^-324` a `1.7 * 10^308` com uma precisão de 15-16 dígitos.

Se um dos operandos de um operador binário for de um tipo de ponto flutuante, o outro operando deverá ser de um tipo integral ou de um tipo de ponto flutuante, e a operação será avaliada da seguinte maneira:

- Se um dos operandos for de um tipo integral, esse operando será convertido no tipo de ponto flutuante do outro operando.
- Em seguida, se qualquer um dos operandos for do tipo `double`, o outro operando será convertido em `double`, a operação será executada usando pelo menos `double` intervalo e precisão, e o tipo do resultado será `double` (ou `bool` para os operadores relacionais).
- Caso contrário, a operação é executada usando pelo menos `float` intervalo e precisão, e o tipo do resultado é `float` (ou `bool` para os operadores relacionais).

Os operadores de ponto flutuante, incluindo os operadores de atribuição, nunca produzem exceções. Em vez disso, em situações excepcionais, as operações de ponto flutuante produzem zero, infinito ou NaN, conforme descrito abaixo:

- Se o resultado de uma operação de ponto flutuante for muito pequeno para o formato de destino, o resultado da operação se tornará positivo zero ou negativo zero.
- Se o resultado de uma operação de ponto flutuante for muito grande para o formato de destino, o resultado da operação se tornará infinito positivo ou infinito

negativo.

- Se uma operação de ponto flutuante for inválida, o resultado da operação se tornará NaN.
- Se um ou ambos os operandos de uma operação de ponto flutuante forem NaN, o resultado da operação se tornará NaN.

As operações de ponto flutuante podem ser executadas com precisão mais alta do que o tipo de resultado da operação. Por exemplo, algumas arquiteturas de hardware dão suporte a um tipo de ponto flutuante "estendido" ou "longo Duplo" com maior intervalo e precisão do que o `double` tipo e executa implicitamente todas as operações de ponto flutuante usando esse tipo de precisão mais alto. Somente o custo excessivo no desempenho pode fazer com que tais arquiteturas de hardware sejam feitas para executar operações de ponto flutuante com menos precisão e, em vez de exigir uma implementação para perder o desempenho e a precisão, o C# permite que um tipo de precisão mais alto seja usado para todas as operações de ponto flutuante. Além de fornecer resultados mais precisos, isso raramente tem efeitos mensuráveis. No entanto, em expressões do formulário `x * y / z`, em que a multiplicação produz um resultado fora do `double` intervalo, mas a divisão subsequente traz o resultado temporário de volta para o `double` intervalo, o fato de que a expressão é avaliada em um formato de intervalo superior pode fazer com que um resultado finito seja produzido em vez de um infinito.

## O tipo decimal

O tipo `decimal` é um tipo de dados de 128 bits adequado para cálculos financeiros e monetários. O `decimal` tipo pode representar valores variando de `1.0 * 10^-28` a aproximadamente `7.9 * 10^28` com 28-29 dígitos significativos.

O conjunto finito de valores do tipo é `decimal` do formulário `(-1)^s * c * 10^-e`, em que o sinal `s` é 0 ou 1, o coeficiente `c` é fornecido por `0 <= c < 2^96` e a escala `e` é tal `0 <= e <= 28`. O `decimal` tipo não dá suporte a zeros assinados, infinitos ou NaN. Um `decimal` é representado como um inteiro de 96 bits dimensionado por uma potência de dez. Para `decimal` `s` com um valor absoluto menor que `1.0m`, o valor é exato para a casa de 28 decimais, mas não mais. Para `decimal` `s` com um valor absoluto maior ou igual a `1.0m`, o valor é exato para 28 ou 29 dígitos. Ao contrário dos `float` `double` tipos de dados e, os números de fração decimais, como 0,1, podem ser representados exatamente na `decimal` representação. Nas `float` `double` representações e, esses números geralmente são frações infinitas, tornando essas representações mais propensas a erros de arredondamento.

Se um dos operandos de um operador binário for do tipo `decimal`, o outro operando deverá ser de um tipo integral ou do tipo `decimal`. Se um operando de tipo integral estiver presente, ele será convertido em `decimal` antes que a operação seja executada.

O resultado de uma operação em valores do tipo `decimal` é que isso resultaria no cálculo de um resultado exato (preservando a escala, conforme definido para cada operador) e, em seguida, arredondando para se ajustar à representação. Os resultados são arredondados para o valor representável mais próximo e, quando um resultado é igualmente próximo a dois valores representáveis, para o valor que tem um número par na posição de dígito menos significativa (isso é conhecido como "arredondamento do banco"). Um resultado zero sempre tem um sinal de 0 e uma escala de 0.

Se uma operação aritmética decimal produzir um valor menor ou igual a  $5 * 10^{-29}$  em um valor absoluto, o resultado da operação se tornará zero. Se uma `decimal` operação aritmética produzir um resultado muito grande para o `decimal` formato, um `System.OverflowException` será lançado.

O `decimal` tipo tem maior precisão, mas um intervalo menor do que os tipos de ponto flutuante. Assim, as conversões dos tipos de ponto flutuante `decimal` podem produzir exceções de estouro, e as conversões de `decimal` para os tipos de ponto flutuante podem causar perda de precisão. Por esses motivos, não existem conversões implícitas entre os tipos de ponto flutuante e `decimal`, sem conversões explícitas, não é possível misturar ponto flutuante e `decimal` operandos na mesma expressão.

## O tipo `bool`

O `bool` tipo representa as quantidades lógicas booleanas. Os valores possíveis do tipo `bool` são `true` e `false`.

Não existem conversões padrão entre os `bool` outros tipos. Em particular, o `bool` tipo é distinto e separado dos tipos inteiros, e um `bool` valor não pode ser usado no lugar de um valor integral e vice-versa.

Nas linguagens C e C++, um valor integral zero ou de ponto flutuante, ou um ponteiro nulo pode ser convertido para o valor booleano `false` e um valor de ponto flutuante ou integral diferente de zero, ou um ponteiro não nulo pode ser convertido para o valor booleano `true`. No C#, essas conversões são realizadas comparando explicitamente um valor integral ou de ponto flutuante para zero ou comparando explicitamente uma referência de objeto para `null`.

## Tipos de enumeração

Um tipo de enumeração é um tipo distinto com constantes nomeadas. Cada tipo de enumeração tem um tipo subjacente, que deve ser,,, `byte` `sbyte` `short` `ushort` `int` `uint` `long` ou `ulong`. O conjunto de valores do tipo de enumeração é o mesmo que o conjunto de valores do tipo subjacente. Os valores do tipo de enumeração não são restritos aos valores das constantes nomeadas. Os tipos de enumeração são definidos por meio de declarações de enumeração ([declarações enum](#)).

## Tipos anuláveis

Um tipo anulável pode representar todos os valores de seu *tipo subjacente*, além de um valor nulo adicional. Um tipo anulável é gravado `T?`, em que `T` é o tipo subjacente. Essa sintaxe é abreviada para `System.Nullable<T>`, e os dois formulários podem ser usados de forma intercambiável.

Um *tipo de valor não anulável* inversamente é qualquer tipo de valor diferente de `System.Nullable<T>` e sua abreviação `T?` (para qualquer `T`), além de qualquer parâmetro de tipo restrito a ser um tipo de valor não anulável (ou seja, qualquer parâmetro de tipo com uma `struct` restrição). O `System.Nullable<T>` tipo especifica a restrição de tipo de valor para `T` ([restrições de parâmetro de tipo](#)), o que significa que o tipo subjacente de um tipo anulável pode ser qualquer tipo de valor não anulável. O tipo subjacente de um tipo anulável não pode ser um tipo anulável ou um tipo de referência. Por exemplo, `int??` e `string?` são tipos inválidos.

Uma instância de um tipo anulável `T?` tem duas propriedades públicas somente leitura:

- Uma `HasValue` Propriedade do tipo `bool`
- Uma `Value` Propriedade do tipo `T`

Uma instância do que `HasValue` é verdadeira é considerada não nula. Uma instância não nula contém um valor conhecido e `Value` retorna esse valor.

Uma instância do que `HasValue` é falsa é considerada nula. Uma instância nula tem um valor indefinido. A tentativa de ler o `Value` de uma instância nula faz com que um seja `System.InvalidOperationException` gerado. O processo de acessar a `Value` propriedade de uma instância anulável é chamado de **desencapsulamento**.

Além do construtor padrão, todo tipo anulável `T?` tem um construtor público que usa um único argumento do tipo `T`. Dado um valor `x` do tipo `T`, uma invocação de construtor do formulário

C#

```
new T?(x)
```

Cria uma instância não nula do `T?` para a qual a `value` propriedade é `x`. O processo de criação de uma instância não nula de um tipo anulável para um determinado valor é chamado de *encapsulamento*.

Conversões implícitas estão disponíveis do `null` literal para `T?` ([conversões de literal NULL](#)) e de `T` para `T?` ([conversões anuláveis implícitas](#)).

## Tipos de referência

Um tipo de referência é um tipo de classe, um tipo de interface, um tipo de matriz ou um tipo de delegado.

antlr

```
reference_type
: class_type
| interface_type
| array_type
| delegate_type
;

class_type
: type_name
| 'object'
| 'dynamic'
| 'string'
;

interface_type
: type_name
;

array_type
: non_array_type rank_specifier+
;

non_array_type
: type
;

rank_specifier
: '[' dim_separator* ']'
;

dim_separator
```

```
: ','  
;  
  
delegate_type  
: type_name  
;
```

Um valor de tipo de referência é uma referência a uma *\*instância\_* do tipo, o último conhecido como *\_Object*. O valor especial `null` é compatível com todos os tipos de referência e indica a ausência de uma instância.

## Tipos de aula

Um tipo de classe define uma estrutura de dados que contém membros de dados (constantes e campos), membros de função (métodos, propriedades, eventos, indexadores, operadores, construtores de instância, destruidores e construtores estáticos) e tipos aninhados. Os tipos de classe dão suporte à herança, um mecanismo pelo qual classes derivadas podem estender e especializar classes base. Instâncias de tipos de classe são criadas usando *object\_creation\_expressions* ([expressões de criação de objeto](#)).

Os tipos de classe são descritos em [classes](#).

Determinados tipos de classe predefinidos têm significado especial na linguagem C#, conforme descrito na tabela a seguir.

Tipo de classe	Descrição
<code>System.Object</code>	A classe base definitiva de todos os outros tipos. Consulte <a href="#">o tipo de objeto</a> .
<code>System.String</code>	O tipo de cadeia de caracteres da linguagem C#. Consulte <a href="#">o tipo de cadeia de caracteres</a> .
<code>System.ValueType</code>	A classe base de todos os tipos de valor. Consulte <a href="#">o tipo System. ValueType</a> .
<code>System.Enum</code>	A classe base de todos os tipos enum. Consulte <a href="#">enums</a> .
<code>System.Array</code>	A classe base de todos os tipos de matriz. Consulte <a href="#">Matrizes</a> .
<code>System.Delegate</code>	A classe base de todos os tipos delegados. Consulte <a href="#">delegados</a> .
<code>System.Exception</code>	A classe base de todos os tipos de exceção. Consulte <a href="#">exceções</a> .

## O tipo de objeto

O `object` tipo de classe é a classe base definitiva de todos os outros tipos. Todo tipo em C#, direta ou indiretamente, deriva do `object` tipo de classe.

A palavra-chave `object` é simplesmente um alias para a classe predefinida `System.Object`.

## O tipo dinâmico

O `dynamic` tipo, como `object`, pode fazer referência a qualquer objeto. Quando os operadores são aplicados a expressões do tipo `dynamic`, sua resolução é adiada até que o programa seja executado. Portanto, se o operador não puder ser aplicado legalmente ao objeto referenciado, nenhum erro será fornecido durante a compilação. Em vez disso, uma exceção será lançada quando a resolução do operador falhar em tempo de execução.

Sua finalidade é permitir a vinculação dinâmica, que é descrita em detalhes na [vinculação dinâmica](#).

`dynamic` é considerado idêntico a `object`, exceto nos seguintes aspectos:

- As operações em expressões do tipo `dynamic` podem ser vinculadas dinamicamente ([associação dinâmica](#)).
- A inferência de tipos ([inferência de tipos](#)) será preferida `dynamic` `object` se ambos forem candidatos.

Devido a essa equivalência, o seguinte contém:

- Há uma conversão de identidade implícita entre `object` e `dynamic` entre os tipos construídos que são iguais ao substituir `dynamic` por `object`
- Conversões implícitas e explícitas de `object` para `dynamic` também se aplicam a `dynamic` e de `dynamic`.
- As assinaturas de método que são iguais ao substituir `dynamic` por `object` são consideradas a mesma assinatura
- O tipo `dynamic` é indistinguível de `object` em tempo de execução.
- Uma expressão do tipo `dynamic` é referenciada como uma *expressão dinâmica*.

## O tipo de cadeia de caracteres

O `string` tipo é um tipo de classe lacrado herdado diretamente do `object`. As instâncias da `string` classe representam as cadeias de caracteres Unicode.

Os valores do `string` tipo podem ser gravados como literais de cadeia de caracteres ([literais de cadeia de caracteres](#)).

A palavra-chave `string` é simplesmente um alias para a classe predefinida `System.String`.

## Tipos de interface

Uma interface define um contrato. Uma classe ou struct que implementa uma interface deve aderir a seu contrato. Uma interface pode herdar de várias interfaces base e uma classe ou estrutura pode implementar várias interfaces.

Os tipos de interface são descritos em [interfaces](#).

## Tipos de matriz

Uma matriz é uma estrutura de dados que contém zero ou mais variáveis que são acessadas por meio de índices computados. As variáveis contidas em uma matriz, também chamadas de elementos da matriz, são todas do mesmo tipo, e esse tipo é chamado de tipo de elemento da matriz.

Os tipos de matriz são descritos em [matrizes](#).

## Tipos delegados

Um delegado é uma estrutura de dados que se refere a um ou mais métodos. Para métodos de instância, ele também se refere às suas instâncias de objeto correspondentes.

O equivalente mais próximo de um delegado em C ou C++ é um ponteiro de função, mas enquanto um ponteiro de função só pode referenciar funções estáticas, um delegado pode fazer referência a métodos estáticos e de instância. No último caso, o delegado armazena não apenas uma referência ao ponto de entrada do método, mas também uma referência à instância do objeto na qual invocar o método.

Os tipos delegados são descritos em [delegados](#).

## Conversão boxing e unboxing

O conceito de boxing e unboxing é fundamental para o sistema de tipos do C#. Ele fornece uma ponte entre *value\_type*s e *reference\_type*s, permitindo que qualquer valor

de uma *value\_type* seja convertido de e para o tipo `object`. Boxing e unboxing permitem uma exibição unificada do sistema de tipos, em que um valor de qualquer tipo pode ser tratado, por fim, como um objeto.

## Conversões de Boxing

Uma conversão boxing permite que um *value\_type* seja convertido implicitamente em um *reference\_type*. Existem as seguintes conversões de boxing:

- De qualquer *value\_type* para o tipo `object`.
- De qualquer *value\_type* para o tipo `System.ValueType`.
- De qualquer *non\_nullable\_value\_type* a qualquer *interface\_type* implementada pelo *value\_type*.
- De qualquer *nullable\_type* a qualquer *interface\_type* implementada pelo tipo subjacente do *nullable\_type*.
- De qualquer *enum\_type* para o tipo `System.Enum`.
- De qualquer *nullable\_type* com um *enum\_type* subjacente para o tipo `System.Enum`.
- Observe que uma conversão implícita de um parâmetro de tipo será executada como uma conversão boxing se em tempo de execução acabar convertendo de um tipo de valor para um tipo de referência ([conversões implícitas envolvendo parâmetros de tipo](#)).

Boxing um valor de uma *non\_nullable\_value\_type* consiste em alocar uma instância de objeto e copiar o valor de *non\_nullable\_value\_type* nessa instância.

Boxing um valor de uma *nullable\_type* produz uma referência nula se for o `null` valor (`HasValue` is `false`), ou o resultado de quebrar e Boxing o valor subjacente, caso contrário.

O processo real de boxing, um valor de uma *non\_nullable\_value\_type* é mais bem explicado ao imaginar a existência de uma **classe Boxing** genérica, que se comporta como se fosse declarada da seguinte maneira:

C#

```
sealed class Box<T>: System.ValueType
{
    T value;

    public Box(T t) {
        value = t;
    }
}
```

A Boxing de um valor `v` do tipo `T` agora consiste em executar a expressão `new Box<T>(v)` e retornar a instância resultante como um valor do tipo `object`. Portanto, as instruções

```
C#
```

```
int i = 123;
object box = i;
```

corresponde conceitualmente a

```
C#
```

```
int i = 123;
object box = new Box<int>(i);
```

Uma classe Boxing como `Box<T>` acima não existe de fato e o tipo dinâmico de um valor boxed não é, na verdade, um tipo de classe. Em vez disso, um valor em caixa do tipo `T` tem o tipo dinâmico `T` e uma verificação de tipo dinâmico usando o `is` operador pode simplesmente referenciar o tipo `T`. Por exemplo,

```
C#
```

```
int i = 123;
object box = i;
if (box is int) {
    Console.WriteLine("Box contains an int");
}
```

produzirá a cadeia de caracteres "`Box contains an int`" no console.

Uma conversão de Boxing implica fazer uma cópia do valor em caixa. Isso é diferente de uma conversão de um *reference\_type* para tipo `object`, no qual o valor continua referenciando a mesma instância e simplesmente é considerado o tipo menos derivado `object`. Por exemplo, dada a declaração

```
C#
```

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
    }
}
```

```
    this.y = y;
}
}
```

as instruções a seguir

C#

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.WriteLine((Point)box).x;
```

produzirá o valor 10 no console, pois a operação Boxing implícita que ocorre na atribuição de `p` para `box` faz com que o valor de seja `p` copiado. `Point` Foi declarado um `class` em vez disso, o valor 20 seria output porque `p` e `box` faria referência à mesma instância.

## Conversões de unboxing

Uma conversão unboxing permite que um *reference\_type* seja explicitamente convertido em um *value\_type*. Existem as seguintes conversões de não conboxing:

- Do tipo `object` para qualquer *value\_type*.
- Do tipo `System.ValueType` para qualquer *value\_type*.
- De qualquer *interface\_type* a qualquer *non\_nullable\_value\_type* que implemente o *interface\_type*.
- De qualquer *interface\_type* a qualquer *nullable\_type* cujo tipo subjacente implemente o *interface\_type*.
- Do tipo `System.Enum` para qualquer *enum\_type*.
- Do tipo `System.Enum` para qualquer *nullable\_type* com um *enum\_type* subjacente.
- Observe que uma conversão explícita em um parâmetro de tipo será executada como uma conversão sem Boxing se, em tempo de execução, ele terminar a conversão de um tipo de referência para um tipo de valor ([conversões dinâmicas explícitas](#)).

Uma operação de unboxing para um *non\_nullable\_value\_type* consiste na primeira verificação de que a instância do objeto é um valor em caixa do *non\_nullable\_value\_type* especificado e, em seguida, copiar o valor para fora da instância.

Unboxing para um *nullable\_type* produz o valor nulo do *nullable\_type* se o operando de origem for `null` ou o resultado encapsulado de uma instância de objeto para o tipo subjacente de *nullable\_type* caso contrário.

Fazendo referência à classe de conversão de erros imaginário descrita na seção anterior, uma conversão unboxing de um objeto `box` em um `value_type T` consiste em executar a expressão `((Box<T>)box).value`. Portanto, as instruções

```
C#
```

```
object box = 123;
int i = (int)box;
```

corresponde conceitualmente a

```
C#
```

```
object box = new Box<int>(123);
int i = ((Box<int>)box).value;
```

Para que uma conversão sem Boxing para um determinado `non_nullable_value_type` seja realizada em tempo de execução, o valor do operando de origem deve ser uma referência a um valor de caixa desse `non_nullable_value_type`. Se o operando de origem for `null`, um `System.NullReferenceException` será lançado. Se o operando de origem for uma referência a um objeto incompatível, um `System.InvalidCastException` será gerado.

Para que uma conversão sem Boxing para um determinado `nullable_type` seja realizada em tempo de execução, o valor do operando de origem deve ser `null` ou uma referência a um valor em caixa da `non_nullable_value_type` subjacente do `nullable_type`. Se o operando de origem for uma referência a um objeto incompatível, um `System.InvalidCastException` será gerado.

## Tipos construídos

Uma declaração de tipo genérico, por si só, denota um *tipo genérico desassociado* – que é usado como um "plano gráfico" para formar muitos tipos diferentes, por meio da aplicação de \_argumentos de tipo\*. Os argumentos de tipo são gravados entre colchetes angulares (`<` e `>`) imediatamente após o nome do tipo genérico. Um tipo que inclui pelo menos um argumento de tipo é chamado de *tipo construído*. Um tipo construído pode ser usado na maioria dos lugares no idioma em que um nome de tipo pode aparecer. Um tipo genérico não associado só pode ser usado dentro de um `_typeof_expression` \* (o operador `typeof`).

Os tipos construídos também podem ser usados em expressões como nomes simples ([nomes simples](#)) ou ao acessar um membro ([acesso de membro](#)).

Quando um *namespace\_or\_type\_name* é avaliado, somente tipos genéricos com o número correto de parâmetros de tipo são considerados. Portanto, é possível usar o mesmo identificador para identificar diferentes tipos, desde que os tipos tenham números diferentes de parâmetros de tipo. Isso é útil ao misturar classes genéricas e não genéricas no mesmo programa:

```
C#  
  
namespace Widgets  
{  
    class Queue {...}  
    class Queue<TElement> {...}  
}  
  
namespace MyApplication  
{  
    using Widgets;  
  
    class X  
    {  
        Queue q1;           // Non-generic Widgets.Queue  
        Queue<int> q2;     // Generic Widgets.Queue  
    }  
}
```

Um *type\_name* pode identificar um tipo construído, mesmo que ele não especifique parâmetros de tipo diretamente. Isso pode ocorrer quando um tipo é aninhado dentro de uma declaração de classe genérica, e o tipo de instância da declaração recipiente é usado implicitamente para pesquisa de nome ([tipos aninhados em classes genéricas](#)):

```
C#  
  
class Outer<T>  
{  
    public class Inner {...}  
  
    public Inner i;           // Type of i is Outer<T>.Inner  
}
```

Em código não seguro, um tipo construído não pode ser usado como um *unmanaged\_type* ([tipos de ponteiro](#)).

## Argumentos de tipo

Cada argumento em uma lista de argumentos de tipo é simplesmente um *tipo*.

```
type_argument_list
  : '<' type_arguments '>'
  ;

type_arguments
  : type_argument (',' type_argument)*
  ;
;

type_argument
  : type
  ;
```

Em código não seguro ([código não seguro](#)), um *type\_argument* não pode ser um tipo de ponteiro. Cada argumento de tipo deve atender a qualquer restrição no parâmetro de tipo correspondente ([restrições de parâmetro de tipo](#)).

## Tipos abertos e fechados

Todos os tipos podem ser classificados como \* tipos **abertos** \_ ou \_ *tipos fechados* \*. Um tipo aberto é um tipo que envolve parâmetros de tipo. Mais especificamente:

- Um parâmetro de tipo define um tipo aberto.
- Um tipo de matriz é um tipo aberto se e somente se seu tipo de elemento for um tipo aberto.
- Um tipo construído é um tipo aberto se e somente se um ou mais dos seus argumentos de tipo for um tipo aberto. Um tipo aninhado construído é um tipo aberto se e somente se um ou mais dos seus argumentos de tipo ou os argumentos de tipo de seus tipos contiverem um tipo aberto.

Um tipo fechado é um tipo que não é um tipo aberto.

Em tempo de execução, todo o código dentro de uma declaração de tipo genérico é executado no contexto de um tipo construído fechado que foi criado pela aplicação de argumentos de tipo à declaração genérica. Cada parâmetro de tipo dentro do tipo genérico está associado a um tipo de tempo de execução específico. O processamento em tempo de execução de todas as instruções e expressões sempre ocorre com tipos fechados e os tipos abertos ocorrem somente durante o processamento de tempo de compilação.

Cada tipo construído fechado tem seu próprio conjunto de variáveis estáticas, que não são compartilhadas com nenhum outro tipo construído fechado. Como um tipo aberto não existe em tempo de execução, não há variáveis estáticas associadas a um tipo aberto. Dois tipos construídos fechados são o mesmo tipo se forem construídos do

mesmo tipo genérico não associado e seus argumentos de tipo correspondentes forem do mesmo tipo.

## Tipos vinculados e desvinculados

O termo **\*desassociado tipo** se refere a um tipo não genérico ou a um tipo genérico não associado. O termo **\_associado tipo\*** refere-se a um tipo não genérico ou construído.

Um tipo não associado refere-se à entidade declarada por uma declaração de tipo. Um tipo genérico não associado não é, em si, um tipo e não pode ser usado como o tipo de uma variável, um argumento ou um valor de retorno, ou como um tipo base. A única construção na qual um tipo genérico não associado pode ser referenciado é a `typeof` expressão ([o operador `typeof`](#)).

## Atendendo às restrições

Sempre que um tipo construído ou um método genérico é referenciado, os argumentos de tipo fornecidos são verificados em relação às restrições de parâmetro de tipo declaradas no tipo ou método genérico ([restrições de parâmetro de tipo](#)). Para cada `where` cláusula, o argumento de tipo `A` que corresponde ao parâmetro de tipo nomeado é verificado em relação a cada restrição da seguinte maneira:

- Se a restrição for um tipo de classe, um tipo de interface ou um parâmetro de tipo, permita `C` representar essa restrição com os argumentos de tipo fornecidos substituídos por qualquer parâmetro de tipo que apareça na restrição. Para atender à restrição, ele deve ser o caso em que o tipo `A` é conversível para digitar `C` por um dos seguintes:
  - Uma conversão de identidade ([conversão de identidade](#))
  - Uma conversão de referência implícita ([conversões de referência implícita](#))
  - Uma conversão boxing ([conversões Boxing](#)), desde que o tipo a seja um tipo de valor não anulável.
  - Uma referência implícita, Boxing ou conversão de parâmetro de tipo de um parâmetro de tipo `A` para `C`.
- Se a restrição for a restrição de tipo de referência (`class`), o tipo `A` deverá satisfazer um dos seguintes:
  - `A` é um tipo de interface, tipo de classe, tipo delegado ou tipo de matriz.  
Observe que `System.ValueType` e `System.Enum` são tipos de referência que atendem a essa restrição.

- `A` é um parâmetro de tipo que é conhecido como um tipo de referência ([restrições de parâmetro de tipo](#)).
- Se a restrição for a restrição de tipo de valor (`struct`), o tipo `A` deverá satisfazer um dos seguintes:
  - `A` é um tipo de struct ou tipo de enumeração, mas não um tipo anulável.  
Observe que `System.ValueType` e `System.Enum` são tipos de referência que não atendem a essa restrição.
  - `A` é um parâmetro de tipo que tem a restrição de tipo de valor ([restrições de parâmetro de tipo](#)).
- Se a restrição for a restrição de construtor `new()`, o tipo `A` não deve ser `abstract` e deve ter um construtor público sem parâmetros. Isso será atendido se uma das seguintes opções for verdadeira:
  - `A` é um tipo de valor, já que todos os tipos de valor têm um construtor público padrão ([construtores padrão](#)).
  - `A` é um parâmetro de tipo que tem a restrição de Construtor ([restrições de parâmetro de tipo](#)).
  - `A` é um parâmetro de tipo que tem a restrição de tipo de valor ([restrições de parâmetro de tipo](#)).
  - `A` é uma classe que não é `abstract` e contém um construtor declarado explicitamente `public` sem parâmetros.
  - `A` Não é `abstract` e tem um construtor padrão ([construtores padrão](#)).

Ocorrerá um erro de tempo de compilação se uma ou mais das restrições de um parâmetro de tipo não forem atendidas pelos argumentos de tipo fornecidos.

Como os parâmetros de tipo não são herdados, as restrições nunca são herdadas. No exemplo a seguir, é `D` necessário especificar a restrição em seu parâmetro de tipo `T` para que `T` satisfaça a restrição imposta pela classe base `B<T>`. Por outro lado, `E` a classe não precisa especificar uma restrição, pois `List<T>` implementa `IEnumerable` para `Any T`.

C#

```
class B<T> where T: IEnumerable {...}

class D<T>: B<T> where T: IEnumerable {...}

class E<T>: B<List<T>> {...}
```

## Parâmetros de tipo

Um parâmetro de tipo é um identificador que designa um tipo de valor ou tipo de referência ao qual o parâmetro está associado em tempo de execução.

```
antlr  
  
type_parameter  
    : identifier  
    ;
```

Como um parâmetro de tipo pode ser instanciado com muitos argumentos de tipo real diferentes, os parâmetros de tipo têm operações e restrições ligeiramente diferentes dos outros tipos. Elas incluem:

- Um parâmetro de tipo não pode ser usado diretamente para declarar uma classe base ([classe base](#)) ou uma interface ([listas de parâmetros de tipo Variant](#)).
- As regras para pesquisa de membro em parâmetros de tipo dependem das restrições, se houver, aplicadas ao parâmetro de tipo. Eles são detalhados na [pesquisa de membros](#).
- As conversões disponíveis para um parâmetro de tipo dependem das restrições, se houver, aplicadas ao parâmetro de tipo. Eles são detalhados em [conversões implícitas envolvendo parâmetros de tipo](#) e [conversões dinâmicas explícitas](#).
- O literal `null` não pode ser convertido em um tipo fornecido por um parâmetro de tipo, exceto se o parâmetro de tipo for conhecido como um tipo de referência ([conversões implícitas envolvendo parâmetros de tipo](#)). No entanto, uma `default` expressão ([expressões de valor padrão](#)) pode ser usada em vez disso. Além disso, um valor com um tipo fornecido por um parâmetro de tipo pode ser comparado `null` com `==` o uso de `e` `!=` (operadores de igualdade de tipo de [referência](#)), a menos que o parâmetro de tipo tenha a restrição de tipo de valor.
- Uma `new` expressão ([expressões de criação de objeto](#)) só poderá ser usada com um parâmetro de tipo se o parâmetro de tipo for restrito por uma `constructor_constraint` ou a restrição de tipo de valor (restrições de [parâmetro de tipo](#)).
- Um parâmetro de tipo não pode ser usado em nenhum lugar dentro de um atributo.
- Um parâmetro de tipo não pode ser usado em um acesso de membro ([acesso de membro](#)) ou nome de tipo ([namespace e nomes de tipo](#)) para identificar um membro estático ou um tipo aninhado.
- Em código não seguro, um parâmetro de tipo não pode ser usado como um `unmanaged_type` ([tipos de ponteiro](#)).

Como um tipo, os parâmetros de tipo são puramente uma construção de tempo de compilação. Em tempo de execução, cada parâmetro de tipo é associado a um tipo de

tempo de execução que foi especificado fornecendo um argumento de tipo para a declaração de tipo genérico. Assim, o tipo de uma variável declarada com um parâmetro de tipo será, em tempo de execução, ser um tipo construído fechado ([tipos abertos e fechados](#)). A execução em tempo de execução de todas as instruções e expressões que envolvem parâmetros de tipo usa o tipo real que foi fornecido como o argumento de tipo para esse parâmetro.

## Tipos de árvore de expressão

\*As árvores de expressão \_ permitem que as expressões lambda sejam representadas como estruturas de dados em vez de código executável. As árvores de expressão são valores de *tipos de árvore de expressão*\* do formulário

`System.Linq.Expressions.Expression<D>` , onde `D` é qualquer tipo de delegado. Para o restante desta especificação, iremos nos referir a esses tipos usando a abreviação `Expression<D>` .

Se existir uma conversão de uma expressão lambda para um tipo delegado `D` , uma conversão também existirá para o tipo de árvore de expressão `Expression<D>` .

Enquanto a conversão de uma expressão lambda em um tipo delegado gera um delegado que referencia o código executável para a expressão lambda, a conversão para um tipo de árvore de expressão cria uma representação de árvore de expressão da expressão lambda.

As árvores de expressão são representações de dados na memória eficientes de expressões lambda e tornam a estrutura da expressão lambda transparente e explícita.

Assim como um tipo delegado `D` , `Expression<D>` diz-se que os tipos de parâmetro e de retorno são os mesmos do `D` .

O exemplo a seguir representa uma expressão lambda tanto como código executável quanto como uma árvore de expressão. Como existe uma conversão para

`Func<int,int>` , uma conversão também existe para `Expression<Func<int,int>>` :

C#

```
Func<int,int> del = x => x + 1;           // Code  
Expression<Func<int,int>> exp = x => x + 1; // Data
```

Seguindo essas atribuições, o delegado `del` faz referência a um método que retorna `x + 1` , e a árvore de expressão `exp` faz referência a uma estrutura de dados que descreve a expressão `x => x + 1` .

A definição exata do tipo genérico `Expression<D>`, bem como as regras precisas para construir uma árvore de expressão quando uma expressão lambda é convertida em um tipo de árvore de expressão, estão fora do escopo desta especificação.

Duas coisas são importantes para tornar explícitas:

- Nem todas as expressões lambda podem ser convertidas em árvores de expressão. Por exemplo, expressões lambda com corpos de instrução e expressões lambda contendo expressões de atribuição não podem ser representadas. Nesses casos, uma conversão ainda existe, mas falhará em tempo de compilação. Essas exceções são detalhadas em [conversões de função anônimas](#).
- `Expression<D>` oferece um método `Compile` de instância que produz um delegado do tipo `D`:

C#

```
Func<int,int> del2 = exp.Compile();
```

Invocar esse delegado faz com que o código representado pela árvore de expressão seja executado. Assim, dadas as definições acima, `del` e `del2` são equivalentes, e as duas instruções a seguir terão o mesmo efeito:

C#

```
int i1 = del(1);  
  
int i2 = del2(1);
```

Depois de executar esse código `i1`, `i2` os dois terão o valor `2`.

# Variáveis

Artigo • 16/09/2021

As variáveis representam locais de armazenamento. Cada variável tem um tipo que determina quais valores podem ser armazenados na variável. O c# é um idioma de tipo seguro, e o compilador C# garante que os valores armazenados em variáveis sejam sempre do tipo apropriado. O valor de uma variável pode ser alterado por atribuição ou pelo uso dos `++` operadores e `--`.

Uma variável deve ser **definitivamente atribuída** ([atribuição definitiva](#)) antes que seu valor possa ser obtido.

Conforme descrito nas seções a seguir, as variáveis são **\*inicialmente atribuídas \_ ou \_ inicialmente não atribuídas \***. Uma variável atribuída inicialmente tem um valor inicial bem definido e é sempre considerada definitivamente atribuída. Uma variável inicialmente não atribuída não tem valor inicial. Para que uma variável inicialmente não atribuída seja considerada definitivamente atribuída em determinado local, uma atribuição para a variável deve ocorrer em cada caminho de execução possível, levando a esse local.

## Categorias variáveis

O C# define sete categorias de variáveis: variáveis estáticas, variáveis de instância, elementos de matriz, parâmetros de valor, parâmetros de referência, parâmetros de saída e variáveis locais. As seções a seguir descrevem cada uma dessas categorias.

No exemplo

```
C#
class A
{
    public static int x;
    int y;

    void F(int[] v, int a, ref int b, out int c) {
        int i = 1;
        c = a + b++;
    }
}
```

`x` é uma variável estática, `y` é uma variável de instância, `v[0]` é um elemento de matriz, `a` é um parâmetro de valor, é um parâmetro de `b` referência, `c` é um parâmetro de

saída e `i` é uma variável local.

## Variáveis estáticas

Um campo declarado com o `static` modificador é chamado de uma *variável estática*.

Uma variável estática entra em existência antes da execução do construtor estático ([construtores estáticos](#)) para seu tipo recipiente e deixa de existir quando o domínio do aplicativo associado deixar de existir.

O valor inicial de uma variável estática é o valor padrão ([valores padrão](#)) do tipo da variável.

Para fins de verificação de atribuição definitiva, uma variável estática é considerada inicialmente atribuída.

## Variáveis de instância

Um campo declarado sem o `static` modificador é chamado de *variável de instância*.

### Variáveis de instância em classes

Uma variável de instância de uma classe entra em existência quando uma nova instância dessa classe é criada e deixa de existir quando não há nenhuma referência a essa instância e o destruidor da instância (se houver) tiver sido executado.

O valor inicial de uma variável de instância de uma classe é o valor padrão ([valores padrão](#)) do tipo da variável.

Para fins de verificação de atribuição definitiva, uma variável de instância de uma classe é considerada inicialmente atribuída.

### Variáveis de instância em structs

Uma variável de instância de uma struct tem exatamente o mesmo tempo de vida que a variável de struct à qual ela pertence. Em outras palavras, quando uma variável de um tipo struct entra em existência ou deixa de existir, também as variáveis de instância do struct.

O estado de atribuição inicial de uma variável de instância de uma struct é o mesmo da variável struct que a contém. Em outras palavras, quando uma variável de struct é considerada inicialmente atribuída, também são suas variáveis de instância e, quando

uma variável de struct é considerada inicialmente não atribuída, suas variáveis de instância são, da mesma forma, não atribuídas.

## Elementos da matriz

Os elementos de uma matriz entram em existência quando uma instância de matriz é criada e deixa de existir quando não há nenhuma referência a essa instância de matriz.

O valor inicial de cada um dos elementos de uma matriz é o valor padrão ([valores padrão](#)) do tipo dos elementos da matriz.

Para fins de verificação de atribuição definitiva, um elemento de matriz é considerado inicialmente atribuído.

## Parâmetros de valor

Um parâmetro declarado sem um `ref` `out` modificador ou é um *parâmetro de valor*.

Um parâmetro de valor entra em existência na invocação do membro da função (método, Construtor de instância, acessador ou operador) ou função anônima à qual o parâmetro pertence, e é inicializado com o valor do argumento fornecido na invocação. Um parâmetro de valor normalmente deixa de existir no retorno do membro da função ou da função anônima. No entanto, se o parâmetro de valor for capturado por uma função anônima ([expressões de função anônima](#)), seu tempo de vida se estenderá pelo menos até que a árvore de delegação ou expressão criada a partir dessa função anônima esteja qualificada para a coleta de lixo.

Para fins de verificação de atribuição definitiva, um parâmetro `value` é considerado inicialmente atribuído.

## Parâmetros de referência

Um parâmetro declarado com um `ref` modificador é um *parâmetro de referência*.

Um parâmetro de referência não cria um novo local de armazenamento. Em vez disso, um parâmetro de referência representa o mesmo local de armazenamento que a variável fornecida como o argumento no membro da função ou invocação de função anônima. Assim, o valor de um parâmetro de referência é sempre o mesmo que a variável subjacente.

As regras de atribuição definidas a seguir se aplicam aos parâmetros de referência. Observe as diferentes regras para os parâmetros de saída descritos em [parâmetros de](#)

saída.

- Uma variável deve ser definitivamente atribuída ([atribuição definida](#)) antes de poder ser passada como um parâmetro de referência em um membro de função ou uma invocação de delegado.
- Dentro de um membro de função ou função anônima, um parâmetro de referência é considerado inicialmente atribuído.

Dentro de um método de instância ou acessador de instância de um tipo struct, a `this` palavra-chave se comporta exatamente como um parâmetro de referência do tipo struct ([esse acesso](#)).

## Parâmetros de saída

Um parâmetro declarado com um `out` modificador é um **parâmetro de saída**.

Um parâmetro de saída não cria um novo local de armazenamento. Em vez disso, um parâmetro de saída representa o mesmo local de armazenamento que a variável fornecida como o argumento no membro da função ou na invocação de delegado. Assim, o valor de um parâmetro de saída é sempre o mesmo que a variável subjacente.

As regras de atribuição definidas a seguir se aplicam aos parâmetros de saída. Observe as diferentes regras para parâmetros de referência descritos em [parâmetros de referência](#).

- Uma variável não precisa ser definitivamente atribuída antes que possa ser passada como um parâmetro de saída em um membro de função ou invocação de delegado.
- Após a conclusão normal de um membro de função ou de uma invocação de delegado, cada variável passada como um parâmetro de saída é considerada atribuída nesse caminho de execução.
- Dentro de um membro de função ou função anônima, um parâmetro de saída é considerado inicialmente não atribuído.
- Cada parâmetro de saída de um membro de função ou função anônima deve ser definitivamente atribuído ([atribuição definitiva](#)) antes que o membro de função ou função anônima retorne normalmente.

Dentro de um construtor de instância de um tipo struct, a `this` palavra-chave se comporta exatamente como um parâmetro de saída do tipo struct ([esse acesso](#)).

## Variáveis locais

Uma \*variável local \_ é declarada por um `local_variable_declaration` \*, que pode ocorrer em um *bloco*, um *for\_statement*, um *switch\_statement* ou um *using\_statement*; ou por um *foreach\_statement* ou um *specific\_catch\_clause* para um *try\_statement*.

O tempo de vida de uma variável local é a parte da execução do programa durante o qual o armazenamento tem a garantia de ser reservado para ele. Esse tempo de vida estende pelo menos da entrada para o *bloco*, *for\_statement*, *switch\_statement*, *using\_statement*, *foreach\_statement* ou *specific\_catch\_clause* com o qual está associado, até a execução desse *bloco*, *for\_statement*, *switch\_statement*, *using\_statement*, *foreach\_statement* ou *specific\_catch\_clause* termina de qualquer forma. (Inserir um *bloco* anexado ou chamar um método suspende, mas não termina, execução do *bloco* atual, *for\_statement*, *switch\_statement*, *using\_statement*, *foreach\_statement* ou *specific\_catch\_clause*.) Se a variável local for capturada por uma função anônima ([variáveis externas capturadas](#)), seu tempo de vida se estenderá pelo menos até que a árvore de delegação ou de expressão criada a partir da função anônima, juntamente com quaisquer outros objetos que venham a fazer referência à variável capturada, sejam elegíveis para coleta de lixo.

Se o *bloco* pai, *for\_statement*, *switch\_statement*, *using\_statement*, *foreach\_statement* ou *specific\_catch\_clause* for inserido recursivamente, uma nova instância da variável local será criada a cada vez, e seu *local\_variable\_initializer*, se houver, será avaliado a cada vez.

Uma variável local introduzida por um *local\_variable\_declaration* não é inicializada automaticamente e, portanto, não tem valor padrão. Para fins de verificação de atribuição definitiva, uma variável local introduzida por um *local\_variable\_declaration* é considerada inicialmente não atribuída. Um *local\_variable\_declaration* pode incluir um *local\_variable\_initializer*, caso em que a variável é considerada definitivamente atribuída somente após a expressão de inicialização ([instruções de declaração](#)).

Dentro do escopo de uma variável local introduzida por um *local\_variable\_declaration*, é um erro de tempo de compilação para se referir a essa variável local em uma posição textual que precede sua *local\_variable\_declarator*. Se a declaração de variável local for implícita ([declarações de variável local](#)), também será um erro para se referir à variável dentro de seu *local\_variable\_declarator*.

Uma variável local introduzida por um *foreach\_statement* ou um *specific\_catch\_clause* é considerada definitivamente atribuída em seu escopo inteiro.

O tempo de vida real de uma variável local é dependente de implementação. Por exemplo, um compilador pode determinar estaticamente que uma variável local em um bloco seja usada apenas para uma pequena parte desse bloco. Usando essa análise, o

compilador pode gerar código que resulta no armazenamento da variável que tem um tempo de vida menor do que o bloco que a contém.

O armazenamento referido por uma variável de referência local é recuperado independentemente do tempo de vida dessa variável de referência local ([Gerenciamento de memória automático](#)).

## Valores padrão

As seguintes categorias de variáveis são inicializadas automaticamente para seus valores padrão:

- Variáveis estáticas.
- Variáveis de instância de instâncias de classe.
- Elementos da matriz.

O valor padrão de uma variável depende do tipo da variável e é determinado da seguinte maneira:

- Para uma variável de um *value\_type*, o valor padrão é o mesmo que o valor calculado pelo construtor padrão do *Value\_type*([construtores padrão](#)).
- Para uma variável de um *reference\_type*, o valor padrão é `null` .

A inicialização para valores padrão normalmente é feita por ter o Gerenciador de memória ou o coletor de lixo inicializar a memória para todos os bits-zero antes que ele seja alocado para uso. Por esse motivo, é conveniente usar todos os bits-zero para representar a referência nula.

## Atribuição definida

Em um determinado local no código executável de um membro de função, uma variável é considerada **definitivamente atribuída** se o compilador puder provar, por uma análise de fluxo estático específica ([regras exatas para determinar a atribuição definitiva](#)), se a variável tiver sido inicializada automaticamente ou se tiver sido o destino de pelo menos uma atribuição. Indicado informalmente, as regras de atribuição definitiva são:

- Uma variável atribuída inicialmente ([variáveis inicialmente atribuídas](#)) é sempre considerada definitivamente atribuída.
- Uma variável inicialmente não atribuída ([variáveis inicialmente não atribuídas](#)) será considerada definitivamente atribuída em um determinado local se todos os caminhos de execução possíveis que levam a esse local contiverem pelo menos um dos seguintes itens:

- Uma atribuição simples ([atribuição simples](#)) na qual a variável é o operando esquerdo.
- Uma expressão de invocação ([expressões de invocação](#)) ou expressão de criação de objeto ([expressões de criação de objeto](#)) que passa a variável como um parâmetro de saída.
- Para uma variável local, uma declaração de variável local ([declarações de variável local](#)) que inclui um inicializador de variável.

A especificação formal subjacente às regras informais acima é descrita em [variáveis atribuídas inicialmente, variáveis inicialmente não atribuídas regras precisas para determinar a atribuição definitiva](#).

Os Estados de atribuição definitivos de variáveis de instância de uma variável *struct\_type* são acompanhados individualmente, bem como coletivamente. Além das regras acima, as regras a seguir se aplicam a *struct\_type* variáveis e suas variáveis de instância:

- Uma variável de instância é considerada definitivamente atribuída se a variável que a contém *struct\_type* é considerada definitivamente atribuída.
- Uma variável *struct\_type* será considerada definitivamente atribuída se cada uma de suas variáveis de instância for considerada definitivamente atribuída.

A atribuição definitiva é um requisito nos seguintes contextos:

- Uma variável deve ser atribuída definitivamente em cada local em que seu valor é obtido. Isso garante que os valores indefinidos nunca ocorram. A ocorrência de uma variável em uma expressão é considerada para obter o valor da variável, exceto quando
  - a variável é o operando esquerdo de uma atribuição simples,
  - a variável é passada como um parâmetro de saída ou
  - a variável é uma *struct\_type* variável e ocorre como o operando esquerdo de um acesso de membro.
- Uma variável deve ser definitivamente atribuída em cada local em que é passada como um parâmetro de referência. Isso garante que o membro da função que está sendo invocado pode considerar o parâmetro de referência inicialmente atribuído.
- Todos os parâmetros de saída de um membro de função devem ser definitivamente atribuídos em cada local em que o membro da função retorna (por meio de uma `return` instrução ou por meio da execução que chega ao final do corpo do membro da função). Isso garante que os membros da função não retornem valores indefinidos nos parâmetros de saída, permitindo que o compilador considere uma invocação de membro de função que usa uma variável como um parâmetro de saída equivalente a uma atribuição para a variável.

- A `this` variável de um construtor de instância de `struct_type` deve ser definitivamente atribuída em cada local em que o construtor de instância retorna.

## Variáveis inicialmente atribuídas

As seguintes categorias de variáveis são classificadas como atribuídas inicialmente:

- Variáveis estáticas.
- Variáveis de instância de instâncias de classe.
- Variáveis de instância de variáveis de struct inicialmente atribuídas.
- Elementos da matriz.
- Parâmetros de valor.
- Parâmetros de referência.
- Variáveis declaradas em uma `catch` cláusula ou em uma `foreach` instrução.

## Variáveis inicialmente não atribuídas

As seguintes categorias de variáveis são classificadas como inicialmente não atribuídas:

- Variáveis de instância de variáveis struct inicialmente não atribuídas.
- Parâmetros de saída, incluindo a `this` variável de construtores de instância de struct.
- Variáveis locais, exceto aquelas declaradas em uma `catch` cláusula ou uma `foreach` instrução.

## Regras precisas para determinar a atribuição definitiva

Para determinar se cada variável usada é definitivamente atribuída, o compilador deve usar um processo que seja equivalente ao descrito nesta seção.

O compilador processa o corpo de cada membro de função que tem uma ou mais variáveis não atribuídas inicialmente. Para cada variável `v` inicialmente não atribuída, o compilador determina um **\*estado de atribuição definitivo \_ para \_v \*** em cada um dos seguintes pontos no membro da função:

- No início de cada instrução
- No ponto de extremidade ([pontos de extremidade e acessibilidade](#)) de cada instrução
- Em cada arco que transfere o controle para outra instrução ou para o ponto final de uma instrução
- No início de cada expressão

- No final de cada expressão

O estado de atribuição definitivo de  $v$  pode ser:

- Definitivamente atribuído. Isso indica que em todos os fluxos de controle possíveis para esse ponto,  $v$  recebeu um valor.
- Não é definitivamente atribuído. Para o estado de uma variável no final de uma expressão do tipo `bool`, o estado de uma variável que não está definitivamente atribuída pode (mas não necessariamente) se enquadrar em um dos seguintes subcaminhos:
  - Atribuído definitivamente após a expressão `true`. Esse estado indica que  $v$  é definitivamente atribuído se a expressão booleana for avaliada como `true`, mas não será necessariamente atribuído se a expressão booleana for avaliada como `false`.
  - Definitivamente atribuída após expressão `false`. Esse estado indica que  $v$  é definitivamente atribuído se a expressão booleana for avaliada como `false`, mas não será necessariamente atribuída se a expressão booleana for avaliada como `true`.

As regras a seguir regem como o estado de uma variável  $v$  é determinado em cada local.

## Regras gerais para instruções

- $v$  não é definitivamente atribuído no início de um corpo de membro de função.
- o  $v$  é definitivamente atribuído no início de qualquer instrução inacessível.
- O estado de atribuição definitivo de  $v$  no início de qualquer outra instrução é determinado verificando o estado de atribuição definido de  $v$  em todas as transferências de fluxo de controle direcionadas ao início dessa instrução. Se (e somente se)  $v$  for definitivamente atribuído a todas as transferências de fluxo de controle,  $v$  será definitivamente atribuído no início da instrução. O conjunto de possíveis transferências de fluxo de controle é determinado da mesma forma que para verificar a acessibilidade da instrução ([pontos de extremidade e acessibilidade](#)).
- O estado de atribuição definitivo de  $v$  no ponto de extremidade de uma instrução `Block`, `checked`, `unchecked`, `if`, `while`, `do`, `for`, `foreach`, `lock`, `using` ou `switch` é determinado verificando o estado de atribuição definido de  $v$  em todas as transferências de fluxo de controle direcionadas ao ponto de extremidade dessa instrução. Se  $v$  for definitivamente atribuído a todas essas transferências de fluxo de controle, então  $v$  será definitivamente atribuído no ponto de extremidade da instrução. ,  $v$  não é definitivamente atribuído no ponto de extremidade da

instrução. O conjunto de possíveis transferências de fluxo de controle é determinado da mesma forma que para verificar a acessibilidade da instrução ([pontos de extremidade e acessibilidade](#)).

## Instruções Block, instruções marcadas e desmarcadas

O estado de atribuição definitivo de  $v$  na transferência de controle para a primeira instrução da lista de instruções no bloco (ou até o ponto final do bloco, se a lista de instruções estiver vazia) é igual à instrução de atribuição definitiva de  $v$  antes da `checked` instrução ou do bloco `unchecked`.

## Instruções de expressão

Para uma instrução de expressão  $stmt$  que consiste na expressão *Expression*:

- $v$  tem o mesmo estado de atribuição definitivo no início de  $expr$  como no início de  $stmt$ .
- Se *for* definitivamente atribuído ao final de  $expr$ , ele será definitivamente atribuído no ponto final de  $stmt$ ; Ele não é definitivamente atribuído no ponto de extremidade de  $stmt$ .

## Instruções de declaração

- Se  $stmt$  for uma instrução de declaração sem inicializadores,  $v$  terá o mesmo estado de atribuição definitivo no ponto de extremidade de  $stmt$  como no início de  $stmt$ .
- Se  $stmt$  for uma instrução de declaração com inicializadores, o estado de atribuição definitivo para  $v$  será determinado como se  $stmt$  fosse uma lista de instruções, com uma instrução de atribuição para cada declaração com um inicializador (na ordem de declaração).

## Instruções If

Para uma `if` instrução  $stmt$  do formulário:

C#

```
if ( expr ) then_stmt else else_stmt
```

- $v$  tem o mesmo estado de atribuição definitivo no início de  $expr$  como no início de  $stmt$ .

- Se  $v$  for definitivamente atribuído ao final de  $expr$ , ele será definitivamente atribuído na transferência de fluxo de controle para  $then\_stmt$  e para  $else\_stmt$  ou para o ponto de extremidade de  $stmt$  se não houver nenhuma cláusula  $else$ .
- Se  $v$  tiver o estado "definitivamente atribuído após expressão verdadeira" no final de  $expr$ , ele será definitivamente atribuído na transferência de fluxo de controle para  $then\_stmt$  e não definitivamente atribuído na transferência de fluxo de controle para  $else\_stmt$  ou para o ponto de extremidade de  $stmt$  se não houver nenhuma cláusula  $else$ .
- Se  $v$  tiver o estado "definitivamente atribuído após expressão falsa" no final de  $expr$ , ele será definitivamente atribuído na transferência de fluxo de controle para  $else\_stmt$  e não definitivamente atribuído na transferência de fluxo de controle para  $then\_stmt$ . Ele é definitivamente atribuído no ponto de extremidade de  $stmt$  se e somente se ele for definitivamente atribuído no ponto de extremidade de  $then\_stmt$ .
- Caso contrário,  $v$  é considerado não definitivamente atribuído na transferência de fluxo de controle para o  $then\_stmt$  ou  $else\_stmt$ , ou para o ponto de extremidade de  $stmt$  se não houver nenhuma cláusula  $else$ .

## Instruções switch

Em uma `switch` instrução  $stmt$  com uma  $expr$  de expressão de controle:

- O estado de atribuição definitivo de  $v$  no início de  $expr$  é o mesmo que o estado de  $v$  no início de  $stmt$ .
- O estado de atribuição definitivo de  $v$  na transferência de fluxo de controle para uma lista de instrução de bloqueio de switch acessível é igual ao estado de atribuição definido de  $v$  no final de  $expr$ .

## Instruções while

Para uma `while` instrução  $stmt$  do formulário:

```
C#
```

```
while ( expr ) while_body
```

- $v$  tem o mesmo estado de atribuição definitivo no início de  $expr$  como no início de  $stmt$ .
- Se  $v$  for definitivamente atribuído ao final de  $expr$ , ele será definitivamente atribuído na transferência de fluxo de controle para  $while\_body$  e para o ponto final de  $stmt$ .

- Se  $v$  tiver o estado "definitivamente atribuído após expressão verdadeira" no final de  $expr$ , ele será definitivamente atribuído na transferência de fluxo de controle para  $while\_body$ , mas não definitivamente atribuído no ponto de extremidade de  $stmt$ .
- Se  $v$  tiver o estado "definitivamente atribuído após expressão falsa" no final de  $expr$ , ele será definitivamente atribuído na transferência de fluxo de controle para o ponto final de  $stmt$ , mas não definitivamente atribuído na transferência de fluxo de controle para  $while\_body$ .

## Instruções do

Para uma `do` instrução  $stmt$  do formulário:

C#

```
do do_body while ( expr ) ;
```

- $v$  tem o mesmo estado de atribuição definitivo na transferência de fluxo de controle do início de  $stmt$  para  $do\_body$  como no início de  $stmt$ .
- $v$  tem o mesmo estado de atribuição definitivo no início de  $expr$  como no ponto de extremidade de  $do\_body$ .
- Se  $v$  for definitivamente atribuído ao final de  $expr$ , ele será definitivamente atribuído na transferência de fluxo de controle para o ponto final de  $stmt$ .
- Se  $v$  tiver o estado "definitivamente atribuído após expressão falsa" no final de  $expr$ , ele será definitivamente atribuído na transferência de fluxo de controle para o ponto final de  $stmt$ .

## Instruções for

Verificação de atribuição definitiva para uma `for` instrução do formulário:

C#

```
for ( for_initializer ; for_condition ; for_iterator ) embedded_statement
```

é feito como se a instrução fosse gravada:

C#

```
{
    for_initializer ;
    while ( for_condition ) {
```

```
    embedded_statement ;
    for_iterator ;
}
}
```

Se o *for\_condition* for omitido da `for` instrução, a avaliação da atribuição definitiva continuará como se *for\_condition* substituídos `true` por na expansão acima.

## Instruções Break, continue e goto

O estado de atribuição definitivo de *v* na transferência de fluxo de controle causada por uma `break` `continue` instrução, ou `goto` é igual ao estado de atribuição definido de *v* no início da instrução.

## Instruções throw

Para uma instrução *stmt* do formulário

```
C#
throw expr ;
```

O estado de atribuição definitivo de *v* no início de *expr* é o mesmo que o estado de atribuição definido de *v* no início de *stmt*.

## Instruções de retorno

Para uma instrução *stmt* do formulário

```
C#
return expr ;
```

- O estado de atribuição definitivo de *v* no início de *expr* é o mesmo que o estado de atribuição definido de *v* no início de *stmt*.
- Se *v* for um parâmetro de saída, ele deverá ser definitivamente atribuído a:
  - Depois de *expr*
  - ou no final do `finally` bloco de um `try - finally` ou `try - catch - finally` que inclui a `return` instrução.

Para uma instrução *stmt* do formulário:

C#

```
return ;
```

- Se *v* for um parâmetro de saída, ele deverá ser definitivamente atribuído a:
  - antes de *stmt*
  - ou no final do `finally` bloco de um `try - finally` ou `try - catch - finally` que inclui a `return` instrução.

## Instruções Try-Catch

Para uma instrução *stmt* do formulário:

C#

```
try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
```

- O estado de atribuição definitivo de *v* no início de *try\_block* é o mesmo que o estado de atribuição definido de *v* no início de *stmt*.
- O estado de atribuição definitivo de *v* no início de *catch\_block\_i* (para qualquer *i*) é igual ao estado de atribuição definitivo de *v* no início de *stmt*.
- O estado de atribuição definitivo de *v* no ponto de extremidade de *stmt* é definitivamente atribuído se (e somente se) *v* for definitivamente atribuído no ponto de extremidade de *try\_block* e a cada *catch\_block\_i* (para cada *i* de 1 a *n*).

## Instruções try – finally

Para uma `try` instrução *stmt* do formulário:

C#

```
try try_block finally finally_block
```

- O estado de atribuição definitivo de *v* no início de *try\_block* é o mesmo que o estado de atribuição definido de *v* no início de *stmt*.
- O estado de atribuição definitivo de *v* no início de *finally\_block* é o mesmo que o estado de atribuição definido de *v* no início de *stmt*.
- O estado de atribuição definitivo de *v* no ponto de extremidade de *stmt* é definitivamente atribuído se (e somente se) pelo menos uma das seguintes opções

for verdadeira:

- o *v* é definitivamente atribuído no ponto de extremidade de *try\_block*
- o *v* é definitivamente atribuído no ponto de extremidade de *finally\_block*

Se for feita uma transferência de fluxo de controle (por exemplo, uma `goto` instrução) que começa dentro de *try\_block* e termina fora do *try\_block*, então *v* também será considerado definitivamente atribuído na transferência do fluxo de controle se *v* for definitivamente atribuído no ponto de extremidade de *finally\_block*. (Isso não é apenas se — se *v* for definitivamente atribuído por outro motivo nessa transferência de fluxo de controle, ele ainda será considerado definitivamente atribuído.)

## Instruções try – catch-finally

Análise de atribuição definitiva para uma `try - catch - finally` instrução do formulário:

C#

```
try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
finally *finally_block*
```

é feito como se a instrução fosse uma `try - finally` instrução delimitando uma `try - catch` instrução:

C#

```
try {
    try try_block
    catch(...) catch_block_1
    ...
    catch(...) catch_block_n
}
finally finally_block
```

O exemplo a seguir demonstra como os diferentes blocos de uma `try` instrução ([a instrução try](#)) afetam a atribuição definitiva.

C#

```
class A
{
    static void F() {
```

```

    int i, j;
    try {
        goto LABEL;
        // neither i nor j definitely assigned
        i = 1;
        // i definitely assigned
    }

    catch {
        // neither i nor j definitely assigned
        i = 3;
        // i definitely assigned
    }

    finally {
        // neither i nor j definitely assigned
        j = 5;
        // j definitely assigned
    }
    // i and j definitely assigned
LABEL:;
    // j definitely assigned
}

}

```

## Instruções Foreach

Para uma `foreach` instrução *stmt* do formulário:

C#

```
foreach ( type identifier in expr ) embedded_statement
```

- O estado de atribuição definitivo de *v* no início de *expr* é o mesmo que o estado de *v* no início de *stmt*.
- O estado de atribuição definitivo de *v* na transferência de fluxo de controle para *embedded\_statement* ou para o ponto final de *stmt* é o mesmo que o estado de *v* no final de *expr*.

## Como usar instruções

Para uma `using` instrução *stmt* do formulário:

C#

```
using ( resource_acquisition ) embedded_statement
```

- O estado de atribuição definitivo de  $v$  no início de *resource\_acquisition* é o mesmo que o estado de  $v$  no início de *stmt*.
- O estado de atribuição definitivo de  $v$  na transferência de fluxo de controle para *embedded\_statement* é o mesmo que o estado de  $v$  no final de *resource\_acquisition*.

## Instruções Lock

Para uma `lock` instrução *stmt* do formulário:

C#

```
lock ( expr ) embedded_statement
```

- O estado de atribuição definitivo de  $v$  no início de *expr* é o mesmo que o estado de  $v$  no início de *stmt*.
- O estado de atribuição definitivo de  $v$  na transferência de fluxo de controle para *embedded\_statement* é o mesmo que o estado de  $v$  no final de *expr*.

## Instruções yield

Para uma `yield return` instrução *stmt* do formulário:

C#

```
yield return expr ;
```

- O estado de atribuição definitivo de  $v$  no início de *expr* é o mesmo que o estado de  $v$  no início de *stmt*.
- O estado de atribuição definitivo de  $v$  no final de *stmt* é igual ao estado de  $v$  no final de *expr*.
- Uma `yield break` instrução não tem nenhum efeito sobre o estado de atribuição definitivo.

## Regras gerais para expressões simples

A regra a seguir se aplica a esses tipos de expressões: literais ([literais](#)), nomes simples ([nomes simples](#)), expressões de acesso de membro ([acesso de membro](#)), expressões de acesso base não indexadas ([acesso de base](#)), `typeof` expressões ([o operador typeof](#)),

expressões de valor padrão (expressões de [valor padrão](#)) e [nameof](#) expressões ([expressões nameof](#)).

- O estado de atribuição definitivo de *v* no final de tal expressão é igual ao estado de atribuição definido de *v* no início da expressão.

## Regras gerais para expressões com expressões inseridas

As regras a seguir se aplicam a esses tipos de expressões: expressões entre parênteses ([expressões entre parênteses](#)), expressões de acesso de elemento ([acesso de elemento](#)), expressões de acesso de base com indexação ([acesso de base](#)), expressões de incremento e decréscimo ([incremento de sufixo e diminuição de operadores](#), [incremento de prefixo e operadores de decréscimo](#)), [expressões de conversão](#) unário +, -, ~, \* expressões, binary +, -, \*, /, % ..... <<, >>, <, <=, >, >=, ==, !=, is, as, &, |, ^ expressões ([operadores aritméticos](#), [operadores de deslocamento](#), [operadores relacionais e de teste de tipo](#), [operadores lógicos](#)), expressões de atribuição de composição ([atribuição composta](#)) checked e unchecked expressões ([os operadores marcados e não verificados](#)), além de expressões de criação de matriz e delegado

Cada uma dessas expressões tem uma ou mais subexpressões que são avaliadas incondicionalmente em uma ordem fixa. Por exemplo, o % operador binário avalia o lado esquerdo do operador e, em seguida, o lado direito. Uma operação de indexação avalia a expressão indexada e, em seguida, avalia cada uma das expressões de índice, na ordem da esquerda para a direita. Para uma *expr* de expressão, que tem subexpressões *E1, E2,..., en*, avaliadas nessa ordem:

- O estado de atribuição definitivo de *v* no início do *E1* é o mesmo que o estado de atribuição definido no início de *expr*.
- O estado de atribuição definitivo de *v* no início de *Ei* (*i* maior que um) é o mesmo que o estado de atribuição definido no final da subexpressão anterior.
- O estado de atribuição definitivo de *v* no final de *expr* é igual ao estado de atribuição definido no final de *en*

## Expressões de invocação e expressões de criação de objeto

Para uma *expr* de expressão de invocação do formulário:

C#

```
primary_expression ( arg1 , arg2 , ... , argN )
```

ou uma expressão de criação de objeto do formulário:

C#

```
new type ( arg1 , arg2 , ... , argN )
```

- Para uma expressão de invocação, o estado de atribuição definitivo de *v* antes de *primary\_expression* é o mesmo que o estado de *v* antes de *expr*.
- Para uma expressão de invocação, o estado de atribuição definitivo de *v* antes de *arg1* é igual ao estado de *v* após *primary\_expression*.
- Para uma expressão de criação de objeto, o estado de atribuição definitivo de *v* antes de *arg1* é o mesmo que o estado de *v* antes de *expr*.
- Para cada argumento *Argi*, o estado de atribuição definitivo de *v* após *Argi* é determinado pelas regras de expressão normal, ignorando quaisquer `ref` `out` modificadores ou.
- Para cada argumento *Argi* para qualquer *i* maior que um, o estado de atribuição definitivo de *v* antes de *Argi* é o mesmo que o estado de *v* após o ARG anterior.
- Se a variável *v* for passada como um `out` argumento (ou seja, um argumento do formulário `out v`) em qualquer um dos argumentos, o estado de *v* After *expr* será atribuído definitivamente., o estado de *v* After *expr* é o mesmo que o estado de *v* após *argN*.
- Para inicializadores de matriz([expressões de criação de matriz](#)), inicializadores de objeto ([inicializadores de objeto](#)), inicializadores de coleção ([inicializadores de coleção](#)) e inicializadores de objeto anônimos ([expressões de criação de objeto anônimo](#)), o estado de atribuição definitivo é determinado pela expansão de que essas construções são definidas em termos de.

## Expressões de atribuição simples

Para uma *expr* de expressão do formulário `w = expr_rhs` :

- O estado de atribuição definitivo de *v* antes de *expr\_rhs* é o mesmo que o estado de atribuição definido de *v* antes de *expr*.
- O estado de atribuição definitivo de *v* após *expr* é determinado por:
  - Se *w* for a mesma variável que *v*, o estado de atribuição definitivo de *v* após *expr* será atribuído definitivamente.
  - Caso contrário, se a atribuição ocorrer dentro do construtor de instância de um tipo struct, se *w* for um acesso de propriedade que designa uma propriedade automaticamente implementada *p* na instância que está sendo construída e *v* for o campo de apoio oculto de *p*, o estado de atribuição definido de *v* após *expr* será atribuído definitivamente.

- o Caso contrário, o estado de atribuição definitivo de *v* após *expr* será o mesmo que o estado de atribuição definido de *v* após *expr\_rhs*.

## Expressões && (condicional e)

Para uma *expr* de expressão do formulário `expr_first && expr_second` :

- O estado de atribuição definitivo de *v* antes de *expr\_first* é o mesmo que o estado de atribuição definido de *v* antes de *expr*.
- O estado de atribuição definitivo de *v* antes de *expr\_second* será definitivamente atribuído se o estado de *v* após *expr\_first* for definitivamente atribuído ou "definitivamente atribuído após a expressão verdadeira". Caso contrário, ele não é definitivamente atribuído.
- O estado de atribuição definitivo de *v* após *expr* é determinado por:
  - o Se *expr\_first* for uma expressão constante com o valor `false`, o estado de atribuição definitivo de *v* após *expr* será o mesmo que o estado de atribuição definido de *v* após *expr\_first*.
  - o Caso contrário, se o estado de *v* após *expr\_first* for definitivamente atribuído, o estado de *v* após *expr* será atribuído definitivamente.
  - o Caso contrário, se o estado de *v* após *expr\_second* for definitivamente atribuído, e o estado de *v* After *expr\_first* for "definitivamente atribuído após a expressão falsa", o estado de *v* após *expr* será atribuído definitivamente.
  - o Caso contrário, se o estado de *v* após *expr\_second* for definitivamente atribuído ou "definitivamente atribuído após a expressão verdadeira", o estado de *v* após *expr* será "definitivamente atribuído após a expressão verdadeira".
  - o Caso contrário, se o estado de *v* após *expr\_first* for "definitivamente atribuído após a expressão falsa", e o estado de *v* após *expr\_second* for "definitivamente atribuído após a expressão falsa", o estado de *v* após *expr* será "definitivamente atribuído após a expressão falsa".
  - o Caso contrário, o estado de *v* após *expr* não será atribuído definitivamente.

No exemplo

```
C#
class A
{
    static void F(int x, int y) {
        int i;
        if (x >= 0 && (i = y) >= 0) {
            // i definitely assigned
        }
        else {
            // i not definitely assigned
        }
    }
}
```

```

        }
        // i not definitely assigned
    }
}

```

a variável `i` é considerada definitivamente atribuída em uma das instruções inseridas de uma `if` instrução, mas não no outro. Na `if` instrução no método `F`, a variável `i` é definitivamente atribuída na primeira instrução inserida porque a execução da expressão `(i = y)` sempre precede a execução dessa instrução inserida. Por outro lado, a variável `i` não é definitivamente atribuída na segunda instrução incorporada, pois `x >= 0` pode ter testado false, resultando na não atribuição da variável `i`.

## || expressões (condicionais ou)

Para uma `expr` de expressão do formulário `expr_first || expr_second` :

- O estado de atribuição definitivo de `v` antes de `expr_first` é o mesmo que o estado de atribuição definido de `v` antes de `expr`.
- O estado de atribuição definitivo de `v` antes de `expr_second` será definitivamente atribuído se o estado de `v` após `expr_first` for definitivamente atribuído ou "definitivamente atribuído após a expressão falsa". Caso contrário, ele não é definitivamente atribuído.
- A instrução de atribuição definitiva de `v` após `expr` é determinada por:
  - Se `expr_first` for uma expressão constante com o valor `true`, o estado de atribuição definitivo de `v` após `expr` será o mesmo que o estado de atribuição definido de `v` após `expr_first`.
  - Caso contrário, se o estado de `v` após `expr_first` for definitivamente atribuído, o estado de `v` após `expr` será atribuído definitivamente.
  - Caso contrário, se o estado de `v` após `expr_second` for definitivamente atribuído, e o estado de `v` After `expr_first` for "definitivamente atribuído após a expressão verdadeira", o estado de `v` após `expr` será atribuído definitivamente.
  - Caso contrário, se o estado de `v` após `expr_second` for definitivamente atribuído ou "definitivamente atribuído após a expressão falsa", o estado de `v` após `expr` será "definitivamente atribuído após a expressão falsa".
  - Caso contrário, se o estado de `v` após `expr_first` for "definitivamente atribuída após a expressão verdadeira", e o estado de `v` após `expr_second` for "definitivamente atribuído após a expressão verdadeira", o estado de `v` após `expr` será "definitivamente atribuído após a expressão verdadeira".
  - Caso contrário, o estado de `v` após `expr` não será atribuído definitivamente.

No exemplo

C#

```
class A
{
    static void G(int x, int y) {
        int i;
        if (x >= 0 || (i = y) >= 0) {
            // i not definitely assigned
        }
        else {
            // i definitely assigned
        }
        // i not definitely assigned
    }
}
```

a variável `i` é considerada definitivamente atribuída em uma das instruções inseridas de uma `if` instrução, mas não no outro. Na `if` instrução no método `G`, a variável `i` é definitivamente atribuída na segunda instrução inserida porque a execução da expressão `(i = y)` sempre precede a execução dessa instrução inserida. Por outro lado, a variável `i` não é definitivamente atribuída na primeira instrução inserida, pois `x >= 0` pode ter testado verdadeiro, resultando na não atribuição da variável `i`.

## ! (negação lógica) expressões

Para uma `expr` de expressão do formulário `! expr_operand`:

- O estado de atribuição definitivo de `v` antes de `expr_operand` é o mesmo que o estado de atribuição definido de `v` antes de `expr`.
- O estado de atribuição definitivo de `v` após `expr` é determinado por:
  - Se o estado de `v` após \* `expr_operand` \* for definitivamente atribuído, o estado de `v` após `expr` será atribuído definitivamente.
  - Se o estado de `v` após \* `expr_operand` \* não for definitivamente atribuído, o estado de `v` após `expr` não será atribuído definitivamente.
  - Se o estado de `v` após \* `expr_operand` \* for "definitivamente atribuído após a expressão falsa", o estado de `v` após `expr` será "definitivamente atribuído após a expressão verdadeira".
  - Se o estado de `v` após \* `expr_operand` \* for "definitivamente atribuído após a expressão verdadeira", o estado de `v` após `expr` será "definitivamente atribuído após a expressão falsa".

## ?? (União nula) expressões

Para uma *expr* de expressão do formulário `expr_first ?? expr_second` :

- O estado de atribuição definitivo de *v* antes de *expr\_first* é o mesmo que o estado de atribuição definido de *v* antes de *expr*.
- O estado de atribuição definitivo de *v* antes de *expr\_second* é o mesmo que o estado de atribuição definido de *v* após *expr\_first*.
- A instrução de atribuição definitiva de *v* após *expr* é determinada por:
  - Se *expr\_first* for uma expressão constante ([expressões constantes](#)) com valor `NULL`, o estado de *v* após *expr* será o mesmo que o estado de *v* após *expr\_second*.
- Caso contrário, o estado de *v* After *expr* será o mesmo que o estado de atribuição definido de *v* após *expr\_first*.

## **expressões?: (condicionais)**

Para uma *expr* de expressão do formulário `expr_cond ? expr_true : expr_false` :

- O estado de atribuição definitivo de *v* antes de *expr\_cond* é o mesmo que o estado de *v* antes de *expr*.
- O estado de atribuição definitivo de *v* antes de *expr\_true* é definitivamente atribuído se e somente se uma das seguintes isenções:
  - *expr\_cond* é uma expressão constante com o valor `false`
  - o estado de *v* após *expr\_cond* é definitivamente atribuído ou "definitivamente atribuído após a expressão verdadeira".
- O estado de atribuição definitivo de *v* antes de *expr\_false* é definitivamente atribuído se e somente se uma das seguintes isenções:
  - *expr\_cond* é uma expressão constante com o valor `true`
  - o estado de *v* após *expr\_cond* é definitivamente atribuído ou "definitivamente atribuído após expressão falsa".
- O estado de atribuição definitivo de *v* após *expr* é determinado por:
  - Se *expr\_cond* for uma expressão constante ([expressões constantes](#)) com valor `true` , o estado de *v* após *expr* será o mesmo que o estado de *v* após *expr\_true*.
  - Caso contrário, se *expr\_cond* for uma expressão constante ([expressões constantes](#)) com valor `false` , o estado de *v* após *expr* será o mesmo que o estado de *v* após *expr\_false*.
  - Caso contrário, se o estado de *v* após *expr\_true* for definitivamente atribuído e o estado de *v* após *expr\_false* for definitivamente atribuído, o estado de *v* após *expr* será atribuído definitivamente.
  - Caso contrário, o estado de *v* após *expr* não será atribuído definitivamente.

## Funções anônimas

Para um *lambda\_expression* ou *expr* de *anonymous\_method\_expression* com um corpo (*bloco* ou *expressão*) *corpo*:

- O estado de atribuição definitivo de uma variável externa *v* para o *corpo* é o mesmo que o estado de *v* antes de *expr*. Ou seja, o estado de atribuição definitivo de variáveis externas é herdado do contexto da função anônima.
- O estado de atribuição definitivo de uma variável externa *v* depois de *expr* é o mesmo que o estado de *v* antes de *expr*.

O exemplo

```
C#  
  
delegate bool Filter(int i);  
  
void F() {  
    int max;  
  
    // Error, max is not definitely assigned  
    Filter f = (int n) => n < max;  
  
    max = 5;  
    DoWork(f);  
}
```

gera um erro de tempo de compilação porque `max` não é definitivamente atribuído onde a função anônima é declarada. O exemplo

```
C#  
  
delegate void D();  
  
void F() {  
    int n;  
    D d = () => { n = 1; };  
  
    d();  
  
    // Error, n is not definitely assigned  
    Console.WriteLine(n);  
}
```

também gera um erro de tempo de compilação porque a atribuição para `n` na função anônima não tem nenhum efeito sobre o estado de atribuição definitivo de `n` fora da função anônima.

# Referências de variáveis

Uma *variable\_reference* é uma *expressão* que é classificada como uma variável. Um *variable\_reference* denota um local de armazenamento que pode ser acessado para buscar o valor atual e para armazenar um novo valor.

```
antlr  
  
variable_reference  
: expression  
;
```

Em C e C++, um *variable\_reference* é conhecido como um *lvalue*.

## Atomicidade de referências de variáveis

Leituras e gravações dos seguintes tipos de dados são Atomic: `bool` , `char` , `byte` ...  
`sbyte` `short` `ushort` , `uint` , `int` , `float` e tipos de referência. Além disso, leituras e gravações de tipos de enumeração com um tipo subjacente na lista anterior também são atômicas. As leituras e gravações de outros tipos, incluindo `long` , `ulong` `double` e `decimal` , bem como tipos definidos pelo usuário, não têm garantia de serem atômicas. Além das funções de biblioteca projetadas para essa finalidade, não há nenhuma garantia de leitura-modificação-gravação atômica, como no caso de incremento ou decréscimo.

# Conversões

Artigo • 16/09/2021

Uma conversão <sup>\*</sup> permite que uma expressão seja tratada como sendo de um tipo específico. Uma conversão pode fazer com que uma expressão de um determinado tipo seja tratada como tendo um tipo diferente, ou pode causar uma expressão sem um tipo para obter um tipo. As conversões podem ser *implícitas* ou *Explicit* <sup>\*</sup>, e isso determina se uma conversão explícita é necessária. Por exemplo, a conversão de tipo `int` para tipo `long` é implícita, portanto, as expressões do tipo `int` podem ser tratadas implicitamente como tipo `long`. A conversão oposta, de tipo `long` para tipo `int`, é explícita e, portanto, uma conversão explícita é necessária.

C#

```
int a = 123;
long b = a;           // implicit conversion from int to long
int c = (int) b;     // explicit conversion from long to int
```

Algumas conversões são definidas pelo idioma. Os programas também podem definir suas próprias conversões ([conversões definidas pelo usuário](#)).

## Conversões implícitas

As conversões a seguir são classificadas como conversões implícitas:

- Conversões de identidade
- Conversões numéricas implícitas
- Conversões implícitas de enumeração
- Conversões de cadeia de caracteres interpoladas implícitas
- Conversões anuláveis implícitas
- Conversões literais nulas
- Conversões de referência implícitas
- Conversões de Boxing
- Conversões dinâmicas implícitas
- Conversões de expressão de constante implícita
- Conversões implícitas definidas pelo usuário
- Conversões de função anônima
- Conversões de grupo de métodos

Conversões implícitas podem ocorrer em várias situações, incluindo invocações de membro de função ([verificação de tempo de compilação da resolução dinâmica de](#)

sobrecarga), expressões de conversão ([expressões de conversão](#)) e atribuições ([operadores de atribuição](#)).

As conversões implícitas predefinidas sempre são bem sucedidos e nunca causam a geração de exceções. Conversões implícitas definidas pelo usuário corretamente criadas também devem apresentar essas características.

Para fins de conversão, os tipos `object` e `dynamic` são considerados equivalentes.

No entanto, conversões dinâmicas (conversões [dinâmicas implícitas](#) e [conversões dinâmicas explícitas](#)) se aplicam somente a expressões do tipo `dynamic` ([o tipo dinâmico](#)).

## Conversão de identidade

Uma conversão de identidade converte de qualquer tipo para o mesmo tipo. Essa conversão existe de modo que uma entidade que já tenha um tipo necessário seja considerada para ser conversível para esse tipo.

- Como `object` e `dynamic` são considerados equivalentes, há uma conversão de identidade entre `object` e `dynamic` entre os tipos construídos que são iguais ao substituir todas as ocorrências de `dynamic` por `object`.

## Conversões numéricas implícitas

As conversões numéricas implícitas são:

- De `sbyte` para `short`, `int`, `long`, `float`, `double` ou `decimal`.
- De `byte` para `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` ou `decimal`
- De `short` para `int`, `long`, `float`, `double` ou `decimal`.
- De `ushort` para `int`, `uint`, `long`, `ulong`, `float`, `double` ou `decimal`.
- De `int` para `long`, `float`, `double` ou `decimal`.
- De `uint` para `long`, `ulong`, `float`, `double` ou `decimal`.
- De `long` para `float`, `double` ou `decimal`.
- De `ulong` para `float`, `double` ou `decimal`.
- De `char` para `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` ou `decimal`.
- De `float` a `double`.

As conversões de `int`, `uint`, `long` ou `ulong` para `float` e de `long` ou `ulong` para `double` podem causar uma perda de precisão, mas nunca causarão uma perda de

magnitude. As outras conversões numéricas implícitas nunca perdem nenhuma informação.

Não há conversões implícitas para o `char` tipo, de modo que os valores dos outros tipos inteiros não são convertidos automaticamente para o `char` tipo.

## Conversões implícitas de enumeração

Uma conversão implícita de enumeração permite que o *decimal\_integer\_literal* `0` seja convertido em qualquer *enum\_type* e em qualquer *nullable\_type* cujo tipo subjacente seja um *enum\_type*. No último caso, a conversão é avaliada convertendo para o *enum\_type* subjacente e encapsulando o resultado ([tipos anuláveis](#)).

## Conversões de cadeia de caracteres interpoladas implícitas

Uma conversão de cadeia de caracteres interpolada implícita permite que um *interpolated\_string\_expression* ([cadeias de caracteres interpoladas](#)) seja convertido em `System.IFormattable` ou `System.FormattableString` (que implementa `System.IFormattable`).

Quando essa conversão é aplicada, um valor de cadeia de caracteres não é composto da cadeia de caracteres interpolada. Em vez disso, uma instância do `System.FormattableString` é criada, conforme descrito em [cadeias de caracteres interpoladas](#).

## Conversões anuláveis implícitas

Conversões implícitas predefinidas que operam em tipos de valores não anuláveis também podem ser usadas com formulários anuláveis desses tipos. Para cada uma das conversões implícitas e numéricas predefinidas que convertem de um tipo de valor não anulável `S` para um tipo de valor não anulável `T`, existem as seguintes conversões anuláveis implícitas:

- Uma conversão implícita de `S?` para `T?` .
- Uma conversão implícita de `S` para `T?` .

A avaliação de uma conversão anulável implícita com base em uma conversão subjacente do `S` para `T` prossegue da seguinte maneira:

- Se a conversão anulável for de `S?` para `T?` :

- Se o valor de origem for `NULL` (a `HasValue` propriedade é `false`), o resultado será o valor nulo do tipo `T?`.
- Caso contrário, a conversão é avaliada como um desempacotamento de `S?` para `S`, seguida pela conversão subjacente de `S` para `T`, seguida por um encapsulamento ([tipos anuláveis](#)) de `T` para `T?`.
- Se a conversão anulável for de `S` para `T?`, a conversão será avaliada como a conversão subjacente de `S` para `T` seguida de um encapsulamento de `T` para `T?`.

## Conversões literais nulas

Uma conversão implícita existe do `null` literal para qualquer tipo anulável. Essa conversão produz o valor nulo ([tipos anuláveis](#)) do tipo anulável especificado.

## Conversões de referência implícitas

As conversões de referência implícitas são:

- De qualquer `reference_type` para `object` e `dynamic`.
- De qualquer `class_type` `S` a qualquer `class_type` `T`, fornecida `S` é derivada de `T`.
- De qualquer `class_type` `S` a qualquer `interface_type` `T`, fornecida `S` implementa `T`.
- De qualquer `interface_type` `S` a qualquer `interface_type` `T`, fornecida `S` é derivada de `T`.
- De um `array_type` `S` com um tipo de elemento `SE` para um `array_type` `T` com um tipo de elemento `TE`, desde que todas as seguintes opções sejam verdadeiras:
  - `S` e `T` difere somente no tipo de elemento. Em outras palavras, `S` e `T` têm o mesmo número de dimensões.
  - Ambos `SE` e `TE` são `reference_type`s.
  - Existe uma conversão de referência implícita de `SE` para `TE`.
- De qualquer `array_type` para `System.Array` o e as interfaces que ele implementa.
- De um tipo de matriz unidimensional `S[]` para `System.Collections.Generic.IList<T>` o e suas interfaces base, desde que haja uma conversão implícita de identidade ou referência de `S` para `T`.
- De qualquer `delegate_type` para `System.Delegate` o e as interfaces que ele implementa.
- Do literal nulo para qualquer `reference_type`.
- De qualquer `reference_type` a uma `reference_type` `T` se ele tiver uma conversão implícita de identidade ou referência para uma `reference_type` `T0` e `T0` tiver uma conversão de identidade para `T`.

- De qualquer *reference\_type* a uma interface ou tipo delegado *T*, se ele tiver uma conversão implícita de identidade ou referência para uma interface ou tipo delegado *T0* e *T0* for conversível ([conversão de variância](#)) para *T*.
- Conversões implícitas envolvendo parâmetros de tipo que são conhecidos como tipos de referência. Confira [conversões implícitas envolvendo parâmetros de tipo](#) para obter mais detalhes sobre conversões implícitas envolvendo parâmetros de tipo.

As conversões de referência implícitas são aquelas que são conversões entre *reference\_type*s que podem ser comprovadas sempre com sucesso e, portanto, não exigem verificações em tempo de execução.

As conversões de referência, implícita ou explícita, nunca alteram a identidade referencial do objeto que está sendo convertido. Em outras palavras, embora uma conversão de referência possa alterar o tipo da referência, ela nunca altera o tipo ou o valor do objeto que está sendo referenciado.

## Conversões de Boxing

Uma conversão boxing permite que um *value\_type* seja convertido implicitamente em um tipo de referência. Existe uma conversão boxing de qualquer *non\_nullable\_value\_type* para `object` e para `dynamic` `System.ValueType` e para qualquer *interface\_type* implementada pelo *non\_nullable\_value\_type*. Além disso, um *enum\_type* pode ser convertido para o tipo `System.Enum`.

Uma conversão boxing existe de um *nullable\_type* para um tipo de referência, se e somente se uma conversão boxing existir do *non\_nullable\_value\_type* subjacente para o tipo de referência.

Um tipo de valor tem uma conversão boxing para um tipo de interface *I* se ele tiver uma conversão boxing em um tipo de interface *I0* e *I0* tiver uma conversão de identidade para *I*.

Um tipo de valor tem uma conversão boxing para um tipo de interface *I* se ele tiver uma conversão boxing para uma interface ou tipo delegado *I0* e *I0* for conversível ([conversão de variância](#)) para *I*.

Boxing um valor de uma *non\_nullable\_value\_type* consiste em alocar uma instância de objeto e copiar o valor de *value\_type* nessa instância. Uma struct pode ser encaixada no tipo `System.ValueType`, pois essa é uma classe base para todas as structs ([herança](#)).

Boxing um valor de uma *nullable\_type* procede da seguinte maneira:

- Se o valor de origem for `NULL` (a `HasValue` propriedade é `false`), o resultado será uma referência nula do tipo de destino.
- Caso contrário, o resultado é uma referência a um encaixado `T` produzido por desencapsulamento e conversão de código-fonte do valor de origem.

As conversões Boxing são descritas em mais detalhes nas [conversões Boxing](#).

## Conversões dinâmicas implícitas

Existe uma conversão dinâmica implícita de uma expressão do tipo `dynamic` para qualquer tipo `T`. A conversão é vinculada dinamicamente ([associação dinâmica](#)), o que significa que uma conversão implícita será procurada em tempo de execução do tipo de tempo de execução da expressão para `T`. Se nenhuma conversão for encontrada, uma exceção de tempo de execução será lançada.

Observe que essa conversão implícita aparentemente viola o Conselho no início de [conversões implícitas](#) que uma conversão implícita nunca deve causar uma exceção. No entanto, ela não é a conversão em si, mas a *localização* da conversão que causa a exceção. O risco de exceções em tempo de execução é inerente ao uso de associação dinâmica. Se a vinculação dinâmica da conversão não for desejada, a expressão poderá ser convertida primeiro para `object` e, em seguida, para o tipo desejado.

O exemplo a seguir ilustra as conversões dinâmicas implícitas:

C#

```
object o = "object"
dynamic d = "dynamic";

string s1 = o; // Fails at compile-time -- no conversion exists
string s2 = d; // Compiles and succeeds at run-time
int i      = d; // Compiles but fails at run-time -- no conversion exists
```

As atribuições para `s2` e `i` ambas empregam conversões dinâmicas implícitas, em que a Associação das operações é suspensa até o tempo de execução. Em tempo de execução, as conversões implícitas são buscadas do tipo de tempo de execução de `d` -- `string` -- para o tipo de destino. Uma conversão é encontrada para `string`, mas não para `int`.

## Conversões de expressão de constante implícita

Uma conversão de expressão constante implícita permite as seguintes conversões:

- Um *constant\_expression* ([expressões constantes](#)) do tipo `int` pode ser convertido em tipo `sbyte` , `byte` , ou, `short` desde que `ushort` `uint` `ulong` o valor da *constant\_expression* esteja dentro do intervalo do tipo de destino.
- Uma *constant\_expression* do tipo `long` pode ser convertida para o tipo `ulong` , desde que o valor da *constant\_expression* não seja negativo.

## Conversões implícitas envolvendo parâmetros de tipo

Existem as seguintes conversões implícitas para um determinado parâmetro de tipo `T` :

- De `T` para sua classe base efetiva `C` , de `T` para qualquer classe base de `C` , e de `T` para qualquer interface implementada pelo `C` . Em tempo de execução, se `T` for um tipo de valor, a conversão será executada como uma conversão boxing. Caso contrário, a conversão é executada como uma conversão de referência implícita ou conversão de identidade.
- De `T` para um tipo de interface `I` no `T` conjunto de interfaces efetivas e de `T` para qualquer interface base do `I` . Em tempo de execução, se `T` for um tipo de valor, a conversão será executada como uma conversão boxing. Caso contrário, a conversão é executada como uma conversão de referência implícita ou conversão de identidade.
- De `T` para um parâmetro de tipo `U` , fornecido `T` depende de `U` ([restrições de parâmetro de tipo](#)). Em tempo de execução, se `U` for um tipo de valor, será, `T` `U` necessariamente, o mesmo tipo e nenhuma conversão será executada. Caso contrário, se `T` for um tipo de valor, a conversão será executada como uma conversão boxing. Caso contrário, a conversão é executada como uma conversão de referência implícita ou conversão de identidade.
- Do literal nulo para `T` , fornecido `T` é conhecido como um tipo de referência.
- De `T` para um tipo de referência `I` se ele tiver uma conversão implícita para um tipo de referência `S0` e `S0` tiver uma conversão de identidade para `S` . Em tempo de execução, a conversão é executada da mesma maneira que a conversão para `S0` .
- De `T` para um tipo de interface `I` se ele tem uma conversão implícita para uma interface ou tipo delegado `I0` e `I0` é convertido em variância para `I` ([conversão de variância](#)). Em tempo de execução, se `T` for um tipo de valor, a conversão será executada como uma conversão boxing. Caso contrário, a conversão é executada como uma conversão de referência implícita ou conversão de identidade.

Se `T` for conhecido como um tipo de referência ([restrições de parâmetro de tipo](#)), as conversões acima serão todas classificadas como conversões de referência implícita ([conversões de referência implícitas](#)). Se `T` não for conhecido como um tipo de

referência, as conversões acima serão classificadas como conversões Boxing ([conversões Boxing](#)).

## Conversões implícitas definidas pelo usuário

Uma conversão implícita definida pelo usuário consiste em uma conversão implícita padrão opcional, seguida pela execução de um operador de conversão implícita definido pelo usuário, seguido por outra conversão implícita padrão opcional. As regras exatas para avaliar conversões implícitas definidas pelo usuário são descritas em [processamento de conversões implícitas definidas pelo usuário](#).

## Conversões de função anônima e conversões de grupo de métodos

Funções anônimas e grupos de métodos não têm tipos próprios, mas podem ser convertidos implicitamente em tipos delegados ou tipos de árvore de expressão. As conversões de função anônimas são descritas mais detalhadamente em [conversões de função anônima](#) e conversões de grupo de métodos em [conversões de grupos de métodos](#).

## Conversões explícitas

As conversões a seguir são classificadas como conversões explícitas:

- Todas as conversões implícitas.
- Conversões numéricas explícitas.
- Conversões de enumeração explícitas.
- Conversões anuláveis explícitas.
- Conversões de referência explícita.
- Conversões de interface explícitas.
- Conversões de unboxing.
- Conversões dinâmicas explícitas
- Conversões explícitas definidas pelo usuário.

Conversões explícitas podem ocorrer em expressões de conversão ([expressões de conversão](#)).

O conjunto de conversões explícitas inclui todas as conversões implícitas. Isso significa que as expressões de conversão redundantes são permitidas.

As conversões explícitas que não são conversões implícitas são conversões que não podem ser comprovadas sempre com sucesso, conversões que são conhecidas por possivelmente perderem informações e conversões em domínios de tipos suficientemente diferentes para a notação explícita de mérito.

## Conversões numéricas explícitas

As conversões numéricas explícitas são as conversões de um *numeric\_type* para outro *numeric\_type* para o qual uma conversão numérica implícita ([conversões numéricas implícitas](#)) ainda não existe:

- De `sbyte` para `byte` , `ushort` , `uint` , `ulong` OU `char` .
- De `byte` para `sbyte` e `char` .
- De `short` para `sbyte` , `byte` , `ushort` , `uint` , `ulong` OU `char` .
- Do `ushort` para `sbyte` , `byte` , `short` OU `char` .
- De `int` para `sbyte` , `byte` , `short` , `ushort` , `uint` `ulong` OU `char` .
- De `uint` para `sbyte` , `byte` , `short` , `ushort` , `int` OU `char` .
- De `long` para `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `ulong` OU `char` .
- De `ulong` para `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` OU `char` .
- Do `char` para `sbyte` , `byte` ou `short` .
- De `float` para `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` ou `decimal` .
- De `double` para `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` , `float` ou `decimal` .
- De `decimal` para `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` , `float` ou `double` .

Como as conversões explícitas incluem todas as conversões numéricas implícitas e explícitas, sempre é possível converter de qualquer *numeric\_type* em qualquer outro *numeric\_type* usando uma expressão de conversão ([expressões de conversão](#)).

As conversões numéricas explícitas possivelmente perdem informações ou possivelmente causam a geração de exceções. Uma conversão numérica explícita é processada da seguinte maneira:

- Para uma conversão de um tipo integral para outro tipo integral, o processamento depende do contexto de verificação de estouro ([os operadores marcados e desmarcados](#)) nos quais a conversão ocorre:
  - Em um `checked` contexto, a conversão terá sucesso se o valor do operando de origem estiver dentro do intervalo do tipo de destino, mas lançará um

`System.OverflowException` se o valor do operando de origem estiver fora do intervalo do tipo de destino.

- Em um `unchecked` contexto, a conversão sempre é bem sucedido e prossegue da seguinte maneira:
  - Se o tipo de origem for maior do que o tipo de destino, então o valor de origem será truncado descartando seus bits "extra" mais significativos. O resultado é então tratado como um valor do tipo de destino.
  - Se o tipo de origem for menor do que o tipo de destino, então o valor de origem será estendido por sinal ou por zero para que tenha o mesmo tamanho que o tipo de destino. A extensão por sinal será usada se o tipo de origem tiver sinal; a extensão por zero será usada se o tipo de origem não tiver sinal. O resultado é então tratado como um valor do tipo de destino.
  - Se o tipo de origem tiver o mesmo tamanho que o tipo de destino, então o valor de origem será tratado como um valor do tipo de destino.
- Para uma conversão de `decimal` para um tipo integral, o valor de origem é arredondado para zero para o valor integral mais próximo, e esse valor integral se torna o resultado da conversão. Se o valor integral resultante estiver fora do intervalo do tipo de destino, um `System.OverflowException` será lançado.
- Para uma conversão de `float` ou `double` para um tipo integral, o processamento depende do contexto de verificação de estouro ([os operadores marcados e desmarcados](#)) nos quais a conversão ocorre:
  - Em um `checked` contexto, a conversão continua da seguinte maneira:
    - Se o valor do operando for NaN ou infinito, um `System.OverflowException` será gerado.
    - Caso contrário, o operando de origem será arredondado em direção a zero para o valor integral mais próximo. Se esse valor integral estiver dentro do intervalo do tipo de destino, esse valor será o resultado da conversão.
    - Caso contrário, uma `System.OverflowException` será gerada.
  - Em um `unchecked` contexto, a conversão sempre é bem sucedido e prossegue da seguinte maneira:
    - Se o valor do operando for NaN ou infinito, o resultado da conversão será um valor não especificado do tipo de destino.
    - Caso contrário, o operando de origem será arredondado em direção a zero para o valor integral mais próximo. Se esse valor integral estiver dentro do intervalo do tipo de destino, esse valor será o resultado da conversão.
    - Caso contrário, o resultado da conversão é um valor não especificado do tipo de destino.
- Para uma conversão de `double` para `float`, o `double` valor é arredondado para o valor mais próximo `float`. Se o `double` valor for muito pequeno para representar como um `float`, o resultado se tornará positivo zero ou negativo zero. Se o

`double` valor for muito grande para representar como um `float`, o resultado se tornará infinito positivo ou infinito negativo. Se o `double` valor for NaN, o resultado também será Nan.

- Para uma conversão de `float` ou `double` para `decimal`, o valor de origem é convertido em `decimal` representação e arredondado para o número mais próximo após a casa decimal 28, se necessário ([o tipo decimal](#)). Se o valor de origem for muito pequeno para representar como um `decimal`, o resultado se tornará zero. Se o valor de Source for NaN, Infinity ou muito grande para representar como um `decimal`, um `System.OverflowException` será lançado.
- Para uma conversão de `decimal` para `float` ou `double`, o `decimal` valor é arredondado para o valor mais próximo `double` ou `float`. Embora essa conversão possa perder a precisão, ela nunca faz com que uma exceção seja gerada.

## Conversões de enumeração explícitas

As conversões de enumeração explícitas são:

- De `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double` ou `decimal` para qualquer `enum_type`.
- De qualquer `enum_type` para `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double` ou `decimal`.
- De qualquer `enum_type` a qualquer outro `enum_type`.

Uma conversão de enumeração explícita entre dois tipos é processada tratando qualquer `enum_type` participante como o tipo subjacente do `enum_type` e, em seguida, executando uma conversão numérica implícita ou explícita entre os tipos resultantes. Por exemplo, dado um `enum_type` `E` com o tipo subjacente de `int`, uma conversão de `E` para `byte` é processada como uma conversão numérica explícita ([conversões numéricas explícitas](#)) de `int` para `byte` e uma conversão de `byte` para `E` é processada como uma conversão numérica implícita ([conversões numéricas implícitas](#)) de `byte` para `int`.

## Conversões anuláveis explícitas

As [\*\*conversões anuláveis explícitas\*\*](#) permitem conversões explícitas predefinidas que operam em tipos de valores não anuláveis para também serem usadas com formulários anuláveis desses tipos. Para cada uma das conversões explícitas predefinidas que convertem de um tipo de valor não anulável `S` para um tipo de valor não anulável `T` ([conversão de identidade](#), [conversões numéricas implícitas](#), conversões [implícitas de](#)

enumeração, conversões numéricas explícitas e conversões de enumeração explícitas), existem as seguintes conversões anuláveis:

- Uma conversão explícita de `S?` para `T?`.
- Uma conversão explícita de `S` para `T?`.
- Uma conversão explícita de `S?` para `T`.

A avaliação de uma conversão anulável com base em uma conversão subjacente do `S` para `T` prossegue da seguinte maneira:

- Se a conversão anulável for de `S?` para `T?`:
  - Se o valor de origem for `NULL` (a `HasValue` propriedade é `false`), o resultado será o valor nulo do tipo `T?`.
  - Caso contrário, a conversão é avaliada como um desempacotamento de `S?` para `S`, seguida pela conversão subjacente de `S` para `T`, seguida de um encapsulamento de `T` para `T?`.
- Se a conversão anulável for de `S` para `T?`, a conversão será avaliada como a conversão subjacente de `S` para `T` seguida de um encapsulamento de `T` para `T?`.
- Se a conversão anulável for de `S?` para `T`, a conversão será avaliada como um desempacotamento de `S?` para `S` seguido pela conversão subjacente de `S` para `T`.

Observe que uma tentativa de desencapsular um valor anulável gerará uma exceção se o valor for `null`.

## Conversões de referência explícitas

As conversões de referência explícitas são:

- De `object` e `dynamic` para qualquer outro *reference\_type*.
- De qualquer *class\_type* `S` a qualquer *class\_type* `T`, fornecido `S` é uma classe base de `T`.
- De qualquer *class\_type* `S` a qualquer *interface\_type* `T`, fornecido `S` não é lacrado e fornecido não `S` implementa `T`.
- De qualquer *interface\_type* `S` a qualquer *class\_type* `T`, fornecida `T` não é uma implementação selada ou fornecida `T S`.
- De qualquer *interface\_type* `S` a qualquer *interface\_type* `T`, fornecido `S` não é derivado de `T`.
- De um *array\_type* `S` com um tipo de elemento `SE` para um *array\_type* `T` com um tipo de elemento `TE`, desde que todas as seguintes opções sejam verdadeiras:

- `S` e `T` difere somente no tipo de elemento. Em outras palavras, `S` e `T` têm o mesmo número de dimensões.
- Ambos `SE` e `TE` são *reference\_type*s.
- Existe uma conversão de referência explícita de `SE` para `TE`.
- `System.Array` E as interfaces que ele implementa para qualquer *array\_type*.
- De um tipo de matriz unidimensional `S[]` para `System.Collections.Generic.IList<T>` e suas interfaces base, desde que haja uma conversão de referência explícita de `S` para `T`.
- De `System.Collections.Generic.IList<S>` e suas interfaces base para um tipo de matriz unidimensional `T[]`, desde que haja uma conversão explícita de identidade ou referência de `S` para `T`.
- `System.Delegate` E as interfaces que ele implementa para qualquer *delegate\_type*.
- De um tipo de referência para um tipo de referência `T` se ele tiver uma conversão de referência explícita para um tipo de referência `T0` e `T0` tiver uma conversão de identidade `T`.
- De um tipo de referência para uma interface ou tipo delegado `T` se ele tiver uma conversão de referência explícita para um tipo de interface ou delegado `T0` e `T0` for conversível em variância `T` ou se `T` for convertido em variância para `T0` (conversão de variância).
- De `D<S1...Sn>` para `D<T1...Tn>` onde `D<X1...Xn>` é um tipo de delegado genérico, `D<S1...Sn>` não é compatível com or idêntico a `D<T1...Tn>` e para cada parâmetro `Xi` de tipo das `D` seguintes isenções:
  - Se `Xi` for invariável, `Si` será idêntico a `Ti`.
  - Se `Xi` for covariant, haverá uma identidade implícita ou explícita ou conversão de referência de `Si` para `Ti`.
  - Se `Xi` for contravariant, então `Si` será `Ti` idêntico ou ambos os tipos de referência.
- Conversões explícitas envolvendo parâmetros de tipo que são conhecidos como tipos de referência. Para obter mais detalhes sobre conversões explícitas envolvendo parâmetros de tipo, consulte [conversões explícitas envolvendo parâmetros de tipo](#).

As conversões de referência explícitas são as conversões entre os tipos de referência que exigem verificações de tempo de execução para garantir que estejam corretos.

Para que uma conversão de referência explícita tenha sucesso em tempo de execução, o valor do operando de origem deve ser `null`, ou o tipo real do objeto referenciado pelo operando de origem deve ser um tipo que pode ser convertido para o tipo de destino por uma conversão de referência implícita ([conversões de referência implícita](#)) ou

conversão boxing ([conversões Boxing](#)). Se uma conversão de referência explícita falhar, um `System.InvalidCastException` será lançado.

As conversões de referência, implícita ou explícita, nunca alteram a identidade referencial do objeto que está sendo convertido. Em outras palavras, embora uma conversão de referência possa alterar o tipo da referência, ela nunca altera o tipo ou o valor do objeto que está sendo referenciado.

## Conversões de unboxing

Uma conversão sem Boxing permite que um tipo de referência seja explicitamente convertido em um *value\_type*. Existe uma conversão de irboxing dos tipos `object`, `dynamic` e `System.ValueType` para qualquer *non\_nullable\_value\_type* e de qualquer *interface\_type* a qualquer *non\_nullable\_value\_type* que implemente o *interface\_type*. Além disso, o tipo `System.Enum` pode ser desemoldurado para qualquer *enum\_type*.

Uma conversão unboxing existe de um tipo de referência para um *nullable\_type* se existir uma conversão unboxing do tipo de referência para o *non\_nullable\_value\_type* subjacente do *nullable\_type*.

Um tipo de valor `s` tem uma conversão unboxing de um tipo de interface `I` se ele tiver uma conversão unboxing de um tipo de interface `I0` e `I0` tiver uma conversão de identidade para `I`.

Um tipo de valor `s` tem uma conversão unboxing de um tipo de interface `I` se ele tem uma conversão unboxing de uma interface ou um tipo delegado, `I0` e `I0` é convertido em variância ou pode ser `I` convertido em `I` variância `I0` (conversão de [variância](#)).

Uma operação de unboxing consiste na primeira verificação de que a instância do objeto é um valor em caixa do determinado *value\_type* e, em seguida, copiar o valor para fora da instância. Unboxing uma referência nula a um *nullable\_type* produz o valor nulo do *nullable\_type*. Uma struct pode ser desemoldurada do tipo `System.ValueType`, pois essa é uma classe base para todas as structs ([herança](#)).

As conversões de unboxing são descritas em detalhes em [conversões unboxing](#).

## Conversões dinâmicas explícitas

Existe uma conversão dinâmica explícita de uma expressão do tipo `dynamic` para qualquer tipo `T`. A conversão é vinculada dinamicamente ([associação dinâmica](#)), o que significa que uma conversão explícita será procurada em tempo de execução do tipo de

tempo de execução da expressão para `T`. Se nenhuma conversão for encontrada, uma exceção de tempo de execução será lançada.

Se a vinculação dinâmica da conversão não for desejada, a expressão poderá ser convertida primeiro para `object` e, em seguida, para o tipo desejado.

Suponha que a seguinte classe seja definida:

```
C#  
  
class C  
{  
    int i;  
  
    public C(int i) { this.i = i; }  
  
    public static explicit operator C(string s)  
    {  
        return new C(int.Parse(s));  
    }  
}
```

O exemplo a seguir ilustra as conversões dinâmicas explícitas:

```
C#  
  
object o = "1";  
dynamic d = "2";  
  
var c1 = (C)o; // Compiles, but explicit reference conversion fails  
var c2 = (C)d; // Compiles and user defined conversion succeeds
```

A melhor conversão de `o` para `C` é encontrada em tempo de compilação para ser uma conversão de referência explícita. Isso falha em tempo de execução, porque `"1"` não é, na verdade, uma `C`. A conversão de `d` para `C` no entanto, como uma conversão dinâmica explícita, é suspensa para tempo de execução, onde uma conversão definida pelo usuário do tipo de tempo de execução de `d -- string --> C` é encontrada e é bem sucedido.

## Conversões explícitas envolvendo parâmetros de tipo

Existem as seguintes conversões explícitas para um determinado parâmetro de tipo `T`:

- Da classe base efetiva `C` de `T` para `T` e de qualquer classe base de `C` para `T`. Em tempo de execução, se `T` for um tipo de valor, a conversão será executada como

uma conversão não boxing. Caso contrário, a conversão é executada como uma conversão de referência explícita ou conversão de identidade.

- De qualquer tipo de interface para  $T$ . Em tempo de execução, se  $T$  for um tipo de valor, a conversão será executada como uma conversão não boxing. Caso contrário, a conversão é executada como uma conversão de referência explícita ou conversão de identidade.
- De  $T$  para qualquer *interface\_type*  $I$  fornecido, ainda não há uma conversão implícita de  $T$  para  $I$ . Em tempo de execução, se  $T$  for um tipo de valor, a conversão será executada como uma conversão boxing seguida de uma conversão de referência explícita. Caso contrário, a conversão é executada como uma conversão de referência explícita ou conversão de identidade.
- De um parâmetro de tipo  $U$  para  $T$ , fornecido  $T$  depende de  $U$  (restrições de parâmetro de tipo). Em tempo de execução, se  $U$  for um tipo de valor, será,  $T \rightarrow U$  necessariamente, o mesmo tipo e nenhuma conversão será executada. Caso contrário, se  $T$  for um tipo de valor, a conversão será executada como uma conversão não boxing. Caso contrário, a conversão é executada como uma conversão de referência explícita ou conversão de identidade.

Se  $T$  for conhecido como um tipo de referência, as conversões acima serão todas classificadas como conversões de referência explícita ([conversões de referência explícitas](#)). Se  $T$  não for conhecido como um tipo de referência, as conversões acima serão classificadas como conversões não Boxing ([conversões unboxing](#)).

As regras acima não permitem uma conversão explícita direta de um parâmetro de tipo irrestrito para um tipo que não seja de interface, o que pode ser surpreendente. A razão para essa regra é evitar confusão e tornar a semântica dessas conversões clara. Por exemplo, considere a seguinte declaração:

```
C#  
  
class X<T>  
{  
    public static long F(T t) {  
        return (long)t;           // Error  
    }  
}
```

Se a conversão explícita direta de  $t$  para  $int$  fosse permitida, uma delas poderia esperar que  $X<int>.F(7)$  retornasse  $7L$ . No entanto, isso não ocorre porque as conversões numéricas padrão só são consideradas quando os tipos são conhecidos como numéricos no tempo de associação. Para tornar a semântica clara, o exemplo acima deve ser escrito em vez disso:

C#

```
class X<T>
{
    public static long F(T t) {
        return (long)(object)t;           // Ok, but will only work when T is
    long
    }
}
```

Agora, esse código será compilado, mas a execução `X<int>.F(7)` geraria uma exceção em tempo de execução, pois um Boxed `int` não pode ser convertido diretamente em um `long`.

## Conversões explícitas definidas pelo usuário

Uma conversão explícita definida pelo usuário consiste em uma conversão explícita padrão opcional, seguida pela execução de um operador de conversão implícita ou explícita definido pelo usuário, seguido por outra conversão opcional explícita padrão. As regras exatas para avaliar conversões explícitas definidas pelo usuário são descritas em [processamento de conversões explícitas definidas pelo usuário](#).

## Conversões padrão

As conversões padrão são aquelas conversões predefinidas que podem ocorrer como parte de uma conversão definida pelo usuário.

## Conversões implícitas padrão

As conversões implícitas a seguir são classificadas como conversões implícitas padrão:

- Conversões de identidade ([conversão de identidade](#))
- Conversões numéricas implícitas ([conversões numéricas implícitas](#))
- Conversões anuláveis implícitas ([conversões anuláveis implícitas](#))
- Conversões implícitas de referência ([conversões de referência implícitas](#))
- Conversões Boxing ([conversões Boxing](#))
- Conversões de expressão de constante implícita ([conversões de expressão de constante implícita](#))
- Conversões implícitas envolvendo parâmetros de tipo ([conversões implícitas envolvendo parâmetros de tipo](#))

As conversões implícitas padrão excluem especificamente conversões implícitas definidas pelo usuário.

## Conversões explícitas padrão

As conversões explícitas padrão são todas as conversões implícitas padrão mais o subconjunto das conversões explícitas para as quais existe uma conversão implícita padrão oposta. Em outras palavras, se existir uma conversão implícita padrão de um tipo `A` para um tipo `B`, uma conversão explícita padrão existirá de tipo `A` para tipo `B` e de tipo `B` para tipo `A`.

## Conversões definidas pelo usuário

O C# permite que as conversões implícitas e explícitas predefinidas sejam aumentadas por **conversões definidas pelo usuário**. Conversões definidas pelo usuário são introduzidas pela declaração de operadores de conversão ([operadores de conversão](#)) em tipos de classe e struct.

## Conversões permitidas definidas pelo usuário

O C# permite que apenas determinadas conversões definidas pelo usuário sejam declaradas. Em particular, não é possível redefinir uma conversão implícita ou explícita já existente.

Para um determinado tipo de origem `S` e tipo `T` de destino, se `S` ou `T` forem tipos anuláveis, avise `s0` e `t0` faça referência aos seus tipos subjacentes, caso contrário, `s0` e `t0` sejam iguais a `S` e `T` respectivamente. Uma classe ou estrutura tem permissão para declarar uma conversão de um tipo de origem `S` para um tipo de destino `T` somente se todas as seguintes opções forem verdadeiras:

- `s0` e `t0` são tipos diferentes.
- `s0` ou `t0` é o tipo de classe ou struct no qual a declaração do operador ocorre.
- Nem `s0` nem `t0` é um *interface\_type*.
- Excluindo conversões definidas pelo usuário, uma conversão não existe de `S` ou para `T` `T` `S`.

As restrições que se aplicam a conversões definidas pelo usuário são discutidas mais detalhadamente nos [operadores de conversão](#).

## Operadores de conversão levantados

Dado um operador de conversão definido pelo usuário que converte de um tipo de valor não anulável `s` para um tipo de valor não anulável `T`, existe um **operador de conversão levantado** que converte de `s?` para `T?`. Esse operador de conversão levantado executa um desempacotamento do `s?` para `s` seguido pela conversão definida pelo usuário de `s` para `T` seguida por um encapsulamento de `T` para `T?`, exceto que um valor nulo é `s?` convertido diretamente em um valor nulo `T?`.

Um operador de conversão elevado tem a mesma classificação implícita ou explícita que seu operador de conversão subjacente definido pelo usuário. O termo "conversão definida pelo usuário" aplica-se ao uso de operadores de conversão que foram definidos pelo usuário e com elevação.

## Avaliação de conversões definidas pelo usuário

Uma conversão definida pelo usuário converte um valor de seu tipo, chamado de *tipo de origem* \_, para outro tipo, chamado de \_*tipo de destino*\*.. Avaliação de um centro de conversão definido pelo usuário para encontrar o \_ mais específico\* operador de conversão definido pelo usuário para os tipos de origem e destino específicos. Essa determinação é dividida em várias etapas:

- Localizando o conjunto de classes e structs dos quais operadores de conversão definidos pelo usuário serão considerados. Esse conjunto consiste no tipo de origem e suas classes base, e o tipo de destino e suas classes base (com as suposições implícitas que somente classes e structs podem declarar operadores definidos pelo usuário e que não têm classes base). Para os fins desta etapa, se o tipo de origem ou de destino for um *nullable\_type*, seu tipo subjacente será usado em seu lugar.
- Desse conjunto de tipos, determinando quais operadores de conversão levantados e definidos pelo usuário são aplicáveis. Para que um operador de conversão seja aplicável, deve ser possível executar uma conversão padrão ([conversões padrão](#)) do tipo de origem para o tipo de operando do operador e deve ser possível executar uma conversão padrão do tipo de resultado do operador para o tipo de destino.
- Do conjunto de operadores aplicáveis definidos pelo usuário, determinando qual operador não é ambigamente o mais específico. Em termos gerais, o operador mais específico é o operador cujo tipo de operando é "mais próximo" do tipo de origem e cujo tipo de resultado é "mais próximo" ao tipo de destino. Os operadores de conversão definidos pelo usuário são preferenciais em relação aos operadores de conversão levantados. As regras exatas para estabelecer o operador de conversão mais específico definido pelo usuário são definidas nas seções a seguir.

Depois que um operador de conversão definido pelo usuário mais específico tiver sido identificado, a execução real da conversão definida pelo usuário envolverá até três etapas:

- Primeiro, se necessário, executando uma conversão padrão do tipo de origem para o tipo de operando do operador de conversão definido pelo usuário ou levantado.
- Em seguida, invocando o operador de conversão definido pelo usuário ou elevação para executar a conversão.
- Por fim, se necessário, executar uma conversão padrão do tipo de resultado do operador de conversão definido pelo usuário ou de forma elevada para o tipo de destino.

A avaliação de uma conversão definida pelo usuário nunca envolve mais de um operador de conversão mais definido pelo usuário ou levantado. Em outras palavras, uma conversão de tipo `S` para tipo `T` nunca executará primeiro uma conversão definida pelo usuário de `S` para `X` e, em seguida, executará uma conversão definida pelo usuário de `X` para `T`.

As definições exatas de avaliação de conversões implícitas ou explícitas definidas pelo usuário são dadas nas seções a seguir. As definições fazem uso dos seguintes termos:

- Se uma conversão implícita padrão ([conversões implícitas padrão](#)) existir de um tipo `A` para um tipo `B`, e se nem `A` `B` for *interface\_type* s, `A` será dito que é \* englobado **por** `B` e `B` é dito como `_***A`.
- O **tipo mais abrangente** em um conjunto de tipos é aquele que abrange todos os outros tipos no conjunto. Se nenhum tipo único abrange todos os outros tipos, o conjunto não terá mais de abrange o tipo. Em termos mais intuitivos, o tipo mais abrangente é o tipo "maior" no conjunto — o tipo para o qual cada um dos outros tipos pode ser convertido implicitamente.
- O **tipo mais abrangente** em um conjunto de tipos é aquele que é abrangido por todos os outros tipos no conjunto. Se nenhum tipo único for abrangido por todos os outros tipos, o conjunto não terá o tipo mais abrangido. Em termos mais intuitivos, o tipo mais abrangente é o tipo "menor" no conjunto — aquele tipo que pode ser convertido implicitamente em cada um dos outros tipos.

## Processamento de conversões implícitas definidas pelo usuário

Uma conversão implícita definida pelo usuário do tipo `S` para `T` o tipo é processada da seguinte maneira:

- Determine os tipos  $s_0$  e  $t_0$ . If  $s$  ou  $t$  são tipos anuláveis,  $s_0$  e  $t_0$  são seus tipos subjacentes, caso contrário,  $s_0$  e  $t_0$  são iguais a  $s$  e  $t$  respectivamente.
- Localize o conjunto de tipos,  $D$ , do qual os operadores de conversão definidos pelo usuário serão considerados. Esse conjunto consiste em  $s_0$  (if  $s_0$  é uma classe ou struct), as classes base de  $s_0$  (if  $s_0$  é uma classe) e  $t_0$  (if  $t_0$  é uma Class ou struct).
- Localize o conjunto de operadores de conversão aplicáveis definidos pelo usuário e com elevação,  $U$ . Esse conjunto consiste nos operadores de conversão implícitas definidos pelo usuário e levantados declarados pelas classes ou estruturas no  $D$  que são convertidas de um tipo que abrange  $s$  um tipo que é abrangido por  $t$ . Se  $U$  estiver vazio, a conversão será indefinida e ocorrerá um erro de tempo de compilação.
- Localize o tipo de fonte mais específico,  $s_x$ , dos operadores em  $U$ :
  - Se qualquer um dos operadores em  $U$  converter de  $s$ ,  $s_x$  for  $s$ .
  - Caso contrário,  $s_x$  é o tipo mais abrangente no conjunto combinado de tipos de origem dos operadores no  $U$ . Se não for possível encontrar exatamente um tipo mais abrangente, a conversão será ambígua e ocorrerá um erro de tempo de compilação.
- Localize o tipo de destino mais específico,  $t_x$ , dos operadores em  $U$ :
  - Se qualquer um dos operadores em  $U$  converter para  $t$ ,  $t_x$  for  $t$ .
  - Caso contrário,  $t_x$  é o tipo mais abrangente no conjunto combinado de tipos de destino dos operadores no  $U$ . Se não for possível encontrar exatamente um tipo mais abrangente, a conversão será ambígua e ocorrerá um erro de tempo de compilação.
- Encontre o operador de conversão mais específico:
  - Se  $U$  contiver exatamente um operador de conversão definido pelo usuário que converte de  $s_x$  para  $t_x$ , esse será o operador de conversão mais específico.
  - Caso contrário, se  $U$  contiver exatamente um operador de conversão levantado que converta de  $s_x$  para  $t_x$ , esse será o operador de conversão mais específico.
  - Caso contrário, a conversão é ambígua e ocorre um erro de tempo de compilação.
- Por fim, aplique a conversão:
  - Se  $s$  não for  $s_x$ , uma conversão implícita padrão de  $s$  para  $s_x$  será executada.
  - O operador de conversão mais específico é chamado para converter de  $s_x$  para  $t_x$ .
  - Se  $t_x$  não for  $t$ , uma conversão implícita padrão de  $t_x$  para  $t$  será executada.

# Processamento de conversões explícitas definidas pelo usuário

Uma conversão explícita definida pelo usuário do tipo  $S$  para  $T$  o tipo é processada da seguinte maneira:

- Determine os tipos  $S_0$  e  $T_0$ . If  $S$  ou  $T$  são tipos anuláveis,  $S_0$  e  $T_0$  são seus tipos subjacentes, caso contrário,  $S_0$  e  $T_0$  são iguais a  $S$  e  $T$  respectivamente.
- Localize o conjunto de tipos,  $D$ , do qual os operadores de conversão definidos pelo usuário serão considerados. Esse conjunto consiste em  $S_0$  (se  $S_0$  for uma classe ou struct), as classes base de  $S_0$  (se  $S_0$  for uma classe),  $T_0$  (se  $T_0$  for uma classe ou estrutura) e as classes base de  $T_0$  (se  $T_0$  for uma classe).
- Localize o conjunto de operadores de conversão aplicáveis definidos pelo usuário e com elevação,  $U$ . Esse conjunto consiste nos operadores de conversão implícitos ou explícitos definidos pelo usuário, declarados pelas classes ou structs no  $D$  que são convertidos de um tipo que abrange ou é derivado de  $S$  para um tipo que abrange ou abrange o pelo  $T$ . Se  $U$  estiver vazio, a conversão será indefinida e ocorrerá um erro de tempo de compilação.
- Localize o tipo de fonte mais específico,  $SX$ , dos operadores em  $U$ :
  - Se qualquer um dos operadores em  $U$  converter de  $S$ ,  $SX$  for  $S$ .
  - Caso contrário, se qualquer um dos operadores em  $U$  converter dos tipos que englobam  $S$ ,  $SX$  será o tipo mais abrangido no conjunto combinado de tipos de origem desses operadores. Se nenhum tipo mais abrangido puder ser encontrado, a conversão será ambígua e ocorrerá um erro de tempo de compilação.
  - Caso contrário,  $SX$  é o tipo mais abrangente no conjunto combinado de tipos de origem dos operadores no  $U$ . Se não for possível encontrar exatamente um tipo mais abrangente, a conversão será ambígua e ocorrerá um erro de tempo de compilação.
- Localize o tipo de destino mais específico,  $TX$ , dos operadores em  $U$ :
  - Se qualquer um dos operadores em  $U$  converter para  $T$ ,  $TX$  for  $T$ .
  - Caso contrário, se qualquer um dos operadores em  $U$  converter para tipos que são incluídos pelo  $T$ ,  $TX$  será o tipo mais abrangente no conjunto combinado de tipos de destino desses operadores. Se não for possível encontrar exatamente um tipo mais abrangente, a conversão será ambígua e ocorrerá um erro de tempo de compilação.
  - Caso contrário,  $TX$  é o tipo mais abrangido no conjunto combinado de tipos de destino dos operadores no  $U$ . Se nenhum tipo mais abrangido puder ser

encontrado, a conversão será ambígua e ocorrerá um erro de tempo de compilação.

- Encontre o operador de conversão mais específico:
  - Se `U` contiver exatamente um operador de conversão definido pelo usuário que converte de `sx` para `TX`, esse será o operador de conversão mais específico.
  - Caso contrário, se `U` contiver exatamente um operador de conversão levantado que converta de `sx` para `TX`, esse será o operador de conversão mais específico.
  - Caso contrário, a conversão é ambígua e ocorre um erro de tempo de compilação.
- Por fim, aplique a conversão:
  - Se `S` não for `sx`, uma conversão explícita padrão de `S` para `sx` será executada.
  - O operador de conversão mais específico definido pelo usuário é chamado para converter de `sx` para `TX`.
  - Se `TX` não for `T`, uma conversão explícita padrão de `TX` para `T` será executada.

## Conversões de função anônima

Uma *anonymous\_method\_expression* ou *lambda\_expression* é classificada como uma função anônima ([expressões de função anônimas](#)). A expressão não tem um tipo, mas pode ser convertida implicitamente em um tipo de delegado ou tipo de árvore de expressão compatível. Especificamente, uma função anônima `F` é compatível com um tipo delegado `D` fornecido:

- Se `F` contiver um *anonymous\_function\_signature*, `D` e `F` terá o mesmo número de parâmetros.
- Se não `F` contiver um *anonymous\_function\_signature*, `D` poderá ter zero ou mais parâmetros de qualquer tipo, desde que nenhum parâmetro de `D` tenha o `out` modificador de parâmetro.
- Se o `F` tiver uma lista de parâmetros tipados explicitamente, cada parâmetro no `D` terá o mesmo tipo e modificadores que o parâmetro correspondente no `F`.
- Se o `F` tiver uma lista de parâmetros tipados implicitamente, o `D` não terá `ref` ou `out` parâmetros ou.
- Se o corpo de `F` for uma expressão e `D` tiver um tipo de `void` retorno ou `F` for `Async` e `D` tiver o tipo de retorno `Task`, quando cada parâmetro de `F` for dado o tipo do parâmetro correspondente no `D`, o corpo de `F` será uma expressão válida (wrt [Expressions](#)) que seria permitida como uma *statement\_expression* (instruções de [expressão](#)).

- Se o corpo de `F` for um bloco de instrução e `D` tiver um `void` tipo de retorno ou `F` for `Async` e `D` tiver o tipo de retorno `Task`, quando cada parâmetro de `F` for dado o tipo do parâmetro correspondente no `D`, o corpo de `F` será um bloco de instrução válido ([blocos](#)<sub>WRT</sub>) em que nenhuma `return` instrução especifica uma expressão.
- Se o corpo de `F` for uma expressão, e `F` for não `Async` e `D` tiver um tipo de retorno não `void T`, ou `F` for `Async` e `D` tiver um tipo de retorno `Task<T>`, quando cada parâmetro de `F` for dado o tipo do parâmetro correspondente no `D`, o corpo de `F` será uma expressão válida ([Expressions](#)) que é implicitamente conversível para `T`.
- Se o corpo de `F` for um bloco de instrução, e `F` é não `Async` e `D` tem um tipo de retorno não `void T`, ou `F` é `Async` e `D` tem um tipo de retorno `Task<T>`, quando cada parâmetro de `F` recebe o tipo do parâmetro correspondente no `D`, o corpo de `F` é um bloco de instrução válido ([blocos](#)<sub>WRT</sub>) com um ponto de extremidade não acessível no qual cada `return` instrução especifica uma expressão que é implicitamente conversível para `T`.

Para fins de brevidade, esta seção usa a forma abreviada para os tipos de tarefa `Task` e `Task<T>` ([funções assíncronas](#)).

Uma expressão lambda `F` é compatível com um tipo de árvore de expressão `Expression<D>` se `F` for compatível com o tipo delegado `D`. Observe que isso não se aplica a métodos anônimos, somente expressões lambda.

Determinadas expressões lambda não podem ser convertidas em tipos de árvore de expressões: mesmo que a conversão *exista*, ela falhará em tempo de compilação. Esse é o caso se a expressão lambda:

- Tem um corpo de *bloco*
- Contém operadores de atribuição simples ou compostos
- Contém uma expressão vinculada dinamicamente
- É `Async`

Os exemplos a seguir usam um tipo de delegado genérico `Func<A,R>` que representa uma função que usa um argumento do tipo `A` e retorna um valor do tipo `R`:

C#

```
delegate R Func<A,R>(A arg);
```

Nas atribuições

C#

```
Func<int,int> f1 = x => x + 1;           // Ok  
Func<int,double> f2 = x => x + 1;         // Ok  
Func<double,int> f3 = x => x + 1;         // Error  
Func<int, Task<int>> f4 = async x => x + 1; // Ok
```

os tipos de parâmetro e de retorno de cada função anônima são determinados do tipo da variável à qual a função anônima é atribuída.

A primeira atribuição converte com êxito a função anônima no tipo delegado `Func<int,int>` porque, quando `x` é o tipo fornecido `int`, `x+1` é uma expressão válida que é implicitamente conversível para o tipo `int`.

Da mesma forma, a segunda atribuição converte com êxito a função anônima no tipo delegado `Func<int,double>` porque o resultado de `x+1` (do tipo `int`) é implicitamente conversível para o tipo `double`.

No entanto, a terceira atribuição é um erro de tempo de compilação porque, quando `x` é determinado tipo `double`, o resultado de `x+1` (do tipo `double`) não é implicitamente conversível para o tipo `int`.

A quarta atribuição converte com êxito a função Async anônima no tipo delegado `Func<int, Task<int>>` porque o resultado de `x+1` (do tipo `int`) é implicitamente conversível para o tipo de resultado `int` do tipo de tarefa `Task<int>`.

As funções anônimas podem influenciar a resolução de sobrecarga e participar da inferência de tipos. Consulte [membros da função](#) para obter mais detalhes.

## Avaliação de conversões de função anônima para tipos delegados

A conversão de uma função anônima em um tipo delegate produz uma instância delegada que faz referência à função anônima e ao conjunto (possivelmente vazio) de variáveis externas capturadas que estão ativas no momento da avaliação. Quando o delegado é invocado, o corpo da função anônima é executado. O código no corpo é executado usando o conjunto de variáveis externas capturadas referenciadas pelo delegado.

A lista de invocação de um delegado produzido por uma função anônima contém uma única entrada. O objeto de destino exato e o método de destino do delegado não são especificados. Em particular, ele não é especificado se o objeto de destino do delegado é `null`, o `this` valor do membro da função delimitadora ou algum outro objeto.

Conversões de funções anônimas, semanticamente idênticas com o mesmo conjunto (possivelmente vazio) de instâncias de variáveis externas capturadas para os mesmos tipos delegados, são permitidas (mas não obrigatórias) para retornar a mesma instância de delegado. O termo semanticamente idêntico é usado aqui para significar que a execução das funções anônimas irá, em todos os casos, produzir os mesmos efeitos, considerando os mesmos argumentos. Essa regra permite que um código como o seguinte seja otimizado.

C#

```
delegate double Function(double x);

class Test
{
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void F(double[] a, double[] b) {
        a = Apply(a, (double x) => Math.Sin(x));
        b = Apply(b, (double y) => Math.Sin(y));
        ...
    }
}
```

Como os dois delegados de função anônimas têm o mesmo conjunto (vazio) de variáveis externas capturadas e, como as funções anônimas são semanticamente idênticas, o compilador tem permissão para fazer com que os delegados se refiram ao mesmo método de destino. Na verdade, o compilador tem permissão para retornar a mesma instância de delegado de ambas as expressões de função anônimas.

## Avaliação de conversões de função anônima para tipos de árvore de expressão

A conversão de uma função anônima em um tipo de árvore de expressão produz uma árvore de expressão ([tipos de árvore de expressão](#)). Mais precisamente, a avaliação da conversão da função anônima leva à construção de uma estrutura de objeto que

representa a estrutura da própria função anônima. A estrutura precisa da árvore de expressão, bem como o processo exato para criá-la, são definidas para implementação.

## Exemplo de implementação

Esta seção descreve uma possível implementação de conversões de funções anônimas em termos de outras construções de C#. A implementação descrita aqui se baseia nos mesmos princípios usados pelo compilador do Microsoft C#, mas não é, de maneira alguma, uma implementação obrigatória, nem é a única possível. Ele apenas menciona conversões em árvores de expressão, pois a semântica exata está fora do escopo desta especificação.

O restante desta seção fornece vários exemplos de código que contém funções anônimas com características diferentes. Para cada exemplo, é fornecida uma tradução correspondente ao código que usa apenas outras construções C#. Nos exemplos, o identificador `D` é assumido por representar o seguinte tipo de delegado:

```
C#  
  
public delegate void D();
```

A forma mais simples de uma função anônima é aquela que captura nenhuma variável externa:

```
C#  
  
class Test  
{  
    static void F() {  
        D d = () => { Console.WriteLine("test"); };  
    }  
}
```

Isso pode ser convertido em uma instanciação de delegado que faz referência a um método estático gerado pelo compilador no qual o código da função anônima é colocado:

```
C#  
  
class Test  
{  
    static void F() {  
        D d = new D(__Method1);  
    }  
}
```

```
    static void __Method1() {
        Console.WriteLine("test");
    }
}
```

No exemplo a seguir, a função anônima faz referência a membros da instância de `this`:

C#

```
class Test
{
    int x;

    void F() {
        D d = () => { Console.WriteLine(x); };
    }
}
```

Isso pode ser convertido em um método de instância gerado pelo compilador que contém o código da função anônima:

C#

```
class Test
{
    int x;

    void F() {
        D d = new D(__Method1);
    }

    void __Method1() {
        Console.WriteLine(x);
    }
}
```

Neste exemplo, a função anônima captura uma variável local:

C#

```
class Test
{
    void F() {
        int y = 123;
        D d = () => { Console.WriteLine(y); };
    }
}
```

O tempo de vida da variável local agora deve ser estendido para pelo menos o tempo de vida do delegado da função anônima. Isso pode ser obtido por meio da "guindaste" da variável local em um campo de uma classe gerada pelo compilador. A instanciação da variável local ([instanciação de variáveis locais](#)) corresponde à criação de uma instância da classe gerada pelo compilador e o acesso à variável local corresponde ao acesso a um campo na instância da classe gerada pelo compilador. Além disso, a função anônima torna-se um método de instância da classe gerada pelo compilador:

C#

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.y = 123;
        D d = new D(__locals1.__Method1);
    }

    class __Locals1
    {
        public int y;

        public void __Method1() {
            Console.WriteLine(y);
        }
    }
}
```

Por fim, a função anônima a seguir captura `this`, bem como duas variáveis locais com tempos de vida diferentes:

C#

```
class Test
{
    int x;

    void F() {
        int y = 123;
        for (int i = 0; i < 10; i++) {
            int z = i * 2;
            D d = () => { Console.WriteLine(x + y + z); };
        }
    }
}
```

Aqui, uma classe gerada pelo compilador é criada para cada bloco de instrução no qual os locais são capturados, de modo que os locais nos diferentes blocos possam ter

tempos de vida independentes. Uma instância do `_Locals2`, a classe gerada pelo compilador para o bloco de instrução interna, contém a variável local `z` e um campo que faz referência a uma instância do `_Locals1`. Uma instância do `_Locals1`, a classe gerada pelo compilador para o bloco de instrução externa, contém a variável local `y` e um campo que faz referência `this` ao membro da função de circunscrição. Com essas estruturas de dados, é possível alcançar todas as variáveis externas capturadas por meio de uma instância do `_Locals2` e o código da função anônima pode, portanto, ser implementado como um método de instância dessa classe.

```
C#  
  
class Test  
{  
    void F() {  
        _Locals1 __locals1 = new _Locals1();  
        __locals1._this = this;  
        __locals1.y = 123;  
        for (int i = 0; i < 10; i++) {  
            _Locals2 __locals2 = new _Locals2();  
            __locals2._locals1 = __locals1;  
            __locals2.z = i * 2;  
            D d = new D(__locals2._Method1);  
        }  
    }  
  
    class __Locals1  
    {  
        public Test __this;  
        public int y;  
    }  
  
    class __Locals2  
    {  
        public __Locals1 __locals1;  
        public int z;  
  
        public void __Method1() {  
            Console.WriteLine(__locals1.__this.x + __locals1.y + z);  
        }  
    }  
}
```

A mesma técnica aplicada aqui para capturar variáveis locais também pode ser usada ao converter funções anônimas em árvores de expressão: referências aos objetos gerados pelo compilador podem ser armazenadas na árvore de expressão e o acesso às variáveis locais pode ser representado como acessos de campo nesses objetos. A vantagem dessa abordagem é que ela permite que as variáveis locais "levantadas" sejam compartilhadas entre delegados e árvores de expressão.

# Conversões de grupo de métodos

Uma conversão implícita ([conversões implícitas](#)) existe de um grupo de métodos ([classificações de expressão](#)) para um tipo de delegado compatível. Dado um tipo delegado  $D$  e uma expressão  $E$  que é classificada como um grupo de métodos, uma conversão implícita existe de  $E$  para  $D$  se  $E$  contiver pelo menos um método aplicável em seu formato normal ([membro de função aplicável](#)) a uma lista de argumentos construída pelo uso dos tipos de parâmetro e modificadores de  $D$ , conforme descrito a seguir.

O aplicativo de tempo de compilação de uma conversão de um grupo de métodos  $E$  para um tipo delegado  $D$  é descrito a seguir. Observe que a existência de uma conversão implícita de  $E$  para não  $D$  garante que o aplicativo de tempo de compilação da conversão terá sucesso sem erros.

- Um único método  $M$  é selecionado, correspondendo a uma invocação de método ([invocações de método](#)) do formulário  $E(A)$ , com as seguintes modificações:
  - A lista  $A$  de argumentos é uma lista de expressões, cada uma classificada como variável e com o tipo e o modificador (`ref` ou `out`) do parâmetro correspondente no *formal\_parameter\_list* de  $D$ .
  - Os métodos candidatos considerados são apenas os métodos que são aplicáveis em seu formato normal ([membro de função aplicável](#)), não aqueles aplicáveis apenas em sua forma expandida.
- Se o algoritmo das [invocações de método](#) produzir um erro, ocorrerá um erro de tempo de compilação. Caso contrário, o algoritmo produz um único método melhor  $M$  com o mesmo número de parâmetros que  $D$  e a conversão é considerada como existente.
- O método selecionado  $M$  deve ser compatível ([delegar compatibilidade](#)) com o tipo delegado  $D$  ou, caso contrário, ocorrerá um erro de tempo de compilação.
- Se o método selecionado  $M$  for um método de instância, a expressão de instância associada a  $E$  determinará o objeto de destino do delegado.
- Se o método selecionado  $M$  for um método de extensão indicado por meio de um acesso de membro em uma expressão de instância, essa expressão de instância determinará o objeto de destino do delegado.
- O resultado da conversão é um valor do tipo  $D$ , ou seja, um delegado recém-criado que se refere ao método selecionado e ao objeto de destino.
- Observe que esse processo pode levar à criação de um delegado para um método de extensão, se o algoritmo das [invocações de método](#) não conseguir encontrar um método de instância, mas tiver êxito no processamento da invocação de  $E(A)$  como uma invocação de método de extensão ([invocações de método de extensão](#)).

[extensão](#)). Um delegado criado então captura o método de extensão, bem como seu primeiro argumento.

O exemplo a seguir demonstra as conversões de grupo de métodos:

C#

```
delegate string D1(object o);

delegate object D2(string s);

delegate object D3();

delegate string D4(object o, params object[] a);

delegate string D5(int i);

class Test
{
    static string F(object o) {...}

    static void G() {
        D1 d1 = F;                // Ok
        D2 d2 = F;                // Ok
        D3 d3 = F;                // Error -- not applicable
        D4 d4 = F;                // Error -- not applicable in normal form
        D5 d5 = F;                // Error -- applicable but not compatible

    }
}
```

A atribuição para `d1` converter implicitamente o grupo de métodos `F` em um valor do tipo `D1`.

A atribuição para `d2` mostra como é possível criar um delegado para um método que tenha tipos de parâmetro menos derivados (contravariant) e um tipo de retorno mais derivado (Covariance).

A atribuição para `d3` mostra como nenhuma conversão existe se o método não for aplicável.

A atribuição para `d4` mostra como o método deve ser aplicável em seu formato normal.

A atribuição para `d5` mostra como os tipos de parâmetro e de retorno do delegado e do método têm permissão para diferir apenas para tipos de referência.

Assim como acontece com todas as outras conversões implícitas e explícitas, o operador cast pode ser usado para executar explicitamente uma conversão de grupo de métodos.

Portanto, o exemplo

```
C#
```

```
object obj = new EventHandler(myDialog.OkClick);
```

em vez disso, poderia ser escrito

```
C#
```

```
object obj = (EventHandler)myDialog.OkClick;
```

Os grupos de métodos podem influenciar a resolução de sobrecarga e participar da inferência de tipos. Consulte [membros da função](#) para obter mais detalhes.

A avaliação de tempo de execução de uma conversão de grupo de métodos procede da seguinte maneira:

- Se o método selecionado em tempo de compilação for um método de instância, ou for um método de extensão que é acessado como um método de instância, o objeto de destino do delegado é determinado da expressão de instância associada a `E` :
  - A expressão de instância é avaliada. Se essa avaliação causar uma exceção, nenhuma etapa adicional será executada.
  - Se a expressão de instância for de um *reference\_type*, o valor calculado pela expressão de instância se tornará o objeto de destino. Se o método selecionado for um método de instância e o objeto de destino for `null`, um `System.NullReferenceException` será lançado e nenhuma etapa adicional será executada.
  - Se a expressão de instância for de um *value\_type*, uma operação Boxing ([conversões Boxing](#)) será executada para converter o valor em um objeto e esse objeto se tornará o objeto de destino.
- Caso contrário, o método selecionado faz parte de uma chamada de método estático e o objeto de destino do delegado é `null` .
- Uma nova instância do tipo delegado `D` é alocada. Se não houver memória suficiente disponível para alocar a nova instância, um `System.OutOfMemoryException` será lançado e nenhuma etapa adicional será executada.
- A nova instância de delegado é inicializada com uma referência ao método que foi determinado em tempo de compilação e uma referência ao objeto de destino calculado acima.

# Expressões

Artigo • 16/09/2021

Uma expressão é uma sequência de operadores e operandos. Este capítulo define a sintaxe, a ordem de avaliação de operandos e operadores e o significado de expressões.

## Classificações de expressão

Uma expressão é classificada como uma das seguintes:

- Um valor. Cada valor tem um tipo associado.
- Uma variável. Cada variável tem um tipo associado, ou seja, o tipo declarado da variável.
- Um namespace. Uma expressão com essa classificação só pode aparecer como o lado esquerdo de uma *member\_access* ([acesso de membro](#)). Em qualquer outro contexto, uma expressão classificada como um namespace causa um erro em tempo de compilação.
- Um tipo. Uma expressão com essa classificação só pode aparecer como o lado esquerdo de uma *member\_access* ([acesso de membro](#)) ou como um operando para o `as` operador ([o operador as](#)), o `is` operador ([o operador is](#)) ou o `typeof` operador ([o operador typeof](#)). Em qualquer outro contexto, uma expressão classificada como um tipo causa um erro de tempo de compilação.
- Um grupo de métodos, que é um conjunto de métodos sobrecarregados resultante de uma pesquisa de membro ([pesquisa de membro](#)). Um grupo de métodos pode ter uma expressão de instância associada e uma lista de argumentos de tipo associado. Quando um método de instância é invocado, o resultado da avaliação da expressão de instância se torna a instância representada por `this` ([esse acesso](#)). Um grupo de métodos é permitido em *um invocation\_expression* ([expressões de invocação](#)), *um delegate\_creation\_expression* ([expressões de criação de delegado](#)) e como o lado esquerdo de um operador `is` e pode ser convertido implicitamente em um tipo de delegado compatível ([conversões de grupo de métodos](#)). Em qualquer outro contexto, uma expressão classificada como um grupo de métodos causa um erro em tempo de compilação.
- Um literal nulo. Uma expressão com essa classificação pode ser convertida implicitamente em um tipo de referência ou tipo anulável.
- Uma função anônima. Uma expressão com essa classificação pode ser convertida implicitamente em um tipo de representante ou tipo de árvore de expressão compatível.

- Um acesso de propriedade. Cada acesso de propriedade tem um tipo associado, ou seja, o tipo da propriedade. Além disso, um acesso de propriedade pode ter uma expressão de instância associada. Quando um acessador (o `get` `set` bloco ou) de um acesso de propriedade de instância é invocado, o resultado da avaliação da expressão de instância se torna a instância representada por `this` ([esse acesso](#)).
- Um acesso de evento. Cada acesso de evento tem um tipo associado, ou seja, o tipo do evento. Além disso, um acesso de evento pode ter uma expressão de instância associada. Um acesso de evento pode aparecer como o operando à esquerda dos `+=` `-=` operadores e ([atribuição de evento](#)). Em qualquer outro contexto, uma expressão classificada como um acesso de evento causa um erro de tempo de compilação.
- Um acesso ao indexador. Cada acesso ao indexador tem um tipo associado, ou seja, o tipo de elemento do indexador. Além disso, um acesso ao indexador tem uma expressão de instância associada e uma lista de argumentos associados. Quando um acessador (o `get` `set` bloco ou) de um acesso do indexador é invocado, o resultado da avaliação da expressão de instância se torna a instância representada pelo `this` ([esse acesso](#)) e o resultado da avaliação da lista de argumentos se torna a lista de parâmetros da invocação.
- Nada. Isso ocorre quando a expressão é uma invocação de um método com um tipo de retorno de `void`. Uma expressão classificada como `Nothing` só é válida no contexto de uma *statement\_expression* ([instruções de expressão](#)).

O resultado final de uma expressão nunca é um namespace, tipo, grupo de métodos ou acesso de evento. Em vez disso, conforme observado acima, essas categorias de expressões são construções intermediárias que só são permitidas em determinados contextos.

Um acesso de propriedade ou de indexador é sempre reclassificado como um valor executando uma invocação do *acessador get* ou o *acessador set*. O acessador específico é determinado pelo contexto da propriedade ou do acesso do indexador: se o acesso for o destino de uma atribuição, o *acessador set* será invocado para atribuir um novo valor ([atribuição simples](#)). Caso contrário, o *acessador get* é invocado para obter o valor atual ([valores de expressões](#)).

## Valores de expressões

A maioria das construções que envolvem uma expressão, por fim, exige que a expressão denotasse um *valor*. Nesses casos, se a expressão real denota um namespace, um tipo, um grupo de métodos ou nada, ocorrerá um erro em tempo de compilação. No entanto, se a expressão denota um acesso de propriedade, um acesso de indexador ou

uma variável, o valor da propriedade, do indexador ou da variável é substituído implicitamente:

- O valor de uma variável é simplesmente o valor atualmente armazenado no local de armazenamento identificado pela variável. Uma variável deve ser considerada definitivamente atribuída ([atribuição definitiva](#)) antes que seu valor possa ser obtido ou, caso contrário, ocorrerá um erro de tempo de compilação.
- O valor de uma expressão de acesso à propriedade é obtido invocando o *acessador get* da propriedade. Se a propriedade não tiver nenhum *acessador get*, ocorrerá um erro em tempo de compilação. Caso contrário, uma invocação de membro de função ([verificação de tempo de compilação da resolução dinâmica de sobrecarga](#)) será executada e o resultado da invocação se tornará o valor da expressão de acesso à propriedade.
- O valor de uma expressão de acesso do indexador é obtido invocando o *acessador get* do indexador. Se o indexador não tiver nenhum *acessador get*, ocorrerá um erro em tempo de compilação. Caso contrário, uma invocação de membro de função ([verificação de tempo de compilação da resolução dinâmica de sobrecarga](#)) é executada com a lista de argumentos associada à expressão de acesso do indexador e o resultado da invocação se torna o valor da expressão de acesso do indexador.

## Associação estática e dinâmica

O processo de determinar o significado de uma operação com base no tipo ou valor de expressões constituintes (argumentos, operandos, receptores) é geralmente conhecido como **Associação**. Por exemplo, o significado de uma chamada de método é determinado com base no tipo de receptor e argumentos. O significado de um operador é determinado com base no tipo de seus operandos.

Em C#, o significado de uma operação geralmente é determinado em tempo de compilação, com base no tipo de tempo de compilação de suas expressões constituintes. Da mesma forma, se uma expressão contiver um erro, o erro será detectado e relatado pelo compilador. Essa abordagem é conhecida como **associação estática**.

No entanto, se uma expressão for uma expressão dinâmica (ou seja, tiver o tipo `dynamic`), isso indicará que qualquer associação na qual ela participa deve ser baseada em seu tipo de tempo de execução (ou seja, o tipo real do objeto que ele denota em tempo de execução) em vez do tipo que ele tem em tempo de compilação. A associação de tal operação é, portanto, adiada até o momento em que a operação deve ser executada durante a execução do programa. Isso é conhecido como **associação dinâmica**.

Quando uma operação é vinculada dinamicamente, pouca ou nenhuma verificação é executada pelo compilador. Em vez disso, se a associação de tempo de execução falhar, os erros serão relatados como exceções em tempo de execução.

As seguintes operações em C# estão sujeitas à associação:

- Acesso de membro: `e.M`
- Invocação de método: `e.M(e1, ..., eN)`
- Invocação de delegado: `e(e1, ..., eN)`
- Acesso ao elemento: `e[e1, ..., eN]`
- Criação de objeto: `new C(e1, ..., eN)`
- Operadores unários sobrecarregados: `+ - ! ~ ++ -- true , false`
- Operadores binários sobrecarregados: `+, -, *, /, %, &, &&, |, ||, ??, ^, <<`  
`>>, ==, !=, >, <, >=, <=`
- Operadores de atribuição: `=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`
- Conversões implícitas e explícitas

Quando não há expressões dinâmicas envolvidas, o C# assume como padrão a associação estática, o que significa que os tipos de tempo de compilação de expressões constituintes são usados no processo de seleção. No entanto, quando uma das expressões constituintes nas operações listadas acima é uma expressão dinâmica, a operação é vinculada dinamicamente.

## Tempo de associação

A associação estática ocorre no tempo de compilação, enquanto a associação dinâmica ocorre em tempo de execução. Nas seções a seguir, o termo **Associação de tempo refere-** se ao tempo de compilação ou tempo de execução, dependendo de quando a associação ocorre.

O exemplo a seguir ilustra as noções de associação estática e dinâmica e de tempo de vinculação:

```
C#  
  
object o = 5;  
dynamic d = 5;  
  
Console.WriteLine(5); // static binding to Console.WriteLine(int)  
Console.WriteLine(o); // static binding to Console.WriteLine(object)  
Console.WriteLine(d); // dynamic binding to Console.WriteLine(int)
```

As duas primeiras chamadas são vinculadas estaticamente: a sobrecarga de `Console.WriteLine` é escolhida com base no tipo de tempo de compilação de seu argumento. Portanto, o tempo de associação é de tempo de compilação.

A terceira chamada é vinculada dinamicamente: a sobrecarga de `Console.WriteLine` é escolhida com base no tipo de tempo de execução de seu argumento. Isso acontece porque o argumento é uma expressão dinâmica – seu tipo de tempo de compilação é `dynamic`. Portanto, o tempo de associação para a terceira chamada é tempo de execução.

## Associação dinâmica

A finalidade da vinculação dinâmica é permitir que programas em C# interajam com **objetos dinâmicos**, ou seja, objetos que não seguem as regras normais do sistema de tipos do C#. Objetos dinâmicos podem ser objetos de outras linguagens de programação com sistemas de tipos diferentes, ou podem ser objetos que são configurados programaticamente para implementar sua própria semântica de ligação para operações diferentes.

O mecanismo pelo qual um objeto dinâmico implementa sua própria semântica é a implementação definida. Uma determinada interface – novamente definida--é implementada por objetos dinâmicos para sinalizar para o tempo de execução do C# que eles têm semântica especial. Portanto, sempre que as operações em um objeto dinâmico forem vinculadas dinamicamente, sua própria semântica de ligação, em vez das do C#, conforme especificado neste documento, assumirá.

Embora a finalidade da vinculação dinâmica seja permitir a interoperabilidade com objetos dinâmicos, o C# permite a vinculação dinâmica em todos os objetos, sejam eles dinâmicos ou não. Isso permite uma integração mais suave de objetos dinâmicos, pois os resultados das operações neles não podem ser objetos dinâmicos, mas ainda são de um tipo desconhecido para o programador em tempo de compilação. Além disso, a ligação dinâmica pode ajudar a eliminar códigos baseados em reflexão propensos a erros, mesmo quando nenhum objeto envolvido é um objeto dinâmico.

As seções a seguir descrevem cada construção no idioma exatamente quando a vinculação dinâmica é aplicada, qual é a verificação de tempo de compilação – se houver, se houver, e qual será o resultado de tempo de compilação e a classificação de expressão.

## Tipos de expressões constituintes

Quando uma operação é vinculada estaticamente, o tipo de uma expressão constituinte (por exemplo, um receptor, um argumento, um índice ou um operando) é sempre considerado o tipo de tempo de compilação dessa expressão.

Quando uma operação é vinculada dinamicamente, o tipo de uma expressão constituinte é determinado de maneiras diferentes, dependendo do tipo de tempo de compilação da expressão constituinte:

- Uma expressão constituinte de tipo de tempo de compilação `dynamic` é considerada como tendo o tipo do valor real que a expressão avalia em tempo de execução
- Uma expressão constituinte cujo tipo de tempo de compilação é um parâmetro de tipo é considerado como tendo o tipo ao qual o parâmetro de tipo está associado em tempo de execução
- Caso contrário, a expressão constituinte será considerada com seu tipo de tempo de compilação.

## Operadores

As expressões são construídas de *operandos e operadores*\*.. Os operadores de uma expressão indicam quais operações devem ser aplicadas aos operandos. Exemplos de operadores incluem `+`, `-`, `*`, `/` e `new`. Exemplos de operandos incluem literais, campos, variáveis locais e expressões.

Há três tipos de operadores:

- Operadores unários. Os operadores unários usam um operando e usam a notação de prefixo (como `--x`) ou a notação de sufixo (como `x++`).
- Operadores binários. Os operadores binários usam dois operandos e todos usam notação de infixo (como `x + y`).
- Operador ternário. Há apenas um operador ternário, `? :`, ele usa três operandos e usa a notação de infixo (`c ? x : y`).

A ordem de avaliação dos operadores em uma expressão é determinada pela *precedência e Associação* dos operadores ([precedência de operador e Associação](#)).

Os operandos em uma expressão são avaliados da esquerda para a direita. Por exemplo, no `F(i) + G(i++) * H(i)`, o método `F` é chamado usando o valor antigo de `i`, então `G` o método é chamado com o valor antigo de `i` e, finalmente, `H` o método é chamado com o novo valor de `i`. Isso é separado de e não relacionado à precedência de operador.

Determinados operadores podem ser *sobre carregados*. A sobrecarga de operador permite que as implementações de operador definidas pelo usuário sejam especificadas para operações em que um ou ambos os operandos são de uma classe definida pelo usuário ou tipo struct ([sobrecarga de operador](#)).

## Precedência e associatividade do operador

Quando uma expressão contém vários operadores, a \*precedência \_ dos operadores controla a ordem na qual os operadores individuais são avaliados. Por exemplo, a expressão `x + y * z` é avaliada como `x + (y * z)` porque o operador `*` tem precedência maior do que o operador binário `+`. A precedência de um operador é estabelecida pela definição da sua produção de gramática associada. Por exemplo, um *additive\_expression* consiste em uma sequência de *multiplicative\_expressions* separados por `+` ou `-` operadores or, dando, assim, os operadores `+` e a `-` precedência mais baixa do que os `*` / `/` operadores, e `%`.

A tabela a seguir resume todos os operadores em ordem de precedência do mais alto para o mais baixo:

Seção	Categoria	Operadores
Expressões primárias	Primário	<code>x.y f(x) a[x] x++ x-- new typeof default checked unchecked delegate</code>
Operadores unários	Unário	<code>+ - ! ~ ++x --x (T)x</code>
Operadores aritméticos	Multiplicativo	<code>* / %</code>
Operadores aritméticos	Aditiva	<code>+</code>
Operadores shift	Shift	<code>&lt;&lt; &gt;&gt;</code>
Operadores de teste de tipo e relacional	Teste de tipo e relacional	<code>&lt; &gt; &lt;= &gt;= is as</code>
Operadores de teste de tipo e relacional	Igualitário	<code>== !=</code>
Operadores lógicos	AND lógico	<code>&amp;</code>
Operadores lógicos	XOR lógico	<code>^</code>
Operadores lógicos	OR lógico	<code> </code>
Operadores lógicos condicionais	AND condicional	<code>&amp;&amp;</code>
Operadores lógicos condicionais	OR condicional	<code>  </code>

Seção	Categoria	Operadores
O operador de coalescência nula	Coalescência nula	??
Operador condicional	Condisional	?:
Operadores de atribuição, expressões de função anônimas	Atribuição e expressão lambda	= *= /= %= += -= <=> &= ^=  = =>

Quando ocorre um operando entre dois operadores com a mesma precedência, a associatividade dos operadores controla a ordem na qual as operações são executadas:

- Exceto para os operadores de atribuição e o operador de União nulo, todos os operadores binários são *associativos à esquerda*, o que significa que as operações são executadas da esquerda para a direita. Por exemplo, `x + y + z` é avaliado como `(x + y) + z`.
- Os operadores de atribuição, o operador de União nulo e o operador condicional (`?:`) são *associativos à direita*, o que significa que as operações são executadas da direita para a esquerda. Por exemplo, `x = y = z` é avaliado como `x = (y = z)`.

Precedência e associatividade podem ser controladas usando parênteses. Por exemplo, `x + y * z` primeiro multiplica `y` por `z` e, em seguida, adiciona o resultado a `x`, mas `(x + y) * z` primeiro adiciona `x` e `y` e, em seguida, multiplica o resultado por `z`.

## Sobrecarga de operador

Todos os operadores unários e binários têm implementações predefinidas que estão automaticamente disponíveis em qualquer expressão. Além das implementações predefinidas, as implementações definidas pelo usuário podem ser introduzidas com a inclusão de `operator` declarações em classes e Structs ([operadores](#)). Implementações de operador definidas pelo usuário sempre têm precedência sobre implementações de operador predefinidas: somente quando não houver nenhuma implementação de operador definida pelo usuário aplicável as implementações de operador predefinidas serão consideradas, conforme descrito em resolução de [sobrecarga de operador unário](#) e [resolução de sobrecarga de operador binário](#).

Os *operadores unários sobrecarregados* são:

C#

```
+ - ! ~ ++ -- true false
```

Embora `true` e `false` não sejam usados explicitamente em expressões (e, portanto, não sejam incluídos na tabela de precedência em [precedência de operador e Associação](#)), eles são considerados operadores porque são invocados em vários contextos de expressão: expressões booleanas ([expressões booleanas](#)) e expressões que envolvem o condicional ([operador condicional](#)) e operadores lógicos condicionais ([operadores lógicos condicionais](#)).

Os *operadores binários sobrecarregados* são:

C#

```
+ - * / % & | ^ << >> == != > < >= <=
```

Somente os operadores listados acima podem ser sobrecarregados. Em particular, não é possível sobrecarregar o acesso de membro, a invocação de método ou os operadores `=`, `&&`, `||`, `??`, `? :`, `=>`, `checked`, `unchecked`, `new`, `typeof`, `default`, `as` e `is`.

Quando um operador binário está sobrecarregado, o operador de atribuição correspondente, se houver, também estará implicitamente sobrecarregado. Por exemplo, uma sobrecarga de operador `*` também é uma sobrecarga de operador `*=`. Isso é descrito mais detalhadamente na [atribuição composta](#). Observe que o próprio operador de atribuição (`=`) não pode ser sobrecarregado. Uma atribuição sempre executa uma cópia de bits simples de um valor em uma variável.

As operações de conversão, como `(T)x`, são sobrecarregadas fornecendo conversões definidas pelo usuário ([conversões definidas pelo usuário](#)).

O acesso ao elemento, como `a[x]`, não é considerado um operador sobrecarregado. Em vez disso, há suporte para a indexação definida pelo usuário por meio de indexadores ([indexadores](#)).

Em expressões, os operadores são referenciados usando a notação de operador e, em declarações, os operadores são referenciados usando a notação funcional. A tabela a seguir mostra a relação entre as notações de operador e funcional para operadores unários e binários. Na primeira entrada, *op* denota qualquer operador de prefixo unário sobrecarregado. Na segunda entrada, *op* denota o sufixo e os operadores unários `++` e `-`. Na terceira entrada, *op* denota qualquer operador binário sobrecarregado.

Notação de operador	Notação funcional
<code>op x</code>	<code>operator op(x)</code>
<code>x op</code>	<code>operator op(x)</code>

Notação de operador	Notação funcional
<code>x op y</code>	<code>operator op(x,y)</code>

As declarações de operador definidas pelo usuário sempre exigem que pelo menos um dos parâmetros seja do tipo de classe ou struct que contém a declaração do operador. Portanto, não é possível que um operador definido pelo usuário tenha a mesma assinatura que um operador predefinido.

As declarações do operador definidas pelo usuário não podem modificar a sintaxe, a precedência ou a associação de um operador. Por exemplo, o `/` operador é sempre um operador binário, sempre tem o nível de precedência especificado em [precedência de operador e Associação](#), e é sempre associativo à esquerda.

Embora seja possível que um operador definido pelo usuário execute qualquer computação, as implementações que produzem resultados diferentes daqueles que são intuitivos esperados são altamente desencorajadas. Por exemplo, uma implementação do `operator ==` deve comparar os dois operandos para igualdade e retornar um `bool` resultado apropriado.

As descrições de operadores individuais em [expressões primárias](#) por meio de [operadores lógicos condicionais](#) especificam as implementações predefinidas dos operadores e quaisquer regras adicionais que se aplicam a cada operador. As descrições usam os termos [\*unários de sobrecarga de operador resolução\*](#) \_, [\*resolução de sobrecarga de operador binário\\*\*](#) \_ e [\*promoção numérica \\*\*](#), definições das quais são encontradas nas seções a seguir.

## Resolução de sobrecarga de operador unário

Uma operação do formulário `op x` ou `x op`, em que `op` é um operador unário sobrecarregado e `x` é uma expressão do tipo `x`, é processada da seguinte maneira:

- O conjunto de operadores candidatos definidos pelo usuário fornecido pelo `x` para a operação `operator op(x)` é determinado usando as regras de [operadores definidos pelo usuário candidatos](#).
- Se o conjunto de operadores candidatos definidos pelo usuário não estiver vazio, isso se tornará o conjunto de operadores candidatos para a operação. Caso contrário, as implementações unários predefinidas `operator op`, incluindo suas formas levantadas, se tornarão o conjunto de operadores candidatos para a operação. As implementações predefinidas de um determinado operador são especificadas na descrição do operador ([expressões primárias](#) e [operadores unários](#)).

- As regras de resolução de sobrecarga da [resolução de sobrecarga](#) são aplicadas ao conjunto de operadores candidatos para selecionar o melhor operador em relação à lista de argumentos `(x)`, e esse operador se torna o resultado do processo de resolução de sobrecarga. Se a resolução de sobrecarga falhar ao selecionar um único operador melhor, ocorrerá um erro de tempo de associação.

## Resolução de sobrecarga de operador binário

Uma operação do formulário `x op y`, em que `op` é um operador binário sobrecarregado, `x` é uma expressão do tipo `X` e `y` é uma expressão do tipo `Y`, é processada da seguinte maneira:

- O conjunto de operadores candidatos definidos pelo usuário fornecido pelo `x` e `y` para a operação `operator op(x,y)` é determinado. O conjunto consiste na União dos operadores candidatos fornecidos pelo `x` e nos operadores candidatos fornecidos pelo `y`, cada um determinado usando as regras de [operadores definidos pelo usuário candidatos](#). Se `x` e `y` forem do mesmo tipo, ou se `x` e `y` forem derivados de um tipo base comum, os operadores de candidato compartilhados só ocorrerão no conjunto combinado uma vez.
- Se o conjunto de operadores candidatos definidos pelo usuário não estiver vazio, isso se tornará o conjunto de operadores candidatos para a operação. Caso contrário, as implementações binárias predefinidas `operator op`, incluindo seus formulários levantados, se tornarão o conjunto de operadores candidatos para a operação. As implementações predefinidas de um determinado operador são especificadas na descrição do operador ([operadores aritméticos](#) por meio de [operadores lógicos condicionais](#)). Para operadores enum e delegate predefinidos, os únicos operadores considerados são aqueles definidos por um tipo de enumeração ou de delegado que é o tipo de tempo de associação de um dos operandos.
- As regras de resolução de sobrecarga da [resolução de sobrecarga](#) são aplicadas ao conjunto de operadores candidatos para selecionar o melhor operador em relação à lista de argumentos `(x,y)`, e esse operador se torna o resultado do processo de resolução de sobrecarga. Se a resolução de sobrecarga falhar ao selecionar um único operador melhor, ocorrerá um erro de tempo de associação.

## Operadores definidos pelo usuário candidatos

Dado um tipo `T` e uma operação `operator op(A)`, em que `op` é um operador que é sobrecarregado e `A` é uma lista de argumentos, o conjunto de operadores candidatos

definidos pelo usuário fornecido pelo `for T operator op(A)` é determinado da seguinte maneira:

- Determine o tipo `T0`. Se `T` for um tipo anulável, `T0` será seu tipo subjacente; caso contrário, `T0` será igual a `T`.
- Para todas as `operator op` declarações em `T0` e todas as formas compostas desses operadores, se pelo menos um operador for aplicável ([membro de função aplicável](#)) em relação à lista de argumentos `A`, o conjunto de operadores candidatos consistirá em todos esses operadores aplicáveis no `T0`.
- Caso contrário, se `T0` for `object`, o conjunto de operadores candidatos estará vazio.
- Caso contrário, o conjunto de operadores candidatos fornecidos pelo `T0` é o conjunto de operadores candidatos fornecidos pela classe base direta de `T0`, ou a classe base efetiva de `T0`. If `T0` é um parâmetro de tipo.

## Promoções numéricas

A promoção numérica consiste em executar automaticamente determinadas conversões implícitas dos operandos dos operadores numéricos unários e binários predefinidos. A promoção numérica não é um mecanismo distinto, mas sim um efeito de aplicar a resolução de sobrecarga aos operadores predefinidos. A promoção numérica especificamente não afeta a avaliação de operadores definidos pelo usuário, embora os operadores definidos pelo usuário possam ser implementados para exibir efeitos semelhantes.

Como exemplo de promoção numérica, considere as implementações predefinidas do `*` operador binário:

C#

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

Quando as regras de resolução de sobrecarga ([resolução de sobrecarga](#)) são aplicadas a esse conjunto de operadores, o efeito é selecionar o primeiro dos operadores para os quais conversões implícitas existem nos tipos de operando. Por exemplo, para a operação `b * s`, em que `b` é um `byte` e `s` é um `short`, a resolução de sobrecarga

seleciona `operator *(int,int)` como o melhor operador. Assim, o efeito é que `b` e `s` são convertidos em `int`, e o tipo do resultado é `int`. Da mesma forma, para a operação `i * d`, em que `i` é um `int` e `d` é um `double`, a resolução de sobrecarga seleciona `operator *(double,double)` como o melhor operador.

## Promoções numéricas unários

A promoção numérica unário ocorre para os operandos dos operadores predefinidos `+`, `-` e `~` unários. A promoção numérica unário consiste simplesmente em converter operandos do tipo `sbyte`, `byte`, `short`, `ushort` ou `char` para o tipo `int`. Além disso, para o `-` operador unário, a promoção numérica unário converte os operandos do tipo `uint` para o tipo `long`.

## Promoções numéricas binárias

A promoção numérica binária ocorre para os operandos dos operadores predefinidos `..... + - * / % & | ^ == != > < , >= & <=` binários. A promoção numérica binária converte implicitamente os dois operandos em um tipo comum que, no caso dos operadores não relacionais, também se torna o tipo de resultado da operação. A promoção numérica binária consiste na aplicação das seguintes regras, na ordem em que aparecem aqui:

- Se qualquer operando for do tipo `decimal`, o outro operando será convertido em tipo `decimal` ou um erro de tempo de ligação ocorrerá se o outro operando for do tipo `float` ou `double`.
- Caso contrário, se qualquer operando for do tipo `double`, o outro operando será convertido em tipo `double`.
- Caso contrário, se qualquer operando for do tipo `float`, o outro operando será convertido em tipo `float`.
- Caso contrário, se qualquer operando for do tipo `ulong`, o outro operando será convertido em tipo `ulong`, ou um erro de tempo de ligação ocorrerá se o outro operando for do tipo `sbyte`, `short`, `int` ou `long`.
- Caso contrário, se qualquer operando for do tipo `long`, o outro operando será convertido em tipo `long`.
- Caso contrário, se qualquer operando for do tipo `uint` e o outro operando for do tipo `sbyte`, `short` ou `int`, ambos os operandos serão convertidos para o tipo `long`.
- Caso contrário, se qualquer operando for do tipo `uint`, o outro operando será convertido em tipo `uint`.

- Caso contrário, ambos os operandos serão convertidos para o tipo `int`.

Observe que a primeira regra não permite qualquer operação que misture o `decimal` tipo com os `double` tipos e `float`. A regra segue do fato de que não há conversões implícitas entre o `decimal` tipo e os `double` tipos e `float`.

Observe também que não é possível que um operando seja do tipo `ulong` quando o outro operando é de um tipo integral assinado. O motivo é que não existe nenhum tipo integral que possa representar o intervalo completo `ulong`, bem como os tipos inteiros assinados.

Em ambos os casos acima, uma expressão de conversão pode ser usada para converter explicitamente um operando em um tipo que seja compatível com o outro operando.

No exemplo

C#

```
decimal AddPercent(decimal x, double percent) {
    return x * (1.0 + percent / 100.0);
}
```

um erro de tempo de associação ocorre porque um `decimal` não pode ser multiplicado por um `double`. O erro é resolvido convertendo explicitamente o segundo operando para `decimal`, da seguinte maneira:

C#

```
decimal AddPercent(decimal x, double percent) {
    return x * (decimal)(1.0 + percent / 100.0);
}
```

## Operadores levantados

**Operadores levantados** permitem operadores predefinidos e definidos pelo usuário que operam em tipos de valores não anuláveis para também serem usados com formulários anuláveis desses tipos. Os operadores levantados são construídos a partir de operadores predefinidos e definidos pelo usuário que atendem a determinados requisitos, conforme descrito no seguinte:

- Para os operadores unários

C#

```
+  ++  -  --  !  ~
```

uma forma levantada de um operador existirá se o operando e os tipos de resultado forem tipos de valor não anuláveis. O formulário levantado é construído adicionando um único `?`  modificador ao operando e tipos de resultado. O operador levantado produz um valor nulo se o operando for nulo. Caso contrário, o operador levantado desencapsula o operando, aplica o operador subjacente e encapsula o resultado.

- Para os operadores binários

```
C#
```

```
+  -  *  /  %  &  |  ^  <<  >>
```

Existe uma forma levantada de um operador se os tipos de operando e de resultado são todos os tipos de valor não anuláveis. O formulário levantado é construído adicionando um único `?`  modificador a cada operando e tipo de resultado. O operador levantado produz um valor nulo se um ou ambos os operandos forem nulos (uma exceção sendo os `&` `|`  operadores e do `bool?` tipo, conforme descrito em [operadores lógicos booleanos](#)). Caso contrário, o operador levantado desencapsula os operandos, aplica o operador subjacente e encapsula o resultado.

- Para os operadores de igualdade

```
C#
```

```
==  !=
```

uma forma levantada de um operador existe se os tipos de operando forem tipos de valor não anuláveis e se o tipo de resultado for `bool`. O formulário levantado é construído adicionando um único `?`  modificador a cada tipo de operando. O operador levantado considera dois valores nulos iguais e um valor nulo é diferente de qualquer valor não nulo. Se ambos os operandos forem não nulos, o operador levantado desencapsulará os operandos e aplicará o operador subjacente para produzir o `bool` resultado.

- Para os operadores relacionais

```
C#
```

```
<  >  <=  >=
```

uma forma levantada de um operador existe se os tipos de operando forem tipos de valor não anuláveis e se o tipo de resultado for `bool`. O formulário levantado é construído adicionando um único `?`  modificador a cada tipo de operando. O operador levantado produz o valor `false` se um ou ambos os operandos forem nulos. Caso contrário, o operador levantado desencapsula os operandos e aplica o operador subjacente para produzir o `bool` resultado.

## Pesquisa de membros

Uma pesquisa de membro é o processo pelo qual o significado de um nome no contexto de um tipo é determinado. Uma pesquisa de membro pode ocorrer como parte da avaliação de um *Simple\_name* ([nomes simples](#)) ou uma *member\_access* ([acesso de membro](#)) em uma expressão. Se o *Simple\_name* ou *member\_access* ocorrer como o *primary\_expression* de um *invocation\_expression* ([invocações de método](#)), o membro será chamado de chamada.

Se um membro for um método ou evento, ou se for uma constante, campo ou propriedade de um tipo delegado ([delegados](#)) ou do tipo `dynamic` ([o tipo dinâmico](#)), o membro será dito como *invocável*.

A pesquisa de membros considera não apenas o nome de um membro, mas também o número de parâmetros de tipo que o membro tem e se o membro está acessível. Para fins de pesquisa de membros, os métodos genéricos e os tipos genéricos aninhados têm o número de parâmetros de tipo indicados em suas respectivas declarações e todos os outros membros têm zero parâmetros de tipo.

Uma pesquisa de membro de um nome `N` com `K` parâmetros de tipo em um tipo `T` é processada da seguinte maneira:

- Primeiro, um conjunto de membros acessíveis chamado `N` é determinado:
  - Se `T` for um parâmetro de tipo, o conjunto será a União dos conjuntos de membros acessíveis nomeados `N` em cada um dos tipos especificados como uma restrição primária ou restrição secundária ([restrições de parâmetro de tipo](#)) para `T`, juntamente com o conjunto de membros acessíveis nomeados `N` em `object`.
  - Caso contrário, o conjunto consiste em todos os membros ([acesso de membro](#)) acessíveis chamados `N` no `T`, incluindo membros herdados e os membros acessíveis chamados `N` em `object`. Se `T` for um tipo construído, o conjunto de membros será obtido pela substituição de argumentos de tipo, conforme

descrito em [membros de tipos construídos](#). Os membros que incluem um `override` modificador são excluídos do conjunto.

- Em seguida, se `K` for zero, todos os tipos aninhados cujas declarações incluem parâmetros de tipo serão removidos. Se `K` não for zero, todos os membros com um número diferente de parâmetros de tipo serão removidos. Observe que quando `K` é zero, os métodos com parâmetros de tipo não são removidos, pois o processo de inferência de tipos ([inferência de tipos](#)) pode ser capaz de inferir os argumentos de tipo.
- Em seguida, se o membro for *invocado*, todos os membros não *invocáveis* serão removidos do conjunto.
- Em seguida, os membros que estão ocultos por outros membros são removidos do conjunto. Para cada membro `S.M` no conjunto, em que `S` é o tipo no qual o membro `M` é declarado, as regras a seguir são aplicadas:
  - Se `M` for uma constante, campo, propriedade, evento ou membro de enumeração, todos os membros declarados em um tipo base de `S` serão removidos do conjunto.
  - Se `M` for uma declaração de tipo, todos os não tipos declarados em um tipo base de `S` serão removidos do conjunto, e todas as declarações de tipo com o mesmo número de parâmetros de tipo, conforme `M` declarado em um tipo base de, `S` serão removidas do conjunto.
  - Se `M` for um método, todos os membros não-método declarados em um tipo base de `S` serão removidos do conjunto.
- Em seguida, os membros de interface ocultos por membros de classe são removidos do conjunto. Essa etapa só terá efeito se `T` for um parâmetro de tipo e `T` tiver uma classe base em vigor diferente de `object` e um conjunto de interfaces efetivas não vazias ([restrições de parâmetro de tipo](#)). Para cada membro `S.M` no conjunto, em que `S` é o tipo no qual o membro `M` é declarado, as regras a seguir serão aplicadas se `S` for uma declaração de classe diferente de `object` :
  - Se `M` for uma constante, campo, propriedade, evento, membro de enumeração ou declaração de tipo, todos os membros declarados em uma declaração de interface serão removidos do conjunto.
  - Se `M` for um método, todos os membros que não forem do método declarados em uma declaração de interface serão removidos do conjunto e todos os métodos com a mesma assinatura `M` declarados em uma declaração de interface serão removidos do conjunto.
- Finalmente, ter removido Membros ocultos, o resultado da pesquisa é determinado:
  - Se o conjunto consistir em um único membro que não seja um método, esse membro será o resultado da pesquisa.

- o Caso contrário, se o conjunto contiver apenas métodos, esse grupo de métodos será o resultado da pesquisa.
- o Caso contrário, a pesquisa será ambígua e ocorrerá um erro de tempo de vinculação.

Para pesquisas de membros em tipos diferentes de parâmetros de tipo e interfaces, e pesquisas de membros em interfaces que são estritamente herança única (cada interface na cadeia de herança tem exatamente zero ou uma interface de base direta), o efeito das regras de pesquisa é simplesmente que os membros derivados ocultam os membros de base com o mesmo nome ou assinatura. Essas pesquisas de herança única nunca são ambíguas. As ambiguidades que podem surgir em relação a pesquisas de membros em interfaces de várias heranças são descritas em [acesso de membro de interface](#).

## Tipos base

Para fins de pesquisa de membro, é considerado que um tipo `T` tem os seguintes tipos base:

- Se `T` for `object`, `T` não terá nenhum tipo base.
- Se `T` for um `enum_type`, os tipos de base do `T` são os tipos de classe `System.Enum`, `System.ValueType` e `object`.
- Se `T` for um `struct_type`, os tipos de base do `T` são os tipos de classe `System.ValueType` e `object`.
- Se `T` for um `class_type`, os tipos base do `T` são as classes base do `T`, incluindo o tipo de classe `object`.
- Se `T` for um `interface_type`, os tipos de base do `T` são as interfaces base do `T` e o tipo de classe `object`.
- Se `T` for um `array_type`, os tipos de base do `T` são os tipos de classe `System.Array` e `object`.
- Se `T` for um `delegate_type`, os tipos de base do `T` são os tipos de classe `System.Delegate` e `object`.

## Membros da função

Membros de função são membros que contêm instruções Executáveis. Membros de função são sempre membros de tipos e não podem ser membros de namespaces. O C# define as seguintes categorias de membros de função:

- Métodos

- Propriedades
- Eventos
- Indexadores
- Operadores definidos pelo usuário
- Construtores de instância
- Construtores estáticos
- Destruidores

Exceto para destruidores e construtores estáticos (que não podem ser invocados explicitamente), as instruções contidas nos membros da função são executadas por meio de invocações de membro de função. A sintaxe real para gravar uma invocação de membro de função depende da categoria de membro de função específica.

A lista de argumentos ([listas de argumentos](#)) de uma invocação de membro de função fornece valores reais ou referências de variáveis para os parâmetros do membro da função.

Invocações de métodos genéricos podem empregar inferência de tipos para determinar o conjunto de argumentos de tipo a serem passados para o método. Esse processo é descrito em [inferência de tipos](#).

Invocações de métodos, indexadores, operadores e construtores de instância empregam resolução de sobrecarga para determinar qual de um conjunto candidato de membros de função invocar. Esse processo é descrito em [resolução de sobrecarga](#).

Depois que um membro de função específico tiver sido identificado no tempo de vinculação, possivelmente por meio da resolução de sobrecarga, o processo real de tempo de execução de invocação do membro da função será descrito em [verificação de tempo de compilação da resolução dinâmica de sobrecarga](#).

A tabela a seguir resume o processamento que ocorre em construções envolvendo as seis categorias de membros de função que podem ser invocadas explicitamente. Na tabela, `e`, `x` e `y` e `value` indicam expressões classificadas como variáveis ou valores, `T` indica uma expressão classificada como um tipo, `F` é o nome simples de um método e `P` é o nome simples de uma propriedade.

Constructo	Exemplo	Descrição
Invocação de método	<code>F(x,y)</code>	A resolução de sobrecarga é aplicada para selecionar o melhor método <code>F</code> na classe ou struct que o contém. O método é invocado com a lista de argumentos <code>(x,y)</code> . Se o método não for <code>static</code> , a expressão de instância será <code>this</code> .

Constructo	Exemplo	Descrição
	<code>T.F(x,y)</code>	A resolução de sobrecarga é aplicada para selecionar o melhor método <code>F</code> na classe ou estrutura <code>T</code> . Ocorrerá um erro de tempo de ligação se o método não for <code>static</code> . O método é invocado com a lista de argumentos <code>(x,y)</code> .
	<code>e.F(x,y)</code>	A resolução de sobrecarga é aplicada para selecionar o melhor método <code>F</code> na classe, struct ou interface fornecida pelo tipo de <code>e</code> . Um erro de tempo de associação ocorrerá se o método for <code>static</code> . O método é invocado com a expressão de instância <code>e</code> e a lista de argumentos <code>(x,y)</code> .
Acesso à propriedade	<code>P</code>	O <code>get</code> acessador da propriedade <code>P</code> na classe ou struct que a contém é invocado. Ocorrerá um erro de tempo de compilação se <code>P</code> for somente gravação. Se <code>P</code> não for <code>static</code> , a expressão de instância será <code>this</code> .
	<code>P = value</code>	O <code>set</code> acessador da propriedade <code>P</code> na classe ou struct que a contém é invocado com a lista de argumentos <code>(value)</code> . Ocorrerá um erro de tempo de compilação se <code>P</code> for somente leitura. Se <code>P</code> não for <code>static</code> , a expressão de instância será <code>this</code> .
	<code>T.P</code>	O <code>get</code> acessador da propriedade <code>P</code> na classe ou struct <code>T</code> é invocado. Um erro de tempo de compilação ocorre se <code>P</code> não for <code>static</code> ou se <code>P</code> for somente gravação.
	<code>T.P = value</code>	O <code>set</code> acessador da propriedade <code>P</code> na classe ou struct <code>T</code> é invocado com a lista de argumentos <code>(value)</code> . Um erro de tempo de compilação ocorre se <code>P</code> não for <code>static</code> ou se <code>P</code> for somente leitura.
	<code>e.P</code>	O <code>get</code> acessador da propriedade <code>P</code> na classe, struct ou interface fornecida pelo tipo de <code>e</code> é invocado com a expressão de instância <code>e</code> . Um erro de tempo de associação ocorrerá se <code>P</code> for <code>static</code> ou se <code>P</code> for somente gravação.
	<code>e.P = value</code>	O <code>set</code> acessador da propriedade <code>P</code> na classe, struct ou interface fornecida pelo tipo de <code>e</code> é invocado com a expressão de instância <code>e</code> e a lista de argumentos <code>(value)</code> . Um erro de tempo de associação ocorrerá se <code>P</code> for <code>static</code> ou se <code>P</code> for somente leitura.
Acesso a eventos	<code>E += value</code>	O <code>add</code> acessador do evento <code>E</code> na classe ou struct que o contém é invocado. Se <code>E</code> não for estático, a expressão de instância será <code>this</code> .
	<code>E -= value</code>	O <code>remove</code> acessador do evento <code>E</code> na classe ou struct que o contém é invocado. Se <code>E</code> não for estático, a expressão de instância será <code>this</code> .
	<code>T.E += value</code>	O <code>add</code> acessador do evento <code>E</code> na classe ou struct <code>T</code> é invocado. Ocorrerá um erro de tempo de ligação se <code>E</code> não for estático.

Constructo	Exemplo	Descrição
	<code>T.E -= value</code>	O <code>remove</code> acessador do evento <code>E</code> na classe ou struct <code>T</code> é invocado. Ocorrerá um erro de tempo de ligação se <code>E</code> não for estático.
	<code>e.E += value</code>	O <code>add</code> acessador do evento <code>E</code> na classe, struct ou interface fornecida pelo tipo de <code>e</code> é invocado com a expressão de instância <code>e</code> . Ocorrerá um erro de tempo de associação se <code>E</code> for estático.
	<code>e.E -= value</code>	O <code>remove</code> acessador do evento <code>E</code> na classe, struct ou interface fornecida pelo tipo de <code>e</code> é invocado com a expressão de instância <code>e</code> . Ocorrerá um erro de tempo de associação se <code>E</code> for estático.
Acesso de indexador	<code>e[x,y]</code>	A resolução de sobrecarga é aplicada para selecionar o melhor indexador na classe, struct ou interface fornecida pelo tipo de <code>e</code> . O <code>get</code> acessador do indexador é invocado com a expressão de instância <code>e</code> e a lista de argumentos <code>(x,y)</code> . Ocorrerá um erro de tempo de ligação se o indexador for somente gravação.
	<code>e[x,y] = value</code>	A resolução de sobrecarga é aplicada para selecionar o melhor indexador na classe, struct ou interface fornecida pelo tipo de <code>e</code> . O <code>set</code> acessador do indexador é invocado com a expressão de instância <code>e</code> e a lista de argumentos <code>(x,y,value)</code> . Ocorrerá um erro de tempo de ligação se o indexador for somente leitura.
Invocação de operador	<code>-x</code>	A resolução de sobrecarga é aplicada para selecionar o melhor operador unário na classe ou struct fornecido pelo tipo de <code>x</code> . O operador selecionado é invocado com a lista de argumentos <code>(x)</code> .
	<code>x + y</code>	A resolução de sobrecarga é aplicada para selecionar o melhor operador binário nas classes ou estruturas dadas pelos tipos de <code>x</code> e <code>y</code> . O operador selecionado é invocado com a lista de argumentos <code>(x,y)</code> .
Invocação de construtor de instância	<code>new T(x,y)</code>	A resolução de sobrecarga é aplicada para selecionar o melhor Construtor de instância na classe ou estrutura <code>T</code> . O construtor de instância é invocado com a lista de argumentos <code>(x,y)</code> .

## Listas de argumentos

Cada membro de função e invocação de delegado inclui uma lista de argumentos que fornece valores reais ou referências de variáveis para os parâmetros do membro da função. A sintaxe para especificar a lista de argumentos de uma invocação de membro de função depende da categoria de membro da função:

- Para construtores de instância, métodos, indexadores e delegados, os argumentos são especificados como um *argument\_list*, conforme descrito abaixo. Para

indexadores, ao invocar o `set` acessador, a lista de argumentos também inclui a expressão especificada como o operando à direita do operador de atribuição.

- Para propriedades, a lista de argumentos está vazia ao invocar o `get` acessador e consiste na expressão especificada como o operando à direita do operador de atribuição ao invocar o `set` acessador.
- Para eventos, a lista de argumentos consiste na expressão especificada como o operando à direita do `+ = - =` operador OR.
- Para operadores definidos pelo usuário, a lista de argumentos consiste no único operando do operador unário ou nos dois operandos do operador binário.

Os argumentos de Propriedades ([Propriedades](#)), eventos ([eventos](#)) e operadores definidos pelo usuário ([operadores](#)) são sempre passados como parâmetros de valor ([parâmetros de valor](#)). Os argumentos de indexadores ([indexadores](#)) são sempre passados como parâmetros de valor ([parâmetros de valor](#)) ou matrizes de parâmetro ([matrizes de parâmetro](#)). Parâmetros de referência e saída não têm suporte para essas categorias de membros de função.

Os argumentos de um construtor de instância, método, indexador ou invocação de delegado são especificados como um *argument\_list*:

```
antlr

argument_list
  : argument (',' argument)*
  ;

argument
  : argument_name? argument_value
  ;

argument_name
  : identifier ':'
  ;

argument_value
  : expression
  | 'ref' variable_reference
  | 'out' variable_reference
  ;
```

Um *argument\_list* consiste em um ou mais *s de argumento*, separados por vírgulas. Cada argumento consiste em um *argument\_name* opcional seguido por um *argument\_value*.

Um *argumento* com um *argument\_name* é chamado de um **argumento nomeado**, enquanto que um *argumento* \* sem um *argument\_name* é um **argumento posicional**\*.. É

um erro para que um argumento posicional apareça após um argumento nomeado em um `_argument_list` \*.

O `argument_value` pode ter um dos seguintes formatos:

- Uma *expressão*, indicando que o argumento é passado como um parâmetro de valor ([parâmetros de valor](#)).
- A palavra-chave `ref` seguida por uma `variable_reference` ([referências de variáveis](#)), indicando que o argumento é passado como um parâmetro de referência ([parâmetros de referência](#)). Uma variável deve ser definitivamente atribuída ([atribuição definitiva](#)) antes de poder ser passada como um parâmetro de referência. A palavra-chave `out` seguida por uma `variable_reference` ([referências de variáveis](#)), indicando que o argumento é passado como um parâmetro de saída ([parâmetros de saída](#)). Uma variável é considerada definitivamente atribuída ([atribuição definitiva](#)) após uma invocação de membro de função na qual a variável é passada como um parâmetro de saída.

## Parâmetros correspondentes

Para cada argumento em uma lista de argumentos, deve haver um parâmetro correspondente no membro da função ou no delegado que está sendo invocado.

A lista de parâmetros usada no seguinte é determinada como a seguir:

- Para métodos virtuais e indexadores definidos em `classes`, a lista de parâmetros é escolhida da declaração mais específica ou substituição do membro da função, começando com o tipo estático do receptor e pesquisando suas `classes base`.
- Para métodos de interface e indexadores, a lista de parâmetros é escolhida para formar a definição mais específica do membro, começando com o tipo de interface e pesquisando as `interfaces base`. Se nenhuma lista de parâmetros exclusiva for encontrada, uma lista de parâmetros com nomes inacessíveis e nenhum parâmetro opcional será construído, de modo que as invocações não poderão usar parâmetros nomeados nem omitir argumentos opcionais.
- Para métodos parciais, a lista de parâmetros da declaração de método parcial de definição é usada.
- Para todos os outros membros de função e delegados, há apenas uma única lista de parâmetros, que é a usada.

A posição de um argumento ou parâmetro é definida como o número de argumentos ou parâmetros que o antecedem na lista de argumentos ou na lista de parâmetros.

Os parâmetros correspondentes para argumentos de membro de função são estabelecidos da seguinte maneira:

- Argumentos na *argument\_list* de construtores de instância, métodos, indexadores e delegados:
  - Um argumento posicional em que um parâmetro fixo ocorre na mesma posição na lista de parâmetros corresponde a esse parâmetro.
  - Um argumento posicional de um membro de função com uma matriz de parâmetros invocada em seu formato normal corresponde à matriz de parâmetros, que deve ocorrer na mesma posição na lista de parâmetros.
  - Um argumento posicional de um membro de função com uma matriz de parâmetro invocada em sua forma expandida, em que nenhum parâmetro fixo ocorre na mesma posição na lista de parâmetros, corresponde a um elemento na matriz de parâmetros.
  - Um argumento nomeado corresponde ao parâmetro de mesmo nome na lista de parâmetros.
  - Para indexadores, ao invocar o `set` acessador, a expressão especificada como o operando à direita do operador de atribuição corresponde ao `value` parâmetro implícito da `set` declaração de acessador.
- Para propriedades, ao invocar o `get` acessador, não há argumentos. Ao invocar o `set` acessador, a expressão especificada como o operando à direita do operador de atribuição corresponde ao `value` parâmetro implícito da `set` declaração de acessador.
- Para operadores unários definidos pelo usuário (incluindo conversões), o único operando corresponde ao parâmetro único da declaração do operador.
- Para operadores binários definidos pelo usuário, o operando esquerdo corresponde ao primeiro parâmetro e o operando direito corresponde ao segundo parâmetro da declaração do operador.

## Avaliação de tempo de execução de listas de argumentos

Durante o processamento de tempo de execução de uma invocação de membro de função ([verificação de tempo de compilação da resolução dinâmica de sobrecarga](#)), as expressões ou referências de variáveis de uma lista de argumentos são avaliadas na ordem, da esquerda para a direita, da seguinte maneira:

- Para um parâmetro de valor, a expressão de argumento é avaliada e uma conversão implícita ([conversões implícitas](#)) para o tipo de parâmetro correspondente é executada. O valor resultante torna-se o valor inicial do parâmetro `value` na invocação de membro da função.
- Para um parâmetro de referência ou saída, a referência de variável é avaliada e o local de armazenamento resultante torna-se o local de armazenamento representado pelo parâmetro na invocação de membro da função. Se a referência

de variável fornecida como um parâmetro de referência ou de saída for um elemento de matriz de um *reference\_type*, uma verificação de tempo de execução será executada para garantir que o tipo de elemento da matriz seja idêntico ao tipo do parâmetro. Se essa verificação falhar, um `System.ArrayTypeMismatchException` será lançado.

Métodos, indexadores e construtores de instância podem declarar seu parâmetro mais à direita para ser uma matriz de parâmetros ([matrizes de parâmetros](#)). Esses membros de função são invocados em seu formato normal ou em sua forma expandida, dependendo do que é aplicável ([membro de função aplicável](#)):

- Quando um membro de função com uma matriz de parâmetros é invocado em seu formato normal, o argumento fornecido para a matriz de parâmetros deve ser uma única expressão que é implicitamente conversível ([conversões implícitas](#)) para o tipo de matriz de parâmetros. Nesse caso, a matriz de parâmetros funciona precisamente como um parâmetro de valor.
- Quando um membro de função com uma matriz de parâmetros é invocado em seu formato expandido, a invocação deve especificar zero ou mais argumentos posicionais para a matriz de parâmetros, em que cada argumento é uma expressão que é implicitamente conversível ([conversões implícitas](#)) para o tipo de elemento da matriz de parâmetros. Nesse caso, a invocação cria uma instância do tipo de matriz de parâmetros com um comprimento correspondente ao número de argumentos, inicializa os elementos da instância de matriz com os valores de argumento fornecidos e usa a instância de matriz recém-criada como o argumento real.

As expressões de uma lista de argumentos são sempre avaliadas na ordem em que são gravadas. Portanto, o exemplo

C#

```
class Test
{
    static void F(int x, int y = -1, int z = -2) {
        System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }

    static void Main() {
        int i = 0;
        F(i++, i++, i++);
        F(z: i++, x: i++);
    }
}
```

produz a saída

Console

```
x = 0, y = 1, z = 2
x = 4, y = -1, z = 3
```

As regras de covariância de matriz ([covariância de matriz](#)) permitem que um valor de um tipo de matriz `A[]` seja uma referência a uma instância de um tipo de matriz `B[]`, desde que exista uma conversão de referência implícita de `B` para `A`. Devido a essas regras, quando um elemento de matriz de um *reference\_type* é passado como um parâmetro de referência ou de saída, uma verificação de tempo de execução é necessária para garantir que o tipo de elemento real da matriz seja idêntico ao do parâmetro. No exemplo

C#

```
class Test
{
    static void F(ref object x) {...}

    static void Main() {
        object[] a = new object[10];
        object[] b = new string[10];
        F(ref a[0]);           // Ok
        F(ref b[1]);           // ArrayTypeMismatchException
    }
}
```

a segunda invocação de `F` faz com que uma seja `System.ArrayTypeMismatchException` gerada porque o tipo de elemento real de `b` é `string` e não `object`.

Quando um membro de função com uma matriz de parâmetros é invocado em seu formato expandido, a invocação é processada exatamente como se uma expressão de criação de matriz com um inicializador de matriz ([expressões de criação de matriz](#)) foi inserida em volta dos parâmetros expandidos. Por exemplo, dada a declaração

C#

```
void F(int x, int y, params object[] args);
```

as seguintes invocações da forma expandida do método

C#

```
F(10, 20);
F(10, 20, 30, 40);
```

```
F(10, 20, 1, "hello", 3.0);
```

corresponder exatamente a

C#

```
F(10, 20, new object[] {});  
F(10, 20, new object[] {30, 40});  
F(10, 20, new object[] {1, "hello", 3.0});
```

Em particular, observe que uma matriz vazia é criada quando há zero argumentos fornecidos para a matriz de parâmetros.

Quando os argumentos são omitidos de um membro de função com parâmetros opcionais correspondentes, os argumentos padrão da declaração de membro de função são passados implicitamente. Como elas são sempre constantes, sua avaliação não afetará a ordem de avaliação dos argumentos restantes.

## Inferência de tipos

Quando um método genérico é chamado sem especificar argumentos de tipo, um processo de *inferência de tipos* tenta inferir argumentos de tipo para a chamada. A presença da inferência de tipos permite que uma sintaxe mais conveniente seja usada para chamar um método genérico e permite que o programador Evite especificar informações de tipo redundante. Por exemplo, dada a declaração do método:

C#

```
class Chooser  
{  
    static Random rand = new Random();  
  
    public static T Choose<T>(T first, T second) {  
        return (rand.Next(2) == 0)? first: second;  
    }  
}
```

é possível invocar o `Choose` método sem especificar explicitamente um argumento de tipo:

C#

```
int i = Chooser.Choose(5, 213);           // Calls Choose<int>  
  
string s = Chooser.Choose("foo", "bar");     // Calls Choose<string>
```

Por meio da inferência de tipos, os argumentos de tipo `int` e `string` são determinados dos argumentos para o método.

A inferência de tipos ocorre como parte do processamento do tempo de associação de uma invocação de método ([invocações de método](#)) e ocorre antes da etapa de resolução de sobrecarga da invocação. Quando um grupo de métodos específico é especificado em uma invocação de método e nenhum argumento de tipo é especificado como parte da invocação de método, a inferência de tipos é aplicada a cada método genérico no grupo de métodos. Se a inferência de tipos for realizada com sucesso, os argumentos de tipo deduzidos serão usados para determinar os tipos de argumentos para resolução de sobrecarga subsequente. Se a resolução de sobrecarga escolher um método genérico como o para invocar, os argumentos de tipo inferido serão usados como os argumentos de tipo real para a invocação. Se a inferência de tipos para um determinado método falhar, esse método não participará da resolução de sobrecarga. A falha da inferência de tipos, por si só, não causa um erro de tempo de ligação. No entanto, geralmente leva a um erro de tempo de ligação quando a resolução de sobrecarga não consegue encontrar nenhum método aplicável.

Se o número de argumentos fornecido for diferente do número de parâmetros no método, a inferência falhará imediatamente. Caso contrário, suponha que o método genérico tenha a seguinte assinatura:

C#

```
Tr M<X1, ..., Xn>(T1 x1, ..., Tm xm)
```

Com uma chamada de método do formulário `M(E1...Em)`, a tarefa da inferência de tipos é encontrar argumentos `S1...Sn` de tipo exclusivo para cada um dos parâmetros de tipo `X1...Xn` para que a chamada `M<S1...Sn>(E1...Em)` se torne válida.

Durante o processo de inferência, cada parâmetro `xi` de tipo é *corrigido* para um tipo específico `si` ou não é *corrigido* com um conjunto associado de *limites*. Cada um dos limites é algum tipo `T`. Inicialmente, cada variável `xi` de tipo não é corrigida com um conjunto vazio de limites.

A inferência de tipos ocorre em fases. Cada fase tentará inferir argumentos de tipo para mais variáveis de tipo com base nas conclusões da fase anterior. A primeira fase faz algumas inferências iniciais de limites, enquanto a segunda fase corrige variáveis de tipo para tipos específicos e infere limites adicionais. A segunda fase pode ter que ser repetida várias vezes.

*Observação:* A inferência de tipos ocorre não apenas quando um método genérico é chamado. A inferência de tipos para a conversão de grupos de métodos é descrita na [inferência de tipos para a conversão de grupos de métodos](#) e a localização do melhor tipo comum de um conjunto de expressões é descrita na [localização do melhor tipo comum de um conjunto de expressões](#).

## A primeira fase

Para cada um dos argumentos do método  $E_i$  :

- Se  $E_i$  for uma função anônima, uma *inferência de tipo de parâmetro explícita* ([induções de tipo de parâmetro explícita](#)) será feita de  $E_i$  para  $T_i$
- Caso contrário, se  $E_i$  tiver um tipo  $U$  e  $x_i$  for um parâmetro de valor, uma *inferência de limite inferior* será feita de  $U$  para  $T_i$  .
- Caso contrário, se  $E_i$  tiver um tipo  $U$  e  $x_i$  for um `ref` parâmetro ou `out` , uma *inferência exata* será feita de  $U$  para  $T_i$  .
- Caso contrário, nenhuma inferência é feita para esse argumento.

## A segunda fase

A segunda fase continua da seguinte maneira:

- Todas as variáveis de tipo não *fixas*  $x_i$  que não *dependem de* ([dependência](#))  $x_j$  são corrigidas ([corrigindo](#)).
- Se nenhuma dessas variáveis de tipo existir, todas as variáveis de tipo não *fixas*  $x_i$  serão *corrigidas* para cada uma das seguintes isenções:
  - Há pelo menos uma variável de tipo  $x_j$  que depende de  $x_i$
  - $x_i$  tem um conjunto não vazio de limites
- Se nenhuma das variáveis de tipo existirem e ainda houver variáveis de tipo não *fixas* , a inferência de tipos falhará.
- Caso contrário, se nenhuma variável de tipo não *fixa* existir, a inferência de tipos terá sucesso.
- Caso contrário, para todos os argumentos  $E_i$  com o tipo de parâmetro correspondente,  $T_i$  em que os *tipos de saída* ([tipos de saída](#)) contêm variáveis de tipo não *fixas*  $x_j$  , mas os *tipos de entrada* ([tipos de entrada](#)) não, uma *inferência de tipo de saída* ([inferências de tipo de saída](#)) é feita de  $E_i$  para  $T_i$  . Em seguida, a segunda fase é repetida.

## Tipos de entrada

Se  $E$  for um grupo de métodos ou uma função anônima digitada implicitamente e  $T$  for um tipo delegado ou tipo de árvore de expressão, todos os tipos de parâmetro de  $T$  serão *tipos de entrada* de  $E$  com o tipo  $T$ .

## Tipos de saída

Se  $E$  for um grupo de métodos ou uma função anônima e  $T$  for um tipo delegado ou tipo de árvore de expressão, o tipo de retorno de  $T$  será um *tipo de saída*  $E$  com o tipo  $T$ .

## Dependência

Uma variável de tipo não fixo  $x_i$  depende diretamente de uma variável de tipo não fixa  $x_j$  se, para algum argumento  $E_k$  com tipo  $T_k$ ,  $x_j$  ocorrer em um *tipo de entrada* de  $E_k$  com tipo  $T_k$  e  $x_i$  ocorrer em um *tipo de saída* de  $E_k$  com tipo  $T_k$ .

$x_j$  depende de  $x_i$ . Se  $x_j$  depende diretamente do  $x_i$  ou se  $x_i$  depende diretamente de  $x_k$  e  $x_k$  depende de  $x_j$ . Assim, "depende" é o fechamento transitivo, mas não reflexivo, de "depende diretamente de".

## Inferências de tipo de saída

Uma *inferência de tipo de saída* é feita de uma expressão  $E$  para um tipo  $T$  da seguinte maneira:

- Se  $E$  é uma função anônima com tipo de retorno inferido  $U$  ([tipo de retorno inferido](#)) e  $T$  é um tipo delegado ou tipo de árvore de expressão com tipo de retorno  $T_b$ , uma *inferência de limite inferior* ([inferências de limite inferior](#)) é feita de  $U$  para  $T_b$ .
- Caso contrário, se  $E$  for um grupo de métodos e  $T$  for um tipo delegado ou tipo de árvore de expressão com tipos de parâmetro  $T_1 \dots T_k$  e tipo  $T_b$  de retorno, e a resolução de sobrecarga de  $E$  com os tipos  $T_1 \dots T_k$  gerar um único método com tipo de retorno  $U$ , uma *inferência de limite inferior* será feita de  $U$  para  $T_b$ .
- Caso contrário, se  $E$  for uma expressão com tipo  $U$ , uma *inferência de limite inferior* será feita de  $U$  para  $T$ .
- Caso contrário, nenhuma inferência será feita.

## Inferências de tipo de parâmetro explícito

Uma *inferência de tipo de parâmetro explícita* é feita de uma expressão  $E$  para um tipo  $T$  da seguinte maneira:

- Se  $E$  é uma função anônima explicitamente tipada com tipos de parâmetro  $U_1 \dots U_k$  e  $T$  é um tipo delegado ou tipo de árvore de expressão com tipos  $V_1 \dots V_k$  de parâmetro, então cada  $U_i$  uma das *inferências exatas* é feita de  $U_i$  para o correspondente  $V_i$ .

## Inferências exatas

Uma *inferência exata* de um tipo  $U$  para um tipo  $V$  é feita da seguinte maneira:

- Se  $V$  for um dos não corrigidos  $X_i$ ,  $U$  será adicionado ao conjunto de limites exatos para  $X_i$ .
- Caso contrário, os conjuntos  $V_1 \dots V_k$  e  $U_1 \dots U_k$  são determinados verificando se algum dos casos a seguir se aplica:
  - $V$  é um tipo  $V_1[\dots]$  de matriz e  $U$  é um tipo  $U_1[\dots]$  de matriz da mesma classificação
  - $V$  é o tipo  $V_1?$  e  $U$  é o tipo  $U_1?$
  - $V$  é um tipo construído  $c < V_1 \dots V_k >$  e  $U$  é um tipo construído  $c < U_1 \dots U_k >$

Se qualquer um desses casos se aplicar, uma *inferência exata* será feita de cada  $U_i$  para o correspondente  $V_i$ .

- Caso contrário, nenhuma inferência será feita.

## Inferências de limite inferior

Uma *inferência de limite inferior* de um tipo  $U$  para um tipo  $V$  é feita da seguinte maneira:

- Se  $V$  for um dos não corrigidos  $X_i$ ,  $U$  será adicionado ao conjunto de limites inferiores para  $X_i$ .
- Caso contrário, se  $V$  for o tipo  $V_1?$  e  $U$  for o tipo  $U_1?$ , uma inferência de limite inferior será feita de  $U_1$  para  $V_1$ .
- Caso contrário, os conjuntos  $U_1 \dots U_k$  e  $V_1 \dots V_k$  são determinados verificando se algum dos casos a seguir se aplica:

- $v$  é um tipo de matriz  $v_1[\dots]$  e  $u$  é um tipo  $u_1[\dots]$  de matriz (ou um parâmetro de tipo cujo tipo base efetivo é  $u_1[\dots]$ ) da mesma classificação
- $v$  é um de `IEnumerable<V1>`, `ICollection<V1>` ou `IList<V1>` e  $u$  é um tipo de matriz unidimensional  $u_1[]$  (ou um parâmetro de tipo cujo tipo base é efetivo  $u_1[]$ )
- $v$  é uma classe construída, struct, interface ou tipo delegado  $c<v_1\dots v_k>$  e há um tipo exclusivo  $c<u_1\dots u_k>$ , de modo que  $u$  (ou, se  $u$  for um parâmetro de tipo, sua classe base efetiva ou qualquer membro de seu conjunto de interface efetivo) seja idêntico a, herda de (direta ou indiretamente) ou implementa (direta ou indiretamente)  $c<u_1\dots u_k>$ .

(A restrição de "exclusividade" significa que, na interface do caso `c<T> {} class U: c<X>, c<Y> {}`, nenhuma inferência é feita quando se faz referência  $U$  a, `c<T>` porque  $U$  poderia ser  $X$  ou  $Y$ .)

Se qualquer um desses casos se aplicar, uma inferência será feita *de cada  $u_i$  para o correspondente da seguinte  $v_i$*  maneira:

- Se  $u_i$  não for conhecido como um tipo de referência, uma *inferência exata* será feita
- Caso contrário, se  $u$  for um tipo de matriz, uma *inferência de limite inferior* será feita
- Caso contrário, se  $v$  for  $c<v_1\dots v_k>$ , a inferência dependerá do parâmetro de tipo  $i$ -th de  $c$  :
  - Se for covariant, uma *inferência de limite inferior* será feita.
  - Se for contravariant, uma *inferência de limite superior* será feita.
  - Se for invariável, uma *inferência exata* será feita.
- Caso contrário, nenhuma inferência será feita.

## Inferências de limite superior

Uma *inferência de limite superior* de um tipo  $u$  para um tipo  $v$  é feita da seguinte maneira:

- Se  $v$  for um dos não corrigidos  $x_i$ ,  $u$  será adicionado ao conjunto de limites superiores para  $x_i$ .
- Caso contrário, os conjuntos  $v_1\dots v_k$  e  $u_1\dots u_k$  são determinados verificando se algum dos casos a seguir se aplica:

- $U$  é um tipo  $U_1[\dots]$  de matriz e  $V$  é um tipo  $V_1[\dots]$  de matriz da mesma classificação
- $U$  é um de `IEnumerable<Ue>`, `ICollection<Ue>` ou `IList<Ue>` e  $V$  é um tipo de matriz unidimensional  $V_e[]$
- $U$  é o tipo  $U_1?$  e  $V$  é o tipo  $V_1?$
- $U$  é classe construída, struct, interface ou tipo delegado  $C<U_1\dots U_k>$  e  $V$  é uma classe, struct, interface ou tipo delegado, que é idêntica a, herda de (direta ou indiretamente) ou implementa (direta ou indiretamente) um tipo exclusivo  $C<V_1\dots V_k>$

(A restrição de "exclusividade" significa que, se tivermos `interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{}`, nenhuma inferência será feita ao se inferir de `C<U1>` para `V<Q>`. As inferências não são feitas no `U1 X<Q>` ou no `Y<Q>`.)

Se qualquer um desses casos se aplicar, uma inferência será feita de cada  $U_i$  para o correspondente da seguinte  $V_i$  maneira:

- Se  $U_i$  não for conhecido como um tipo de referência, uma *inferência exata* será feita
- Caso contrário, se  $V$  for um tipo de matriz, uma *inferência de limite superior* será feita
- Caso contrário, se  $U$  for  $C<U_1\dots U_k>$ , a inferência dependerá do parâmetro de tipo i-th de  $C$  :
  - Se for covariant, uma *inferência de limite superior* será feita.
  - Se for contravariant, uma *inferência de limite inferior* será feita.
  - Se for invariável, uma *inferência exata* será feita.
- Caso contrário, nenhuma inferência será feita.

## Resolvendo

Uma variável  $x_i$  de tipo não fixo com um conjunto de limites é *fixa* da seguinte maneira:

- O conjunto de *tipos candidatos*  $U_j$  começa como o conjunto de todos os tipos no conjunto de limites para  $x_i$ .
- Em seguida, examinamos cada associado por  $x_i$  vez: para cada limite exato  $U$  de  $x_i$  todos os tipos  $U_j$  que não são idênticos a  $U$  são removidos do conjunto candidato. Para cada limite inferior  $U$  de  $x_i$  todos os tipos  $U_j$  para os quais *não*

há uma conversão implícita de `U` são removidos do conjunto candidato. Para cada limite superior `U` de `Xi` todos os tipos `Uj` dos quais não há uma conversão implícita a ser `U` removida do conjunto de candidatos.

- Se entre os tipos candidatos restantes `Uj`, há um tipo exclusivo `V` do qual há uma conversão implícita para todos os outros tipos candidatos e, em seguida, `Xi` é corrigido para `V`.
- Caso contrário, a inferência de tipos falhará.

## Tipo de retorno inferido

O tipo de retorno inferido de uma função anônima `F` é usado durante a inferência de tipos e resolução de sobrecarga. O tipo de retorno inferido só pode ser determinado para uma função anônima em que todos os tipos de parâmetro são conhecidos, seja porque eles são explicitamente fornecidos, fornecidos por meio de uma conversão de função anônima ou inferido durante a inferência de tipos em uma invocação de método genérico delimitador.

O *tipo de resultado deduzido* é determinado da seguinte maneira:

- Se o corpo de `F` for uma *expressão* que tem um tipo, o tipo de resultado inferido de `F` será o tipo dessa expressão.
- Se o corpo de `F` for um *bloco* e o conjunto de expressões nas instruções do bloco `return` tiver um tipo mais comum `T` ([encontrando o melhor tipo comum de um conjunto de expressões](#)), o tipo de resultado inferido de `F` será `T`.
- Caso contrário, não será possível inferir um tipo de resultado para `F`.

O *tipo de retorno inferido* é determinado da seguinte maneira:

- Se `F` é `Async` e o corpo de `F` é uma expressão classificada como `Nothing` ([classificações de expressão](#)) ou um bloco de instruções em que nenhuma instrução de retorno tem expressões, o tipo de retorno inferido é `System.Threading.Tasks.Task`
- Se `F` for `Async` e tiver um tipo de resultado inferido `T`, o tipo de retorno inferido será `System.Threading.Tasks.Task<T>`.
- Se `F` for não `Async` e tiver um tipo de resultado inferido `T`, o tipo de retorno inferido será `T`.
- Caso contrário, não será possível inferir um tipo de retorno para `F`.

Como exemplo de inferência de tipos envolvendo funções anônimas, considere o `Select` método de extensão declarado na `System.Linq.Enumerable` classe:

C#

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TResult> Select<TSource, TResult>(
            this IEnumerable<TSource> source,
            Func<TSource, TResult> selector)
        {
            foreach (TSource element in source) yield return
            selector(element);
        }
    }
}
```

Supondo que o `System.Linq` namespace tenha sido importado com uma `using` cláusula e dada uma classe `Customer` com uma `Name` Propriedade do tipo `string`, o `Select` método pode ser usado para selecionar os nomes de uma lista de clientes:

C#

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

A invocação do método de extensão ([invocações de método de extensão](#)) do `Select` é processada pela regravação da invocação para uma invocação de método estático:

C#

```
IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);
```

Como os argumentos de tipo não foram especificados explicitamente, a inferência de tipos é usada para inferir os argumentos de tipo. Primeiro, o `customers` argumento está relacionado ao `source` parâmetro, inferindo-se `T` a ser `Customer`. Em seguida, o uso do processo de inferência de tipo de função anônima descrito acima `c` é fornecido `Customer`, e a expressão `c.Name` está relacionada ao tipo de retorno do `selector` parâmetro, inferindo-se `S` a ser `string`. Portanto, a invocação é equivalente a

C#

```
Sequence.Select<Customer, string>(customers, (Customer c) => c.Name)
```

e o resultado é do tipo `IEnumerable<string>`.

O exemplo a seguir demonstra como a inferência de tipo de função anônima permite que informações de tipo "Flow" entrem argumentos em uma invocação de método genérico. Dado o método:

```
C#  
  
static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {  
    return f2(f1(value));  
}
```

Inferência de tipos para a invocação:

```
C#  
  
double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);
```

continua da seguinte maneira: primeiro, o argumento `"1:15:30"` está relacionado ao `value` parâmetro, inferindo-se `X` a ser `string`. Em seguida, o parâmetro da primeira função anônima, `s`, recebe o tipo inferido `string`, e a expressão `TimeSpan.Parse(s)` está relacionada ao tipo de retorno `f1`, inferindo-se `Y` a `System.TimeSpan`. Por fim, o parâmetro da segunda função anônima, `t`, recebe o tipo inferido `System.TimeSpan`, e a expressão `t.TotalSeconds` está relacionada ao tipo de retorno `f2`, inferindo-se `Z` a `double`. Assim, o resultado da invocação é do tipo `double`.

## Inferência de tipos para conversão de grupos de métodos

Semelhante às chamadas de métodos genéricos, a inferência de tipos também deve ser aplicada quando um grupo de método `M` que contém um método genérico é convertido em um determinado tipo de delegado `D` ([conversões de grupo de métodos](#)). Dado um método

```
C#  
  
Tr M<X1...Xn>(T1 x1 ... Tm xm)
```

e o grupo de métodos `M` sendo atribuído ao tipo delegado `D` a tarefa de tipo inferência é encontrar argumentos de tipo `S1...Sn` para que a expressão:

```
C#  
  
M<S1...Sn>
```

torna-se compatível ([delegar declarações](#)) com  $D$ .

Ao contrário do algoritmo de inferência de tipos para chamadas de método genérico, nesse caso, há apenas *tipos* de argumento, sem *expressões* de argumento. Em particular, não existem funções anônimas e, portanto, não há necessidade de várias fases de inferência.

Em vez disso, todos  $x_i$  são considerados não *corrigidos* e uma *inferência de limite inferior* é feita de cada tipo  $U_j$  de argumento de  $D$  para o tipo de parâmetro correspondente  $T_j$  de  $M$ . Se para qualquer um dos  $x_i$  limites não for encontrado, a inferência de tipos falhará. Caso contrário, todos  $x_i$  são *corrigidos* para  $S_i$  o correspondente, que são o resultado da inferência de tipos.

## Encontrando o melhor tipo comum de um conjunto de expressões

Em alguns casos, um tipo comum precisa ser inferido para um conjunto de expressões. Em particular, os tipos de elemento de matrizes tipadas implicitamente e os tipos de retorno de funções anônimas com corpos de *bloco* são encontrados dessa maneira.

Intuitivamente, dado um conjunto de expressões,  $E_1 \dots E_m$  essa inferência deve ser equivalente à chamada de um método

C#

```
Tr M<X>(X x1 ... X xm)
```

pelos  $E_i$  argumentos as.

Mais precisamente, a inferência começa com uma variável de tipo não fixa  $X$ . As *inferências de tipo de saída* são então feitas de cada  $E_i$  para  $X$ . Por fim,  $X$  é fixo e, se for bem-sucedido, o tipo resultante  $S$  será o melhor tipo comum resultante para as expressões. Se não  $S$  existir, as expressões não terão o melhor tipo comum.

## Resolução de sobrecarga

A resolução de sobrecarga é um mecanismo de tempo de ligação para selecionar o melhor membro de função para invocar uma lista de argumentos e um conjunto de membros de funções candidatas. Resolução de sobrecarga seleciona o membro da função a ser invocado nos seguintes contextos distintos em C#:

- Invocação de um método chamado em um *invocation\_expression* ([invocações de método](#)).

- Invocação de um construtor de instância chamado em um *object\_creation\_expression* ([expressões de criação de objeto](#)).
- Invocação de um acessador de indexador por meio de um *element\_access* ([acesso de elemento](#)).
- Invocação de um operador predefinido ou definido pelo usuário referenciado em uma expressão (resolução de [sobrecarga de operador unário](#) e [resolução de sobrecarga de operador binário](#)).

Cada um desses contextos define o conjunto de membros da função candidata e a lista de argumentos de forma exclusiva, conforme descrito em detalhes nas seções listadas acima. Por exemplo, o conjunto de candidatos para uma invocação de método não inclui métodos marcados `override` ([pesquisa de membros](#)) e os métodos em uma classe base não serão candidatos se qualquer método em uma classe derivada for aplicável ([invocações de método](#)).

Depois que os membros da função candidata e a lista de argumentos tiverem sido identificados, a seleção do melhor membro de função será a mesma em todos os casos:

- Dado o conjunto de membros da função candidata aplicável, o melhor membro da função nesse conjunto está localizado. Se o conjunto contiver apenas um membro de função, esse membro de função será o melhor membro de função. Caso contrário, o melhor membro da função é o membro de uma função que é melhor do que todos os outros membros da função em relação à lista de argumentos fornecida, desde que cada membro da função seja comparado com todos os outros membros da função usando as regras no [melhor membro da função](#). Se não houver exatamente um membro da função que seja melhor do que todos os outros membros da função, a invocação do membro da função será ambígua e ocorrerá um erro de tempo de vinculação.

As seções a seguir definem os significados exatos dos termos **\*membro da função aplicável \_ e \_ melhor membro da função \***.

## Membro da função aplicável

Um membro de função é considerado um ***membro de função aplicável*** em relação a uma lista de argumentos `A` quando todos os itens a seguir são verdadeiros:

- Cada argumento em `A` corresponde a um parâmetro na declaração de membro de função, conforme descrito em [parâmetros correspondentes](#), e qualquer parâmetro ao qual nenhum argumento corresponde é um parâmetro opcional.
- Para cada argumento no `A` , o modo de passagem de parâmetro do argumento (ou seja, value `ref` , ou `out` ) é idêntico ao modo de passagem de parâmetro do

parâmetro correspondente e

- para um parâmetro de valor ou uma matriz de parâmetros, existe uma conversão implícita ([conversões implícitas](#)) do argumento para o tipo do parâmetro correspondente ou
- para um `ref` `out` parâmetro ou, o tipo do argumento é idêntico ao tipo do parâmetro correspondente. Afinal, um `ref` parâmetro ou `out` é um alias para o argumento passado.

Para um membro de função que inclui uma matriz de parâmetros, se o membro da função for aplicável pelas regras acima, ele será considerado aplicável em seu **\*formato normal**. Se um membro de função que inclui uma matriz de parâmetros não for aplicável em seu formato normal, o membro da função poderá ser aplicável em seu **formato \_ \* expandido\* \***:

- O formulário expandido é construído pela substituição da matriz de parâmetros na declaração de membro de função por zero ou mais parâmetros de valor do tipo de elemento da matriz de parâmetros, de modo que o número de argumentos na lista de argumentos `A` corresponda ao número total de parâmetros. Se `A` tiver menos argumentos do que o número de parâmetros fixos na declaração de membro da função, a forma expandida do membro da função não poderá ser construída e, portanto, não será aplicável.
- Caso contrário, o formulário expandido será aplicável se para cada argumento no `A` modo de passagem de parâmetro do argumento for idêntico ao modo de passagem de parâmetro do parâmetro correspondente e
  - para um parâmetro de valor fixo ou um parâmetro de valor criado pela expansão, uma conversão implícita ([conversões implícitas](#)) existe do tipo do argumento para o tipo do parâmetro correspondente ou
  - para um `ref` `out` parâmetro ou, o tipo do argumento é idêntico ao tipo do parâmetro correspondente.

## Melhor membro da função

Para fins de determinação do melhor membro de função, uma lista de argumentos desativados `A` é construída contendo apenas as expressões de argumento em si na ordem em que aparecem na lista de argumentos originais.

As listas de parâmetros para cada um dos membros da função candidata são construídos da seguinte maneira:

- O formulário expandido será usado se o membro da função for aplicável somente no formulário expandido.

- Parâmetros opcionais sem argumentos correspondentes são removidos da lista de parâmetros
- Os parâmetros são reordenados para que eles ocorram na mesma posição que o argumento correspondente na lista de argumentos.

Dada uma lista  $A$  de argumentos com um conjunto de expressões de argumento  $\{E_1, E_2, \dots, E_n\}$  e dois membros de função aplicáveis  $M_p$  e  $M_q$  com tipos de parâmetro  $\{P_1, P_2, \dots, P_n\}$  e  $\{Q_1, Q_2, \dots, Q_n\}$ ,  $M_p$  é definido como um **membro de função melhor** do que  $M_q$  se

- para cada argumento, a conversão implícita de  $E_x$  para  $Q_x$  não é melhor do que a conversão implícita de  $E_x$  para  $P_x$  e
- para pelo menos um argumento, a conversão de  $E_x$  para  $P_x$  é melhor do que a conversão de  $E_x$  para  $Q_x$ .

Ao executar essa avaliação, se  $M_p$  ou  $M_q$  for aplicável em sua forma expandida,  $P_x$  ou  $Q_x$  se referir a um parâmetro na forma expandida da lista de parâmetros.

No caso de sequências de tipo de parâmetro  $\{P_1, P_2, \dots, P_n\}$  e  $\{Q_1, Q_2, \dots, Q_n\}$  são equivalentes (ou seja  $P_i$ , cada uma tem uma conversão de identidade para o correspondente  $Q_i$ ), as seguintes regras de quebra de empate são aplicadas, em ordem, para determinar o melhor membro de função.

- Se  $M_p$  for um método não genérico e  $M_q$  for um método genérico,  $M_p$  será melhor do que  $M_q$ .
- Caso contrário, se  $M_p$  é aplicável em seu formato normal e  $M_q$  tem uma `params` matriz e é aplicável apenas em sua forma expandida,  $M_p$  é melhor do que  $M_q$ .
- Caso contrário, se  $M_p$  tiver mais parâmetros declarados do que o  $M_q$ ,  $M_p$  será melhor do que  $M_q$ . Isso pode ocorrer se ambos os métodos tiverem `params` matrizes e se forem aplicáveis somente em suas formas expandidas.
- Caso contrário, se todos os parâmetros de  $M_p$  tiverem um argumento correspondente, enquanto os argumentos padrão precisarão ser substituídos por pelo menos um parâmetro opcional em  $M_q$ ,  $M_p$  é melhor do que  $M_q$ .
- Caso contrário, se  $M_p$  tiver tipos de parâmetro mais específicos do que  $M_q$ ,  $M_p$  será melhor do que  $M_q$ . Permita  $\{R_1, R_2, \dots, R_n\}$  e  $\{S_1, S_2, \dots, S_n\}$  representar os tipos de parâmetro não instanciados e não expandidos de  $M_p$  e  $M_q$ .  $M_p$  os tipos de parâmetro do são mais específicos que o  $M_q$  If, para cada parâmetro,  $R_x$  não é menos específico que  $S_x$  e, para pelo menos um parâmetro,  $R_x$  é mais específico que  $S_x$  :
  - Um parâmetro de tipo é menos específico que um parâmetro sem tipo.

- Recursivamente, um tipo construído é mais específico do que outro tipo construído (com o mesmo número de argumentos de tipo) se pelo menos um argumento de tipo for mais específico e nenhum argumento de tipo for menos específico do que o argumento de tipo correspondente no outro.
- Um tipo de matriz é mais específico do que outro tipo de matriz (com o mesmo número de dimensões) se o tipo de elemento do primeiro for mais específico do que o tipo de elemento do segundo.
- Caso contrário, se um membro for um operador não-comparado e o outro for um operador de aumento, o que não foi levantado será melhor.
- Caso contrário, nenhum membro da função é melhor.

## Melhor conversão de expressão

Dada uma conversão implícita  $c_1$  que converte de uma expressão  $E$  em um tipo  $T_1$  e uma conversão implícita  $c_2$  que converte de uma expressão  $E$  em um tipo  $T_2$ ,  $c_1$  é uma **conversão melhor** do que  $c_2$  se não  $E$  corresponder exatamente  $T_2$  e pelo menos uma das seguintes isenções:

- $E$  correspondências exatas  $T_1$  ([expressão exatamente correspondente](#))
- $T_1$  é um destino de conversão melhor do que  $T_2$  ([melhor destino de conversão](#))

## Expressão exatamente correspondente

Dada uma expressão  $E$  e um tipo  $T$ ,  $E$  corresponde exatamente  $T$  a se uma das seguintes isenções:

- $E$  tem um tipo  $S$  e uma conversão de identidade existe em  $S$  para  $T$
- $E$  é uma função anônima,  $T$  é um tipo delegado  $D$  ou um tipo de árvore de expressão `Expression<D>` e uma das seguintes isenções:
  - Um tipo de retorno inferido  $X$  existe para  $E$  no contexto da lista de parâmetros  $D$  ([tipo de retorno inferido](#)) e uma conversão de identidade existe de  $X$  para o tipo de retorno de  $D$
  - $E$  é não `Async` e  $D$  tem um tipo de retorno  $Y$  ou  $E$  é `Async` e  $D$  tem um tipo de retorno `Task<Y>`, e uma das seguintes contém:
    - O corpo de  $E$  é uma expressão que corresponde exatamente a  $Y$
    - O corpo de  $E$  é um bloco de instruções em que cada instrução de retorno retorna uma expressão que corresponde exatamente a  $Y$

## Melhor destino de conversão

Considerando dois tipos diferentes  $T_1$  e  $T_2$ ,  $T_1$  é um destino de conversão melhor que  $T_2$  do que se não houver conversão implícita de  $T_2$  para  $T_1$ . Existem pelo menos uma das seguintes isenções:

- Uma conversão implícita de  $T_1$  para  $T_2$  existe
- $T_1$  é um tipo delegado  $D_1$  ou um tipo de árvore de expressão  $\text{Expression} < D_1 >$ ,  $T_2$  é um tipo delegado  $D_2$  ou um tipo de árvore de expressão  $\text{Expression} < D_2 >$ ,  $D_1$  tem um tipo  $S_1$  de retorno e uma das seguintes isenções:
  - $D_2$  está anulando retorno
  - $D_2$  tem um tipo de retorno  $S_2$  e  $S_1$  é um destino de conversão melhor do que  $S_2$
- $T_1$  é  $\text{Task} < S_1 >$ ,  $T_2$  é  $\text{Task} < S_2 >$ , e  $S_1$  é um destino de conversão melhor do que  $S_2$
- $T_1$  é  $S_1$  ou  $S_1?$  onde  $S_1$  é um tipo integral assinado e  $T_2$  é  $S_2$  ou  $S_2?$  onde  $S_2$  é um tipo integral não assinado. Especificamente:
  - $S_1$  é `sbyte` e  $S_2$  é `byte`, `ushort`, `uint` ou `ulong`
  - $S_1$  é `short` e  $S_2$  é `ushort`, `uint`, ou `ulong`
  - $S_1$  é `int` e  $S_2$  é `uint`, ou `ulong`
  - $S_1$  é `long` e  $S_2$  é `ulong`

## Sobrecarregando em classes genéricas

Embora as assinaturas como declaradas devam ser exclusivas, é possível que a substituição dos argumentos de tipo resulte em assinaturas idênticas. Nesses casos, as regras de quebra de sobrecarga da resolução de sobrecarga acima escolherão o membro mais específico.

Os exemplos a seguir mostram sobrecargas que são válidas e inválidas de acordo com esta regra:

C#

```
interface I1<T> {...}

interface I2<T> {...}

class G1<U>
{
    int F1(U u);                      // Overload resolution for G<int>.F1
    int F1(int i);                    // will pick non-generic

    void F2(I1<U> a);                // Valid overload
    void F2(I2<U> a);
```

```

}

class G2<U,V>
{
    void F3(U u, V v);           // Valid, but overload resolution for
    void F3(V v, U u);           // G2<int,int>.F3 will fail

    void F4(U u, I1<V> v);     // Valid, but overload resolution for
    void F4(I1<V> v, U u);     // G2<I1<int>,int>.F4 will fail

    void F5(U u1, I1<V> v2);   // Valid overload
    void F5(V v1, U u2);

    void F6(ref U u);          // valid overload
    void F6(out V v);
}

```

## Verificação de tempo de compilação da resolução dinâmica de sobrecarga

Para operações associadas mais dinamicamente, o conjunto de possíveis candidatos para resolução é desconhecido em tempo de compilação. Em determinados casos, no entanto, o conjunto candidato é conhecido em tempo de compilação:

- Chamadas de método estático com argumentos dinâmicos
- O método de instância chama onde o receptor não é uma expressão dinâmica
- Chamadas do indexador onde o receptor não é uma expressão dinâmica
- Chamadas de construtor com argumentos dinâmicos

Nesses casos, uma verificação limitada de tempo de compilação é executada para cada candidato para ver se qualquer uma delas poderia ser aplicada em tempo de execução. Essa verificação consiste nas seguintes etapas:

- Inferência de tipo parcial: qualquer argumento de tipo que não dependa direta ou indiretamente em um argumento do tipo `dynamic` é inferido usando as regras de [inferência de tipo](#). Os argumentos de tipo restantes são desconhecidos.
- Verificação de aplicabilidade parcial: a aplicabilidade é verificada de acordo com o [membro de função aplicável](#), mas ignorando parâmetros cujos tipos são desconhecidos.
- Se nenhum candidato passar nesse teste, ocorrerá um erro em tempo de compilação.

## Invocação de membro de função

Esta seção descreve o processo que ocorre em tempo de execução para invocar um membro de função específico. Supõe-se que um processo de tempo de ligação já determinou o membro específico para invocar, possivelmente aplicando a resolução de sobrecarga a um conjunto de membros de funções candidatas.

Para fins de descrição do processo de invocação, os membros da função são divididos em duas categorias:

- Membros da função estática. Esses são construtores de instância, métodos estáticos, acessadores de propriedade estática e operadores definidos pelo usuário. Membros de função estática são sempre não virtuais.
- Membros da função de instância. Esses são métodos de instância, acessadores de propriedade de instância e acessadores do indexador. Os membros da função de instância são não virtuais ou virtuais e são sempre invocados em uma instância específica. A instância é computada por uma expressão de instância e torna-se acessível dentro do membro da função como `this` ([esse acesso](#)).

O processamento em tempo de execução de uma invocação de membro de função consiste nas etapas a seguir, em que `M` é o membro da função e, se `M` for um membro da instância, `E` é a expressão de instância:

- Se `M` é um membro de função estática:
  - A lista de argumentos é avaliada conforme descrito em [listas de argumentos](#).
  - `M` é invocado.
- Se `M` é um membro de função de instância declarado em um *value\_type*:
  - `E` é avaliado. Se essa avaliação causar uma exceção, nenhuma etapa adicional será executada.
  - Se `E` não for classificado como uma variável, uma variável local temporária do `E` tipo será criada e o valor de `E` será atribuído a essa variável. `E` é então reclassificado como uma referência a essa variável local temporária. A variável temporária é acessível como `this` em `M`, mas não de qualquer outra maneira. Portanto, somente quando `E` é uma variável true é possível que o chamador observe as alterações que o `M` faz `this`.
  - A lista de argumentos é avaliada conforme descrito em [listas de argumentos](#).
  - `M` é invocado. A variável referenciada `E` se torna a variável referenciada por `this`.
- Se `M` é um membro de função de instância declarado em um *reference\_type*:
  - `E` é avaliado. Se essa avaliação causar uma exceção, nenhuma etapa adicional será executada.

- A lista de argumentos é avaliada conforme descrito em [listas de argumentos](#).
- Se o tipo de `E` for um *value\_type*, uma conversão boxing ([conversões Boxing](#)) será executada para converter `E` em Type `object` e `E` será considerada do tipo `object` nas etapas a seguir. Nesse caso, `M` poderia ser apenas um membro de `System.Object`.
- O valor de `E` é verificado para ser válido. Se o valor de `E` for `null`, um `System.NullReferenceException` será lançado e nenhuma etapa adicional será executada.
- A implementação do membro da função a ser invocada é determinada:
  - Se o tipo de tempo de associação de `E` for uma interface, o membro da função a ser invocado será a implementação de `M` fornecida pelo tipo de tempo de execução da instância referenciada pelo `E`. Esse membro de função é determinado pela aplicação das regras de mapeamento de interface ([mapeamento de interface](#)) para determinar a implementação de `M` fornecida pelo tipo de tempo de execução da instância referenciada pelo `E`.
  - Caso contrário, se `M` for um membro da função virtual, o membro da função a ser invocado será a implementação de `M` fornecida pelo tipo de tempo de execução da instância referenciada pelo `E`. Esse membro de função é determinado pela aplicação das regras para determinar a implementação mais derivada ([métodos virtuais](#)) de `M` em relação ao tipo de tempo de execução da instância referenciada pelo `E`.
  - Caso contrário, `M` é um membro de função não virtual e o membro da função a ser invocado é o `M` próprio.
- A implementação do membro da função determinada na etapa acima é invocada. O objeto referenciado `E` se torna o objeto referenciado por `this`.

## Invocações em instâncias em caixas

Um membro de função implementado em um *value\_type* pode ser invocado por meio de uma instância em caixa do que *value\_type* nas seguintes situações:

- Quando o membro da função é um `override` de um método herdado do tipo `object` e é invocado por meio de uma expressão de instância do tipo `object`.
- Quando o membro da função é uma implementação de um membro da função de interface e é invocado por meio de uma expressão de instância de um *interface\_type*.
- Quando o membro da função é invocado por meio de um delegado.

Nessas situações, a instância em caixa é considerada para conter uma variável do *value\_type*, e essa variável se torna a variável referenciada por `this` dentro da invocação de membro da função. Em particular, isso significa que, quando um membro de função é invocado em uma instância em caixa, é possível que o membro da função modifique o valor contido na instância em caixa.

## Expressões primárias

As expressões primárias incluem as formas mais simples de expressões.

antlr

```
primary_expression
: primary_no_array_creation_expression
| array_creation_expression
;

primary_no_array_creation_expression
: literal
| interpolated_string_expression
| simple_name
| parenthesized_expression
| member_access
| invocation_expression
| element_access
| this_access
| base_access
| post_increment_expression
| post_decrement_expression
| object_creation_expression
| delegate_creation_expression
| anonymous_object_creation_expression
| typeof_expression
| checked_expression
| unchecked_expression
| default_value_expression
| nameof_expression
| anonymous_method_expression
| primary_no_array_creation_expression_unsafe
;
```

As expressões primárias são divididas entre *array\_creation\_expression*s e *primary\_no\_array\_creation\_expression*s. Tratar a expressão de criação de matriz dessa maneira, em vez de listá-la junto com outros formulários de expressão simples, permite que a gramática inpermita códigos potencialmente confusos, como

C#

```
object o = new int[3][1];
```

que, de outra forma, seriam interpretados como

C#

```
object o = (new int[3])[1];
```

## Literais

Um *primary\_expression* que consiste em um *literal* ([literais](#)) é classificado como um valor.

## Cadeias de caracteres interpoladas

Um *interpolated\_string\_expression* consiste em um `$` sinal seguido por um literal de cadeia de caracteres regular ou textual, no qual os buracos, delimitados por `{` e `}`, incluem expressões e especificações de formatação. Uma expressão de cadeia de caracteres interpolada é o resultado de um *interpolated\_string\_literal* que foi dividido em tokens individuais, conforme descrito em [literais de cadeia de caracteres interpolados](#).

antlr

```
interpolated_string_expression
: '$' interpolated_regular_string
| '$' interpolated_verbatim_string
;

interpolated_regular_string
: interpolated_regular_string_whole
| interpolated_regular_string_start interpolated_regular_string_body
interpolated_regular_string_end
;

interpolated_regular_string_body
: interpolation (interpolated_regular_string_mid interpolation)*
;

interpolation
: expression
| expression ',' constant_expression
;

interpolated_verbatim_string
: interpolated_verbatim_string_whole
| interpolated_verbatim_string_start interpolated_verbatim_string_body
interpolated_verbatim_string_end
```

```
;  
  
interpolated_verbatim_string_body  
: interpolation (interpolated_verbatim_string_mid interpolation)+  
;
```

O *constant\_expression* em uma interpolação deve ter uma conversão implícita para `int`.

Um *interpolated\_string\_expression* é classificado como um valor. Se ele for imediatamente convertido em `System.IFormattable` ou `System.FormattableString` com uma conversão de cadeia de caracteres interpolada implícita ([conversões de cadeia de caracteres interpoladas implícitas](#)), a expressão de cadeia de caracteres interpolada terá esse tipo. Caso contrário, ele terá o tipo `string`.

Se o tipo de uma cadeia de caracteres interpolada for `System.IFormattable` ou `System.FormattableString`, o significado será uma chamada para `System.Runtime.CompilerServices.FormattableStringFactory.Create`. Se o tipo for `string`, o significado da expressão será uma chamada para `string.Format`. Em ambos os casos, a lista de argumentos da chamada consiste em um literal de cadeia de caracteres de formato com espaços reservados para cada interpolação e um argumento para cada expressão correspondente aos detentores de lugar.

A cadeia de caracteres de formato literal é construída da seguinte maneira, em que `N` é o número de interpolações no *interpolated\_string\_expression*:

- Se um *interpolated\_regular\_string\_whole* ou um *interpolated\_verbatim\_string\_whole* seguir o `$` sinal, o literal da cadeia de caracteres de formato será esse token.
- Caso contrário, o literal de cadeia de caracteres de formato consiste em:
  - Primeiro *interpolated\_regular\_string\_start* ou *interpolated\_verbatim\_string\_start*
  - Em seguida, para cada número `I` de `0` a `N-1`:
    - A representação decimal de `I`
    - Em seguida, se a *interpolação* correspondente tiver um *constant\_expression*, um `,` (vírgula) seguido pela representação decimal do valor do *constant\_expression*
    - Em seguida, o *interpolated\_regular\_string\_mid*, *interpolated\_regular\_string\_end*, *interpolated\_verbatim\_string\_mid* ou *interpolated\_verbatim\_string\_end* imediatamente após a interpolação correspondente.

Os argumentos subsequentes são simplesmente as *expressões* das *interpolações* (se houver), em ordem.

TODO: exemplos.

## Nomes simples

Um *Simple\_name* consiste em um identificador, opcionalmente seguido por uma lista de argumentos de tipo:

```
antlr

simple_name
    : identifier type_argument_list?
    ;
```

Uma *Simple\_name* é uma das formas `I` ou do formulário `I<A1,...,Ak>`, em que `I` é um único identificador e `<A1,...,Ak>` é uma *type\_argument\_list* opcional. Quando nenhum *type\_argument\_list* for especificado, considere `K` ser zero. A *Simple\_name* é avaliada e classificada da seguinte maneira:

- Se `K` for zero e o *Simple\_name* aparecer dentro de um *bloco* e se o espaço de declaração local ([declarações](#)) do *bloco*(ou um *bloco* delimitador) contiver uma variável local, um parâmetro ou uma constante com nome `I`, o *Simple\_name* se referirá a essa variável local, parâmetro ou constante e será classificado como uma variável ou um valor.
- Se `K` for zero e o *Simple\_name* aparecer dentro do corpo de uma declaração de método genérico e se essa declaração incluir um parâmetro de tipo com nome `I`, o *Simple\_name* se referirá a esse parâmetro de tipo.
- Caso contrário, para cada tipo de instância `T` ([o tipo de instância](#)), começando com o tipo de instância da declaração de tipo delimitadora imediatamente e continuando com o tipo de instância de cada declaração de classe ou struct de delimitador (se houver):
  - Se `K` for zero e a declaração de `T` incluir um parâmetro de tipo com `I` o nome, o *Simple\_name* se referirá a esse parâmetro de tipo.
  - Caso contrário, se uma pesquisa de membro ([pesquisa de membro](#)) de `I` em `T` com argumentos de `K` tipo produzir uma correspondência:
    - Se `T` é o tipo de instância do tipo de classe ou struct imediatamente delimitador e a pesquisa identifica um ou mais métodos, o resultado é um grupo de métodos com uma expressão de instância associada de `this`. Se uma lista de argumentos de tipo tiver sido especificada, ela será usada na chamada de um método genérico ([invocações de método](#)).
    - Caso contrário, se `T` for o tipo de instância do tipo de classe ou struct imediatamente delimitadora, se a pesquisa identificar um membro de instância e se a referência ocorrer no corpo de um construtor de instância,

um método de instância ou um acessador de instância, o resultado será o mesmo que um acesso de membro ([acesso de membro](#)) do formulário `this.I`. Isso só pode acontecer quando `K` é zero.

- Caso contrário, o resultado será o mesmo que um acesso de membro ([acesso de membro](#)) do formulário `T.I` ou `T.I<A1,...,Ak>`. Nesse caso, é um erro de tempo de associação para o *Simple\_name* fazer referência a um membro de instância.
- Caso contrário, para cada namespace `N`, começando com o namespace no qual o *Simple\_name* ocorre, continuando com cada namespace delimitador (se houver) e terminando com o namespace global, as etapas a seguir são avaliadas até que uma entidade esteja localizada:
  - Se `K` for zero e `I` for o nome de um namespace em `N`, então:
    - Se o local onde a *Simple\_name* ocorre estiver entre uma declaração de namespace para `N` e a declaração de namespace contiver um *extern\_alias\_directive* ou *using\_alias\_directive* que associa o nome a `I` um namespace ou tipo, o *Simple\_name* será ambíguo e ocorrerá um erro em tempo de compilação.
    - Caso contrário, o *Simple\_name* se refere ao namespace chamado `I` em `N`.
  - Caso contrário, se `N` contiver um tipo acessível com `I` parâmetros Name e `K` Type, então:
    - Se `K` for zero e o local em que o *Simple\_name* ocorrer é incluído por uma declaração de namespace para `N` e a declaração de namespace contém um *extern\_alias\_directive* ou *using\_alias\_directive* que associa o nome a `I` um namespace ou tipo, o *Simple\_name* é ambíguo e ocorre um erro em tempo de compilação.
    - Caso contrário, o *namespace\_or\_type\_name* se refere ao tipo construído com os argumentos de tipo fornecidos.
  - Caso contrário, se o local em que o *Simple\_name* ocorrer for incluído por uma declaração de namespace para `N`:
    - Se `K` for zero e a declaração de namespace contiver um *extern\_alias\_directive* ou *using\_alias\_directive* que associa o nome a `I` um namespace ou tipo importado, a *Simple\_name* se referirá a esse namespace ou tipo.
    - Caso contrário, se os namespaces e as declarações de tipo importados pelo *using\_namespace\_directive*s e *using\_static\_directive*s da declaração de namespace contiverem exatamente um tipo acessível ou um membro estático sem extensão que tenha parâmetros de nome `I` e `K` tipo, o *Simple\_name* se referirá a esse tipo ou membro construído com os argumentos de tipo fornecidos.

- Caso contrário, se os namespaces e os tipos importados pelo *using\_namespace\_directive*s da declaração de namespace contiverem mais de um tipo acessível ou membro estático de método não-extensão que possua parâmetros de nome *I* e *K* tipo, o *Simple\_name* será ambíguo e ocorrerá um erro.

Observe que essa etapa inteira é exatamente paralela à etapa correspondente no processamento de um *namespace\_or\_type\_name* ([namespace e nomes de tipo](#)).

- Caso contrário, o *Simple\_name* será indefinido e ocorrerá um erro em tempo de compilação.

## Expressões entre parênteses

Uma *parenthesized\_expression* consiste em uma *expressão* entre parênteses.

```
antlr

parenthesized_expression
: '(' expression ')'
;
```

Uma *parenthesized\_expression* é avaliada avaliando a *expressão* dentro dos parênteses. Se a *expressão* dentro dos parênteses denota um namespace ou tipo, ocorre um erro de tempo de compilação. Caso contrário, o resultado da *parenthesized\_expression* é o resultado da avaliação da *expressão* contida.

## Acesso de membros

Um *member\_access* consiste em um *primary\_expression*, um *predefined\_type* ou um *qualified\_alias\_member*, seguido por um token ". ", seguido por um *identificador*, opcionalmente seguido por um *type\_argument\_list*.

```
antlr

member_access
: primary_expression '.' identifier type_argument_list?
| predefined_type '.' identifier type_argument_list?
| qualified_alias_member '.' identifier
;

predefined_type
: 'bool'    | 'byte'   | 'char'   | 'decimal' | 'double' | 'float'  | 'int'
| 'long'
| 'object' | 'sbyte' | 'short' | 'string' | 'uint'   | 'ulong' |
```

```
' ushort'  
;
```

A produção *qualified\_alias\_member* é definida em [qualificadores de alias de namespace](#).

Uma *member\_access* é uma das formas  $E.I$  ou do formulário  $E.I<A_1, \dots, A_k>$ , em que  $E$  é uma expressão primária,  $I$  é um único identificador e  $\langle A_1, \dots, A_k \rangle$  é uma *type\_argument\_list* opcional. Quando nenhum *type\_argument\_list* for especificado, considere  $K$  ser zero.

Um *member\_access* com um *primary\_expression* do tipo *dynamic* é vinculado dinamicamente ([associação dinâmica](#)). Nesse caso, o compilador classifica o acesso de membro como um acesso de Propriedade do tipo *dynamic*. As regras abaixo para determinar o significado da *member\_access* são então aplicadas em tempo de execução, usando o tipo de tempo de execução em vez do tipo de tempo de compilação do *primary\_expression*. Se essa classificação de tempo de execução leva a um grupo de métodos, o acesso de membro deve ser o *primary\_expression* de um *invocation\_expression*.

A *member\_access* é avaliada e classificada da seguinte maneira:

- Se  $K$  for zero e  $E$  for um namespace e  $E$  contiver um namespace aninhado com nome  $I$ , o resultado será o namespace.
- Caso contrário, se  $E$  for um namespace e  $E$  contiver um tipo acessível com parâmetros de nome  $I$  e  $K$  tipo, o resultado será aquele tipo construído com os argumentos de tipo fornecidos.
- Se  $E$  for um *predefined\_type* ou um *primary\_expression* classificado como um tipo, se  $E$  não for um parâmetro de tipo, e se uma pesquisa de membro ([pesquisa de membro](#)) de  $I$  no  $E$  com parâmetros de  $K$  tipo produzir uma correspondência, o  $E.I$  será avaliado e classificado da seguinte maneira:
  - Se  $I$  o identificar um tipo, o resultado será aquele tipo construído com os argumentos de tipo fornecidos.
  - Se  $I$  o identificar um ou mais métodos, o resultado será um grupo de métodos sem nenhuma expressão de instância associada. Se uma lista de argumentos de tipo tiver sido especificada, ela será usada na chamada de um método genérico ([invocações de método](#)).
  - Se  $I$  o identificar uma *static* propriedade, o resultado será um acesso de propriedade sem nenhuma expressão de instância associada.
  - Se  $I$  o identificar um *static* campo:
    - Se o campo for *readonly* e a referência ocorrer fora do construtor estático da classe ou struct no qual o campo é declarado, o resultado será um valor,

ou seja, o valor do campo estático  $I$  em  $E$ .

- Caso contrário, o resultado será uma variável, ou seja, o campo estático  $I$  em  $E$ .
  - Se  $I$  o identificar um `static` evento:
    - Se a referência ocorrer na classe ou struct na qual o evento é declarado e o evento tiver sido declarado sem `event_accessor_declarations` (`eventos`),  $E.I$  será processado exatamente como se  $I$  fosse um campo estático.
    - Caso contrário, o resultado será um acesso de evento sem nenhuma expressão de instância associada.
  - Se  $I$  o identificar uma constante, o resultado será um valor, ou seja, o valor dessa constante.
  - Se  $I$  o identificar um membro de enumeração, o resultado será um valor, ou seja, o valor desse membro de enumeração.
  - Caso contrário,  $E.I$  é uma referência de membro inválida e ocorre um erro de tempo de compilação.
- Se  $E$  for um acesso de propriedade, acesso ao indexador, variável ou valor, o tipo de que é  $T$ , e uma pesquisa de membro ([pesquisa de membro](#)) de  $I$  em  $T$  com argumentos de  $K$  tipo produzem uma correspondência, então  $E.I$  é avaliado e classificado da seguinte maneira:
    - Primeiro, se  $E$  for um acesso de propriedade ou indexador, o valor da propriedade ou do acesso do indexador será obtido ([valores de expressões](#)) e  $E$  será reclassificado como um valor.
    - Se  $I$  o identificar um ou mais métodos, o resultado será um grupo de métodos com uma expressão de instância associada de  $E$ . Se uma lista de argumentos de tipo tiver sido especificada, ela será usada na chamada de um método genérico ([invocações de método](#)).
    - Se  $I$  o identificar uma propriedade de instância,
      - Se  $E$  for `this`,  $I$  identifica uma propriedade implementada automaticamente ([Propriedades implementadas automaticamente](#)) sem um setter, e a referência ocorre dentro de um construtor de instância para um tipo de classe ou struct  $T$ , então o resultado é uma variável, ou seja, o campo de apoio oculto para a propriedade automática fornecida por  $I$  na instância do  $T$  fornecida pelo `this`.
      - Caso contrário, o resultado é um acesso de propriedade com uma expressão de instância associada de  $E$ .
    - Se  $T$  é um `class_type` e  $I$  identifica um campo de instância desse `class_type`:
      - Se o valor de  $E$  for `null`, um `System.NullReferenceException` será lançado.
      - Caso contrário, se o campo for `readonly` e a referência ocorrer fora de um construtor de instância da classe na qual o campo é declarado, o resultado

- será um valor, ou seja, o valor do campo  $I$  no objeto referenciado por  $E$ .
- Caso contrário, o resultado será uma variável, ou seja, o campo  $I$  no objeto referenciado por  $E$ .
  - Se  $T$  é um *struct\_type* e  $I$  identifica um campo de instância desse *struct\_type*:
    - Se  $E$  for um valor, ou se o campo for *readonly* e a referência ocorrer fora de um construtor de instância do struct no qual o campo é declarado, o resultado será um valor, ou seja, o valor do campo  $I$  na instância de struct fornecida por  $E$ .
    - Caso contrário, o resultado será uma variável, ou seja, o campo  $I$  na instância de struct fornecida por  $E$ .
  - Se  $I$  o identificar um evento de instância:
    - Se a referência ocorrer dentro da classe ou estrutura na qual o evento é declarado e o evento tiver sido declarado sem *event\_accessor\_declarations* (*eventos*) e a referência não ocorrer como o lado esquerdo de um  $+ =$  operador OR,  $E.I$  será processada exatamente como se  $I$  fosse um campo de instância.
    - Caso contrário, o resultado é um acesso de evento com uma expressão de instância associada de  $E$ .
  - Caso contrário, será feita uma tentativa de processar  $E.I$  como uma invocação de método de extensão ([invocações de método de extensão](#)). Se isso falhar,  $E.I$  será uma referência de membro inválida e ocorrerá um erro de tempo de vinculação.

## Nomes de tipos e nomes simples idênticos

Em um acesso de membro do formulário  $E.I$ , se  $E$  for um único identificador e, se o significado de  $E$  um *Simple\_name* ([nomes simples](#)) for uma constante, campo, propriedade, variável local ou parâmetro com o mesmo tipo que o significado de  $E$  um *type\_name* ([namespace e nomes de tipo](#)), então ambos os significados possíveis de  $E$  são permitidos. Os dois significados possíveis do  $E.I$  são nunca ambíguos, pois  $I$  deve necessariamente ser um membro do tipo  $E$  em ambos os casos. Em outras palavras, a regra simplesmente permite o acesso aos membros estáticos e tipos aninhados de  $E$  onde um erro de tempo de compilação teria ocorrido de outra forma. Por exemplo:

C#

```
struct Color
{
    public static readonly Color White = new Color(...);
    public static readonly Color Black = new Color(...);

    public Color Complement() {...}
```

```

}

class A
{
    public Color Color;           // Field Color of type Color

    void F() {
        Color = Color.Black;     // References Color.Black static
    }

    static void G() {
        Color c = Color.White;   // References Color.White static
    }
}

```

## Ambiguidades gramaticais

As produções para *Simple\_name* (nomes simples) e *member\_access* (acesso de membro) podem dar ao aumento das ambiguidades na gramática para expressões. Por exemplo, a instrução:

C#

```
F(G<A,B>(7));
```

pode ser interpretado como uma chamada para `F` com dois argumentos, `G < A e B > (7)`. Como alternativa, ele pode ser interpretado como uma chamada para `F` com um argumento, que é uma chamada para um método genérico `G` com dois argumentos de tipo e um argumento regular.

Se uma sequência de tokens puder ser analisada (no contexto) como *um Simple\_name* (nomes simples), *member\_access* (acesso de membro) ou *pointer\_member\_access* (acesso de membro de ponteiro) terminando com um *type\_argument\_list* (argumentos de tipo), o token imediatamente após o token de fechamento `>` é examinado. Se for um de

C#

```
( ) ] } : ; , . ? == != | ^
```

em seguida, a *type\_argument\_list* é mantida como parte da *Simple\_name*, *member\_access* ou *pointer\_member\_access* e qualquer outra análise possível da

sequência de tokens é descartada. Caso contrário, o *type\_argument\_list* não será considerado como parte do *Simple\_name*, *member\_access* ou *pointer\_member\_access*, mesmo que não haja outra análise possível da sequência de tokens. Observe que essas regras não são aplicadas ao analisar um *type\_argument\_list* em um *namespace\_or\_type\_name* ([namespace e nomes de tipo](#)). A instrução

```
C#
```

```
F(G<A,B>(7));
```

de acordo com essa regra, será interpretado como uma chamada para `F` com um argumento, que é uma chamada para um método genérico `G` com dois argumentos de tipo e um argumento regular. As instruções

```
C#
```

```
F(G < A, B > 7);  
F(G < A, B >> 7);
```

cada um será interpretado como uma chamada para `F` com dois argumentos. A instrução

```
C#
```

```
x = F < A > +y;
```

será interpretado como um operador menor que, maior que operador, e operador de adição unário, como se a instrução tivesse sido gravada `x = (F < A) > (+y)`, em vez de um *Simple\_name* com um *type\_argument\_list* seguido por um operador Binary Plus. Na instrução

```
C#
```

```
x = y is C<T> + z;
```

os tokens `C<T>` são interpretados como um *namespace\_or\_type\_name* com um *type\_argument\_list*.

## Expressões de invocação

Um *invocation\_expression* é usado para invocar um método.

antlr

```
invocation_expression
    : primary_expression '(' argument_list? ')'
    ;
```

Uma *invocation\_expression* é vinculada dinamicamente ([associação dinâmica](#)) se pelo menos uma das seguintes isenções:

- O *primary\_expression* tem tipo de tempo de compilação `dynamic`.
- Pelo menos um argumento da *argument\_list* opcional tem o tipo de tempo de compilação `dynamic` e o *primary\_expression* não tem um tipo delegado.

Nesse caso, o compilador classifica o *invocation\_expression* como um valor do tipo `dynamic`. As regras abaixo para determinar o significado do *invocation\_expression* são então aplicadas em tempo de execução, usando o tipo de tempo de execução em vez do tipo de tempo de compilação dos argumentos de *primary\_expression* e que têm o tipo de tempo de compilação `dynamic`. Se o *primary\_expression* não tiver o tipo de tempo de compilação `dynamic`, a invocação de método passará por uma verificação de tempo de compilação limitada, conforme descrito em [verificação de tempo de compilação da resolução dinâmica de sobrecarga](#).

O *primary\_expression* de um *invocation\_expression* deve ser um grupo de métodos ou um valor de um *delegate\_type*. Se o *primary\_expression* for um grupo de métodos, a *invocation\_expression* será uma invocação de método ([invocações de método](#)). Se o *primary\_expression* for um valor de um *delegate\_type*, o *invocation\_expression* será uma invocação de delegado ([invocações de delegado](#)). Se o *primary\_expression* não for um grupo de métodos nem um valor de um *delegate\_type*, ocorrerá um erro de tempo de associação.

O *argument\_list* opcional ([listas de argumentos](#)) fornece valores ou referências de variáveis para os parâmetros do método.

O resultado da avaliação de um *invocation\_expression* é classificado da seguinte maneira:

- Se o *invocation\_expression* invocar um método ou delegado que retorna `void`, o resultado será `Nothing`. Uma expressão que é classificada como nada é permitida apenas no contexto de uma *statement\_expression* ([instruções de expressão](#)) ou como o corpo de uma *lambda\_expression* ([expressões de função anônimas](#)). Caso contrário, ocorrerá um erro de tempo de vinculação.
- Caso contrário, o resultado será um valor do tipo retornado pelo método ou delegado.

## Invocações de método

Para uma invocação de método, a *primary\_expression* do *invocation\_expression* deve ser um grupo de métodos. O grupo de métodos identifica o método a ser invocado ou o conjunto de métodos sobrecarregados do qual escolher um método específico a ser invocado. No último caso, a determinação do método específico a ser invocado é baseada no contexto fornecido pelos tipos dos argumentos na *argument\_list*.

O processamento de tempo de associação de uma invocação de método do formulário  $M(A)$ , em que  $M$  é um grupo de métodos (possivelmente incluindo um *type\_argument\_list*) e  $A$  é um *argument\_list* opcional, consiste nas seguintes etapas:

- O conjunto de métodos candidatos para a invocação do método é construído.  
Para cada método  $F$  associado ao grupo de métodos  $M$  :
  - Se  $F$  for não genérico,  $F$  será um candidato quando:
    - $M$  Não tem nenhuma lista de argumentos de tipo e
    - $F$  é aplicável em relação a  $A$  ([membro de função aplicável](#)).
  - Se  $F$  é genérico e  $M$  não tem nenhuma lista de argumentos de tipo,  $F$  é um candidato quando:
    - A inferência de tipos ([inferência de tipos](#)) é bem sucedido, inferindo-se a uma lista de argumentos de tipo para a chamada e
    - Depois que os argumentos de tipo inferido são substituídos pelos parâmetros de tipo de método correspondentes, todos os tipos construídos na lista de parâmetros de  $F$  satisfazem suas restrições ([atendendo às restrições](#)) e a lista de parâmetros de  $F$  é aplicável em relação ao  $A$  ([membro de função aplicável](#)).
  - Se  $F$  é genérico e  $M$  inclui uma lista de argumentos de tipo,  $F$  é um candidato quando:
    - $F$  tem o mesmo número de parâmetros de tipo de método que foram fornecidos na lista de argumentos de tipo e
    - Depois que os argumentos de tipo são substituídos pelos parâmetros de tipo de método correspondentes, todos os tipos construídos na lista de parâmetros de  $F$  satisfazem suas restrições ([atendendo às restrições](#)) e a lista de parâmetros de  $F$  é aplicável em relação ao  $A$  ([membro de função aplicável](#)).
- O conjunto de métodos candidatos é reduzido para conter apenas métodos dos tipos mais derivados: para cada método  $C.F$  no conjunto, em que  $C$  é o tipo no qual o método  $F$  é declarado, todos os métodos declarados em um tipo base de  $C$  são removidos do conjunto. Além disso, se  $C$  for um tipo de classe diferente de `object`, todos os métodos declarados em um tipo de interface serão removidos

do conjunto. (Essa última regra só afeta quando o grupo de métodos era o resultado de uma pesquisa de membro em um parâmetro de tipo com uma classe base efetiva diferente de Object e uma interface efetiva não vazia definida.)

- Se o conjunto resultante de métodos candidatos estiver vazio, o processamento adicional nas etapas a seguir será abandonado e, em vez disso, será feita uma tentativa de processar a invocação como uma invocação de método de extensão ([invocações de método de extensão](#)). Se isso falhar, não existirá nenhum método aplicável e ocorrerá um erro de tempo de vinculação.
- O melhor método do conjunto de métodos candidatos é identificado usando as regras de resolução de sobrecarga da [resolução de sobrecarga](#). Se um único método melhor não puder ser identificado, a invocação do método será ambígua e ocorrerá um erro de tempo de ligação. Ao executar a resolução de sobrecarga, os parâmetros de um método genérico são considerados após a substituição dos argumentos de tipo (fornecidos ou inferidos) para os parâmetros de tipo de método correspondentes.
- A validação final do melhor método escolhido é executada:
  - O método é validado no contexto do grupo de métodos: se o melhor método é um método estático, o grupo de métodos deve ter resultado de um *Simple\_name* ou de um *member\_access* por meio de um tipo. Se o melhor método for um método de instância, o grupo de métodos deverá ter resultado de uma *Simple\_name*, uma *member\_access* por meio de uma variável ou valor ou uma *base\_access*. Se nenhum desses requisitos for verdadeiro, ocorrerá um erro de tempo de ligação.
  - Se o melhor método for um método genérico, os argumentos de tipo (fornecidos ou inferidos) serão verificados em relação às restrições ([atendendo às restrições](#)) declaradas no método genérico. Se qualquer argumento de tipo não atender às restrições correspondentes no parâmetro de tipo, ocorrerá um erro de tempo de associação.

Depois que um método tiver sido selecionado e validado em tempo de vinculação pelas etapas acima, a invocação de tempo de execução real será processada de acordo com as regras de invocação de membro de função descrita na [verificação de tempo de compilação da resolução dinâmica de sobrecarga](#).

O efeito intuitivo das regras de resolução descritas acima é o seguinte: para localizar o método específico invocado por uma invocação de método, comece com o tipo indicado pela invocação do método e continue a cadeia de herança até que pelo menos uma declaração de método aplicável, acessível e não substituída seja encontrada. Em seguida, execute a inferência de tipos e a resolução de sobrecarga no conjunto de métodos aplicáveis, acessíveis e não substituição declarados nesse tipo e invoque o

método, portanto, selecionado. Se nenhum método foi encontrado, tente em vez disso processar a invocação como uma invocação de método de extensão.

## Invocações de método de extensão

Em uma invocação de método ([invocações em instâncias em caixas](#)) de um dos formulários

```
C#  
  
expr . identifier ( )  
  
expr . identifier ( args )  
  
expr . identifier < typeargs > ( )  
  
expr . identifier < typeargs > ( args )
```

Se o processamento normal da invocação não encontrar nenhum método aplicável, será feita uma tentativa de processar a construção como uma invocação de método de extensão. Se *expr* ou qualquer um dos *args* tiver tipo de tempo de compilação `dynamic`, os métodos de extensão não serão aplicados.

O objetivo é encontrar a melhor *type\_name* `c`, para que a invocação de método estático correspondente possa ocorrer:

```
C#  
  
C . identifier ( expr )  
  
C . identifier ( expr , args )  
  
C . identifier < typeargs > ( expr )  
  
C . identifier < typeargs > ( expr , args )
```

Um método de extensão `ci.Mj` será *elegível* se:

- `ci` é uma classe não genérica e não aninhada
- O nome do `Mj` é *identificador*
- `Mj` é acessível e aplicável quando aplicado aos argumentos como um método estático, conforme mostrado acima
- Existe uma identidade implícita, uma referência ou uma conversão boxing de *expr* para o tipo do primeiro parâmetro de `Mj`.

A pesquisa do `c` continua da seguinte maneira:

- Começando com a declaração de namespace delimitadora mais próxima, continuando com cada declaração de namespace delimitadora e terminando com a unidade de compilação que a contém, as tentativas sucessivas são feitas para encontrar um conjunto candidato de métodos de extensão:
  - Se o namespace ou a unidade de compilação fornecida diretamente contiver declarações de tipo não genéricas `ci` com métodos de extensão qualificados `Mj`, o conjunto desses métodos de extensão será o conjunto de candidatos.
  - Se tipos `ci` importados por *using\_static\_declarations* e declarados diretamente em namespaces importados por *using\_namespace\_directive*s no namespace ou na unidade de compilação fornecida diretamente contiverem métodos de extensão qualificados `Mj`, o conjunto desses métodos de extensão será o conjunto de candidatos.
- Se nenhum conjunto candidato for encontrado em qualquer declaração de namespace ou unidade de compilação delimitadora ocorrer um erro em tempo de compilação.
- Caso contrário, a resolução de sobrecarga será aplicada ao conjunto de candidatos, conforme descrito em ([resolução de sobrecarga](#)). Se nenhum único método melhor for encontrado, ocorrerá um erro em tempo de compilação.
- `c` é o tipo no qual o melhor método é declarado como um método de extensão.

Usando `c` como um destino, a chamada de método é processada como uma invocação de método estático ([verificação de tempo de compilação da resolução dinâmica de sobrecarga](#)).

As regras anteriores significam que os métodos de instância têm precedência sobre os métodos de extensão, que os métodos de extensão disponíveis nas declarações de namespace interno têm precedência sobre os métodos de extensão disponíveis nas declarações de namespace externo e que os métodos de extensão declarados diretamente em um namespace têm precedência sobre os métodos de extensão importados para o mesmo namespace com uma diretiva de namespace using. Por exemplo:

C#

```
public static class E
{
    public static void F(this object obj, int i) { }

    public static void F(this object obj, string s) { }
}

class A { }
```

```

class B
{
    public void F(int i) { }
}

class C
{
    public void F(object obj) { }
}

class X
{
    static void Test(A a, B b, C c) {
        a.F(1);                  // E.F(object, int)
        a.F("hello");            // E.F(object, string)

        b.F(1);                  // B.F(int)
        b.F("hello");            // E.F(object, string)

        c.F(1);                  // C.F(object)
        c.F("hello");            // C.F(object)
    }
}

```

No exemplo, o `B` método tem precedência sobre o primeiro método de extensão, e o `C` método de tem precedência sobre os dois métodos de extensão.

C#

```

public static class C
{
    public static void F(this int i) { Console.WriteLine("C.F({0})", i); }
    public static void G(this int i) { Console.WriteLine("C.G({0})", i); }
    public static void H(this int i) { Console.WriteLine("C.H({0})", i); }
}

namespace N1
{
    public static class D
    {
        public static void F(this int i) { Console.WriteLine("D.F({0})", i); }
        public static void G(this int i) { Console.WriteLine("D.G({0})", i); }
    }
}

namespace N2
{
    using N1;

    public static class E

```

```

    {
        public static void F(this int i) { Console.WriteLine("E.F({0})", i);
    }
}

class Test
{
    static void Main(string[] args)
    {
        1.F();
        2.G();
        3.H();
    }
}

```

A saída deste exemplo é:

The screenshot shows a console window with the title "Console". Inside the window, the output of the program is displayed, showing three lines of text: "E.F(1)", "D.G(2)", and "C.H(3)".

D.G tem precedência sobre C.G E.F e tem precedência sobre D.F e C.F .

## Delegar invocações

Para uma invocação de delegado, a *primary\_expression* do *invocation\_expression* deve ser um valor de um *delegate\_type*. Além disso, Considerando que a *delegate\_type* seja um membro de função com a mesma lista de parâmetros que a *delegate\_type*, a *delegate\_type* deve ser aplicável ([membro de função aplicável](#)) em relação ao *argument\_list* do *invocation\_expression*.

O processamento em tempo de execução de uma invocação de delegado do formulário `D(A)` , em que `D` é uma *primary\_expression* de um *delegate\_type* e `A` é um *argument\_list* opcional, consiste nas seguintes etapas:

- `D` é avaliado. Se essa avaliação causar uma exceção, nenhuma etapa adicional será executada.
- O valor de `D` é verificado para ser válido. Se o valor de `D` for `null` , um `System.NullReferenceException` será lançado e nenhuma etapa adicional será executada.
- Caso contrário, `D` é uma referência a uma instância delegada. Invocações de membro de função ([verificação de tempo de compilação de resolução dinâmica de](#)

[sobrecarga](#)) são executadas em cada uma das entidades que podem ser chamadas na lista de invocação do delegado. Para entidades que podem ser chamadas que consistem em um método de instância e instância, a instância para a invocação é a instância contida na entidade que pode ser chamada.

## Acesso a elemento

Um *element\_access* consiste em um *primary\_no\_array\_creation\_expression*, seguido por um `[` token "", seguido por um *argument\_list*, seguido por um `]` token "". O *argument\_list* consiste em um ou mais s de *argumento*, separados por vírgulas.

```
antlr
```

```
element_access
    : primary_no_array_creation_expression '[' expression_list ']'
    ;
```

O *argument\_list* de um *element\_access* não tem permissão para conter `ref` ou `out` argumentos.

Uma *element\_access* é vinculada dinamicamente ([associação dinâmica](#)) se pelo menos uma das seguintes isenções:

- O *primary\_no\_array\_creation\_expression* tem tipo de tempo de compilação `dynamic`
- Pelo menos uma expressão da *argument\_list* tem tipo de tempo de compilação `dynamic` e o *primary\_no\_array\_creation\_expression* não tem um tipo de matriz.

Nesse caso, o compilador classifica o *element\_access* como um valor do tipo `dynamic`. As regras abaixo para determinar o significado da *element\_access* são então aplicadas em tempo de execução, usando o tipo de tempo de execução em vez do tipo de tempo de compilação das expressões *primary\_no\_array\_creation\_expression* e *argument\_list* que têm o tipo de tempo de compilação `dynamic`. Se o *primary\_no\_array\_creation\_expression* não tiver o tipo de tempo de compilação `dynamic`, o acesso ao elemento passará por uma verificação de tempo de compilação limitada, conforme descrito em [verificação de tempo de compilação da resolução dinâmica de sobrecarga](#).

Se a *primary\_no\_array\_creation\_expression* de um *element\_access* for um valor de um *array\_type*, o *element\_access* será um acesso à matriz ([acesso à matriz](#)). Caso contrário, o *primary\_no\_array\_creation\_expression* deve ser uma variável ou um valor de uma classe,

struct ou tipo de interface que tenha um ou mais membros do indexador; nesse caso, a *element\_access* é um acesso do indexador ([acesso ao indexador](#)).

## Acesso de matriz

Para um acesso de matriz, o *primary\_no\_array\_creation\_expression* do *element\_access* deve ser um valor de um *array\_type*. Além disso, o *argument\_list* de um acesso à matriz não tem permissão para conter argumentos nomeados. O número de expressões na *argument\_list* deve ser igual à classificação da *array\_type*, e cada expressão deve ser do tipo `int` , `uint` , `long` , `ulong` ou deve ser conversível implicitamente em um ou mais desses tipos.

O resultado da avaliação de um acesso à matriz é uma variável do tipo de elemento da matriz, ou seja, o elemento de matriz selecionado pelo (s) valor (es) das expressões no *argument\_list*.

O processamento em tempo de execução de um acesso de matriz do formulário `P[A]` , em que `P` é uma *primary\_no\_array\_creation\_expression* de um *array\_type* e `A` é um *argument\_list*, consiste nas seguintes etapas:

- `P` é avaliado. Se essa avaliação causar uma exceção, nenhuma etapa adicional será executada.
- As expressões de índice da *argument\_list* são avaliadas na ordem, da esquerda para a direita. Após a avaliação de cada expressão de índice, uma conversão implícita ([conversões implícitas](#)) para um dos seguintes tipos é executada `int` :  
`uint` , `long` , `ulong` . O primeiro tipo nesta lista para o qual existe uma conversão implícita é escolhido. Por exemplo, se a expressão de índice for do tipo `short` , uma conversão implícita em `int` será executada, já que conversões implícitas de `short` para `int` e de `short` para `long` são possíveis. Se a avaliação de uma expressão de índice ou a conversão implícita subsequente causar uma exceção, não serão avaliadas outras expressões de índice e nenhuma etapa adicional será executada.
- O valor de `P` é verificado para ser válido. Se o valor de `P` for `null` , um `System.NullReferenceException` será lançado e nenhuma etapa adicional será executada.
- O valor de cada expressão na *argument\_list* é verificado em relação aos limites reais de cada dimensão da instância de matriz referenciada por `P` . Se um ou mais valores estiverem fora do intervalo, um `System.IndexOutOfRangeException` será lançado e nenhuma etapa adicional será executada.
- O local do elemento da matriz fornecido pelas expressões de índice é computado e esse local se torna o resultado do acesso à matriz.

## Acesso de indexador

Para um acesso de indexador, o *primary\_no\_array\_creation\_expression* do *element\_access* deve ser uma variável ou um valor de uma classe, struct ou tipo de interface, e esse tipo deve implementar um ou mais indexadores que são aplicáveis em relação ao *argument\_list* do *element\_access*.

O processamento de tempo de vinculação de um acesso de indexador do formulário  $P[A]$ , em que  $P$  é uma *primary\_no\_array\_creation\_expression* de uma classe, estrutura ou tipo de interface  $T$ , e  $A$  é um *argument\_list*, consiste nas seguintes etapas:

- O conjunto de indexadores fornecido pelo  $T$  é construído. O conjunto consiste em todos os indexadores declarados em  $T$  ou em um tipo base  $T$  que não são *override* declarações e que são acessíveis no contexto atual ([acesso de membro](#)).
- O conjunto é reduzido para os indexadores aplicáveis e não ocultos por outros indexadores. As regras a seguir são aplicadas a cada indexador  $S.I$  no conjunto, em que  $S$  é o tipo no qual o indexador  $I$  é declarado:
  - Se  $I$  não for aplicável em relação ao  $A$  ([membro da função aplicável](#)),  $I$  será removido do conjunto.
  - Se  $I$  for aplicável em relação ao  $A$  ([membro de função aplicável](#)), todos os indexadores declarados em um tipo base de  $S$  serão removidos do conjunto.
  - Se  $I$  for aplicável em relação ao  $A$  ([membro de função aplicável](#)) e  $S$  for um tipo de classe diferente de `object`, todos os indexadores declarados em uma interface serão removidos do conjunto.
- Se o conjunto resultante de indexadores candidatos estiver vazio, não existirá nenhum indexador aplicável e ocorrerá um erro de tempo de ligação.
- O melhor indexador do conjunto de indexadores candidatos é identificado usando as regras de resolução de sobrecarga da [resolução de sobrecarga](#). Se um único melhor indexador não puder ser identificado, o acesso ao indexador será ambíguo e ocorrerá um erro de tempo de ligação.
- As expressões de índice da *argument\_list* são avaliadas na ordem, da esquerda para a direita. O resultado do processamento do acesso ao indexador é uma expressão classificada como um acesso de indexador. A expressão de acesso do indexador faz referência ao indexador determinado na etapa acima e tem uma expressão de instância associada  $P$  e uma lista de argumentos associados de  $A$ .

Dependendo do contexto no qual ele é usado, um acesso ao indexador causa a invocação do *acessador get* ou do *acessador set* do indexador. Se o acesso do indexador for o destino de uma atribuição, o *acessador set* será invocado para atribuir um novo valor ([atribuição simples](#)). Em todos os outros casos, o *acessador get* é invocado para obter o valor atual ([valores de expressões](#)).

## Este acesso

Uma *this\_access* consiste na palavra reservada `this`.

```
antlr  
  
this_access  
: 'this'  
;
```

Um *this\_access* é permitido apenas no *bloco* de um construtor de instância, um método de instância ou um acessador de instância. Ele tem um dos seguintes significados:

- Quando `this` é usado em uma *primary\_expression* dentro de um construtor de instância de uma classe, ele é classificado como um valor. O tipo do valor é o tipo de instância ([o tipo de instância](#)) da classe dentro da qual o uso ocorre e o valor é uma referência ao objeto que está sendo construído.
- Quando `this` é usado em um *primary\_expression* dentro de um método de instância ou acessador de instância de uma classe, ele é classificado como um valor. O tipo do valor é o tipo de instância ([o tipo de instância](#)) da classe dentro da qual o uso ocorre e o valor é uma referência ao objeto para o qual o método ou acessador foi invocado.
- Quando `this` é usado em uma *primary\_expression* dentro de um construtor de instância de uma struct, ele é classificado como uma variável. O tipo da variável é o tipo de instância ([o tipo de instância](#)) da estrutura dentro da qual o uso ocorre e a variável representa a estrutura que está sendo construída. A `this` variável de um construtor de instância de um struct se comporta exatamente como um `out` parâmetro do tipo struct — em particular, isso significa que a variável deve ser definitivamente atribuída em cada caminho de execução do construtor da instância.
- Quando `this` é usado em um *primary\_expression* dentro de um método de instância ou acessador de instância de uma struct, ele é classificado como uma variável. O tipo da variável é o tipo de instância ([o tipo de instância](#)) da estrutura na qual o uso ocorre.
  - Se o método ou acessador não for um iterador ([iteradores](#)), a `this` variável representará a estrutura para a qual o método ou acessador foi invocado e se comlatará exatamente como um `ref` parâmetro do tipo struct.
  - Se o método ou o acessador for um iterador, a `this` variável representará uma cópia da estrutura para a qual o método ou acessador foi invocado e se comlatará exatamente como um parâmetro de valor do tipo struct.

O uso de `this` em um *primary\_expression* em um contexto diferente daqueles listados acima é um erro de tempo de compilação. Em particular, não é possível referir-se a um `this` método estático, um acessador de propriedade estática ou em uma *variable\_initializer* de uma declaração de campo.

## Acesso de base

Uma *base\_access* consiste na palavra reservada `base` seguida por um token " `.` " e um identificador ou um *argument\_list* entre colchetes:

antlr

```
base_access
  : 'base' '.' identifier
  | 'base' '[' expression_list ']'
  ;
```

Uma *base\_access* é usada para acessar membros da classe base que são ocultados por membros nomeados de forma semelhante na classe ou struct atual. Um *base\_access* é permitido apenas no *bloco* de um construtor de instância, um método de instância ou um acessador de instância. Quando `base.I` ocorre em uma classe ou struct, `I` o deve indicar um membro da classe base dessa classe ou struct. Da mesma forma, quando `base[E]` ocorre em uma classe, um indexador aplicável deve existir na classe base.

No momento da associação, *base\_access* expressões do formulário `base.I` e `base[E]` são avaliadas exatamente como se elas fossem gravadas `((B)this).I` e `((B)this)[E]`, em que `B` é a classe base da classe ou struct na qual a construção ocorre. Assim, `base.I` e `base[E]` correspondem a `this.I` e `this[E]`, exceto pelo, `this` é exibido como uma instância da classe base.

Quando um *base\_access* referencia um membro de função virtual (um método, uma propriedade ou um indexador), a determinação de qual membro de função invocar em tempo de execução ([verificação de tempo de compilação da resolução dinâmica de sobrecarga](#)) é alterada. O membro da função invocado é determinado pela localização da implementação mais derivada ([métodos virtuais](#)) do membro da função em relação a `B` (em vez de em relação ao tipo de tempo de execução de `this`, como seria usual em um acesso não-base). Portanto, dentro `override` de um de um `virtual` membro de função, um *base\_access* pode ser usado para invocar a implementação herdada do membro da função. Se o membro da função referenciado por uma *base\_access* for `abstract`, ocorrerá um erro de tempo de associação.

# Operadores de incremento e decremento pós-fixados

```
antlr

post_incremet_expression
: primary_expression '++'
;

post_decrement_expression
: primary_expression '--'
;
```

O operando de um incremento de sufixo ou uma operação de decréscimo deve ser uma expressão classificada como uma variável, um acesso de propriedade ou um acesso de indexador. O resultado da operação é um valor do mesmo tipo que o operando.

Se o *primary\_expression* tiver o tipo de tempo de compilação `dynamic`, o operador será vinculado dinamicamente ([associação dinâmica](#)), o *post\_incremet\_expression* ou *post\_decrement\_expression* tem o tipo de tempo `dynamic` de compilação e as regras a seguir são aplicadas em tempo de execução usando o tipo de tempo de execução do *primary\_expression*.

Se o operando de um incremento de sufixo ou uma operação de decréscimo for um acesso de propriedade ou indexador, a propriedade ou o indexador deverá ter um `get` e um `set` acessador. Se esse não for o caso, ocorrerá um erro de tempo de associação.

Resolução de sobrecarga de operador unário ([resolução de sobrecarga de operador unário](#)) é aplicada para selecionar uma implementação de operador específica. `++` Os operadores e predefinidos `--` existem para os seguintes tipos: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` e qualquer tipo de enumeração. Os operadores predefinidos `++` retornam o valor produzido pela adição de 1 ao operando, e os operadores predefinidos `--` retornam o valor produzido pela subtração de 1 do operando. Em um `checked` contexto, se o resultado dessa adição ou subtração estiver fora do intervalo do tipo de resultado e o tipo de resultado for um tipo integral ou tipo de enumeração, um `System.OverflowException` será gerado.

O processamento em tempo de execução de um incremento de sufixo ou uma operação de diminuição do formulário `x++` ou `x--` consiste nas seguintes etapas:

- Se `x` é classificado como uma variável:
  - `x` é avaliado para produzir a variável.
  - O valor de `x` é salvo.
  - O operador selecionado é invocado com o valor salvo `x` como seu argumento.

- O valor retornado pelo operador é armazenado no local fornecido pela avaliação de `x`.
- O valor salvo de `x` se torna o resultado da operação.
- Se `x` é classificado como um acesso de propriedade ou indexador:
  - A expressão de instância (se `x` não for `static`) e a lista de argumentos (se `x` for um acesso de indexador) associada a `x` são avaliadas e os resultados são usados nas `get` invocações subsequentes e `set` acessadores.
  - O `get` acessador do `x` é invocado e o valor retornado é salvo.
  - O operador selecionado é invocado com o valor salvo `x` como seu argumento.
  - O `set` acessador de `x` é invocado com o valor retornado pelo operador como seu `value` argumento.
  - O valor salvo de `x` se torna o resultado da operação.

Os `++` `--` operadores e também dão suporte à notação de prefixo ([operadores de incremento e diminuição de prefixo](#)). Normalmente, o resultado de `x++` ou `x--` é o valor de `x` antes da operação, enquanto o resultado de `++x` ou `--x` é o valor de `x` após a operação. Em ambos os casos, `x` ele tem o mesmo valor após a operação.

Uma `operator ++` `operator --` implementação ou pode ser chamada usando o sufixo ou a notação de prefixo. Não é possível ter implementações de operador separadas para as duas notações.

## O novo operador

O `new` operador é usado para criar novas instâncias de tipos.

Há três formas de `new` expressões:

- As expressões de criação de objeto são usadas para criar novas instâncias de tipos de classe e tipos de valor.
- As expressões de criação de matriz são usadas para criar novas instâncias de tipos de matriz.
- As expressões de criação de representante são usadas para criar novas instâncias de tipos delegados.

O `new` operador implica a criação de uma instância de um tipo, mas não implica necessariamente a alocação dinâmica da memória. Em particular, as instâncias de tipos de valor não exigem memória adicional além das variáveis em que residem, e nenhuma alocação dinâmica ocorre quando `new` é usada para criar instâncias de tipos de valor.

## Expressões de criação de objeto

Um *object\_creation\_expression* é usado para criar uma nova instância de um *class\_type* ou um *value\_type*.

```
antlr

object_creation_expression
: 'new' type '(' argument_list? ')' object_or_collection_initializer?
| 'new' type object_or_collection_initializer
;

object_or_collection_initializer
: object_initializer
| collection_initializer
;
```

O *tipo* de um *object\_creation\_expression* deve ser um *class\_type*, um *value\_type* ou um *type\_parameter*. O *tipo* não pode ser um *abstract class\_type*.

O *argument\_list* opcional ([listas de argumentos](#)) só será permitido se o *tipo* for um *class\_type* ou um *struct\_type*.

Uma expressão de criação de objeto pode omitir a lista de argumentos do construtor e os parênteses delimitadores fornecidos incluem um inicializador de objeto ou inicializador de coleção. Omitir a lista de argumentos do construtor e parênteses delimitadores é equivalente a especificar uma lista de argumentos vazia.

O processamento de uma expressão de criação de objeto que inclui um inicializador de objeto ou inicializador de coleção consiste no primeiro processamento do construtor de instância e no processamento das inicializações de membro ou elemento especificadas pelo inicializador de objeto ([inicializadores de objeto](#)) ou pelo inicializador de coleção ([inicializadores de coleção](#)).

Se qualquer um dos argumentos no *argument\_list* opcional tiver o tipo de tempo de compilação *dynamic*, a *object\_creation\_expression* será vinculada dinamicamente ([associação dinâmica](#)) e as regras a seguir serão aplicadas em tempo de execução usando o tipo de tempo de execução desses argumentos da *argument\_list* que têm o tipo de tempo de compilação *dynamic*. No entanto, a criação do objeto passa por uma verificação de tempo de compilação limitada, conforme descrito em [verificação de tempo de compilação da resolução dinâmica de sobrecarga](#).

O processamento de tempo de associação de um *object\_creation\_expression* do formulário `new T(A)`, em que `T` é um *class\_type* ou um *value\_type* e `A` é um *argument\_list* opcional, consiste nas seguintes etapas:

- Se `T` é um *value\_type* e `A` não está presente:
  - O *object\_creation\_expression* é uma invocação de construtor padrão. O resultado da *object\_creation\_expression* é um valor do tipo, ou seja `T`, o valor padrão para, `T` conforme definido no [tipo System. ValueType](#).
- Caso contrário, se `T` for um *type\_parameter* e `A` não estiver presente:
  - Se nenhuma restrição de tipo de valor ou restrição de Construtor ([restrições de parâmetro de tipo](#)) tiver sido especificada para `T`, ocorrerá um erro de tempo de associação.
  - O resultado da *object\_creation\_expression* é um valor do tipo de tempo de execução ao qual o parâmetro de tipo foi associado, ou seja, o resultado da invocação do construtor padrão desse tipo. O tipo de tempo de execução pode ser um tipo de referência ou um tipo de valor.
- Caso contrário, se `T` for um *class\_type* ou um *struct\_type*:
  - Se `T` for um *abstract class\_type*, ocorrerá um erro em tempo de compilação.
  - O construtor de instância a ser invocado é determinado usando as regras de resolução de sobrecarga da [resolução de sobrecarga](#). O conjunto de construtores de instância candidata consiste em todos os construtores de instância acessíveis declarados no `T` que são aplicáveis em relação a `A` ([membro de função aplicável](#)). Se o conjunto de construtores de instância de candidato estiver vazio, ou se um único Construtor de instância recomendada não puder ser identificado, ocorrerá um erro de tempo de associação.
  - O resultado da *object\_creation\_expression* é um valor do tipo, ou seja `T`, o valor produzido por invocar o construtor da instância determinado na etapa acima.
- Caso contrário, o *object\_creation\_expression* é inválido e ocorre um erro de tempo de ligação.

Mesmo que a *object\_creation\_expression* seja vinculada dinamicamente, o tipo de tempo de compilação ainda será `T`.

O processamento em tempo de execução de um *object\_creation\_expression* do formulário `new T(A)`, em que `T` é *class\_type* ou uma *struct\_type* e `A` é um *argument\_list* opcional, consiste nas seguintes etapas:

- Se `T` é um *class\_type*:
  - Uma nova instância da classe `T` é alocada. Se não houver memória suficiente disponível para alocar a nova instância, um `System.OutOfMemoryException` será lançado e nenhuma etapa adicional será executada.
  - Todos os campos da nova instância são inicializados para seus valores padrão ([valores padrão](#)).
  - O construtor de instância é invocado de acordo com as regras de invocação de membro de função ([verificação de tempo de compilação da resolução dinâmica](#)

de sobrecarga). Uma referência à instância alocada recentemente é passada automaticamente para o construtor de instância e a instância pode ser acessada de dentro desse construtor como `this`.

- Se `T` é um *struct\_type*:
  - Uma instância do tipo `T` é criada alocando-se uma variável local temporária. Como um construtor de instância de um *struct\_type* é necessário para atribuir definitivamente um valor a cada campo da instância que está sendo criada, nenhuma inicialização da variável temporária é necessária.
  - O construtor de instância é invocado de acordo com as regras de invocação de membro de função ([verificação de tempo de compilação da resolução dinâmica de sobrecarga](#)). Uma referência à instância alocada recentemente é passada automaticamente para o construtor de instância e a instância pode ser acessada de dentro desse construtor como `this`.

## Inicializadores de objeto

Um *inicializador de objeto* especifica valores para zero ou mais campos, propriedades ou elementos indexados de um objeto.

```
antlr

object_initializer
: '{' member_initializer_list? '}'
| '{' member_initializer_list ',' '}'
;

member_initializer_list
: member_initializer (',' member_initializer)*
;

member_initializer
: initializer_target '=' initializer_value
;

initializer_target
: identifier
| '[' argument_list ']'
;

initializer_value
: expression
| object_or_collection_initializer
;
```

Um inicializador de objeto consiste em uma sequência de inicializadores de membro, entre os `{ }` tokens e separados por vírgulas. Cada *member\_initializer* designa um

destino para a inicialização. Um *identificador* deve nomear um campo ou Propriedade acessível do objeto que está sendo inicializado, enquanto um *argument\_list* entre colchetes deve especificar argumentos para um indexador acessível no objeto que está sendo inicializado. É um erro para um inicializador de objeto incluir mais de um inicializador de membro para o mesmo campo ou propriedade.

Cada *initializer\_target* é seguida por um sinal de igual e uma expressão, um inicializador de objeto ou um inicializador de coleção. Não é possível que as expressões no inicializador de objeto façam referência ao objeto recém-criado que está sendo inicializado.

Um inicializador de membro que especifica uma expressão após o sinal de igual é processado da mesma maneira que uma atribuição ([atribuição simples](#)) para o destino.

Um inicializador de membro que especifica um inicializador de objeto após o sinal de igual é um *inicializador de objeto aninhado*, ou seja, uma inicialização de um objeto inserido. Em vez de atribuir um novo valor ao campo ou à propriedade, as atribuições no inicializador de objeto aninhado são tratadas como atribuições para membros do campo ou da propriedade. Inicializadores de objeto aninhados não podem ser aplicados a propriedades com um tipo de valor, ou a campos somente leitura com um tipo de valor.

Um inicializador de membro que especifica um inicializador de coleção após o sinal de igual é uma inicialização de uma coleção inserida. Em vez de atribuir uma nova coleção ao campo de destino, à propriedade ou ao indexador, os elementos fornecidos no inicializador são adicionados à coleção referenciada pelo destino. O destino deve ser de um tipo de coleção que satisfaça os requisitos especificados em [inicializadores de coleção](#).

Os argumentos para um inicializador de índice sempre serão avaliados exatamente uma vez. Portanto, mesmo se os argumentos acabarem nunca sendo usados (por exemplo, devido a um inicializador aninhado vazio), eles serão avaliados para seus efeitos colaterais.

A classe a seguir representa um ponto com duas coordenadas:

C#

```
public class Point
{
    int x, y;

    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```

Uma instância do `Point` pode ser criada e inicializada da seguinte maneira:

C#

```
Point a = new Point { X = 0, Y = 1 };
```

que tem o mesmo efeito que

C#

```
Point __a = new Point();
__a.X = 0;
__a.Y = 1;
Point a = __a;
```

em que `__a` é uma variável temporária invisível e inacessível. A classe a seguir representa um retângulo criado a partir de dois pontos:

C#

```
public class Rectangle
{
    Point p1, p2;

    public Point P1 { get { return p1; } set { p1 = value; } }
    public Point P2 { get { return p2; } set { p2 = value; } }
}
```

Uma instância do `Rectangle` pode ser criada e inicializada da seguinte maneira:

C#

```
Rectangle r = new Rectangle {
    P1 = new Point { X = 0, Y = 1 },
    P2 = new Point { X = 2, Y = 3 }
};
```

que tem o mesmo efeito que

C#

```
Rectangle __r = new Rectangle();
Point __p1 = new Point();
__p1.X = 0;
__p1.Y = 1;
__r.P1 = __p1;
```

```
Point __p2 = new Point();
__p2.X = 2;
__p2.Y = 3;
__r.P2 = __p2;
Rectangle r = __r;
```

onde `__r` `__p1` e `__p2` são variáveis temporárias que, de outra forma, são invisíveis e inacessíveis.

`Rectangle` O Construtor If aloca as duas instâncias inseridas `Point`

C#

```
public class Rectangle
{
    Point p1 = new Point();
    Point p2 = new Point();

    public Point P1 { get { return p1; } }
    public Point P2 { get { return p2; } }
}
```

a construção a seguir pode ser usada para inicializar as `Point` instâncias inseridas em vez de atribuir novas instâncias:

C#

```
Rectangle r = new Rectangle {
    P1 = { X = 0, Y = 1 },
    P2 = { X = 2, Y = 3 }
};
```

que tem o mesmo efeito que

C#

```
Rectangle __r = new Rectangle();
__r.P1.X = 0;
__r.P1.Y = 1;
__r.P2.X = 2;
__r.P2.Y = 3;
Rectangle r = __r;
```

Dada uma definição apropriada de C, o exemplo a seguir:

C#

```

var c = new C {
    x = true,
    y = { a = "Hello" },
    z = { 1, 2, 3 },
    ["x"] = 5,
    [0,0] = { "a", "b" },
    [1,2] = {}
};

```

é equivalente a esta série de atribuições:

C#

```

C __c = new C();
__c.x = true;
__c.y.a = "Hello";
__c.z.Add(1);
__c.z.Add(2);
__c.z.Add(3);
string __i1 = "x";
__c[__i1] = 5;
int __i2 = 0, __i3 = 0;
__c[__i2,__i3].Add("a");
__c[__i2,__i3].Add("b");
int __i4 = 1, __i5 = 2;
var c = __c;

```

em que `__c`, etc., são variáveis geradas que são invisíveis e inacessíveis para o código-fonte. Observe que os argumentos para `[0,0]` são avaliados apenas uma vez, e os argumentos para `[1,2]` são avaliados uma vez, mesmo que nunca sejam usados.

## Inicializadores de coleção

Um inicializador de coleção especifica os elementos de uma coleção.

antlr

```

collection_initializer
: '{' element_initializer_list '}'
| '{' element_initializer_list ',' '}'
;

element_initializer_list
: element_initializer (',' element_initializer)*
;

element_initializer
: non_assignment_expression
;
```

```
| '{' expression_list '}'  
;  
  
expression_list  
: expression (',' expression)*  
;
```

Um inicializador de coleção consiste em uma sequência de inicializadores de elemento, entre os `{ }`  tokens e separados por vírgulas. Cada inicializador de elemento especifica um elemento a ser adicionado ao objeto de coleção que está sendo inicializado e consiste em uma lista de expressões entre os `{ }`  tokens e separados por vírgulas. Um inicializador de elemento de uma única expressão pode ser gravado sem chaves, mas não pode ser uma expressão de atribuição para evitar ambigüidade com inicializadores de membro. A produção *non\_assignment\_expression* é definida na [expressão](#).

Veja a seguir um exemplo de uma expressão de criação de objeto que inclui um inicializador de coleção:

C#

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

O objeto de coleção ao qual um inicializador de coleção é aplicado deve ser de um tipo que implementa `System.Collections.IEnumerable` ou ocorre um erro em tempo de compilação. Para cada elemento especificado na ordem, o inicializador de coleção invoca um `Add` método no objeto de destino com a lista de expressões do inicializador de elemento como lista de argumentos, aplicando a resolução de membro normal e a solução de sobrecarga para cada invocação. Portanto, o objeto de coleção deve ter uma instância aplicável ou um método de extensão com o nome `Add` para cada inicializador de elemento.

A classe a seguir representa um contato com um nome e uma lista de números de telefone:

C#

```
public class Contact  
{  
    string name;  
    List<string> phoneNumbers = new List<string>();  
  
    public string Name { get { return name; } set { name = value; } }  
  
    public List<string> PhoneNumbers { get { return phoneNumbers; } }  
}
```

Um `List<Contact>` pode ser criado e inicializado da seguinte maneira:

C#

```
var contacts = new List<Contact> {
    new Contact {
        Name = "Chris Smith",
        PhoneNumbers = { "206-555-0101", "425-882-8080" }
    },
    new Contact {
        Name = "Bob Harris",
        PhoneNumbers = { "650-555-0199" }
    }
};
```

que tem o mesmo efeito que

C#

```
var __clist = new List<Contact>();
Contact __c1 = new Contact();
__c1.Name = "Chris Smith";
__c1.PhoneNumbers.Add("206-555-0101");
__c1.PhoneNumbers.Add("425-882-8080");
__clist.Add(__c1);
Contact __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("650-555-0199");
__clist.Add(__c2);
var contacts = __clist;
```

onde `__clist` `__c1` e `__c2` são variáveis temporárias que, de outra forma, são invisíveis e inacessíveis.

## Expressões de criação de matriz

Um `array_creation_expression` é usado para criar uma nova instância de um `array_type`.

antlr

```
array_creation_expression
  : 'new' non_array_type '[' expression_list ']' rank_specifier*
array_initializer?
  | 'new' array_type array_initializer
  | 'new' rank_specifier array_initializer
  ;
```

Uma expressão de criação de matriz do primeiro formulário aloca uma instância de matriz do tipo que resulta da exclusão de cada uma das expressões individuais da lista de expressões. Por exemplo, a expressão de criação de matriz `new int[10, 20]` produz uma instância de matriz do tipo `int[, ]` e a expressão de criação de matriz `new int[10][, ]` produz uma matriz do tipo `int[][]`. Cada expressão na lista de expressões deve ser do tipo `int`, `uint`, `long` ou `ulong`, ou conversível implicitamente em um ou mais desses tipos. O valor de cada expressão determina o comprimento da dimensão correspondente na instância de matriz alocada recentemente. Como o comprimento de uma dimensão de matriz deve ser não negativo, é um erro de tempo de compilação ter um *constant\_expression* com um valor negativo na lista de expressões.

Exceto em um contexto não seguro ([contextos não seguros](#)), o layout das matrizes não é especificado.

Se uma expressão de criação de matriz do primeiro formulário incluir um inicializador de matriz, cada expressão na lista de expressões deverá ser uma constante e os comprimentos de classificação e de dimensão especificados pela lista de expressões devem corresponder àqueles do inicializador de matriz.

Em uma expressão de criação de matriz do segundo ou terceiro formulário, a classificação do tipo de matriz ou especificador de classificação especificado deve corresponder à do inicializador de matriz. Os comprimentos de dimensão individuais são inferidos do número de elementos em cada um dos níveis de aninhamento correspondentes do inicializador de matriz. Portanto, a expressão

```
C#
```

```
new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

corresponde exatamente a

```
C#
```

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

Uma expressão de criação de matriz do terceiro formulário é conhecida como uma **\*expressão de criação de matriz digitada implicitamente**. Ele é semelhante ao segundo formulário, exceto pelo fato de que o tipo de elemento da matriz não é fornecido explicitamente, mas determinado como o melhor tipo comum ([encontrando o melhor tipo comum de um conjunto de expressões](#)) do conjunto de expressões no inicializador de matriz. Para uma matriz multidimensional, ou seja, uma em que o `_rankSpecifier` \*

contém pelo menos uma vírgula, esse conjunto inclui todas as *expressões* s encontradas em *array\_initializer* s aninhadas.

Inicializadores de matriz são descritos mais detalhadamente em [inicializadores de matriz](#).

O resultado da avaliação de uma expressão de criação de matriz é classificado como um valor, ou seja, uma referência à instância de matriz alocada recentemente. O processamento em tempo de execução de uma expressão de criação de matriz consiste nas seguintes etapas:

- As expressões de comprimento da dimensão da *expression\_list* são avaliadas na ordem, da esquerda para a direita. Após a avaliação de cada expressão, uma conversão implícita ([conversões implícitas](#)) para um dos seguintes tipos é executada: `int` , `uint` , `long` , `ulong` . O primeiro tipo nesta lista para o qual existe uma conversão implícita é escolhido. Se a avaliação de uma expressão ou a conversão implícita subsequente causar uma exceção, nenhuma expressão adicional será avaliada e nenhuma outra etapa será executada.
- Os valores computados para os comprimentos das dimensões são validados da seguinte maneira. Se um ou mais valores forem menores que zero, um `System.OverflowException` será lançado e nenhuma etapa adicional será executada.
- Uma instância de matriz com os tamanhos de dimensão fornecidos é alocada. Se não houver memória suficiente disponível para alocar a nova instância, um `System.OutOfMemoryException` será lançado e nenhuma etapa adicional será executada.
- Todos os elementos da nova instância de matriz são inicializados para seus valores padrão ([valores padrão](#)).
- Se a expressão de criação de matriz contiver um inicializador de matriz, cada expressão no inicializador de matriz será avaliada e atribuída ao elemento de matriz correspondente. As avaliações e as atribuições são executadas na ordem em que as expressões são gravadas no inicializador de matriz – em outras palavras, os elementos são inicializados em ordem de índice crescente, com a dimensão mais à direita aumentando primeiro. Se a avaliação de uma determinada expressão ou a atribuição subsequente ao elemento de matriz correspondente causar uma exceção, nenhum elemento adicional será inicializado (e os elementos restantes terão, portanto, seus valores padrão).

Uma expressão de criação de matriz permite instanciação de uma matriz com elementos de um tipo de matriz, mas os elementos de tal matriz devem ser inicializados manualmente. Por exemplo, a instrução

```
int[][] a = new int[100][];
```

Cria uma matriz unidimensional com 100 elementos do tipo `int[]`. O valor inicial de cada elemento é `null`. Não é possível que a mesma expressão de criação de matriz também instancia as submatrizes e a instrução

C#

```
int[][] a = new int[100][5];           // Error
```

resulta em um erro de tempo de compilação. Em vez disso, a instanciação das submatrizes deve ser executada manualmente, como em

C#

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

Quando uma matriz de matrizes tem uma forma "retangular", ou seja, quando as submatrizes têm o mesmo comprimento, é mais eficiente usar uma matriz multidimensional. No exemplo acima, a instanciação da matriz de matrizes cria 101 objetos — uma matriz externa e submatrizes 100. Por outro lado,

C#

```
int[,] = new int[100, 5];
```

cria apenas um único objeto, uma matriz bidimensional e realiza a alocação em uma única instrução.

Veja a seguir exemplos de expressões de criação de matriz digitadas implicitamente:

C#

```
var a = new[] { 1, 10, 100, 1000 };           // int[]
var b = new[] { 1, 1.5, 2, 2.5 };             // double[]
var c = new[,] { { "hello", null }, { "world", "!" } }; // string[,]
var d = new[] { 1, "one", 2, "two" };          // Error
```

A última expressão causa um erro de tempo de compilação porque nem `int` nem `string` é implicitamente conversível para a outra e, portanto, não há um melhor tipo

comum. Uma expressão de criação de matriz tipada explicitamente deve ser usada neste caso, por exemplo, especificando o tipo a ser `object[]`. Como alternativa, um dos elementos pode ser convertido em um tipo base comum, que se tornaria o tipo de elemento inferido.

As expressões de criação de matriz com tipo implícito podem ser combinadas com inicializadores de objeto anônimo ([expressões de criação de objeto anônimo](#)) para criar estruturas de dados com tipo anônimo. Por exemplo:

C#

```
var contacts = new[] {
    new {
        Name = "Chris Smith",
        PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
    },
    new {
        Name = "Bob Harris",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

## Delegar expressões de criação

Uma `delegate_creation_expression` é usada para criar uma nova instância de um `delegate_type`.

antlr

```
delegate_creation_expression
: 'new' delegate_type '(' expression ')'
;
```

O argumento de uma expressão de criação de delegado deve ser um grupo de métodos, uma função anônima ou um valor do tipo de tempo de compilação `dynamic` ou um `delegate_type`. Se o argumento for um grupo de métodos, ele identificará o método e, para um método de instância, o objeto para o qual criar um delegado. Se o argumento for uma função anônima, ele definirá diretamente os parâmetros e o corpo do método do destino delegado. Se o argumento for um valor, ele identificará uma instância de delegado da qual criar uma cópia.

Se a `expressão` tiver o tipo de tempo de compilação `dynamic`, a `delegate_creation_expression` será vinculada dinamicamente ([associação dinâmica](#)) e as regras abaixo serão aplicadas em tempo de execução usando o tipo de tempo de

execução da *expressão*. Caso contrário, as regras são aplicadas em tempo de compilação.

O processamento de tempo de associação de um *delegate\_creation\_expression* do formulário `new D(E)`, em que `D` é uma *delegate\_type* e `E` é uma *expressão*, consiste nas seguintes etapas:

- Se `E` for um grupo de métodos, a expressão de criação de representante será processada da mesma maneira que uma conversão de grupo de métodos ([conversões de grupos de métodos](#)) de `E` para `D`.
- Se `E` for uma função anônima, a expressão de criação de delegado será processada da mesma maneira que uma conversão de função anônima ([conversões de função anônimas](#)) de `E` para `D`.
- Se `E` for um valor, `E` deve ser compatível ([delegar declarações](#)) com `D`, e o resultado é uma referência a um delegado recém-criado do tipo `D` que se refere à mesma lista de invocação que o `E`. Se `E` não for compatível com `D`, ocorrerá um erro em tempo de compilação.

O processamento em tempo de execução de uma *delegate\_creation\_expression* do formulário `new D(E)`, em que `D` é uma *delegate\_type* e `E` é uma *expressão*, consiste nas seguintes etapas:

- Se `E` for um grupo de métodos, a expressão de criação de representante será avaliada como uma conversão de grupo de métodos ([conversões de grupos de métodos](#)) de `E` para `D`.
- Se `E` for uma função anônima, a criação de delegado será avaliada como uma conversão de função anônima de `E` para `D` ([conversões de função anônimas](#)).
- Se `E` é um valor de uma *delegate\_type*:
  - `E` é avaliado. Se essa avaliação causar uma exceção, nenhuma etapa adicional será executada.
  - Se o valor de `E` for `null`, um `System.NullReferenceException` será lançado e nenhuma etapa adicional será executada.
  - Uma nova instância do tipo delegado `D` é alocada. Se não houver memória suficiente disponível para alocar a nova instância, um `System.OutOfMemoryException` será lançado e nenhuma etapa adicional será executada.
  - A nova instância de delegado é inicializada com a mesma lista de invocação que a instância de delegado fornecida pelo `E`.

A lista de invocação de um delegado é determinada quando o delegado é instanciado e, em seguida, permanece constante para todo o tempo de vida do delegado. Em outras

palavras, não é possível alterar as entidades de destino que podem ser chamadas de um delegado depois que ele tiver sido criado. Quando dois delegados são combinados ou um é removido de outro ([delegar declarações](#)), um novo delegado resulta; nenhum delegado existente tem seu conteúdo alterado.

Não é possível criar um delegado que se refere a uma propriedade, indexador, operador definido pelo usuário, Construtor de instância, destruidor ou construtor estático.

Conforme descrito acima, quando um delegado é criado a partir de um grupo de métodos, a lista de parâmetros formais e o tipo de retorno do delegado determinam qual dos métodos sobrecarregados selecionar. No exemplo

```
C#  
  
delegate double DoubleFunc(double x);  
  
class A  
{  
    DoubleFunc f = new DoubleFunc(Square);  
  
    static float Square(float x) {  
        return x * x;  
    }  
  
    static double Square(double x) {  
        return x * x;  
    }  
}
```

o `A.f` campo é inicializado com um delegado que se refere ao segundo `Square` método porque esse método corresponde exatamente à lista de parâmetros formais e ao tipo de retorno de `DoubleFunc`. Se o segundo `Square` método não estiver presente, ocorreria um erro em tempo de compilação.

## Expressões de criação de objeto anônimo

Um *anonymous\_object\_creation\_expression* é usado para criar um objeto de um tipo anônimo.

```
antlr  
  
anonymous_object_creation_expression  
: 'new' anonymous_object_initializer  
;  
  
anonymous_object_initializer  
: '{' member_declarator_list? '}'
```

```

| '{' member_declarator_list ',' '}'
;

member_declarator_list
: member_declarator (',' member_declarator)*
;

member_declarator
: simple_name
| member_access
| base_access
| null_conditional_member_access
| identifier '=' expression
;

```

Um inicializador de objeto anônimo declara um tipo anônimo e retorna uma instância desse tipo. Um tipo anônimo é um tipo de classe sem nome que herda diretamente do `object`. Os membros de um tipo anônimo são uma sequência de propriedades somente leitura inferidas do inicializador de objeto anônimo usado para criar uma instância do tipo. Especificamente, um inicializador de objeto anônimo do formulário

C#

```
new { p1 = e1, p2 = e2, ..., pn = en }
```

declara um tipo anônimo do formulário

C#

```

class __Anonymous1
{
    private readonly T1 f1;
    private readonly T2 f2;
    ...
    private readonly Tn fn;

    public __Anonymous1(T1 a1, T2 a2, ..., Tn an) {
        f1 = a1;
        f2 = a2;
        ...
        fn = an;
    }

    public T1 p1 { get { return f1; } }
    public T2 p2 { get { return f2; } }
    ...
    public Tn pn { get { return fn; } }

    public override bool Equals(object __o) { ... }
}

```

```
    public override int GetHashCode() { ... }  
}
```

onde cada `Tx` é o tipo da expressão correspondente `ex`. A expressão usada em um *member\_declarator* deve ter um tipo. Portanto, é um erro de tempo de compilação para uma expressão em um *member\_declarator* ser nulo ou uma função anônima. Também é um erro de tempo de compilação para que a expressão tenha um tipo não seguro.

Os nomes de um tipo anônimo e do parâmetro para seu `Equals` método são gerados automaticamente pelo compilador e não podem ser referenciados no texto do programa.

Dentro do mesmo programa, dois inicializadores de objeto anônimos que especificam uma sequência de propriedades dos mesmos nomes e tipos de tempo de compilação na mesma ordem produzirão instâncias do mesmo tipo anônimo.

No exemplo

```
C#  
  
var p1 = new { Name = "Lawnmower", Price = 495.00 };  
var p2 = new { Name = "Shovel", Price = 26.95 };  
p1 = p2;
```

a atribuição na última linha é permitida porque `p1` e `p2` são do mesmo tipo anônimo.

Os `Equals` `GetHashCode` métodos e em tipos anônimos substituem os métodos herdados de `object` são definidos em termos de `Equals` e `GetHashCode` das propriedades, de forma que duas instâncias do mesmo tipo anônimo sejam iguais se e somente se todas as suas propriedades forem iguais.

Um Declarador de membro pode ser abreviado como um nome simples ([inferência de tipo](#)), um acesso de membro ([verificação de tempo de compilação de resolução dinâmica de sobrecarga](#)), um acesso base ([acesso de base](#)) ou um membro condicional nulo de acesso ([expressões condicionais nulas como inicializadores de projeção](#)). Isso é chamado de ***inicializador de projeção*** e é abreviado para uma declaração de e atribuição a uma propriedade com o mesmo nome. Especificamente, os declaradores de membros dos formulários

```
C#  
  
identifier  
expr.identifier
```

são precisamente equivalentes aos seguintes, respectivamente:

```
C#
```

```
identifier = identifier  
identifier = expr.identifier
```

Portanto, em um inicializador de projeção, o *identificador* seleciona o valor e o campo ou propriedade ao qual o valor é atribuído. Intuitivamente, um inicializador de projeção projeta não apenas um valor, mas também o nome do valor.

## O operador `typeof`

O `typeof` operador é usado para obter o `System.Type` objeto para um tipo.

```
antlr
```

```
typeof_expression  
: 'typeof' '(' type ')'  
| 'typeof' '(' unbound_type_name ')'  
| 'typeof' '(' 'void' ')'  
;  
  
unbound_type_name  
: identifier generic_dimension_specifier?  
| identifier '::' identifier generic_dimension_specifier?  
| unbound_type_name '.' identifier generic_dimension_specifier?  
;  
  
generic_dimension_specifier  
: '<' comma* '>'  
;  
  
comma  
: ','  
;
```

A primeira forma de *typeof\_expression* consiste em uma `typeof` palavra-chave seguida por um *tipo* entre parênteses. O resultado de uma expressão desse formulário é o `System.Type` objeto para o tipo indicado. Há apenas um `System.Type` objeto para qualquer tipo específico. Isso significa que, para um tipo `T`, `typeof(T) == typeof(T)` é sempre verdadeiro. O *tipo* não pode ser `dynamic`.

A segunda forma de *typeof\_expression* consiste em uma `typeof` palavra-chave seguida por uma *unbound\_type\_name* entre parênteses. Um *unbound\_type\_name* é muito semelhante a um *type\_name* ([namespace e nomes de tipo](#)), exceto pelo fato de que um

*unbound\_type\_name* contém *generic\_dimensionSpecifier*s onde um *type\_name* contém *type\_argument\_list*s. Quando o operando de uma *typeof\_expression* é uma sequência de tokens que satisfaz as gramáticas de *unbound\_type\_name* e *type\_name*, ou seja, quando ele não contém uma *generic\_dimensionSpecifier* nem uma *type\_argument\_list*, a sequência de tokens é considerada uma *type\_name*. O significado de um *unbound\_type\_name* é determinado da seguinte maneira:

- Converte a sequência de tokens em um *type\_name* substituindo cada *generic\_dimensionSpecifier* por uma *type\_argument\_list* com o mesmo número de vírgulas e a palavra-chave `object` que cada *type\_argument*.
- Avalie as *type\_name* resultantes, ignorando todas as restrições de parâmetro de tipo.
- O *unbound\_type\_name* é resolvido para o tipo genérico não associado associado ao tipo construído resultante ([tipos vinculados e desvinculados](#)).

O resultado da *typeof\_expression* é o `System.Type` objeto para o tipo genérico não associado resultante.

A terceira forma de *typeof\_expression* consiste em uma `typeof` palavra-chave seguida por uma `void` palavra-chave entre parênteses. O resultado de uma expressão desse formulário é o `System.Type` objeto que representa a ausência de um tipo. O objeto de tipo retornado por `typeof(void)` é diferente do objeto de tipo retornado para qualquer tipo. Esse objeto de tipo especial é útil em bibliotecas de classes que permitem a reflexão em métodos no idioma, em que esses métodos desejam ter uma maneira de representar o tipo de retorno de qualquer método, incluindo métodos void, com uma instância do `System.Type`.

O `typeof` operador pode ser usado em um parâmetro de tipo. O resultado é o `System.Type` objeto para o tipo de tempo de execução que foi associado ao parâmetro de tipo. O `typeof` operador também pode ser usado em um tipo construído ou um tipo genérico não associado ([tipos vinculados e desvinculados](#)). O `System.Type` objeto para um tipo genérico não associado não é o mesmo que o `System.Type` objeto do tipo de instância. O tipo de instância é sempre um tipo construído fechado em tempo de execução para que seu `System.Type` objeto dependa dos argumentos de tipo de tempo de execução em uso, enquanto o tipo genérico não associado não tem argumentos de tipo.

O exemplo

C#

```

using System;

class X<T>
{
    public static void PrintTypes() {
        Type[] t = {
            typeof(int),
            typeof(System.Int32),
            typeof(string),
            typeof(double[]),
            typeof(void),
            typeof(T),
            typeof(X<T>),
            typeof(X<X<T>>),
            typeof(X<>)
        };
        for (int i = 0; i < t.Length; i++) {
            Console.WriteLine(t[i]);
        }
    }
}

class Test
{
    static void Main() {
        X<int>.PrintTypes();
    }
}

```

produz a seguinte saída:

Console

```

System.Int32
System.Int32
System.String
System.Double[]
System.Void
System.Int32
X`1[System.Int32]
X`1[X`1[System.Int32]]
X`1[T]

```

Observe que `int` e `System.Int32` são do mesmo tipo.

Observe também que o resultado de não `typeof(X<>)` depende do argumento de tipo, mas o resultado de `typeof(X<T>)`.

## Os operadores verificados e não verificados

Os `checked` `unchecked` operadores e são usados para controlar o *contexto de verificação de estouro* para operações e conversões aritméticas de tipo integral.

```
antlr

checked_expression
: 'checked' '(' expression ')'
;

unchecked_expression
: 'unchecked' '(' expression ')'
;
```

O `checked` operador avalia a expressão contida em um contexto selecionado e o `unchecked` operador avalia a expressão contida em um contexto desmarcado. Um *checked\_expression* ou *unchecked\_expression* corresponde exatamente a um *parenthesized\_expression* ([expressões entre parênteses](#)), exceto que a expressão contida é avaliada no contexto de verificação de estouro especificado.

O contexto de verificação de estouro também pode ser controlado por meio das `checked` `unchecked` instruções e ([as instruções marcadas e não verificadas](#)).

As operações a seguir são afetadas pelo contexto de verificação de estouro estabelecido pelos `checked` `unchecked` operadores e instruções:

- Os operadores predefinidos `++` e `--` unários ([incremento de sufixo e decremento](#)) e operadores de [incremento e diminuição de prefixo](#) quando o operando é de um tipo integral.
- O `-` operador unário predefinido ([operador menos unário](#)), quando o operando é de um tipo integral.
- Os operadores predefinidos `+`, `-`, `*` e `/` binários ([operadores aritméticos](#)), quando ambos os operandos são de tipos integrais.
- Conversões numéricas explícitas ([conversões numéricas explícitas](#)) de um tipo integral para outro tipo integral ou de `float` ou `double` para um tipo integral.

Quando uma das operações acima produz um resultado muito grande para representar no tipo de destino, o contexto no qual a operação é executada controla o comportamento resultante:

- Em um `checked` contexto, se a operação for uma expressão constante ([expressões constantes](#)), ocorrerá um erro em tempo de compilação. Caso contrário, quando a operação for executada em tempo de execução, um `System.OverflowException` será gerado.

- Em um `unchecked` contexto, o resultado é truncado descartando quaisquer bits de ordem superior que não caibam no tipo de destino.

Para expressões não constantes (expressões que são avaliadas em tempo de execução) que não são delimitadas `checked` por `unchecked` operadores ou instruções `or`, o contexto de verificação de estouro padrão é `unchecked` a menos que fatores externos (como comutadores de compilador e configuração de ambiente de execução) chamem para `checked` avaliação.

Para expressões constantes (expressões que podem ser totalmente avaliadas em tempo de compilação), o contexto de verificação de estouro padrão é sempre `checked`. A menos que uma expressão constante seja colocada explicitamente em um `unchecked` contexto, os estouros que ocorrerem durante a avaliação do tempo de compilação da expressão sempre causarão erros de tempo de compilação.

O corpo de uma função anônima não é afetado por `checked` ou `unchecked` contextos em que a função anônima ocorre.

No exemplo

C#

```
class Test
{
    static readonly int x = 1000000;
    static readonly int y = 1000000;

    static int F() {
        return checked(x * y);      // Throws OverflowException
    }

    static int G() {
        return unchecked(x * y);   // Returns -727379968
    }

    static int H() {
        return x * y;              // Depends on default
    }
}
```

Não são relatados erros de tempo de compilação, pois nenhuma das expressões pode ser avaliada em tempo de compilação. Em tempo de execução, o `F` método gera um `System.OverflowException` e o `G` método retorna -727379968 (os bits inferiores de 32 do resultado fora do intervalo). O comportamento do `H` método depende do contexto de verificação de estouro padrão para a compilação, mas é o mesmo `F` ou o mesmo que `G`.

No exemplo

```
C#  
  
class Test  
{  
    const int x = 1000000;  
    const int y = 1000000;  
  
    static int F() {  
        return checked(x * y);      // Compile error, overflow  
    }  
  
    static int G() {  
        return unchecked(x * y);    // Returns -727379968  
    }  
  
    static int H() {  
        return x * y;              // Compile error, overflow  
    }  
}
```

os estouros que ocorrem ao avaliar as expressões constantes no `F` e `H` causam erros de tempo de compilação a serem relatados porque as expressões são avaliadas em um `checked` contexto. Um estouro também ocorre ao avaliar a expressão constante no `G`, mas como a avaliação ocorre em um `unchecked` contexto, o estouro não é relatado.

Os `checked` `unchecked` operadores e afetam apenas o contexto de verificação de estouro para as operações que estão contextualmente contidas nos `(` tokens `""` e `""` `)`. Os operadores não têm nenhum efeito nos membros da função que são invocados como resultado da avaliação da expressão contida. No exemplo

```
C#  
  
class Test  
{  
    static int Multiply(int x, int y) {  
        return x * y;  
    }  
  
    static int F() {  
        return checked(Multiply(1000000, 1000000));  
    }  
}
```

o uso de `checked` no `F` não afeta a avaliação de `x * y` no `Multiply`, portanto `x * y` é avaliado no contexto de verificação de estouro padrão.

O `unchecked` operador é conveniente ao escrever constantes dos tipos integrais assinados em notação hexadecimal. Por exemplo:

C#

```
class Test
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);
    public const int HighBit = unchecked((int)0x80000000);
}
```

Ambas as constantes hexadecimais acima são do tipo `uint`. Como as constantes estão fora do `int` intervalo, sem o `unchecked` operador, as conversões para `int` produzir erros em tempo de compilação.

Os `checked` `unchecked` operadores e instruções permitem que os programadores controlem determinados aspectos de alguns cálculos numéricos. No entanto, o comportamento de alguns operadores numéricos depende dos tipos de dados dos operandos. Por exemplo, multiplicar dois decimais sempre resulta em uma exceção no estouro, mesmo dentro de um `unchecked` constructo explicitamente. Da mesma forma, multiplicar dois floats nunca resulta em uma exceção no estouro, mesmo em um `checked` constructo explicitamente. Além disso, outros operadores nunca são afetados pelo modo de verificação, sejam eles padrão ou explícitos.

## Expressões de valor padrão

Uma expressão de valor padrão é usada para obter o valor padrão ([valores padrão](#)) de um tipo. Normalmente, uma expressão de valor padrão é usada para parâmetros de tipo, pois ela poderá não ser conhecida se o parâmetro de tipo for um tipo de valor ou de referência. (Nenhuma conversão existe do `null` literal para um parâmetro de tipo, a menos que o parâmetro de tipo seja conhecido como um tipo de referência.)

antlr

```
default_value_expression
: 'default' '(' type ')'
;
```

Se o *tipo* em um *default\_value\_expression* for avaliado em tempo de execução para um tipo de referência, o resultado será `null` convertido nesse tipo. Se o *tipo* em um *default\_value\_expression* for avaliado em tempo de execução para um tipo de valor, o resultado será o valor padrão do *value\_type*([construtores padrão](#)).

Uma *default\_value\_expression* é uma expressão constante ([expressões constantes](#)) se o tipo é um tipo de referência ou um parâmetro de tipo que é conhecido como um tipo de referência ([restrições de parâmetro de tipo](#)). Além disso, um *default\_value\_expression* é uma expressão constante se o tipo for um dos tipos de valor a seguir:,,, `sbyte` `byte` `short` `ushort` `int` `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` , `bool` ou qualquer tipo de enumeração.

## Expressões `nameof`

Um *nameof\_expression* é usado para obter o nome de uma entidade de programa como uma cadeia de caracteres constante.

```
antlr

nameof_expression
: 'nameof' '(' named_entity ')'
;

named_entity
: simple_name
| named_entity_target '.' identifier type_argument_list?
;

named_entity_target
: 'this'
| 'base'
| named_entity
| predefined_type
| qualified_alias_member
;
```

Em termos de gramática, o operando *named\_entity* sempre é uma expressão. Como `nameof` não é uma palavra-chave reservada, uma expressão `nameof` é sempre sintaticamente ambígua com uma invocação do nome simples `nameof`. Por motivos de compatibilidade, se uma pesquisa de nome ([nomes simples](#)) do nome `nameof` for bem sucedido, a expressão será tratada como um *invocation\_expression*, independentemente de a invocação ser legal ou não. Caso contrário, é um *nameof\_expression*.

O significado da *named\_entity* de uma *nameof\_expression* é o significado dela como uma expressão; ou seja, como um *Simple\_name*, um *base\_access* ou um *member\_access*. No entanto, onde a pesquisa descrita em [nomes simples](#) e [acesso de membro](#) resulta em um erro porque um membro de instância foi encontrado em um contexto estático, um *nameof\_expression* não produz esse erro.

É um erro de tempo de compilação para um *named\_entity* designar um grupo de métodos para ter um *type\_argument\_list*. É um erro de tempo de compilação para um *named\_entity\_target* ter o tipo `dynamic`.

Uma *nameof\_expression* é uma expressão constante do tipo `string` e não tem nenhum efeito no tempo de execução. Especificamente, sua *named\_entity* não é avaliada e é ignorada para fins de análise de atribuição definitiva ([regras gerais para expressões simples](#)). Seu valor é o último identificador da *named\_entity* antes da *type\_argument\_list* final opcional, transformada da seguinte maneira:

- O prefixo " @ ", se usado, será removido.
- Cada *unicode\_escape\_sequence* é transformada em seu caractere Unicode correspondente.
- Todas as *formatting\_characters* são removidas.

Essas são as mesmas transformações aplicadas em [identificadores](#) ao testar a igualdade entre identificadores.

TODO: exemplos

## Expressões de método anônimo

Uma *anonymous\_method\_expression* é uma das duas maneiras de definir uma função anônima. Eles são descritos mais detalhadamente em [expressões de função anônimas](#).

## Operadores unários

Os `? operadores`, `,,,,`, `+ - ! ~ ++ --`, `CAST` e `await` são chamados de operadores unários.

```
antlr

unary_expression
: primary_expression
| null_conditional_expression
| '+' unary_expression
| '-' unary_expression
| '!' unary_expression
| '~' unary_expression
| pre_increment_expression
| pre_decrement_expression
| cast_expression
| await_expression
| unary_expression_unsafe
;
```

Se o operando de um *unary\_expression* tiver o tipo de tempo de compilação `dynamic`, ele será vinculado dinamicamente ([associação dinâmica](#)). Nesse caso, o tipo de tempo de compilação do *unary\_expression* é `dynamic`, e a resolução descrita abaixo ocorrerá em tempo de execução usando o tipo de tempo de execução do operando.

## Operador condicional nulo

O operador NULL-Conditional aplica uma lista de operações ao operando somente se esse operando não for nulo. Caso contrário, o resultado da aplicação do operador é `null`.

```
antlr

null_conditional_expression
: primary_expression null_conditional_operations
;

null_conditional_operations
: null_conditional_operations? '?' '.' identifier type_argument_list?
| null_conditional_operations? '?' '[' argument_list ']'
| null_conditional_operations '.' identifier type_argument_list?
| null_conditional_operations '[' argument_list ']'
| null_conditional_operations '(' argument_list? ')'
;
```

A lista de operações pode incluir acesso a membros e operações de acesso de elemento (que podem ser nulas-condicionais), bem como invocação.

Por exemplo, a expressão `a.b?[0]? .c()` é uma *null\_conditional\_expression* com um *primary\_expression* `a.b` e *null\_conditional\_operations* `?[0]` (acesso de elemento condicional nulo), `? .c` (acesso de membro condicional nulo) e `()` (invocação).

Para um *null\_conditional\_expression* `E` com um *primary\_expression* `P`, vamos `E0` ser a expressão obtida com a remoção textual da entrelinha de `?` cada uma das *null\_conditional\_operations* delas `E`. Conceitualmente, `E0` é a expressão que será avaliada se nenhuma das verificações nulas representadas por `?` s, encontrar um `null`.

Além disso, vamos `E1` ser a expressão obtida por meio da remoção textual da entrelinha `?` apenas da primeira das *null\_conditional\_operations* em `E`. Isso pode levar a uma *expressão primária* (se houver apenas uma `?`) ou a outra *null\_conditional\_expression*.

Por exemplo, se `E` for a expressão `a.b?[0]? .c()`, `E0` será a expressão `a.b[0].c()` e `E1` será a expressão `a.b[0]? .c()`.

Se  $E_0$  for classificado como Nothing,  $E$  será classificado como Nothing. Caso contrário,  $E$  será classificado como um valor.

$E_0$  e  $E_1$  são usados para determinar o significado de  $E$ :

- Se  $E$  ocorrer como um *statement\_expression* o significado de  $E$  é o mesmo que a instrução

C#

```
if ((object)P != null) E1;
```

Exceto que  $P$  é avaliada apenas uma vez.

- Caso contrário, se  $E_0$  for classificado como nada ocorrerá um erro de tempo de compilação.
- Caso contrário, permita que  $T_0$  seja o tipo de  $E_0$ .
  - Se  $T_0$  for um parâmetro de tipo que não é conhecido como um tipo de referência ou um tipo de valor não anulável, ocorrerá um erro em tempo de compilação.
  - Se  $T_0$  for um tipo de valor não anulável, o tipo de  $E$  é  $T_0?$  e o significado de  $E$  é o mesmo que

C#

```
((object)P == null) ? (T0?)null : E1
```

Exceto que  $P$  é avaliado apenas uma vez.

- Caso contrário, o tipo de  $E$  é  $T_0$ , e o significado de  $E$  é o mesmo que

C#

```
((object)P == null) ? null : E1
```

Exceto que  $P$  é avaliado apenas uma vez.

Se o  $E_1$  próprio for um *null\_conditional\_expression*, essas regras serão aplicadas novamente, aninhando os testes para `null` até que não haja mais `?`, e a expressão tenha sido reduzida até a expressão primária  $E_0$ .

Por exemplo, se a expressão `a.b?[0]?.c()` ocorrer como uma expressão de instrução, como na instrução:

```
C#
```

```
a.b?[0]?.c();
```

seu significado é equivalente a:

```
C#
```

```
if (a.b != null) a.b[0]?.c();
```

que novamente é equivalente a:

```
C#
```

```
if (a.b != null) if (a.b[0] != null) a.b[0].c();
```

Exceto que `a.b` e `a.b[0]` são avaliados apenas uma vez.

Se ocorrer em um contexto em que seu valor é usado, como em:

```
C#
```

```
var x = a.b?[0]?.c();
```

e supondo que o tipo da invocação final não seja um tipo de valor não anulável, seu significado é equivalente a:

```
C#
```

```
var x = (a.b == null) ? null : (a.b[0] == null) ? null : a.b[0].c();
```

Exceto que `a.b` e `a.b[0]` são avaliados apenas uma vez.

## Expressões condicionais nulas como inicializadores de projeção

Uma expressão condicional nula só é permitida como uma *member\_declarator* em uma *anonymous\_object\_creation\_expression* (expressões de [criação de objeto anônimo](#)) se terminar com um acesso de membro (opcionalmente nulo). Na gramática, esse requisito pode ser expresso como:

```
antlr
```

```
null_conditional_member_access
    : primary_expression null_conditional_operations? '?' '.' identifier
      type_argument_list?
    | primary_expression null_conditional_operations '.' identifier
      type_argument_list?
    ;
```

Este é um caso especial da gramática para *null\_conditional\_expression* acima. A produção para *member\_declarator* em [expressões de criação de objeto anônimo](#) inclui apenas *null\_conditional\_member\_access*.

## Expressões condicionais nulas como expressões de instrução

Uma expressão condicional nula só é permitida como uma *statement\_expression* ([instruções de expressão](#)) se terminar com uma invocação. Na gramática, esse requisito pode ser expresso como:

```
antlr
```

```
null_conditional_invocation_expression
    : primary_expression null_conditional_operations '(' argument_list? ')'
    ;
```

Este é um caso especial da gramática para *null\_conditional\_expression* acima. A produção para *statement\_expression* em [instruções de expressão](#), em seguida, inclui apenas *null\_conditional\_invocation\_expression*.

## Operador de adição de unário

Para uma operação do formulário `+x`, a resolução de sobrecarga de operador unário ([resolução de sobrecarga de operador unário](#)) é aplicada para selecionar uma implementação de operador específica. O operando é convertido para o tipo de parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador. Os operadores de adição de unários predefinidos são:

```
C#
```

```
int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
```

```
double operator +(double x);
decimal operator +(decimal x);
```

Para cada um desses operadores, o resultado é simplesmente o valor do operando.

## Operador de subtração de unário

Para uma operação do formulário `-x`, a resolução de sobrecarga de operador unário ([resolução de sobrecarga de operador unário](#)) é aplicada para selecionar uma implementação de operador específica. O operando é convertido para o tipo de parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador. Os operadores de negação predefinidos são:

- Negação de inteiro:

C#

```
int operator -(int x);
long operator -(long x);
```

O resultado é calculado com a subtração `x` de zero. Se o valor de `x` for o menor valor representável do tipo de operando ( $-2^{31}$  para `int` ou  $-2^{63}$  para `long`), a negação matemática de `x` não será representável dentro do tipo de operando. Se isso ocorrer em um `checked` contexto, um `System.OverflowException` será lançado; se ocorrer dentro de um `unchecked` contexto, o resultado será o valor do operando e o estouro não será relatado.

Se o operando do operador de negação for do tipo `uint`, ele será convertido em tipo `long` e o tipo do resultado será `long`. Uma exceção é a regra que permite que o `int` valor  $-2^{31}$  seja gravado como um literal inteiro decimal ([literais inteiros](#)).

Se o operando do operador de negação for do tipo `ulong`, ocorrerá um erro em tempo de compilação. Uma exceção é a regra que permite que o `long` valor  $-9.223.372.036.854.775.808$  ( $-2^{63}$ ) seja gravado como um literal inteiro decimal ([literais inteiros](#)).

- Negação de ponto flutuante:

C#

```
float operator -(float x);
double operator -(double x);
```

O resultado é o valor de `x` com seu sinal invertido. Se `x` for NaN, o resultado também será Nan.

- Negação decimal:

```
C#
```

```
decimal operator -(decimal x);
```

O resultado é calculado com a subtração `x` de zero. A negação decimal é equivalente a usar o operador de subtração unário do tipo `System.Decimal`.

## Operador de negação lógica

Para uma operação do formulário `!x`, a resolução de sobrecarga de operador unário ([resolução de sobrecarga de operador unário](#)) é aplicada para selecionar uma implementação de operador específica. O operando é convertido para o tipo de parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador. Existe apenas um operador de negação lógica predefinido:

```
C#
```

```
bool operator !(bool x);
```

Esse operador computa a negação lógica do operando: se o operando for `true`, o resultado será `false`. Se o operando for `false`, o resultado será `true`.

## Operador de complemento bit a bit

Para uma operação do formulário `~x`, a resolução de sobrecarga de operador unário ([resolução de sobrecarga de operador unário](#)) é aplicada para selecionar uma implementação de operador específica. O operando é convertido para o tipo de parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador. Os operadores predefinidos de Complementos de bits são:

```
C#
```

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

Para cada um desses operadores, o resultado da operação é o complemento de bits `x`.

Todos os tipos `E` de enumeração fornecem implicitamente o seguinte operador de complemento bit a bit:

C#

```
E operator ~(E x);
```

O resultado da avaliação `~x`, em que `x` é uma expressão de um tipo de enumeração `E` com um tipo subjacente `U`, é exatamente o mesmo que `(E)(~(U)x)` a avaliação, exceto que a conversão em `E` é sempre executada como se fosse em um `unchecked` contexto ([os operadores marcados e desmarcados](#)).

## Operadores de incremento e decremento pré-fixados

antlr

```
pre_increment_expression
  : '++' unary_expression
  ;

pre_decrement_expression
  : '--' unary_expression
  ;
```

O operando de uma operação de incremento ou diminuição de prefixo deve ser uma expressão classificada como uma variável, um acesso de propriedade ou um acesso de indexador. O resultado da operação é um valor do mesmo tipo que o operando.

Se o operando de uma operação de incremento ou diminuição de prefixo for um acesso de propriedade ou indexador, a propriedade ou o indexador deverá ter um `get` e um `set` acessador. Se esse não for o caso, ocorrerá um erro de tempo de associação.

Resolução de sobrecarga de operador unário ([resolução de sobrecarga de operador unário](#)) é aplicada para selecionar uma implementação de operador específica. `++` Os operadores e predefinidos `--` existem para os seguintes tipos: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` e qualquer tipo de enumeração. Os operadores predefinidos `++` retornam o valor produzido pela adição de 1 ao operando, e os operadores predefinidos `--` retornam o valor produzido pela subtração de 1 do operando. Em um `checked` contexto, se o resultado dessa adição ou

subtração estiver fora do intervalo do tipo de resultado e o tipo de resultado for um tipo integral ou tipo de enumeração, um `System.OverflowException` será gerado.

O processamento em tempo de execução de um incremento de prefixo ou uma operação de diminuição do formulário `++x` ou `--x` consiste nas seguintes etapas:

- Se `x` é classificado como uma variável:
  - `x` é avaliado para produzir a variável.
  - O operador selecionado é invocado com o valor de `x` como seu argumento.
  - O valor retornado pelo operador é armazenado no local fornecido pela avaliação de `x`.
  - O valor retornado pelo operador torna-se o resultado da operação.
- Se `x` é classificado como um acesso de propriedade ou indexador:
  - A expressão de instância (se `x` não for `static`) e a lista de argumentos (se `x` for um acesso de indexador) associada a `x` são avaliadas e os resultados são usados nas `get` invocações subsequentes e `set` acessadores.
  - O `get` acessador do `x` é invocado.
  - O operador selecionado é invocado com o valor retornado pelo `get` acessador como seu argumento.
  - O `set` acessador de `x` é invocado com o valor retornado pelo operador como seu `value` argumento.
  - O valor retornado pelo operador torna-se o resultado da operação.

Os `++` `--` operadores e também dão suporte à notação de sufixo ([operadores de aumento de sufixo e diminuição](#)). Normalmente, o resultado de `x++` ou `x--` é o valor de `x` antes da operação, enquanto o resultado de `++x` ou `--x` é o valor de `x` após a operação. Em ambos os casos, `x` ele tem o mesmo valor após a operação.

Uma `operator++` `operator--` implementação ou pode ser chamada usando o sufixo ou a notação de prefixo. Não é possível ter implementações de operador separadas para as duas notações.

## Expressões cast

Uma `cast_expression` é usada para converter explicitamente uma expressão em um determinado tipo.

antlr

```
cast_expression
  : '(' type ')' unary_expression
  ;
```

Uma *cast\_expression* do formulário  $(T)E$ , em que  $T$  é um *tipo* e  $E$  é um *unary\_expression*, executa uma conversão explícita ([conversões explícitas](#)) do valor de  $E$  para tipo  $T$ . Se não existir nenhuma conversão explícita de  $E$  para  $T$ , ocorrerá um erro de tempo de vinculação. Caso contrário, o resultado será o valor produzido pela conversão explícita. O resultado é sempre classificado como um valor, mesmo se  $E$  o denota uma variável.

A gramática de um *cast\_expression* leva a determinadas ambiguidades sintáticas. Por exemplo, a expressão  $(x)-y$  pode ser interpretada como uma *cast\_expression* (uma conversão de  $-y$  para tipo  $x$ ) ou como uma *additive\_expression* combinada com uma *parenthesized\_expression* (que computa o valor  $x - y$ ).

Para resolver as ambiguidades *cast\_expression*, a seguinte regra existe: uma sequência de um ou mais *tokens* (espaço em branco) entre parênteses é considerada o início de uma *cast\_expression* somente se pelo menos uma das seguintes opções for verdadeira:

- A sequência de tokens é a gramática correta para um *tipo*, mas não para uma *expressão*.
- A sequência de tokens é a gramática correta para um *tipo*, e o token imediatamente após os parênteses de fechamento é o token "`~`", o token "`!`", o token "`(`", um *identificador* ([sequências de escape de caractere Unicode](#)), um *literal* ([literais](#)) ou qualquer *palavra-chave* ([palavras-chave](#)), exceto `as` e `is`.

O termo "correção gramatical" acima significa apenas que a sequência de tokens deve estar de acordo com a produção grammatical específica. Ele não considera especificamente o significado real de quaisquer identificadores constituintes. Por exemplo, se  $x$  e  $y$  são identificadores,  $x.y$  é a gramática correta para um tipo, mesmo que,  $x.y$  na verdade, não denotará um tipo.

A partir da regra de desambiguidade, ela segue, se  $x$  e  $y$  são identificadores,,  $(x)y$   $(x)(y)$ , e  $(x)(-y)$  são *cast\_expression*s, mas  $(x)-y$  não é, mesmo que  $x$  o identifique um tipo. No entanto, se  $x$  for uma palavra-chave que identifica um tipo predefinido (como `int`), todas as quatro formas serão *cast\_expression*s (porque essa palavra-chave não poderia ser uma expressão por si só).

## Expressões Await

O operador `Await` é usado para suspender a avaliação da função `Async` delimitadora até que a operação assíncrona representada pelo operando tenha sido concluída.

```
antlr
```

```
await_expression
: 'await' unary_expression
;
```

Um *await\_expression* só é permitido no corpo de uma função Async ([funções assíncronas](#)). Na função assíncrona delimitadora mais próxima, um *await\_expression* pode não ocorrer nesses locais:

- Dentro de uma função anônima aninhada (não assíncrona)
- Dentro do bloco de um *lock\_statement*
- Em um contexto sem segurança

Observe que um *await\_expression* não pode ocorrer na maioria dos lugares em um *query\_expression*, pois eles são sintaticamente transformados para usar expressões lambda não assíncronas.

Dentro de uma função Async, `await` não pode ser usado como um identificador. Portanto, não há nenhuma ambiguidade sintática entre Await-Expressions e várias expressões que envolvem identificadores. Fora das funções assíncronas, `await` atua como um identificador normal.

O operando de um *await\_expression* é chamado de **\*tarefa \_**. Ele representa uma operação assíncrona que pode ou não ser concluída no momento em que o *\_await\_expression* \* é avaliado. A finalidade do operador Await é suspender a execução da função Async delimitadora até que a tarefa esperada seja concluída e, em seguida, obter seu resultado.

## Expressões awaitable

A tarefa de uma expressão Await deve ser **awaitable**. Uma expressão `t` é awaitable se uma das seguintes isenções:

- `t` é do tipo de tempo de compilação `dynamic`
- `t` tem uma instância ou método de extensão acessível chamado sem `GetAwaiter` parâmetros e nenhum parâmetro de tipo, e um tipo de retorno `A` para o qual todos os seguintes itens contêm:
  - `A` implementa a interface `System.Runtime.CompilerServices.INotifyCompletion` (daqui em diante, conhecida como `INotifyCompletion` para fins de brevidade)
  - `A` tem uma propriedade de instância legível e acessível `IsCompleted` do tipo `bool`

- A tem um método de instância acessível sem `GetResult` parâmetros e nenhum parâmetro de tipo

A finalidade do `GetAwaiter` método é obter um `*awaiter_` para a tarefa. O tipo A é chamado `_aguardador tipo*` para a expressão Await.

A finalidade da `IsCompleted` propriedade é determinar se a tarefa já foi concluída. Nesse caso, não há necessidade de suspender a avaliação.

A finalidade do `INotifyCompletion.OnCompleted` método é inscrever uma "continuação" para a tarefa; ou seja, um delegado (do tipo `System.Action`) que será invocado assim que a tarefa for concluída.

A finalidade do `GetResult` método é obter o resultado da tarefa quando ela for concluída. Esse resultado pode ser uma conclusão bem-sucedida, possivelmente com um valor de resultado, ou pode ser uma exceção que é gerada pelo `GetResult` método.

## Classificação de expressões Await

A expressão `await t` é classificada da mesma maneira que a expressão `(t).GetAwaiter().GetResult()`. Portanto, se o tipo de retorno de `GetResult` for `void`, o `await_expression` será classificado como `Nothing`. Se ele tiver um tipo de retorno não `void T`, o `await_expression` será classificado como um valor do tipo `T`.

## Avaliação de tempo de execução de expressões Await

Em tempo de execução, a expressão `await t` é avaliada da seguinte maneira:

- Um esperador `a` é obtido avaliando a expressão `(t).GetAwaiter()`.
- R `bool b` é obtido avaliando a expressão `(a).IsCompleted`.
- Se `b` for `false`, a avaliação dependerá se `a` implementará a interface `System.Runtime.CompilerServices.ICriticalNotifyCompletion` (daqui em diante, conhecida como `ICriticalNotifyCompletion` para fins de brevidade). Essa verificação é feita no momento da Associação; ou seja, em tempo `a` de execução, se tiver o tipo de tempo de compilação `dynamic` e, em caso contrário, o tempo de compilação. Permitir `r` denotar o delegado de continuação ([funções assíncronas](#)):
  - Se não `a` implementar `ICriticalNotifyCompletion`, a expressão `(a as (INotifyCompletion)).OnCompleted(r)` será avaliada.
  - Se `a` implementa `ICriticalNotifyCompletion`, a expressão `(a as (ICriticalNotifyCompletion)).UnsafeOnCompleted(r)` é avaliada.

- A avaliação é suspensa e o controle é retornado para o chamador atual da função Async.
- Imediatamente após (se `b` foi `true`), ou após a invocação posterior do delegado de continuação (se `b` foi `false`), a expressão `(a).GetResult()` é avaliada. Se ele retornar um valor, esse valor será o resultado da *await\_expression*. Caso contrário, o resultado será Nothing.

A implementação de um esperador dos métodos de interface

`INotifyCompletion.OnCompleted` e `ICriticalNotifyCompletion.UnsafeOnCompleted` deve fazer com que o delegado `r` seja invocado no máximo uma vez. Caso contrário, o comportamento da função assíncrona delimitadora será indefinido.

## Operadores aritméticos

Os `*` operadores,, `/` `%` , `+` e `-` são chamados de operadores aritméticos.

```
antlr

multiplicative_expression
: unary_expression
| multiplicative_expression '*' unary_expression
| multiplicative_expression '/' unary_expression
| multiplicative_expression '%' unary_expression
;

additive_expression
: multiplicative_expression
| additive_expression '+' multiplicative_expression
| additive_expression '-' multiplicative_expression
;
```

Se um operando de um operador aritmético tiver o tipo de tempo de compilação `dynamic`, a expressão será vinculada dinamicamente ([associação dinâmica](#)). Nesse caso, o tipo de tempo de compilação da expressão é `dynamic`, e a resolução descrita abaixo ocorrerá em tempo de execução usando o tipo de tempo de execução desses operandos que têm o tipo de tempo de compilação `dynamic`.

## Operador de multiplicação

Para uma operação do formulário `x * y`, a resolução de sobrecarga de operador binário (resolução de sobrecarga de [operador binário](#)) é aplicada para selecionar uma implementação de operador específica. Os operandos são convertidos nos tipos de

parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador.

Os operadores de multiplicação predefinidos estão listados abaixo. Todos os operadores computam o produto do  $x$  e do  $y$ .

- Multiplicação de inteiro:

C#

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
```

Em um `checked` contexto, se o produto estiver fora do intervalo do tipo de resultado, um `System.OverflowException` será lançado. Em um `unchecked` contexto, os estouros não são relatados e quaisquer bits de ordem superior significativos fora do intervalo do tipo de resultado são descartados.

- Multiplicação de ponto flutuante:

C#

```
float operator *(float x, float y);
double operator *(double x, double y);
```

O produto é calculado de acordo com as regras de aritmética de IEEE 754. A tabela a seguir lista os resultados de todas as combinações possíveis de valores finitos diferentes de zero, zeros, infinitos e NaNs. Na tabela,  $x$  e  $y$  são valores positivos finitos.  $z$  é o resultado de  $x * y$ . Se o resultado é grande demais para o tipo de destino,  $z$  é infinito. Se o resultado é pequeno demais para o tipo de destino,  $z$  é zero.

	+y	-y	+0	-0	+inf	-inf	NaN
+x	+z	-Z	+0	-0	+inf	-inf	NaN
-X	-Z	+z	-0	+0	-inf	+inf	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+inf	+inf	-inf	NaN	NaN	+inf	-inf	NaN

-inf	-inf	+inf	NaN	NaN	-inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Multiplicação decimal:

C#

```
decimal operator *(decimal x, decimal y);
```

Se o valor resultante for muito grande para representar no `decimal` formato, um `System.OverflowException` será lançado. Se o valor do resultado for muito pequeno para representar no `decimal` formato, o resultado será zero. A escala do resultado, antes de qualquer arredondamento, é a soma das escalas dos dois operandos.

A multiplicação decimal é equivalente a usar o operador de multiplicação do tipo `System.Decimal`.

## Operador de divisão

Para uma operação do formulário `x / y`, a resolução de sobrecarga de operador binário (resolução de sobrecarga de [operador binário](#)) é aplicada para selecionar uma implementação de operador específica. Os operandos são convertidos nos tipos de parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador.

Os operadores de divisão predefinidos estão listados abaixo. Todos os operadores computam o quociente de `x` e `y`.

- Divisão de inteiro:

C#

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

Se o valor do operando direito for zero, um `System.DivideByZeroException` será lançado.

A divisão arredonda o resultado em direção a zero. Portanto, o valor absoluto do resultado é o maior inteiro possível que é menor ou igual ao valor absoluto do quociente dos dois operandos. O resultado é zero ou positivo quando os dois operandos têm o mesmo sinal e zero ou negativos quando os dois operandos têm sinais opostos.

Se o operando esquerdo for o menor valor representável `int` ou `long` o operando direito for `-1`, ocorrerá um estouro. Em um `checked` contexto, isso faz com que uma `System.ArithemeticException` (ou uma subclasse dele) seja gerada. Em um `unchecked` contexto, ele é definido pela implementação para se uma `System.ArithemeticException` (ou uma subclasse) for gerada ou se o estouro não for relatado com o valor resultante sendo o do operando esquerdo.

- Divisão de ponto flutuante:

C#

```
float operator /(float x, float y);
double operator /(double x, double y);
```

O quociente é calculado de acordo com as regras de aritmética de IEEE 754. A tabela a seguir lista os resultados de todas as combinações possíveis de valores finitos diferentes de zero, zeros, infinitos e NaNs. Na tabela, `x` e `y` são valores positivos finitos. `z` é o resultado de `x / y`. Se o resultado é grande demais para o tipo de destino, `z` é infinito. Se o resultado é pequeno demais para o tipo de destino, `z` é zero.

	+y	-y	+0	-0	+inf	-inf	NaN
+x	+z	-z	+inf	-inf	+0	-0	NaN
-X	-Z	+z	-inf	+inf	-0	+0	NaN
+0	+0	-0	Nan	Nan	+0	-0	NaN
-0	-0	+0	Nan	Nan	-0	+0	NaN
+inf	+inf	-inf	+inf	-inf	Nan	Nan	NaN
-inf	-inf	+inf	-inf	+inf	Nan	Nan	NaN
NaN	NaN						

- Divisão decimal:

C#

```
decimal operator /(decimal x, decimal y);
```

Se o valor do operando direito for zero, um `System.DivideByZeroException` será lançado. Se o valor resultante for muito grande para representar no `decimal` formato, um `System.OverflowException` será lançado. Se o valor do resultado for muito pequeno para representar no `decimal` formato, o resultado será zero. A escala do resultado é a menor escala que preservará um resultado igual ao valor decimal representável mais próximo para o resultado matemático verdadeiro.

A divisão decimal é equivalente a usar o operador de divisão do tipo `System.Decimal`.

## Operador de resto

Para uma operação do formulário `x % y`, a resolução de sobrecarga de operador binário (resolução de sobrecarga de [operador binário](#)) é aplicada para selecionar uma implementação de operador específica. Os operandos são convertidos nos tipos de parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador.

Os operadores restantes predefinidos estão listados abaixo. Todos os operadores calculam o restante da divisão entre `x` e `y`.

- Restante do inteiro:

C#

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

O resultado de `x % y` é o valor produzido por `x - (x / y) * y`. Se `y` for zero, um `System.DivideByZeroException` será lançado.

Se o operando esquerdo for o menor `int long` valor, e o operando direito for `-1`, um `System.OverflowException` será lançado. Em nenhum caso, o `x % y` gera uma exceção em que `x / y` não geraria uma exceção.

- Restante do ponto flutuante:

C#

```
float operator %(float x, float y);
double operator %(double x, double y);
```

A tabela a seguir lista os resultados de todas as combinações possíveis de valores finitos diferentes de zero, zeros, infinitos e NaNs. Na tabela,  $x$  e  $y$  são valores positivos finitos.  $z$  é o resultado de  $x \% y$  e é calculado como  $x - n * y$ , em que  $n$  é o maior inteiro possível que é menor ou igual a  $x / y$ . Esse método de computação do restante é análogo ao usado para operandos inteiros, mas difere da definição do IEEE 754 (em que  $n$  é o número inteiro mais próximo de  $x / y$ ).

+y	-y	+0	-0	+inf	-inf	NaN	
+x	+z	+z	NaN	NaN	x	x	NaN
-X	-Z	-Z	NaN	NaN	-X	-X	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
-0	-0	-0	NaN	NaN	-0	-0	NaN
+inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Restante decimal:

C#

```
decimal operator %(decimal x, decimal y);
```

Se o valor do operando direito for zero, um `System.DivideByZeroException` será lançado. A escala do resultado, antes de qualquer arredondamento, é maior das escalas dos dois operandos, e o sinal do resultado, se for diferente de zero, é igual ao de  $x$ .

O restante decimal é equivalente a usar o operador restante do tipo `System.Decimal`.

## Operador de adição

Para uma operação do formulário `x + y`, a resolução de sobrecarga de operador binário (resolução de sobrecarga de [operador binário](#)) é aplicada para selecionar uma implementação de operador específica. Os operandos são convertidos nos tipos de parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador.

Os operadores de adição predefinidos são listados abaixo. Para tipos numéricos e de enumeração, os operadores de adição predefinidos computam a soma dos dois operandos. Quando um ou ambos os operandos são do tipo cadeia de caracteres, os operadores de adição predefinidos concatenam a representação de cadeia de caracteres dos operandos.

- Adição de inteiro:

C#

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

Em um `checked` contexto, se a soma estiver fora do intervalo do tipo de resultado, um `System.OverflowException` será lançado. Em um `unchecked` contexto, os estouros não são relatados e quaisquer bits de ordem superior significativos fora do intervalo do tipo de resultado são descartados.

- Adição de ponto flutuante:

C#

```
float operator +(float x, float y);
double operator +(double x, double y);
```

A soma é calculada de acordo com as regras de aritmética de IEEE 754. A tabela a seguir lista os resultados de todas as combinações possíveis de valores finitos diferentes de zero, zeros, infinitos e NaNs. Na tabela, `x` e `y` são valores finitos diferentes de zero e `z` é o resultado de `x + y`. Se `x` e `y` tiverem a mesma magnitude, mas sinais opostos, `z` será zero positivo. Se `x + y` for muito grande para representar no tipo de destino, `z` é um infinito com o mesmo sinal de `x + y`.

a	+0	-0	+inf	-inf	NaN
x	z	x	x	+inf	-inf
					NaN

+0	a	+0	+0	+inf	-inf	NaN
-0	a	+0	-0	+inf	-inf	NaN
+inf	+inf	+inf	+inf	+inf	NaN	NaN
-inf	-inf	-inf	-inf	NaN	-inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Adição de decimal:

C#

```
decimal operator +(decimal x, decimal y);
```

Se o valor resultante for muito grande para representar no `decimal` formato, um `System.OverflowException` será lançado. A escala do resultado, antes de qualquer arredondamento, é maior das escalas dos dois operandos.

A adição de decimal é equivalente a usar o operador de adição do tipo `System.Decimal`.

- Adição de enumeração. Cada tipo de enumeração fornece implicitamente os seguintes operadores predefinidos, em que `E` é o tipo de enumeração e `U` é o tipo subjacente de `E`:

C#

```
E operator +(E x, U y);
E operator +(U x, E y);
```

Em tempo de execução, esses operadores são avaliados exatamente como `(E)((U)x + (U)y)`.

- Concatenação de cadeia de caracteres:

C#

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

Essas sobrecargas do operador binário `+` executam a concatenação de cadeia de caracteres. Se um operando da concatenação de cadeia de caracteres for `null`, uma cadeia de caracteres vazia será substituída. Caso contrário, qualquer argumento que não seja de cadeia de caracteres é convertido em sua representação de cadeia de caracteres invocando o `ToString` método virtual herdado do tipo `object`. Se `ToString` retornar `null`, uma cadeia de caracteres vazia será substituída.

```
C#  
  
using System;  
  
class Test  
{  
    static void Main() {  
        string s = null;  
        Console.WriteLine("s = >" + s + "<");           // displays s = ><  
        int i = 1;  
        Console.WriteLine("i = " + i);                     // displays i = 1  
        float f = 1.2300E+15F;  
        Console.WriteLine("f = " + f);                     // displays f =  
        1.23E+15  
        decimal d = 2.900m;  
        Console.WriteLine("d = " + d);                     // displays d =  
        2.900  
    }  
}
```

O resultado do operador de concatenação de cadeia de caracteres é uma cadeia de caracteres que consiste nos caracteres do operando esquerdo seguidos dos caracteres do operando à direita. O operador de concatenação de cadeia de caracteres nunca retorna um `null` valor. Um `System.OutOfMemoryException` pode ser gerado se não houver memória suficiente disponível para alocar a cadeia de caracteres resultante.

- Combinação de delegado. Cada tipo delegado implicitamente fornece o seguinte operador predefinido, em que `D` é o tipo delegado:

```
C#  
  
D operator +(D x, D y);
```

O `+` operador binário executa a combinação de delegado quando ambos os operandos são de algum tipo delegado `D`. (Se os operandos tiverem tipos delegados diferentes, ocorrerá um erro de tempo de associação.) Se o primeiro

operando for `null`, o resultado da operação será o valor do segundo operando (mesmo que também esteja `null`). Caso contrário, se o segundo operando for `null`, o resultado da operação será o valor do primeiro operando. Caso contrário, o resultado da operação é uma nova instância de delegado que, quando invocada, invoca o primeiro operando e, em seguida, invoca o segundo operando. Para obter exemplos de combinação de delegado, consulte [operador de subtração](#) e [invocação de delegado](#). Como `System.Delegate` não é um tipo delegado, `operator +` não está definido para ele.

## Operador de subtração

Para uma operação do formulário `x - y`, a resolução de sobrecarga de operador binário (resolução de sobrecarga de[operador binário](#)) é aplicada para selecionar uma implementação de operador específica. Os operandos são convertidos nos tipos de parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador.

Os operadores de subtração predefinidos estão listados abaixo. Todos os operadores são subtraídos `y - x`.

- Subtração de inteiro:

```
C#  
  
int operator -(int x, int y);  
uint operator -(uint x, uint y);  
long operator -(long x, long y);  
ulong operator -(ulong x, ulong y);
```

Em um `checked` contexto, se a diferença estiver fora do intervalo do tipo de resultado, um `System.OverflowException` será lançado. Em um `unchecked` contexto, os estouros não são relatados e quaisquer bits de ordem superior significativos fora do intervalo do tipo de resultado são descartados.

- Subtração de ponto flutuante:

```
C#  
  
float operator -(float x, float y);  
double operator -(double x, double y);
```

A diferença é calculada de acordo com as regras de aritmética de IEEE 754. A tabela a seguir lista os resultados de todas as combinações possíveis de valores

finitos diferentes de zero, zeros, infinitos e NaNs. Na tabela, `x` e `y` são valores finitos diferentes de zero e `z` é o resultado de `x - y`. Se `x` e `y` forem iguais, `z` será zero positivo. Se `x - y` for muito grande para representar no tipo de destino, `z` é um infinito com o mesmo sinal de `x - y`.

a	+0	-0	+inf	-inf	Nan	
x	z	x	x	-inf	+inf	Nan
+0	-y	+0	+0	-inf	+inf	Nan
-0	-y	-0	+0	-inf	+inf	Nan
+inf	+inf	+inf	+inf	Nan	+inf	Nan
-inf	-inf	-inf	-inf	-inf	Nan	Nan
Nan	Nan	Nan	Nan	Nan	Nan	Nan

- Subtração decimal:

```
C#
decimal operator -(decimal x, decimal y);
```

Se o valor resultante for muito grande para representar no `decimal` formato, um `System.OverflowException` será lançado. A escala do resultado, antes de qualquer arredondamento, é maior das escalas dos dois operandos.

A subtração decimal é equivalente a usar o operador de subtração do tipo `System.Decimal`.

- Subtração de enumeração. Cada tipo de enumeração fornece implicitamente o seguinte operador predefinido, em que `E` é o tipo de enumeração e `U` é o tipo subjacente de `E`:

```
C#
U operator -(E x, E y);
```

Esse operador é avaliado exatamente como `(U)((U)x - (U)y)`. Em outras palavras, o operador computa a diferença entre os valores ordinais de `x` e `y` o tipo do resultado é o tipo subjacente da enumeração.

C#

```
E operator -(E x, U y);
```

Esse operador é avaliado exatamente como  $(E)((U)x - y)$ . Em outras palavras, o operador subtrai um valor do tipo subjacente da enumeração, produzindo um valor da enumeração.

- Remoção de delegado. Cada tipo delegado implicitamente fornece o seguinte operador predefinido, em que D é o tipo delegado:

C#

```
D operator -(D x, D y);
```

O - operador binário executa a remoção de delegado quando ambos os operandos são de algum tipo delegado D. Se os operandos tiverem tipos delegados diferentes, ocorrerá um erro de tempo de associação. Se o primeiro operando for null, o resultado da operação será null. Caso contrário, se o segundo operando for null, o resultado da operação será o valor do primeiro operando. Caso contrário, os dois operandos representam listas de invocação ([declarações de delegado](#)) que têm uma ou mais entradas e o resultado é uma nova lista de invocação que consiste na lista do primeiro operando com as entradas do segundo operando removidas, desde que a lista do segundo operando seja uma sublista contígua adequada do primeiro. (Para determinar a igualdade da sublista, as entradas correspondentes são comparadas com o operador de igualdade de delegado ([operadores de igualdade de delegado](#)).) Caso contrário, o resultado será o valor do operando esquerdo. Nenhuma das listas de operandos é alterada no processo. Se a lista do segundo operando corresponder a várias sublistas de entradas contíguas na lista do primeiro operando, a sublista correspondente à direita de entradas contíguas será removida. Se a remoção resultar em uma lista vazia, o resultado será null. Por exemplo:

C#

```
delegate void D(int x);

class C
{
    public static void M1(int i) { /* ... */ }
    public static void M2(int i) { /* ... */ }
}

class Test
```

```

{
    static void Main() {
        D cd1 = new D(C.M1);
        D cd2 = new D(C.M2);
        D cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
        cd3 -= cd1; // => M1 + M2 + M2

        cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd2; // => M2 + M1

        cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd2; // => M1 + M1

        cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd1; // => M1 + M2

        cd3 = cd1 + cd2 + cd2 + cd1; // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd1; // => M1 + M2 + M2 + M1
    }
}

```

## Operadores shift

Os `<<` e `>>` operadores são usados para executar operações de mudança de bits.

antlr

```

shift_expression
: additive_expression
| shift_expression '<<' additive_expression
| shift_expression right_shift additive_expression
;

```

Se um operando de um *shift\_expression* tiver o tipo de tempo de compilação `dynamic`, a expressão será vinculada dinamicamente ([associação dinâmica](#)). Nesse caso, o tipo de tempo de compilação da expressão é `dynamic`, e a resolução descrita abaixo ocorrerá em tempo de execução usando o tipo de tempo de execução desses operandos que têm o tipo de tempo de compilação `dynamic`.

Para uma operação da forma `x << count` ou `x >> count`, a resolução de sobrecarga de operador binário ([resolução de sobrecarga de operador binário](#)) é aplicada para selecionar uma implementação de operador específica. Os operandos são convertidos nos tipos de parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador.

Ao declarar um operador de deslocamento sobreescarregado, o tipo do primeiro operando sempre deve ser a classe ou struct que contém a declaração do operador e o tipo do segundo operando sempre deve ser `int`.

Os operadores de alternância predefinidos estão listados abaixo.

- Deslocar para a esquerda:

C#

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);
```

O `<<` operador muda `x` para a esquerda por um número de bits calculados, conforme descrito abaixo.

Os bits de ordem superior fora do intervalo do tipo de resultado de `x` são descartados, os bits restantes são deslocados para a esquerda e as posições de bit em branco de ordem inferior são definidas como zero.

- Deslocar para a direita:

C#

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

O `>>` operador muda `x` para a direita por um número de bits calculados, conforme descrito abaixo.

Quando `x` é do tipo `int` ou `long`, os bits de ordem inferior de `x` são descartados, os bits restantes são deslocados para a direita e as posições de bits vazias de ordem superior são definidas como zero se `x` não forem negativas e definidas como um se `x` forem negativas.

Quando `x` é do tipo `uint` ou `ulong`, os bits de ordem inferior de `x` são descartados, os bits restantes são deslocados para a direita e as posições de bits vazias de ordem superior são definidas como zero.

Para os operadores predefinidos, o número de bits a serem deslocados é calculado da seguinte maneira:

- Quando o tipo de `x` é `int` ou `uint`, a contagem de deslocamento é fornecida pelos cinco bits de ordem inferior de `count`. Em outras palavras, a contagem de deslocamento é calculada a partir de `count & 0x1F`.
- Quando o tipo de `x` é `long` ou `ulong`, a contagem de deslocamento é fornecida pelos seis bits de ordem inferior de `count`. Em outras palavras, a contagem de deslocamento é calculada a partir de `count & 0x3F`.

Se a contagem de deslocamento resultante for zero, os operadores de alternância simplesmente retornarão o valor de `x`.

As operações de Shift nunca causam estouros e produzem os mesmos resultados `checked` e `unchecked` contextos.

Quando o operando esquerdo do `>>` operador for de um tipo integral assinado, o operador executará um deslocamento aritmético à direita no qual o valor do bit mais significativo (o bit de sinal) do operando será propagado para as posições de bit vazio de ordem superior. Quando o operando esquerdo do `>>` operador é de um tipo integral não assinado, o operador executa um direito de deslocamento lógico onde as posições de bits vazias de ordem superior são sempre definidas como zero. Para executar a operação oposta da inferida do tipo de operando, as conversões explícitas podem ser usadas. Por exemplo, se `x` for uma variável do tipo `int`, a operação `unchecked((int)((uint)x >> y))` executará um deslocamento lógico à direita de `x`.

## Operadores de teste de tipo e relacional

Os `==` operadores, `!=`, `<`, `>`, `<=`, `>=`, `is` e `as` são chamados de operadores relacionais e de teste de tipo.

```
antlr

relational_expression
: shift_expression
| relational_expression '<' shift_expression
| relational_expression '>' shift_expression
| relational_expression '<=' shift_expression
| relational_expression '>=' shift_expression
| relational_expression 'is' type
| relational_expression 'as' type
;

equality_expression
: relational_expression
| equality_expression '==' relational_expression
| equality_expression '!=' relational_expression
;
```

O `is` operador é descrito no [operador is](#) e o `as` operador é descrito no [operador as](#).

Os `==`, `!=`, `<`, `>`, `<=` e `>=` são **operadores de comparação**.

Se um operando de um operador de comparação tiver o tipo de tempo de compilação `dynamic`, a expressão será vinculada dinamicamente ([associação dinâmica](#)). Nesse caso, o tipo de tempo de compilação da expressão é `dynamic`, e a resolução descrita abaixo ocorrerá em tempo de execução usando o tipo de tempo de execução desses operandos que têm o tipo de tempo de compilação `dynamic`.

Para uma operação do formulário `x op y`, em que `op` é um operador de comparação, a resolução de sobrecarga ([resolução de sobrecarga de operador binário](#)) é aplicada para selecionar uma implementação de operador específica. Os operandos são convertidos nos tipos de parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador.

Os operadores de comparação predefinidos são descritos nas seções a seguir. Todos os operadores de comparação predefinidos retornam um resultado do tipo `bool`, conforme descrito na tabela a seguir.

Operação	Resultado
<code>x == y</code>	<code>true</code> Se <code>x</code> for igual a <code>y</code> , <code>false</code> caso contrário,
<code>x != y</code>	<code>true</code> Se <code>x</code> não for igual a <code>y</code> , <code>false</code> caso contrário,
<code>x &lt; y</code>	<code>true</code> se <code>x</code> for menor que <code>y</code> , caso contrário, <code>false</code>
<code>x &gt; y</code>	<code>true</code> se <code>x</code> for maior que <code>y</code> , caso contrário, <code>false</code>
<code>x &lt;= y</code>	<code>true</code> se <code>x</code> for menor ou igual a <code>y</code> , caso contrário, <code>false</code>
<code>x &gt;= y</code>	<code>true</code> se <code>x</code> for maior ou igual a <code>y</code> , caso contrário, <code>false</code>

## Operadores de comparação de inteiros

Os operadores de comparação de inteiros predefinidos são:

C#

```
bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);
```

```
bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);
```

Cada um desses operadores compara os valores numéricos dos dois operandos inteiros e retorna um `bool` valor que indica se a relação específica é `true` ou `false`.

## Operadores de comparação de ponto flutuante

Os operadores de comparação de ponto flutuante predefinidos são:

C#

```
bool operator ==(float x, float y);
bool operator ==(double x, double y);

bool operator !=(float x, float y);
bool operator !=(double x, double y);

bool operator <(float x, float y);
bool operator <(double x, double y);

bool operator >(float x, float y);
bool operator >(double x, double y);

bool operator <=(float x, float y);
bool operator <=(double x, double y);

bool operator >=(float x, float y);
bool operator >=(double x, double y);
```

Os operadores comparam os operandos de acordo com as regras do padrão IEEE 754:

- Se qualquer operando for NaN, o resultado será `false` para todos os operadores, exceto `!=` para os quais o resultado é `true`. Para quaisquer dois operandos, `x != y` sempre produz o mesmo resultado que `!(x == y)`. No entanto, quando um ou ambos os operandos são NaN, os `<` operadores,, `>` `<=` e `>=` não produzem os mesmos resultados que a negação lógica do operador oposto. Por exemplo, se um dos `x` e `y` for NaN, então `x < y` é `false`, mas `!(x >= y)` é `true`.
- Quando nenhum operando é NaN, os operadores comparam os valores dos dois operandos de ponto flutuante em relação à ordenação

C#

```
-inf < -max < ... < -min < -0.0 == +0.0 < +min < ... < +max < +inf
```

onde `min` e `max` são os valores finitos menores e maiores positivos que podem ser representados no formato de ponto flutuante fornecido. Os efeitos notáveis dessa ordenação são:

- Zeros negativos e positivos são considerados iguais.
- Um infinito negativo é considerado menor que todos os outros valores, mas é igual a outro infinito negativo.
- Um infinito positivo é considerado maior que todos os outros valores, mas é igual a outro infinito positivo.

## Operadores de comparação decimal

Os operadores de comparação decimal predefinidos são:

C#

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

Cada um desses operadores compara os valores numéricos dos dois operando decimais e retorna um `bool` valor que indica se a relação específica é `true` ou `false`. Cada comparação decimal é equivalente a usar o operador relacional ou de igualdade correspondente do tipo `System.Decimal`.

## Operadores de igualdade booleanos

Os operadores de igualdade booleanos predefinidos são:

C#

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

O resultado de `==` é `true` se `x` e `y` são `true` ou se ambos forem `x` e `y` `false`. Caso contrário, o resultado será `false`.

O resultado de `!=` é `false` se `x` e `y` são `true` ou se ambos forem `x` e `y` `false`. Caso contrário, o resultado será `true`. Quando os operandos forem do tipo `bool`, o `!=` operador produzirá o mesmo resultado que o `^` operador.

## Operadores de comparação de enumeração

Cada tipo de enumeração fornece implicitamente os seguintes operadores de comparação predefinidos:

C#

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

O resultado da avaliação `x op y`, onde `x` e `y` são expressões de um tipo de enumeração `E` com um tipo subjacente `U`, e `op` é um dos operadores de comparação, é exatamente o mesmo que avaliar `((U)x) op ((U)y)`. Em outras palavras, os operadores de comparação de tipo de enumeração apenas comparam os valores inteiros subjacentes dos dois operandos.

## Operadores de igualdade de tipo de referência

Os operadores de igualdade do tipo de referência predefinido são:

C#

```
bool operator ==(object x, object y);
bool operator !=(object x, object y);
```

Os operadores retornam o resultado da comparação das duas referências para igualdade ou não igualdade.

Como os operadores de igualdade do tipo de referência predefinido aceitam operandos do tipo `object`, eles se aplicam a todos os tipos que não declaram `operator ==` ou `operator !=`. Os membros e aplicáveis. Por outro lado, qualquer operador de igualdade definido pelo usuário aplicável efetivamente oculta os operadores de igualdade do tipo de referência predefinido.

Os operadores de igualdade do tipo de referência predefinidos exigem um dos seguintes:

- Ambos os operandos são um valor de um tipo conhecido como um *reference\_type* ou o literal `null`. Além disso, uma conversão de referência explícita ([conversões de referência explícitas](#)) existe no tipo de um dos operandos para o tipo do outro operando.
- Um operando é um valor do tipo `T`, em que `T` é um *type\_parameter* e o outro operando é o literal `null`. Além disso, `T` não tem a restrição de tipo de valor.

A menos que uma dessas condições seja verdadeira, ocorre um erro de tempo de associação. As principais implicações dessas regras são:

- É um erro de tempo de ligação para usar os operadores de igualdade do tipo de referência predefinido para comparar duas referências que são conhecidas para serem diferentes no tempo de associação. Por exemplo, se os tipos de tempo de Associação dos operandos forem dois tipos de classe `A` e `B`, e se nem forem `A` `B` derivados do outro, seria impossível para os dois operandos referenciarem o mesmo objeto. Assim, a operação é considerada um erro de tempo de associação.
- Os operadores de igualdade do tipo de referência predefinido não permitem que os operandos de tipo de valor sejam comparados. Portanto, a menos que um tipo de struct declare seus próprios operadores de igualdade, não é possível comparar valores desse tipo struct.
- Os operadores de igualdade do tipo de referência predefinido nunca causam a ocorrência de operações Boxing para seus operandos. Seria sem sentido executar essas operações boxing, já que as referências às instâncias emolduradas recentemente alocadas seriam necessariamente diferentes de todas as outras referências.

- Se um operando de um tipo de parâmetro de tipo `T` for comparado a `null`, e o tipo de tempo de execução de `T` for um tipo de valor, o resultado da comparação será `false`.

O exemplo a seguir verifica se um argumento de um tipo de parâmetro de tipo irrestrito é `null`.

C#

```
class C<T>
{
    void F(T x) {
        if (x == null) throw new ArgumentNullException();
        ...
    }
}
```

A `x == null` construção é permitida mesmo que `T` possa representar um tipo de valor, e o resultado é simplesmente definido como `false` quando `T` é um tipo de valor.

Para uma operação do formulário `x == y` ou `x != y`, se aplicável `operator ==` ou `operator !=` existir, as regras de resolução de sobrecarga de operador ([resolução de sobrecarga de operador binário](#)) selecionarão esse operador em vez do operador de igualdade do tipo de referência predefinido. No entanto, sempre é possível selecionar o operador de igualdade do tipo de referência predefinido, convertendo explicitamente um ou ambos os operandos para o tipo `object`. O exemplo

C#

```
using System;

class Test
{
    static void Main() {
        string s = "Test";
        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine((object)s == t);
        Console.WriteLine(s == (object)t);
        Console.WriteLine((object)s == (object)t);
    }
}
```

produz a saída

Console

```
True  
False  
False  
False
```

As `s` e `t` variáveis se referem a duas instâncias distintas `string` contendo os mesmos caracteres. A primeira comparação gera `True` porque o operador de igualdade de cadeia de caracteres predefinido ([operadores de igualdade de cadeia de caracteres](#)) é selecionado quando ambos os operandos são do tipo `string`. As comparações restantes são todas saídas `False` porque o tipo de referência predefinido operador de igualdade é selecionado quando um ou ambos os operandos são do tipo `object`.

Observe que a técnica acima não é significativa para tipos de valor. O exemplo

C#

```
class Test  
{  
    static void Main()  
    {  
        int i = 123;  
        int j = 123;  
        System.Console.WriteLine((object)i == (object)j);  
    }  
}
```

saídas `False` porque as conversões criam referências a duas instâncias separadas de valores de caixa `int`.

## Operadores de igualdade de cadeia de caracteres

Os operadores de igualdade de cadeia de caracteres predefinidos são:

C#

```
bool operator ==(string x, string y);  
bool operator !=(string x, string y);
```

Dois `string` valores são considerados iguais quando uma das seguintes opções é verdadeira:

- Ambos os valores são `null`.
- Ambos os valores são referências não nulas a instâncias de cadeia de caracteres que têm comprimentos idênticos e caracteres idênticos em cada posição de caractere.

Os operadores de igualdade de cadeia de caracteres comparam valores de cadeia de caracteres em vez de referências. Quando duas instâncias de cadeias de caracteres separadas contêm exatamente a mesma sequência de caracteres, os valores das cadeias são iguais, mas as referências são diferentes. Conforme descrito em [operadores de igualdade de tipo de referência](#), os operadores de igualdade de tipo de referência podem ser usados para comparar referências de cadeia de caracteres em vez de valores de cadeia de caracteres.

## Delegar operadores de igualdade

Cada tipo de delegado fornece implicitamente os seguintes operadores de comparação predefinidos:

C#

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

Duas instâncias de delegado são consideradas iguais da seguinte maneira:

- Se uma das instâncias de delegado for `null`, elas serão iguais se e somente se ambas estiverem `null`.
- Se os delegados tiverem um tipo de tempo de execução diferente, nunca serão iguais.
- Se ambas as instâncias de delegado tiverem uma lista de invocação ([declarações de delegado](#)), essas instâncias serão iguais se e somente se suas listas de invocação tiverem o mesmo comprimento, e cada entrada na lista de invocação de uma for igual (conforme definido abaixo) à entrada correspondente, na ordem, na lista de invocação do outro.

As regras a seguir regem a igualdade de entradas da lista de invocação:

- Se duas entradas da lista de invocação fizerem referência ao mesmo método estático, as entradas serão iguais.
- Se duas entradas da lista de invocação fizerem referência ao mesmo método não-estático no mesmo objeto de destino (conforme definido pelos operadores de igualdade de referência), as entradas serão iguais.
- As entradas da lista de invocação produzidas da avaliação de *anonymous\_method\_expressions* ou *lambda\_expressions* semanticamente idênticas com o mesmo conjunto (possivelmente vazio) de instâncias de variáveis externas capturadas são permitidas (mas não obrigatórias) para serem iguais.

# Operadores de igualdade e nulo

Os `==` `!=` operadores e permitem que um operando seja um valor de um tipo anulável e o outro para ser o `null` literal, mesmo que nenhum operador predefinido ou definido pelo usuário (em forma não elevada ou levantada) exista para a operação.

Para uma operação de um dos formulários

C#

```
x == null  
null == x  
x != null  
null != x
```

em que `x` é uma expressão de um tipo anulável, se a resolução de sobrecarga de operador ([resolução de sobrecarga de operador binário](#)) falhar ao localizar um operador aplicável, o resultado será computado a partir da `HasValue` propriedade de `x`. Especificamente, os dois primeiros formulários são convertidos em `!x.HasValue` e os últimos dois formulários são traduzidos para o `x.HasValue`.

## Operador is

O `is` operador é usado para verificar dinamicamente se o tipo de tempo de execução de um objeto é compatível com um determinado tipo. O resultado da operação `E is T`, em que `E` é uma expressão e `T` é um tipo, é um valor booleano que indica se é `E` possível converter com êxito `T` o tipo por uma conversão de referência, uma conversão boxing ou uma conversão unboxing. A operação é avaliada da seguinte maneira, após os argumentos de tipo terem sido substituídos por todos os parâmetros de tipo:

- Se `E` for uma função anônima, ocorrerá um erro de tempo de compilação
- Se `E` for um grupo de métodos ou o `null` literal, de se o tipo de `E` for um tipo de referência ou um tipo anulável e o valor de `E` for NULL, o resultado será false.
- Caso contrário, permita `D` representar o tipo dinâmico de da `E` seguinte maneira:
  - Se o tipo de `E` for um tipo de referência, `D` será o tipo de tempo de execução da referência de instância por `E`.
  - Se o tipo de `E` for um tipo anulável, `D` será o tipo subjacente desse tipo anulável.
  - Se o tipo de `E` for um tipo de valor não anulável, `D` será o tipo de `E`.
- O resultado da operação depende de `D` e da `T` seguinte maneira:

- Se  $T$  for um tipo de referência, o resultado será true se  $D$  e  $T$  o mesmo tipo, se  $D$  for um tipo de referência e uma conversão de referência implícita de  $D$  para  $T$  Exists, ou se  $D$  for um tipo de valor e uma conversão de Boxing de  $D$  para  $T$  Exists.
- Se  $T$  for um tipo anulável, o resultado será true se  $D$  for o tipo subjacente de  $T$ .
- Se  $T$  for um tipo de valor não anulável, o resultado será true se  $D$  e  $T$  for do mesmo tipo.
- Caso contrário, o resultado será false.

Observe que as conversões definidas pelo usuário não são consideradas pelo `is` operador.

## Operador `as`

O `as` operador é usado para converter explicitamente um valor em um determinado tipo de referência ou tipo anulável. Ao contrário de uma expressão de conversão ([expressões de conversão](#)), o `as` operador nunca gera uma exceção. Em vez disso, se a conversão indicada não for possível, o valor resultante será `null`.

Em uma operação do formulário  $E \text{ as } T$ ,  $E$  deve ser uma expressão e  $T$  deve ser um tipo de referência, um parâmetro de tipo conhecido como um tipo de referência ou um tipo anulável. Além disso, pelo menos um dos seguintes deve ser verdadeiro ou um erro de tempo de compilação ocorre:

- Uma identidade ([conversão de identidade](#)), permite valor nulo implícito ([conversões anuláveis implícitas](#)), referência implícita ([conversões de referência implícita](#)), conversão de Boxing ([conversões de Boxing](#)), nulo explícito ([conversões anuláveis explícitas](#)), referência explícita ([conversões de referência explícita](#)) ou unboxing ([conversões unboxing](#)) existe de  $E$  para  $T$ .
- O tipo de  $E$  ou  $T$  é um tipo aberto.
- $E$  é o `null` literal.

Se o tipo de tempo de compilação de  $E$  não for `dynamic`, a operação  $E \text{ as } T$  produzirá o mesmo resultado que

C#

```
E is T ? (T)(E) : (T)null
```

exceto que `E` é avaliado apenas uma vez. O compilador pode ser ideal para otimizar `E` as `T` para executar no máximo uma verificação de tipo dinâmico, em oposição às duas verificações de tipo dinâmico, IMPLÍCITAS pela expansão acima.

Se o tipo de tempo de compilação de `E` for `dynamic`, ao contrário do operador `cast`, o `as` operador não será vinculado dinamicamente ([associação dinâmica](#)). Portanto, a expansão, nesse caso, é:

```
C#
```

```
E is T ? (T)(object)(E) : (T)null
```

Observe que algumas conversões, como conversões definidas pelo usuário, não são possíveis com o `as` operador e, em vez disso, devem ser executadas usando expressões de conversão.

No exemplo

```
C#
```

```
class X
{
    public string F(object o) {
        return o as string;           // OK, string is a reference type
    }

    public T G<T>(object o) where T: Attribute {
        return o as T;              // Ok, T has a class constraint
    }

    public U H<U>(object o) {
        return o as U;              // Error, U is unconstrained
    }
}
```

o parâmetro `T` de tipo de `G` é conhecido como um tipo de referência, porque ele tem a restrição de classe. O parâmetro `U` de tipo de `H` não é no entanto; portanto, o uso do `as` operador in não `H` é permitido.

## Operadores lógicos

Os `&` `^` operadores, e `|` são chamados de operadores lógicos.

antlr

```

and_expression
: equality_expression
| and_expression '&' equality_expression
;

exclusive_or_expression
: and_expression
| exclusive_or_expression '^' and_expression
;

inclusive_or_expression
: exclusive_or_expression
| inclusive_or_expression '||' exclusive_or_expression
;

```

Se um operando de um operador lógico tiver o tipo de tempo de compilação `dynamic`, a expressão será vinculada dinamicamente ([associação dinâmica](#)). Nesse caso, o tipo de tempo de compilação da expressão é `dynamic`, e a resolução descrita abaixo ocorrerá em tempo de execução usando o tipo de tempo de execução desses operandos que têm o tipo de tempo de compilação `dynamic`.

Para uma operação do formulário `x op y`, em que `op` é um dos operadores lógicos, a resolução de sobrecarga ([resolução de sobrecarga do operador binário](#)) é aplicada para selecionar uma implementação de operador específica. Os operandos são convertidos nos tipos de parâmetro do operador selecionado e o tipo de resultado é o tipo de retorno do operador.

Os operadores lógicos predefinidos são descritos nas seções a seguir.

## Operadores lógicos de inteiros

Os operadores lógicos de inteiro predefinidos são:

C#

```

int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);
uint operator ^(uint x, uint y);

```

```
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);
```

O `&` operador computa o bit lógico `AND` dos dois operandos, `|` e o operador computa a lógica de bits de bit e lógico dos dois `OR` operandos, e o `^` operador computa o exclusivo lógico bit-a `OR` de dois operandos. Não são possíveis estouros dessas operações.

## Operadores lógicos de enumeração

Cada tipo de enumeração `E` fornece implicitamente os seguintes operadores lógicos predefinidos:

C#

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```

O resultado da avaliação `x op y`, onde `x` e `y` são expressões de um tipo de enumeração `E` com um tipo subjacente `U`, e `op` é um dos operadores lógicos, é exatamente o mesmo que avaliar `(E)((U)x op (U)y)`. Em outras palavras, os operadores lógicos do tipo de enumeração simplesmente executam a operação lógica no tipo subjacente dos dois operandos.

## Operadores lógicos booleanos

Os operadores lógicos booleanos predefinidos são:

C#

```
bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);
```

O resultado de `x & y` será `true` se ambos `x` e `y` forem `true`. Caso contrário, o resultado será `false`.

O resultado de `x | y` é `true` se `x` ou `y` é `true`. Caso contrário, o resultado será `false`.

O resultado de `x ^ y` é `true` If `x true` e `y is false`, ou `x é false` e `y é true`. Caso contrário, o resultado será `false`. Quando os operandos são do tipo `bool`, o `^`

operador computa o mesmo resultado que o `!=` operador.

## Operadores lógicos booleanos anuláveis

O tipo booleano Anulável `bool?` pode representar três valores, `true`, `false` e `null`, e é conceitualmente semelhante ao tipo de três valores usado para expressões booleanas no SQL. Para garantir que os resultados produzidos pelos `&` operadores e `|` para `bool?` operandos sejam consistentes com a lógica de três valores do SQL, os seguintes operadores predefinidos são fornecidos:

C#

```
bool? operator &(bool? x, bool? y);
bool? operator |(bool? x, bool? y);
```

A tabela a seguir lista os resultados produzidos por esses operadores para todas as combinações dos valores `true`, `false` e `null`.

x	y	<code>x &amp; y</code>	<code>x   y</code>
<code>true</code>	<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>
<code>true</code>	<code>null</code>	<code>null</code>	<code>true</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>
<code>false</code>	<code>false</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>null</code>	<code>false</code>	<code>null</code>
<code>null</code>	<code>true</code>	<code>null</code>	<code>true</code>
<code>null</code>	<code>false</code>	<code>false</code>	<code>null</code>
<code>null</code>	<code>null</code>	<code>null</code>	<code>null</code>

## Operadores lógicos condicionais

Os operadores `&&` e `||` são chamados de operadores lógicos condicionais. Eles também são chamados de operadores lógicos de "curto-circuito".

antlr

```

conditional_and_expression
: inclusive_or_expression
| conditional_and_expression '&&' inclusive_or_expression
;

conditional_or_expression
: conditional_and_expression
| conditional_or_expression '||' conditional_and_expression
;

```

Os `&&` `||` operadores e são versões condicionais dos `&` `|` operadores e:

- A operação `x && y` corresponde à operação `x & y`, exceto que `y` só será avaliada se `x` não for `false`.
- A operação `x || y` corresponde à operação `x | y`, exceto que `y` só será avaliada se `x` não for `true`.

Se um operando de um operador lógico condicional tiver o tipo de tempo de compilação `dynamic`, a expressão será vinculada dinamicamente ([associação dinâmica](#)).

Nesse caso, o tipo de tempo de compilação da expressão é `dynamic`, e a resolução descrita abaixo ocorrerá em tempo de execução usando o tipo de tempo de execução desses operandos que têm o tipo de tempo de compilação `dynamic`.

Uma operação do formulário `x && y` ou `x || y` é processada pela aplicação da resolução de sobrecarga ([resolução de sobrecarga do operador binário](#)) como se a operação fosse gravada `x & y` ou `x | y`. E, em seguida,

- Se a resolução de sobrecarga não encontrar um único operador melhor, ou se a resolução de sobrecarga selecionar um dos operadores lógicos inteiros predefinidos, ocorrerá um erro de tempo de associação.
- Caso contrário, se o operador selecionado for um dos operadores lógicos boolianos predefinidos ([operadores lógicos boolianos](#)) ou operadores lógicos boolianos anuláveis ([operadores lógicos boolianos anuláveis](#)), a operação será processada conforme descrito em [operadores lógicos condicionais boolianos](#).
- Caso contrário, o operador selecionado é um operador definido pelo usuário e a operação é processada conforme descrito em [operadores lógicos condicionais definidos pelo usuário](#).

Não é possível sobrepor diretamente os operadores lógicos condicionais. No entanto, como os operadores lógicos condicionais são avaliados em termos de operadores lógicos regulares, as sobreporas dos operadores lógicos regulares são, com certas restrições, também consideradas sobreporas dos operadores lógicos

condicionais. Isso é descrito mais detalhadamente em [operadores lógicos condicionais definidos pelo usuário](#).

## Operadores lógicos condicionais booleanos

Quando os operandos de `&&` ou `||` são do tipo `bool`, ou quando os operandos são de tipos que não definem uma aplicável `operator &` ou `operator |`, mas definem conversões implícitas para `bool` o, a operação é processada da seguinte maneira:

- A operação `x && y` é avaliada como `x ? y : false`. Em outras palavras, `x` é avaliado primeiro e convertido em tipo `bool`. Em seguida, se `x` for `true`, `y` será avaliado e convertido em tipo `bool`, e isso se tornará o resultado da operação. Caso contrário, o resultado da operação será `false`.
- A operação `x || y` é avaliada como `x ? true : y`. Em outras palavras, `x` é avaliado primeiro e convertido em tipo `bool`. Em seguida, se `x` for `true`, o resultado da operação será `true`. Caso contrário, `y` é avaliado e convertido para `bool` o tipo, e isso se torna o resultado da operação.

## Operadores lógicos condicionais definidos pelo usuário

Quando os operandos de `&&` ou `||` são de tipos que declaram um definido pelo usuário aplicável `operator &` ou `operator |`, ambos devem ser verdadeiros, em que `T` é o tipo no qual o operador selecionado é declarado:

- O tipo de retorno e o tipo de cada parâmetro do operador selecionado devem ser `T`. Em outras palavras, o operador deve calcular o lógico `AND` ou lógico `OR` de dois operandos do tipo `T` e deve retornar um resultado do tipo `T`.
- `T` deve conter declarações de `operator true` e `operator false`.

Um erro de tempo de associação ocorre se um desses requisitos não for atendido. Caso contrário, `&&` a `||` operação ou é avaliada pela combinação do definido pelo usuário `operator true` ou `operator false` com o operador selecionado definido pelo usuário:

- A operação `x && y` é avaliada como `T.false(x) ? x : T.&(x, y)`, em que `T.false(x)` é uma invocação do `operator false` declarado em `T` e `T.&(x, y)` é uma invocação do selecionado `operator &`. Em outras palavras, o `x` é avaliado pela primeira vez e `operator false` é invocado no resultado para determinar se `x` é definitivamente falso. Em seguida, se `x` for definitivamente false, o resultado da operação será o valor calculado anteriormente para `x`. Caso contrário, `y` é

avaliado, e o selecionado `operator &` é invocado no valor calculado anteriormente para `x` e o valor calculado para `y` para produzir o resultado da operação.

- A operação `x || y` é avaliada como `T.true(x) ? x : T.|(x, y)`, em que `T.true(x)` é uma invocação do `operator true` declarado em `T` e `T.|(x,y)` é uma invocação do selecionado `operator |`. Em outras palavras, o `x` é avaliado pela primeira vez e `operator true` é invocado no resultado para determinar se `x` é definitivamente verdadeiro. Em seguida, se `x` for definitivamente verdadeiro, o resultado da operação será o valor calculado anteriormente para `x`. Caso contrário, `y` é avaliado, e o selecionado `operator |` é invocado no valor calculado anteriormente para `x` e o valor calculado para `y` para produzir o resultado da operação.

Em qualquer uma dessas operações, a expressão fornecida por `x` é avaliada apenas uma vez e a expressão fornecida por `y` não é avaliada ou avaliada exatamente uma vez.

Para obter um exemplo de um tipo que implementa `operator true` e `operator false`, consulte [tipo booleano do banco de dados](#).

## O operador de coalescência nula

O `??` operador é chamado de operador de União nulo.

antlr

```
null_coalescing_expression
: conditional_or_expression
| conditional_or_expression '??' null_coalescing_expression
;
```

Uma expressão de União nula do formulário `a ?? b` requer `a` que seja de um tipo anulável ou tipo de referência. Se `a` for não nulo, o resultado será `a ?? b a`; caso contrário, o resultado será `b`. A operação será avaliada `b` somente se `a` for NULL.

O operador de União nulo é associativo à direita, o que significa que as operações são agrupadas da direita para a esquerda. Por exemplo, uma expressão do formulário `a ?? b ?? c` é avaliada como `a ?? (b ?? c)`. Em termos gerais, uma expressão do formulário `E1 ?? E2 ?? ... ?? En` retorna o primeiro dos operandos que não é nulo ou nulo se todos os operandos forem nulos.

O tipo da expressão `a ?? b` depende de quais conversões implícitas estão disponíveis nos operandos. Em ordem de preferência, o tipo de `a ?? b` é `A0`, `A`, ou `B`, em que `A`

é o tipo de `a` (fornecido que `a` tem um tipo), `B` é o tipo de `b` (fornecido que `b` tem um tipo) e `A0` é o tipo subjacente de `A`. If `A` é um tipo anulável ou, `A` caso contrário.

Especificamente, `a ?? b` o é processado da seguinte maneira:

- Se `A` existir e não for um tipo anulável ou um tipo de referência, ocorrerá um erro em tempo de compilação.
- Se `b` for uma expressão dinâmica, o tipo de resultado será `dynamic`. Em tempo de execução, `a` é avaliado pela primeira vez. Se `a` não for NULL, `a` será convertido em dinâmico, e isso se tornará o resultado. Caso contrário, `b` será avaliado, e isso se tornará o resultado.
- Caso contrário, se `A` existir e for um tipo anulável e existir uma conversão implícita de `b` para `A0`, o tipo de resultado será `A0`. Em tempo de execução, `a` é avaliado pela primeira vez. Se `a` não for NULL, `a` será desencapsulado para `A0` o tipo e isso se tornará o resultado. Caso contrário, `b` é avaliado e convertido para `A0` o tipo, e isso se torna o resultado.
- Caso contrário, se `A` existir e houver uma conversão implícita de `b` para `A`, o tipo de resultado será `A`. Em tempo de execução, `a` é avaliado pela primeira vez. Se `a` não for NULL, `a` torna-se o resultado. Caso contrário, `b` é avaliado e convertido para `A` o tipo, e isso se torna o resultado.
- Caso contrário, se houver `b` um tipo `B` e uma conversão implícita de `a` para `B`, o tipo de resultado será `B`. Em tempo de execução, `a` é avaliado pela primeira vez. Se `a` não for NULL, `a` será desencapsulado para `A0` o tipo (se `A` existir e for anulável) e convertido para `B` o tipo, e isso se tornará o resultado. Caso contrário, `b` será avaliado e se tornará o resultado.
- Caso contrário, `a` e `b` são incompatíveis e ocorre um erro de tempo de compilação.

## Operador condicional

O `?:` operador é chamado de operador condicional. Ele ocorre às vezes também chamado de operador ternário.

antlr

```
conditional_expression
: null_coalescing_expression
| null_coalescing_expression '?' expression ':' expression
;
```

Uma expressão condicional do formulário `b ? x : y` primeiro avalia a condição `b`. Em seguida, se `b` for `true`, `x` será avaliado e se tornará o resultado da operação. Caso contrário, `y` será avaliado e se tornará o resultado da operação. Uma expressão condicional nunca avalia `x` e `y`.

O operador condicional é associativo à direita, o que significa que as operações são agrupadas da direita para a esquerda. Por exemplo, uma expressão do formulário `a ? b : c ? d : e` é avaliada como `a ? b : (c ? d : e)`.

O primeiro operando do `?:` operador deve ser uma expressão que possa ser convertida implicitamente `bool` ou uma expressão de um tipo que implementa `operator true`. Se nenhum desses requisitos for atendido, ocorrerá um erro em tempo de compilação.

O segundo e o terceiro operandos, `x` e `y`, do `?:` operador Control o tipo da expressão condicional.

- Se `x` tiver tipo `X` e `y` tiver tipo `Y`,
  - Se uma conversão implícita ([conversões implícitas](#)) existir de `x` para `Y`, mas não de `Y` para `X`, `Y` será o tipo da expressão condicional.
  - Se uma conversão implícita ([conversões implícitas](#)) existir de `Y` para `X`, mas não de `X` para `Y`, `X` será o tipo da expressão condicional.
  - Caso contrário, nenhum tipo de expressão pode ser determinado e ocorre um erro em tempo de compilação.
- Se apenas um dos `x` e `y` tiver um tipo, e `x` e `y`, forem conversíveis implicitamente para esse tipo, esse será o tipo da expressão condicional.
- Caso contrário, nenhum tipo de expressão pode ser determinado e ocorre um erro em tempo de compilação.

O processamento em tempo de execução de uma expressão condicional do formulário `b ? x : y` consiste nas seguintes etapas:

- Primeiro, `b` é avaliado e o `bool` valor de `b` é determinado:
  - Se uma conversão implícita do tipo de `b` a `bool` existir, essa conversão implícita será executada para produzir um `bool` valor.
  - Caso contrário, o `operator true` definido pelo tipo de `b` é invocado para produzir um `bool` valor.
- Se o `bool` valor produzido pela etapa acima for `true`, `x` será avaliado e convertido para o tipo da expressão condicional, e isso se tornará o resultado da expressão condicional.
- Caso contrário, `y` é avaliado e convertido para o tipo da expressão condicional, e isso se torna o resultado da expressão condicional.

# Expressões de função anônima

Uma *função anônima* é uma expressão que representa uma definição de método "embutida". Uma função anônima não tem um valor ou tipo próprio, mas é conversível em um delegate compatível ou tipo de árvore de expressão. A avaliação de uma conversão de função anônima depende do tipo de destino da conversão: se for um tipo delegado, a conversão será avaliada como um valor delegado que referencia o método que a função anônima define. Se for um tipo de árvore de expressão, a conversão será avaliada como uma árvore de expressão que representa a estrutura do método como uma estrutura de objeto.

Por motivos históricos, há dois tipos sintáticos de funções anônimas, ou seja, *lambda\_expression*s e *anonymous\_method\_expression*s. Para quase todas as finalidades, *lambda\_expression*s são mais concisas e expressivas do que *anonymous\_method\_expression*s, que permanecem na linguagem para compatibilidade com versões anteriores.

```
antlr
```

```
lambda_expression
: anonymous_function_signature '=>' anonymous_function_body
;

anonymous_method_expression
: 'delegate' explicit_anonymous_function_signature? block
;

anonymous_function_signature
: explicit_anonymous_function_signature
| implicit_anonymous_function_signature
;

explicit_anonymous_function_signature
: '(' explicit_anonymous_function_parameter_list? ')'
;

explicit_anonymous_function_parameter_list
: explicit_anonymous_function_parameter (',' explicit_anonymous_function_parameter)*
;

explicit_anonymous_function_parameter
: anonymous_function_parameter_modifier? type identifier
;

anonymous_function_parameter_modifier
: 'ref'
| 'out'
;
```

```

implicit_anonymous_function_signature
  : '(' implicit_anonymous_function_parameter_list? ')'
  | implicit_anonymous_function_parameter
  ;

implicit_anonymous_function_parameter_list
  : implicit_anonymous_function_parameter ( ',' implicit_anonymous_function_parameter)*
  ;

implicit_anonymous_function_parameter
  : identifier
  ;

anonymous_function_body
  : expression
  | block
  ;

```

O `=>` operador tem a mesma precedência de atribuição (`=`) e é associativo à direita.

Uma função anônima com o `async` modificador é uma função assíncrona e segue as regras descritas em [funções assíncronas](#).

Os parâmetros de uma função anônima na forma de um *lambda\_expression* podem ser digitados explícita ou implicitamente. Em uma lista de parâmetros explicitamente tipados, o tipo de cada parâmetro é explicitamente declarado. Em uma lista de parâmetros Tipados implicitamente, os tipos dos parâmetros são inferidos do contexto no qual a função anônima ocorre — especificamente, quando a função anônima é convertida em um tipo de delegado compatível ou tipo de árvore de expressão, esse tipo fornece os tipos de parâmetro ([conversões de função anônima](#)).

Em uma função anônima com um único parâmetro tipado implicitamente, os parênteses podem ser omitidos da lista de parâmetros. Em outras palavras, uma função anônima do formulário

C#

```
( param ) => expr
```

pode ser abreviado como

C#

```
param => expr
```

A lista de parâmetros de uma função anônima na forma de uma *anonymous\_method\_expression* é opcional. Se for fornecido, os parâmetros deverão ser tipados explicitamente. Caso contrário, a função anônima é conversível em um delegado com qualquer lista de parâmetros que não contenha `out` parâmetros.

Um corpo de *bloco* de uma função anônima pode ser acessado ([pontos de extremidade e acessibilidade](#)), a menos que a função anônima ocorra dentro de uma instrução inacessível.

Veja a seguir alguns exemplos de funções anônimas:

C#

```
x => x + 1                                // Implicitly typed, expression body
x => { return x + 1; }                      // Implicitly typed, statement body
(int x) => x + 1                            // Explicitly typed, expression body
(int x) => { return x + 1; }                  // Explicitly typed, statement body
(x, y) => x * y                            // Multiple parameters
() => Console.WriteLine()                   // No parameters
async (t1,t2) => await t1 + await t2        // Async
delegate (int x) { return x + 1; }             // Anonymous method expression
delegate { return 1 + 1; }                     // Parameter list omitted
```

O comportamento de *lambda\_expression*s e *anonymous\_method\_expression*s é o mesmo, exceto os seguintes pontos:

- *anonymous\_method\_expression*s permitem que a lista de parâmetros seja omitida inteiramente, produzindo convertibilidade para delegar tipos de qualquer lista de parâmetros de valor.
- *lambda\_expression*s permitem que os tipos de parâmetro sejam omitidos e inferidos, enquanto *anonymous\_method\_expression*s exigem que os tipos de parâmetros sejam declarados explicitamente.
- O corpo de uma *lambda\_expression* pode ser uma expressão ou um bloco de instrução, enquanto o corpo de uma *anonymous\_method\_expression* deve ser um bloco de instruções.
- Somente *lambda\_expression*s têm conversões para tipos de árvore de expressão compatíveis ([tipos de árvore de expressão](#)).

## Assinaturas de funções anônimas

O *anonymous\_function\_signature* opcional de uma função anônima define os nomes e, opcionalmente, os tipos dos parâmetros formais para a função anônima. O escopo dos parâmetros da função anônima é o *anonymous\_function\_body*. ([Escopos](#)) Junto com a lista de parâmetros (se fornecido), o corpo do método anônimo constitui um espaço de

declaração ([declarações](#)). Portanto, é um erro de tempo de compilação para o nome de um parâmetro da função anônima corresponder ao nome de uma variável local, constante local ou parâmetro cujo escopo inclui a *anonymous\_method\_expression* ou *lambda\_expression*.

Se uma função anônima tiver um *explicit\_anonymous\_function\_signature*, o conjunto de tipos de delegados compatíveis e tipos de árvore de expressão será restrito àqueles que têm os mesmos tipos de parâmetro e modificadores na mesma ordem. Em contraste com conversões de grupos de métodos ([conversões de grupos de métodos](#)), não há suporte para a variância de contrato de tipos de parâmetro de função anônima. Se uma função anônima não tiver um *anonymous\_function\_signature*, o conjunto de tipos de delegados compatíveis e tipos de árvore de expressão será restrito àqueles que não têm `out` parâmetros.

Observe que um *anonymous\_function\_signature* não pode incluir atributos ou uma matriz de parâmetros. No entanto, um *anonymous\_function\_signature* pode ser compatível com um tipo delegado cuja lista de parâmetros contenha uma matriz de parâmetros.

Observe também que a conversão para um tipo de árvore de expressão, mesmo se compatível, ainda pode falhar em tempo de compilação ([tipos de árvore de expressão](#)).

## Corpos de funções anônimas

O corpo (*expressão* ou *bloco*) de uma função anônima está sujeito às seguintes regras:

- Se a função anônima incluir uma assinatura, os parâmetros especificados na assinatura estarão disponíveis no corpo. Se a função anônima não tiver nenhuma assinatura, ela poderá ser convertida em um tipo de representante ou tipo de expressão com parâmetros ([conversões de função anônimas](#)), mas os parâmetros não poderão ser acessados no corpo.
- Exceto para `ref` os `out` parâmetros ou especificados na assinatura (se houver) da função anônima delimitadora mais próxima, trata-se de um erro de tempo de compilação para o corpo acessar um `ref` `out` parâmetro ou.
- Quando o tipo de `this` é um tipo de struct, é um erro de tempo de compilação para o corpo acessar `this`. Isso é verdadeiro se o acesso for explícito (como em `this.x`) ou implícito (como em `x` onde `x` é um membro de instância do struct). Essa regra simplesmente proíbe esse acesso e não afeta se a pesquisa de membros resulta em um membro da estrutura.
- O corpo tem acesso às variáveis externas ([variáveis externas](#)) da função anônima. O acesso de uma variável externa fará referência à instância da variável que está ativa

no momento em que o *lambda\_expression* ou *anonymous\_method\_expression* é avaliado ([avaliação de expressões de função anônimas](#)).

- É um erro de tempo de compilação para o corpo conter uma instrução `goto`, `break` instrução ou instrução `continue` cujo destino está fora do corpo ou dentro do corpo de uma função anônima contida.
- Uma `return` instrução no corpo retorna o controle de uma invocação da função anônima delimitadora mais próxima, não do membro da função delimitadora. Uma expressão especificada em uma `return` instrução deve ser conversível implicitamente para o tipo de retorno do tipo delegado ou de árvore de expressão para o qual o *lambda\_expression* ou *anonymous\_method\_expression* de delimitador mais próximo é convertido ([conversões de função anônimas](#)).

Ele é explicitamente não especificado se há qualquer maneira de executar o bloco de uma função anônima além da avaliação e invocação do *lambda\_expression* ou *anonymous\_method\_expression*. Em particular, o compilador pode optar por implementar uma função anônima sintetizando um ou mais métodos ou tipos nomeados. Os nomes de quaisquer elementos sintetizados devem ser de um formulário reservado para uso do compilador.

## Resolução de sobrecarga e funções anônimas

As funções anônimas em uma lista de argumentos participam da inferência de tipos e da resolução de sobrecarga. Veja a [inferência de tipos](#) e a [resolução de sobrecarga](#) para as regras exatas.

O exemplo a seguir ilustra o efeito de funções anônimas na resolução de sobrecarga.

C#

```
class ItemList<T>: List<T>
{
    public int Sum(Func<T,int> selector) {
        int sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }

    public double Sum(Func<T,double> selector) {
        double sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }
}
```

A `ItemList<T>` classe tem dois `Sum` métodos. Cada um tem um `selector` argumento, que extrai o valor para somar de um item de lista. O valor extraído pode ser um `int` ou a `double` e a soma resultante é, da mesma forma, um `int` ou um `double`.

Os `Sum` métodos poderiam, por exemplo, ser usados para computar somas de uma lista de linhas de detalhes em uma ordem.

C#

```
class Detail
{
    public int UnitCount;
    public double UnitPrice;
    ...
}

void ComputeSums() {
    ItemList<Detail> orderDetails = GetOrderDetails(...);
    int totalUnits = orderDetails.Sum(d => d.UnitCount);
    double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
    ...
}
```

Na primeira invocação de `orderDetails.Sum`, ambos os `Sum` métodos são aplicáveis porque a função anônima `d => d.UnitCount` é compatível com `Func<Detail,int>` e `Func<Detail,double>`. No entanto, a resolução de sobrecarga escolhe o primeiro `Sum` método porque a conversão para `Func<Detail,int>` é melhor do que a conversão para `Func<Detail,double>`.

Na segunda invocação de `orderDetails.Sum`, somente o segundo `Sum` método é aplicável porque a função anônima `d => d.UnitPrice * d.UnitCount` produz um valor do tipo `double`. Portanto, a resolução de sobrecarga escolhe o segundo `Sum` método para essa invocação.

## Funções anônimas e associação dinâmica

Uma função anônima não pode ser um receptor, argumento ou operando de uma operação vinculada dinamicamente.

## Variáveis externas

Qualquer variável local, parâmetro de valor ou matriz de parâmetros cujo escopo inclui o *lambda\_expression* ou *anonymous\_method\_expression* é chamado de uma **variável externa** da função anônima. Em um membro da função de instância de uma classe, o

`this` valor é considerado um parâmetro de valor e é uma variável externa de qualquer função anônima contida no membro da função.

## Variáveis externas capturadas

Quando uma variável externa é referenciada por uma função anônima, diz-se que a variável externa foi *capturada* pela função anônima. Normalmente, o tempo de vida de uma variável local é limitado à execução do bloco ou da instrução com a qual está associado ([variáveis locais](#)). No entanto, o tempo de vida de uma variável externa capturada é estendido pelo menos até que a árvore de expressão ou delegado criada na função anônima se torne elegível para a coleta de lixo.

No exemplo

C#

```
using System;

delegate int D();

class Test
{
    static D F() {
        int x = 0;
        D result = () => ++x;
        return result;
    }

    static void Main() {
        D d = F();
        Console.WriteLine(d());
        Console.WriteLine(d());
        Console.WriteLine(d());
    }
}
```

a variável local `x` é capturada pela função anônima, e o tempo de vida de `x` é estendido pelo menos até que o delegado retornado `F` se torne qualificado para coleta de lixo (o que não ocorre até o final do programa). Como cada invocação da função anônima opera na mesma instância do `x`, a saída do exemplo é:

Console

```
1
2
3
```

Quando uma variável local ou um parâmetro de valor é capturado por uma função anônima, a variável local ou o parâmetro não é mais considerado uma variável fixa ([variáveis fixas e móveis](#)), mas, em vez disso, é considerado uma variável móvel. Portanto, qualquer `unsafe` código que pega o endereço de uma variável externa capturada deve primeiro usar a `fixed` instrução para corrigir a variável.

Observe que, diferentemente de uma variável não capturada, uma variável local capturada pode ser exposta simultaneamente a vários threads de execução.

## Instanciação de variáveis locais

Uma variável local é considerada como *instanciada* quando a execução entra no escopo da variável. Por exemplo, quando o método a seguir é invocado, a variável local `x` é instanciada e inicializada três vezes — uma vez para cada iteração do loop.

```
C#  
  
static void F() {  
    for (int i = 0; i < 3; i++) {  
        int x = i * 2 + 1;  
        ...  
    }  
}
```

No entanto, mover a declaração de `x` fora do loop resulta em uma única instanciação de `x`:

```
C#  
  
static void F() {  
    int x;  
    for (int i = 0; i < 3; i++) {  
        x = i * 2 + 1;  
        ...  
    }  
}
```

Quando não é capturado, não há como observar exatamente com que frequência uma variável local é instanciada — como os tempos de vida das instâncias são não contíguos, é possível que cada instanciação simplesmente use o mesmo local de armazenamento. No entanto, quando uma função anônima captura uma variável local, os efeitos da instanciação se tornam aparentes.

O exemplo

C#

```
using System;

delegate void D();

class Test
{
    static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = () => { Console.WriteLine(x); };
        }
        return result;
    }

    static void Main() {
        foreach (D d in F()) d();
    }
}
```

produz a saída:

Console

```
1
3
5
```

No entanto, quando a declaração de `x` é movida para fora do loop:

C#

```
static D[] F() {
    D[] result = new D[3];
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        result[i] = () => { Console.WriteLine(x); };
    }
    return result;
}
```

a saída é:

Console

```
5
5
```

Se um loop for declarar uma variável de iteração, essa variável será considerada como sendo declarada fora do loop. Portanto, se o exemplo for alterado para capturar a própria variável de iteração:

C#

```
static D[] F() {
    D[] result = new D[3];
    for (int i = 0; i < 3; i++) {
        result[i] = () => { Console.WriteLine(i); };
    }
    return result;
}
```

somente uma instância da variável de iteração é capturada, o que produz a saída:

Console

```
3
3
3
```

É possível que os delegados de função anônima compartilhem algumas variáveis capturadas, embora tenham instâncias separadas de outros. Por exemplo, se `F` for alterado para

C#

```
static D[] F() {
    D[] result = new D[3];
    int x = 0;
    for (int i = 0; i < 3; i++) {
        int y = 0;
        result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
    }
    return result;
}
```

os três delegados capturam a mesma instância do `x`, mas instâncias separadas do `y`, e a saída é:

Console

```
1 1
2 1
```

Funções anônimas separadas podem capturar a mesma instância de uma variável externa. No exemplo:

```
C#  
  
using System;  
  
delegate void Setter(int value);  
  
delegate int Getter();  
  
class Test  
{  
    static void Main()  
    {  
        int x = 0;  
        Setter s = (int value) => { x = value; };  
        Getter g = () => { return x; };  
        s(5);  
        Console.WriteLine(g());  
        s(10);  
        Console.WriteLine(g());  
    }  
}
```

as duas funções anônimas capturam a mesma instância da variável local `x` e, portanto, podem "se comunicar" por meio dessa variável. A saída do exemplo é:

```
Console  
  
5  
10
```

## Avaliação de expressões de função anônimas

Uma função anônima `F` sempre deve ser convertida para um tipo `delegate D` ou um tipo de árvore de expressão `E`, seja diretamente ou por meio da execução de uma expressão de criação de delegado `new D(F)`. Essa conversão determina o resultado da função anônima, conforme descrito em [conversões de função anônimas](#).

## Expressões de consulta

As *expressões de consulta* fornecem uma sintaxe de linguagem integrada para consultas que são semelhantes a linguagens de consulta relacionais e hierárquicas, como SQL e

## XQuery.

antlr

```
query_expression
  : from_clause query_body
  ;

from_clause
  : 'from' type? identifier 'in' expression
  ;

query_body
  : query_body_clauses? select_or_group_clause query_continuation?
  ;

query_body_clauses
  : query_body_clause
  | query_body_clauses query_body_clause
  ;
query_body_clause
  : from_clause
  | let_clause
  | where_clause
  | join_clause
  | join_into_clause
  | orderby_clause
  ;
let_clause
  : 'let' identifier '=' expression
  ;
where_clause
  : 'where' boolean_expression
  ;
join_clause
  : 'join' type? identifier 'in' expression 'on' expression 'equals'
expression
  ;
join_into_clause
  : 'join' type? identifier 'in' expression 'on' expression 'equals'
expression 'into' identifier
  ;
orderby_clause
  : 'orderby' orderings
  ;
orderings
  : ordering (',' ordering)*
```

```

;

ordering
: expression ordering_direction?
;

ordering_direction
: 'ascending'
| 'descending'
;

select_or_group_clause
: select_clause
| group_clause
;

select_clause
: 'select' expression
;

group_clause
: 'group' expression 'by' expression
;

query_continuation
: 'into' identifier query_body
;

```

Uma expressão de consulta começa com uma `from` cláusula e termina com uma `select` `group` cláusula or. A `from` cláusula inicial pode ser seguida por zero ou mais `from` cláusulas,, `let` `where` `join` ou `orderby`. Cada `from` cláusula é um gerador que apresenta uma **variável \* Range \_**, que varia sobre os elementos de uma **sequência \_ \* \***. Cada `let` cláusula introduz uma variável de intervalo que representa um valor calculado por meio de variáveis de intervalo anteriores. Cada `where` cláusula é um filtro que exclui itens do resultado. Cada `join` cláusula compara as chaves especificadas da sequência de origem com chaves de outra sequência, gerando pares correspondentes. Cada `orderby` cláusula reordena os itens de acordo com os critérios especificados. A `select` cláusula or final `group` especifica a forma do resultado em termos das variáveis de intervalo. Por fim, uma `into` cláusula pode ser usada para "unir" consultas tratando os resultados de uma consulta como um gerador em uma consulta subsequente.

## Ambiguidades em expressões de consulta

As expressões de consulta contêm um número de "palavras-chave contextuais", ou seja, identificadores que têm significado especial em um determinado contexto. Especificamente, eles são,,,,,, `from` `where` `join` `on` `equals` `into` `let` `orderby` ,

`ascending`, `descending`, `select group` e `by`. Para evitar ambigüidades em expressões de consulta causadas pelo uso misto desses identificadores como palavras-chave ou nomes simples, esses identificadores são considerados palavras-chave quando ocorrem em qualquer lugar dentro de uma expressão de consulta.

Para essa finalidade, uma expressão de consulta é qualquer expressão que comece com "`from identifier`" seguido por qualquer token, exceto "`;`", "`=`" ou "`,`".

Para usar essas palavras como identificadores dentro de uma expressão de consulta, elas podem ser prefixadas com "`@`" ([identificadores](#)).

## Conversão de expressão de consulta

A linguagem C# não especifica a semântica de execução das expressões de consulta. Em vez disso, as expressões de consulta são convertidas em invocações de métodos que aderem ao *padrão de expressão de consulta* ([o padrão de expressão de consulta](#)).

Especificamente, as expressões de consulta são convertidas em invocações de métodos chamados `Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `GroupBy` e `Cast`. Esses métodos devem ter assinaturas e tipos de resultados específicos, conforme descrito no [padrão de expressão de consulta](#). Esses métodos podem ser métodos de instância do objeto que está sendo consultado ou métodos de extensão que são externos ao objeto e implementam a execução real da consulta.

A conversão de expressões de consulta para invocações de método é um mapeamento sintático que ocorre antes que qualquer associação de tipo ou resolução de sobrecarga tenha sido executada. A conversão tem a garantia de estar sintaticamente correta, mas não há garantia de que ele produza código C# semanticamente correto. Após a tradução das expressões de consulta, as invocações de método resultantes são processadas como invocações de método regular, e isso pode, por sua vez, revelar erros, por exemplo, se os métodos não existirem, se os argumentos tiverem tipos incorretos ou se os métodos forem genéricos e a inferência de tipos falhar.

Uma expressão de consulta é processada pela aplicação repetida das seguintes traduções até que nenhuma redução adicional seja possível. As traduções são listadas em ordem de aplicativo: cada seção pressupõe que as traduções nas seções anteriores foram executadas exaustivamente e, uma vez esgotadas, uma seção não será revisitada posteriormente no processamento da mesma expressão de consulta.

A atribuição de variáveis de intervalo não é permitida em expressões de consulta. No entanto, uma implementação C# é permitida nem sempre impor essa restrição, uma vez

que isso pode, às vezes, não ser possível com o esquema de tradução sintático apresentado aqui.

Determinadas traduções injetam variáveis de intervalo com identificadores transparentes denotados pelo `*`. As propriedades especiais de identificadores transparentes são discutidas mais detalhadamente em [identificadores transparentes](#).

## Cláusulas Select e GroupBy com continuações

Uma expressão de consulta com uma continuação

```
C#
```

```
from ... into x ...
```

é convertido em

```
C#
```

```
from x in ( from ... ) ...
```

As traduções nas seções a seguir pressupõem que as consultas não têm `into` continuação.

O exemplo

```
C#
```

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

é convertido em

```
C#
```

```
from g in
    from c in customers
        group c by c.Country
select new { Country = g.Key, CustCount = g.Count() }
```

a tradução final do que é

```
C#
```

```
customers.  
GroupBy(c => c.Country).  
Select(g => new { Country = g.Key, CustCount = g.Count() })
```

## Tipos de variável de intervalo explícito

Uma `from` cláusula que especifica explicitamente um tipo de variável de intervalo

C#

```
from T x in e
```

é convertido em

C#

```
from x in ( e ) . Cast < T > ( )
```

Uma `join` cláusula que especifica explicitamente um tipo de variável de intervalo

C#

```
join T x in e on k1 equals k2
```

é convertido em

C#

```
join x in ( e ) . Cast < T > ( ) on k1 equals k2
```

As traduções nas seções a seguir pressupõem que as consultas não têm nenhum tipo de variável de intervalo explícito.

O exemplo

C#

```
from Customer c in customers  
where c.City == "London"  
select c
```

é convertido em

```
C#
```

```
from c in customers.Cast<Customer>()
where c.City == "London"
select c
```

a tradução final do que é

```
C#
```

```
customers.
Cast<Customer>().
Where(c => c.City == "London")
```

Os tipos de variável de intervalo explícitos são úteis para consultar coleções que implementam a interface não genérica `IEnumerable`, mas não a `IEnumerable<T>` interface genérica. No exemplo acima, esse seria o caso se `customers` fosse do tipo `ArrayList`.

## Degerar expressões de consulta

Uma expressão de consulta do formulário

```
C#
```

```
from x in e select x
```

é convertido em

```
C#
```

```
( e ) . Select ( x => x )
```

O exemplo

```
C#
```

```
from c in customers
select c
```

é convertido em

```
C#
```

```
customers.Select(c => c)
```

Uma expressão de consulta de degeneração é aquela que seleciona de trivial os elementos da origem. Uma fase posterior da tradução remove as consultas de degeneração introduzidas por outras etapas de conversão, substituindo-as por sua fonte. No entanto, é importante garantir que o resultado de uma expressão de consulta nunca seja o próprio objeto de origem, pois isso revelaria o tipo e a identidade da origem para o cliente da consulta. Portanto, essa etapa protege as consultas de degeneração escritas diretamente no código-fonte chamando explicitamente `Select` na origem. Em seguida, é até os implementadores do `Select` e outros operadores de consulta para garantir que esses métodos nunca retornem o próprio objeto de origem.

## De, Let, Where, cláusulas Join e OrderBy

Uma expressão de consulta com uma segunda `from` cláusula seguida de uma `select` cláusula

```
C#
```

```
from x1 in e1
from x2 in e2
select v
```

é convertido em

```
C#
```

```
( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )
```

Uma expressão de consulta com uma segunda `from` cláusula seguida por algo diferente de uma `select` cláusula:

```
C#
```

```
from x1 in e1
from x2 in e2
...
```

é convertido em

```
C#
```

```
from * in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )
...
...
```

Uma expressão de consulta com uma `let` cláusula

C#

```
from x in e
let y = f
...
...
```

é convertido em

C#

```
from * in ( e ) . Select ( x => new { x , y = f } )
...
...
```

Uma expressão de consulta com uma `where` cláusula

C#

```
from x in e
where f
...
...
```

é convertido em

C#

```
from x in ( e ) . Where ( x => f )
...
...
```

Uma expressão de consulta com uma `join` cláusula sem um `into` seguido por uma `select` cláusula

C#

```
from x1 in e1
join x2 in e2 on k1 equals k2
select v
```

é convertido em

```
C#
```

```
( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )
```

Uma expressão de consulta com uma `join` cláusula sem um `into` seguido por algo diferente de uma `select` cláusula

```
C#
```

```
from x1 in e1
join x2 in e2 on k1 equals k2
...
```

é convertido em

```
C#
```

```
from * in ( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1
, x2 } )
...
```

Uma expressão de consulta com uma `join` cláusula com um `into` seguido por uma `select` cláusula

```
C#
```

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
select v
```

é convertido em

```
C#
```

```
( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )
```

Uma expressão de consulta com uma `join` cláusula com um `into` seguido por algo diferente de uma `select` cláusula

```
C#
```

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
...
```

é convertido em

```
C#
```

```
from * in ( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new {  
    x1 , g })  
...
```

Uma expressão de consulta com uma `orderby` cláusula

```
C#
```

```
from x in e  
orderby k1 , k2 , ... , kn  
...
```

é convertido em

```
C#
```

```
from x in ( e ) .  
OrderBy ( x => k1 ) .  
ThenBy ( x => k2 ) .  
...  
ThenBy ( x => kn )  
...
```

Se uma cláusula de ordenação especificar um `descending` indicador de direção, uma invocação `OrderByDescending` ou `ThenByDescending` será produzida em vez disso.

As traduções a seguir pressupõem que não há `let` `where` `join` cláusulas,, ou `orderby` , e não mais do que a `from` cláusula inicial em cada expressão de consulta.

O exemplo

```
C#
```

```
from c in customers  
from o in c.Orders  
select new { c.Name, o.OrderID, o.Total }
```

é convertido em

```
C#
```

```
customers.  
SelectMany(c => c.Orders,  
          (c,o) => new { c.Name, o.OrderID, o.Total }  
)
```

O exemplo

C#

```
from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }
```

é convertido em

C#

```
from * in customers.  
    SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }
```

a tradução final do que é

C#

```
customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(x => x.o.Total).  
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

onde `x` é um identificador gerado pelo compilador que, de outra forma, é invisível e inacessível.

O exemplo

C#

```
from o in orders  
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)  
where t >= 1000  
select new { o.OrderID, Total = t }
```

é convertido em

```
C#
```

```
from * in orders.  
    Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) })  
where t >= 1000  
select new { o.OrderID, Total = t }
```

a tradução final do que é

```
C#
```

```
orders.  
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).  
Where(x => x.t >= 1000).  
Select(x => new { x.o.OrderID, Total = x.t })
```

onde `x` é um identificador gerado pelo compilador que, de outra forma, é invisível e inacessível.

O exemplo

```
C#
```

```
from c in customers  
join o in orders on c.CustomerID equals o.CustomerID  
select new { c.Name, o.OrderDate, o.Total }
```

é convertido em

```
C#
```

```
customers.Join(orders, c => c.CustomerID, o => o.CustomerID,  
(c, o) => new { c.Name, o.OrderDate, o.Total })
```

O exemplo

```
C#
```

```
from c in customers  
join o in orders on c.CustomerID equals o.CustomerID into co  
let n = co.Count()  
where n >= 10  
select new { c.Name, OrderCount = n }
```

é convertido em

C#

```
from * in customers.
    GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
              (c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

a tradução final do que é

C#

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
          (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n })
```

onde `x` e `y` são identificadores gerados pelo compilador que, de outra forma, são invisíveis e inacessíveis.

O exemplo

C#

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

tem a tradução final

C#

```
orders.
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

## Cláusulas de seleção

Uma expressão de consulta do formulário

C#

```
from x in e select v
```

é convertido em

```
C#
```

```
( e ) . Select ( x => v )
```

Exceto quando v é o identificador x, a tradução é simplesmente

```
C#
```

```
( e )
```

Por exemplo,

```
C#
```

```
from c in customers.Where(c => c.City == "London")
select c
```

é simplesmente traduzido em

```
C#
```

```
customers.Where(c => c.City == "London")
```

## Cláusulas GroupBy

Uma expressão de consulta do formulário

```
C#
```

```
from x in e group v by k
```

é convertido em

```
C#
```

```
( e ) . GroupBy ( x => k , x => v )
```

Exceto quando v é o identificador x, a conversão é

```
C#
```

```
( e ) . GroupBy ( x => k )
```

O exemplo

C#

```
from c in customers  
group c.Name by c.Country
```

é convertido em

C#

```
customers.  
GroupBy(c => c.Country, c => c.Name)
```

## Identificadores transparentes

Determinadas traduções injetam variáveis de intervalo com **\*identificadores transparentes** \_ indicados por `_` . Os identificadores transparentes não são um recurso de idioma adequado; elas existem somente como uma etapa intermediária no processo de conversão de expressão de consulta.

Quando uma conversão de consulta injeta um identificador transparente, as etapas de tradução adicionais propagam o identificador transparente para funções anônimas e inicializadores de objeto anônimos. Nesses contextos, os identificadores transparentes têm o seguinte comportamento:

- Quando um identificador transparente ocorre como um parâmetro em uma função anônima, os membros do tipo anônimo associado são automaticamente no escopo no corpo da função anônima.
- Quando um membro com um identificador transparente está no escopo, os membros desse membro também estão no escopo.
- Quando um identificador transparente ocorre como um Declarador de membro em um inicializador de objeto anônimo, ele introduz um membro com um identificador transparente.
- Nas etapas de conversão descritas acima, os identificadores transparentes são sempre introduzidos junto com tipos anônimos, com a intenção de capturar várias variáveis de intervalo como membros de um único objeto. Uma implementação do C# tem permissão para usar um mecanismo diferente de tipos anônimos para agrupar várias variáveis de intervalo. Os exemplos de conversão a seguir

pressupõem que os tipos anônimos são usados e mostram como os identificadores transparentes podem ser traduzidos fora.

O exemplo

```
C#  
  
from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.Total }
```

é convertido em

```
C#  
  
from * in customers.  
    SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.Total }
```

que é ainda mais convertida em

```
C#  
  
customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(* => o.Total).  
Select(* => new { c.Name, o.Total })
```

que, quando os identificadores transparentes são apagados, é equivalente a

```
C#  
  
customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(x => x.o.Total).  
Select(x => new { x.c.Name, x.o.Total })
```

onde `x` é um identificador gerado pelo compilador que, de outra forma, é invisível e inacessível.

O exemplo

```
C#
```

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

é convertido em

C#

```
from * in customers.
    Join(orders, c => c.CustomerID, o => o.CustomerID,
        (c, o) => new { c, o })
    join d in details on o.OrderID equals d.OrderID
    join p in products on d.ProductID equals p.ProductID
    select new { c.Name, o.OrderDate, p.ProductName }
```

que é ainda mais reduzido para

C#

```
customers.
    Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
    Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
    Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p }).
    Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

a tradução final do que é

C#

```
customers.
    Join(orders, c => c.CustomerID, o => o.CustomerID,
        (c, o) => new { c, o }).
    Join(details, x => x.o.OrderID, d => d.OrderID,
        (x, d) => new { x, d }).
    Join(products, y => y.d.ProductID, p => p.ProductID,
        (y, p) => new { y, p }).
    Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

onde `x`, `y` e `z` são identificadores gerados pelo compilador que, de outra forma, são invisíveis e inacessíveis.

## O padrão de expressão de consulta

O *padrão de expressão de consulta* estabelece um padrão de métodos que os tipos podem implementar para dar suporte a expressões de consulta. Como as expressões de consulta são convertidas em invocações de método por meio de um mapeamento sintático, os tipos têm flexibilidade considerável em como implementam o padrão de expressão de consulta. Por exemplo, os métodos do padrão podem ser implementados como métodos de instância ou como métodos de extensão porque os dois têm a mesma sintaxe de invocação, e os métodos podem solicitar delegados ou árvores de expressão porque as funções anônimas são conversíveis para ambos.

A forma recomendada de um tipo genérico `C<T>` que dá suporte ao padrão de expressão de consulta é mostrada abaixo. Um tipo genérico é usado para ilustrar as relações apropriadas entre os tipos de parâmetro e de resultado, mas é possível implementar o padrão para tipos não genéricos também.

C#

```
delegate R Func<T1,R>(T1 arg1);

delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);

class C
{
    public C<T> Cast<T>();
}

class C<T> : C
{
    public C<T> Where(Func<T,bool> predicate);

    public C<U> Select<U>(Func<T,U> selector);

    public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
        Func<T,U,V> resultSelector);

    public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

    public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

    public O<T> OrderBy<K>(Func<T,K> keySelector);

    public O<T> OrderByDescending<K>(Func<T,K> keySelector);

    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}
```

```

class O<T> : C<T>
{
    public O<T> ThenBy<K>(Func<T,K> keySelector);

    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T> : C<T>
{
    public K Key { get; }
}

```

Os métodos acima usam os tipos de delegado genéricos `Func<T1,R>` e `Func<T1,T2,R>` , mas também poderiam ter usado igualmente outros tipos de árvore de expressão ou de representante com as mesmas relações nos tipos de parâmetro e de resultado.

Observe a relação recomendada entre o `C<T>` e o `O<T>` que garante que os `ThenBy` `ThenByDescending` métodos e estejam disponíveis somente no resultado de um `OrderBy` ou `OrderByDescending` . Observe também a forma recomendada do resultado de `GroupBy` --uma sequência de sequências, em que cada sequência interna tem uma `Key` propriedade adicional.

O `System.Linq` namespace fornece uma implementação do padrão de operador de consulta para qualquer tipo que implemente a `System.Collections.Generic.IEnumerable<T>` interface.

## Operadores de atribuição

Os operadores de atribuição atribuem um novo valor a uma variável, uma propriedade, um evento ou um elemento do indexador.

```

antlr

assignment
: unary_expression assignment_operator expression
;

assignment_operator
: '='
| '+='
| '-='
| '*='
| '/='
| '%='
| '&='
| '|='
| '^='

```

```
| '<<='
| right_shift_assignment
;
```

O operando esquerdo de uma atribuição deve ser uma expressão classificada como uma variável, um acesso de propriedade, um acesso de indexador ou um acesso de evento.

O `=` operador é chamado de **operador de atribuição simples**. Ele atribui o valor do operando direito à variável, à propriedade ou ao elemento do indexador fornecido pelo operando esquerdo. O operando esquerdo do operador de atribuição simples não pode ser um acesso de evento (exceto conforme descrito em [eventos de campo](#)). O operador de atribuição simples é descrito em [atribuição simples](#).

Os operadores de atribuição diferentes do `=` operador são chamados de **operadores de atribuição compostos**. Esses operadores executam a operação indicada nos dois operandos e, em seguida, atribuem o valor resultante à variável, à propriedade ou ao elemento do indexador fornecido pelo operando esquerdo. Os operadores de atribuição compostos são descritos em [atribuição composta](#).

Os `+=` `-=` operadores e com uma expressão de acesso de evento como o operando à esquerda são chamados de **operadores de atribuição de eventos**. Nenhum outro operador de atribuição é válido com um acesso de evento como o operando esquerdo. Os operadores de atribuição de eventos são descritos em [atribuição de eventos](#).

Os operadores de atribuição são associativos à direita, o que significa que as operações são agrupadas da direita para a esquerda. Por exemplo, uma expressão do formulário `a = b = c` é avaliada como `a = (b = c)`.

## Atribuição simples

O `=` operador é chamado de operador de atribuição simples.

Se o operando esquerdo de uma atribuição simples estiver no formato `E.P` ou `E[Ei]` onde `E` tiver o tipo de tempo de compilação `dynamic`, a atribuição será vinculada dinamicamente ([associação dinâmica](#)). Nesse caso, o tipo de tempo de compilação da expressão de atribuição é `dynamic`, e a resolução descrita abaixo ocorrerá em tempo de execução com base no tipo de tempo de execução de `E`.

Em uma atribuição simples, o operando à direita deve ser uma expressão que seja conversível implicitamente no tipo do operando esquerdo. A operação atribui o valor do

operando direito à variável, à propriedade ou ao elemento do indexador fornecido pelo operando esquerdo.

O resultado de uma expressão de atribuição simples é o valor atribuído ao operando à esquerda. O resultado tem o mesmo tipo que o operando esquerdo e sempre é classificado como um valor.

Se o operando esquerdo for um acesso de propriedade ou indexador, a propriedade ou o indexador deverá ter um `set` acessador. Se esse não for o caso, ocorrerá um erro de tempo de associação.

O processamento em tempo de execução de uma atribuição simples do formulário `x = y` consiste nas seguintes etapas:

- Se `x` é classificado como uma variável:
  - `x` é avaliado para produzir a variável.
  - `y` é avaliado e, se necessário, convertido para o tipo de `x` por meio de uma conversão implícita ([conversões implícitas](#)).
  - Se a variável fornecida pelo `x` for um elemento de matriz de um *reference\_type*, uma verificação de tempo de execução será executada para garantir que o valor computado `y` seja compatível com a instância de matriz do que `x` é um elemento. A verificação terá sucesso se `y` for `null`, ou se uma conversão de referência implícita ([conversões de referência implícita](#)) existir do tipo real da instância referenciada pelo `y` para o tipo de elemento real da instância de matriz que contém `x`. Caso contrário, uma `System.ArrayTypeMismatchException` será gerada.
  - O valor resultante da avaliação e da conversão de `y` é armazenado no local fornecido pela avaliação de `x`.
- Se `x` é classificado como um acesso de propriedade ou indexador:
  - A expressão de instância (se `x` não for `static`) e a lista de argumentos (se `x` for um acesso de indexador) associada a `x` são avaliadas e os resultados são usados na `set` invocação subsequente do acessador.
  - `y` é avaliado e, se necessário, convertido para o tipo de `x` por meio de uma conversão implícita ([conversões implícitas](#)).
  - O `set` acessador do `x` é invocado com o valor calculado `y` como seu `value` argumento.

As regras de covariância de matriz ([covariância de matriz](#)) permitem que um valor de um tipo de matriz `A[]` seja uma referência a uma instância de um tipo de matriz `B[]`, desde que exista uma conversão de referência implícita de `B` para `A`. Devido a essas regras, a atribuição a um elemento de matriz de um *reference\_type* requer uma

verificação de tempo de execução para garantir que o valor que está sendo atribuído seja compatível com a instância de matriz. No exemplo

```
C#  
  
string[] sa = new string[10];  
object[] oa = sa;  
  
oa[0] = null; // Ok  
oa[1] = "Hello"; // Ok  
oa[2] = new ArrayList(); // ArrayTypeMismatchException
```

a última atribuição faz com que um seja `System.ArrayTypeMismatchException` gerado porque uma instância do `ArrayList` não pode ser armazenada em um elemento de um `string[]`.

Quando uma propriedade ou um indexador declarado em um *struct\_type* é o destino de uma atribuição, a expressão de instância associada à propriedade ou ao acesso do indexador deve ser classificada como uma variável. Se a expressão de instância for classificada como um valor, ocorrerá um erro de tempo de associação. Devido ao [acesso de membro](#), a mesma regra também se aplica a campos.

Dadas as declarações:

```
C#  
  
struct Point  
{  
    int x, y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int X {  
        get { return x; }  
        set { x = value; }  
    }  
  
    public int Y {  
        get { return y; }  
        set { y = value; }  
    }  
}  
  
struct Rectangle  
{  
    Point a, b;
```

```

public Rectangle(Point a, Point b) {
    this.a = a;
    this.b = b;
}

public Point A {
    get { return a; }
    set { a = value; }
}

public Point B {
    get { return b; }
    set { b = value; }
}
}

```

no exemplo

C#

```

Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;

```

as atribuições para `p.X` , `p.Y` , `r.A` e `r.B` são permitidas porque `p` e `r` são variáveis. No entanto, no exemplo

C#

```

Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;

```

as atribuições são todas inválidas, pois `r.A` e `r.B` não são variáveis.

## Atribuição composta

Se o operando esquerdo de uma atribuição composta estiver no formato `E.P` ou `E[Ei]` onde `E` tiver o tipo de tempo de compilação `dynamic` , a atribuição será vinculada dinamicamente ([associação dinâmica](#)). Nesse caso, o tipo de tempo de compilação da

expressão de atribuição é `dynamic`, e a resolução descrita abaixo ocorrerá em tempo de execução com base no tipo de tempo de execução de `E`.

Uma operação do formulário `x op= y` é processada pela aplicação da resolução de sobrecarga de operador binário ([resolução de sobrecarga de operador binário](#)) como se a operação fosse gravada `x op y`. E, em seguida,

- Se o tipo de retorno do operador selecionado for implicitamente conversível para o tipo de `x`, a operação será avaliada como `x = x op y`, exceto que `x` é avaliada apenas uma vez.
- Caso contrário, se o operador selecionado for um operador predefinido, se o tipo de retorno do operador selecionado for explicitamente conversível para o tipo de `x` e se `y` for implicitamente conversível para o tipo `x` ou o operador for um operador de deslocamento, a operação será avaliada como `x = (T)(x op y)`, em que `T` é o tipo de `x`, exceto que `x` é avaliada apenas uma vez.
- Caso contrário, a atribuição composta é inválida e ocorre um erro de tempo de ligação.

O termo "avaliado apenas uma vez" significa que, na avaliação de `x op y`, os resultados de quaisquer expressões constituintes do `x` são temporariamente salvos e reutilizados ao executar a atribuição para o `x`. Por exemplo, na atribuição `A()[B()] += C()`, em que `A` é um método que retorna `int[]` `B` e `C` são métodos retornando `int`, os métodos são invocados apenas uma vez, na ordem `A`, `B`, `C`.

Quando o operando esquerdo de uma atribuição composta é um acesso de propriedade ou indexador, a propriedade ou o indexador deve ter um `get` acessador e um `set` acessador. Se esse não for o caso, ocorrerá um erro de tempo de associação.

A segunda regra acima permite `x op= y` ser avaliada como `x = (T)(x op y)` em determinados contextos. A regra existe de modo que os operadores predefinidos possam ser usados como operadores compostos quando o operando esquerdo for do tipo `sbyte`, `byte`, `short`, `ushort` ou `char`. Mesmo quando ambos os argumentos são de um desses tipos, os operadores predefinidos produzem um resultado do tipo `int`, conforme descrito em [promoções numéricas binárias](#). Portanto, sem uma conversão, não seria possível atribuir o resultado ao operando esquerdo.

O efeito intuitivo da regra para operadores predefinidos é simplesmente `x op= y` permitido se ambos `x op y` `x = y` forem permitidos. No exemplo

```

byte b = 0;
char ch = '\0';
int i = 0;

b += 1;           // Ok
b += 1000;        // Error, b = 1000 not permitted
b += i;           // Error, b = i not permitted
b += (byte)i;     // Ok

ch += 1;          // Error, ch = 1 not permitted
ch += (char)1;    // Ok

```

o motivo intuitivo de cada erro é que uma atribuição simples correspondente também teria sido um erro.

Isso também significa que as operações de atribuição compostas dão suporte a operações levantadas. No exemplo

```
C#
int? i = 0;
i += 1;           // Ok
```

o operador levantado `+(int?,int?)` é usado.

## Atribuição de evento

Se o operando esquerdo de um `+=` `-=` operador or for classificado como um acesso de evento, a expressão será avaliada da seguinte maneira:

- A expressão de instância, se houver, do acesso ao evento é avaliada.
- O operando à direita do `+=` `-=` operador OR é avaliado e, se necessário, convertido para o tipo do operando esquerdo por meio de uma conversão implícita ([conversões implícitas](#)).
- Um acessador de eventos do evento é invocado, com a lista de argumentos que consiste no operando direito, após a avaliação e, se necessário, a conversão. Se o operador era `+=`, o `add` acessador é invocado; se o operador era `-=`, o `remove` acessador é invocado.

Uma expressão de atribuição de evento não produz um valor. Assim, uma expressão de atribuição de evento é válida somente no contexto de uma *statement\_expression* ([instruções de expressão](#)).

# Expression

Uma *expressão* é uma *non\_assignment\_expression* ou uma *atribuição*.

```
antlr

expression
: non_assignment_expression
| assignment
;

non_assignment_expression
: conditional_expression
| lambda_expression
| query_expression
;
```

## Expressões constantes

Uma *constant\_expression* é uma expressão que pode ser totalmente avaliada em tempo de compilação.

```
antlr

constant_expression
: expression
;
```

Uma expressão constante deve ser o `null` literal ou um valor com um dos seguintes tipos: `sbyte` , `byte` `short` , `ushort` , `int` `uint` , `long` , `ulong` , `char` `float` `double` , `decimal` , `bool` , `object` , `string` ou qualquer tipo de enumeração. Somente as seguintes construções são permitidas em expressões constantes:

- Literais (incluindo o `null` literal).
- Referências a `const` membros de tipos de classe e struct.
- Referências a membros de tipos de enumeração.
- Referências a `const` parâmetros ou variáveis locais
- Subexpressão entre parênteses, que são expressões constantes.
- Expressões de conversão, desde que o tipo de destino seja um dos tipos listados acima.
- `checked` e `unchecked` expressões
- Expressões de valor padrão
- Expressões `nameof`

- Os operadores predefinidos `+`, `-`, e `!` são unários.
- Os operadores predefinidos `*, /`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, e `>=` binários, desde que cada operando seja de um tipo listado acima.
- O `?:` operador condicional.

As conversões a seguir são permitidas em expressões constantes:

- Conversões de identidade
- Conversões numéricas
- Conversões de enumeração
- Conversões de expressão de constante
- Conversões de referência implícitas e explícitas, desde que a origem das conversões seja uma expressão constante que é avaliada como o valor `NULL`.

Outras conversões incluindo boxing, unboxing e conversões de referência implícita de valores não nulos não são permitidas em expressões constantes. Por exemplo:

C#

```
class C {
    const object i = 5;           // error: boxing conversion not permitted
    const object str = "hello";   // error: implicit reference conversion
}
```

A inicialização de `i` é um erro porque uma conversão boxing é necessária. A inicialização de `str` é um erro porque uma conversão de referência implícita de um valor não nulo é necessária.

Sempre que uma expressão atende aos requisitos listados acima, a expressão é avaliada em tempo de compilação. Isso é verdadeiro mesmo que a expressão seja uma subexpressão de uma expressão maior que contenha construções não constantes.

A avaliação de tempo de compilação de expressões constantes usa as mesmas regras que a avaliação de tempo de execução de expressões não constantes, exceto que, quando a avaliação de tempo de execução tiver gerado uma exceção, a avaliação de tempo de compilação causará a ocorrência de um erro de tempo de compilação.

A menos que uma expressão constante seja inserida explicitamente em um `unchecked` contexto, os estouros que ocorrerem em operações aritméticas de tipo integral e conversões durante a avaliação de tempo de compilação da expressão sempre causarão erros de tempo de compilação ([expressões constantes](#)).

Expressões constantes ocorrem nos contextos listados abaixo. Nesses contextos, ocorrerá um erro de tempo de compilação se uma expressão não puder ser totalmente

avaliada em tempo de compilação.

- Declarações de constantes ([constantes](#)).
- Declarações de membro de enumeração ([membros enum](#)).
- Argumentos padrão de listas de parâmetros formais ([parâmetros de método](#))
- `case` rótulos de uma `switch` instrução ([a instrução switch](#)).
- `goto case` Statements ([a instrução goto](#)).
- Comprimentos de dimensão em uma expressão de criação de matriz ([expressões de criação de matriz](#)) que inclui um inicializador.
- Atributos ([atributos](#)).

Uma conversão de expressão constante implícita ([conversões de expressão de constante implícita](#)) permite que uma expressão constante do tipo seja `int` convertida em `sbyte` , `byte` `short` `ushort` `uint` ou `ulong` , desde que o valor da expressão constante esteja dentro do intervalo do tipo de destino.

## Expressões booleanas

Uma *Boolean\_expression* é uma expressão que produz um resultado de tipo `bool` ; diretamente ou por meio da aplicação de `operator true` determinados contextos, conforme especificado no seguinte.

```
antlr  
  
boolean_expression  
    : expression  
    ;
```

A expressão condicional de controle de *um if\_statement* ([a instrução if](#)), *while\_statement* ([a instrução while](#)), *do\_statement* ([a instrução do](#)) ou *for\_statement* ([a instrução for](#)) é uma *Boolean\_expression*. A expressão condicional de controle do `?:` operador ([operador condicional](#)) segue as mesmas regras que um *Boolean\_expression*, mas, por motivos de precedência de operador, é classificado como um *conditional\_or\_expression*.

Um *Boolean\_expression* `E` é necessário para poder produzir um valor do tipo, da `bool` seguinte maneira:

- Se `E` for implicitamente conversível para `bool` , em tempo de execução, a conversão implícita será aplicada.
- Caso contrário, a resolução de sobrecarga de operador unário ([resolução de sobrecarga de operador unário](#)) é usada para encontrar uma melhor

implementação exclusiva do operador `true` em `E` e essa implementação é aplicada em tempo de execução.

- Se esse operador não for encontrado, ocorrerá um erro de tempo de associação.

O tipo `DBBool` struct no [tipo de banco de dados booleano](#) fornece um exemplo de um tipo que implementa `operator true` e `operator false`.

# Instruções

Artigo • 16/09/2021

O C# fornece uma variedade de instruções. A maioria dessas instruções será familiar para os desenvolvedores que se programaram em C e C++.

```
antlr

statement
: labeled_statement
| declaration_statement
| embedded_statement
;

embedded_statement
: block
| empty_statement
| expression_statement
| selection_statement
| iteration_statement
| jump_statement
| try_statement
| checked_statement
| unchecked_statement
| lock_statement
| using_statement
| yield_statement
| embedded_statement_unsafe
;
```

O *embedded\_statement* não terminal é usado para instruções que aparecem dentro de outras instruções. O uso de *embedded\_statement* em vez de *instrução* exclui o uso de instruções de declaração e instruções rotuladas nesses contextos. O exemplo

```
C#

void F(bool b) {
    if (b)
        int i = 44;
}
```

resulta em um erro de tempo de compilação porque uma `if` instrução requer uma *embedded\_statement* em vez de uma *instrução* para sua ramificação If. Se esse código fosse permitido, a variável `i` seria declarada, mas nunca poderia ser usada. No entanto, observe que, colocando `i` a declaração de em um bloco, o exemplo é válido.

# Pontos de extremidade e acessibilidade

Cada instrução tem um *ponto de extremidade*. Em termos intuitivos, o ponto de extremidade de uma instrução é o local que segue imediatamente a instrução. As regras de execução para instruções compostas (instruções que contêm instruções inseridas) especificam a ação que é executada quando o controle atinge o ponto de extremidade de uma instrução inserida. Por exemplo, quando o controle atinge o ponto de extremidade de uma instrução em um bloco, o controle é transferido para a próxima instrução no bloco.

Se uma instrução puder ser alcançada pela execução, a instrução será considerada *\*alcançável*. Por outro lado, se não houver nenhuma possibilidade de que uma instrução seja executada, a instrução será considerada *\_inacessível\_\**.

No exemplo

```
C#  
  
void F() {  
    Console.WriteLine("reachable");  
    goto Label;  
    Console.WriteLine("unreachable");  
    Label:  
    Console.WriteLine("reachable");  
}
```

a segunda invocação de `Console.WriteLine` está inacessível porque não há possibilidade de que a instrução seja executada.

Um aviso será relatado se o compilador determinar que uma instrução está inacessível. Especificamente, não é um erro para que uma instrução fique inacessível.

Para determinar se uma determinada instrução ou ponto de extremidade pode ser acessado, o compilador executa a análise de fluxo de acordo com as regras de acessibilidade definidas para cada instrução. A análise de fluxo leva em conta os valores de expressões constantes ([expressões constantes](#)) que controlam o comportamento de instruções, mas os valores possíveis de expressões não constantes não são considerados. Em outras palavras, para fins de análise de fluxo de controle, uma expressão não constante de um determinado tipo é considerada com qualquer valor possível desse tipo.

No exemplo

```
C#
```

```
void F() {
    const int i = 1;
    if (i == 2) Console.WriteLine("unreachable");
}
```

a expressão booliana da `if` instrução é uma expressão constante porque ambos os operandos do `==` operador são constantes. Como a expressão constante é avaliada em tempo de compilação, produzindo o valor `false`, a `Console.WriteLine` invocação é considerada inacessível. No entanto, se `i` for alterado para ser uma variável local

C#

```
void F() {
    int i = 1;
    if (i == 2) Console.WriteLine("reachable");
}
```

a `Console.WriteLine` invocação é considerada acessível, apesar de, na realidade, nunca será executada.

O *bloco* de um membro de função é sempre considerado acessível. Ao avaliar sucessivamente as regras de acessibilidade de cada instrução em um bloco, a acessibilidade de qualquer instrução específica pode ser determinada.

No exemplo

C#

```
void F(int x) {
    Console.WriteLine("start");
    if (x < 0) Console.WriteLine("negative");
}
```

a acessibilidade do segundo `Console.WriteLine` é determinada da seguinte maneira:

- A primeira `Console.WriteLine` instrução de expressão pode ser acessada porque o bloco do `F` método está acessível.
- O ponto de extremidade da primeira `Console.WriteLine` instrução de expressão pode ser acessado porque essa instrução pode ser acessada.
- A `if` instrução pode ser acessada porque o ponto final da primeira `Console.WriteLine` instrução de expressão está acessível.
- A segunda `Console.WriteLine` instrução de expressão está acessível porque a expressão booliana da `if` instrução não tem o valor constante `false`.

Há duas situações em que é um erro de tempo de compilação para o ponto de extremidade de uma instrução ser acessível:

- Como a `switch` instrução não permite que uma seção de comutador "passe" para a próxima seção switch, é um erro de tempo de compilação para o ponto final da lista de instruções de uma seção de switch ser acessível. Se esse erro ocorrer, normalmente é uma indicação de que uma `break` instrução está ausente.
- É um erro de tempo de compilação para o ponto de extremidade do bloco de um membro de função que computa um valor a ser acessado. Se esse erro ocorrer, normalmente é uma indicação de que uma `return` instrução está ausente.

## Blocos

Um *bloco* permite a produção de várias instruções em contextos nos quais uma única instrução é permitida.

```
antlr

block
  : '{' statement_list? '}'
  ;
```

Um *bloco* consiste em um *statement\_list* opcional ([listas de instruções](#)), entre chaves. Se a lista de instruções for omitida, o bloco será considerado vazio.

Um bloco pode conter instruções de declaração ([instruções de declaração](#)). O escopo de uma variável local ou constante declarada em um bloco é o bloco.

Um bloco é executado da seguinte maneira:

- Se o bloco estiver vazio, o controle será transferido para o ponto de extremidade do bloco.
- Se o bloco não estiver vazio, o controle será transferido para a lista de instruções. Quando e se o controle atingir o ponto de extremidade da lista de instruções, o controle será transferido para o ponto de extremidade do bloco.

A lista de instruções de um bloco pode ser acessada se o próprio bloco estiver acessível.

O ponto de extremidade de um bloco pode ser acessado se o bloco estiver vazio ou se o ponto final da lista de instruções estiver acessível.

Um *bloco* que contém uma ou mais `yield` instruções ([a instrução yield](#)) é chamado de bloco de iterador. Os blocos de iteradores são usados para implementar membros de

função como iteradores ([iteradores](#)). Algumas restrições adicionais se aplicam a blocos de iteradores:

- É um erro de tempo de compilação para `return` que uma instrução apareça em um bloco de iterador (mas as `yield return` instruções são permitidas).
- É um erro de tempo de compilação para um bloco de iterador conter um contexto não seguro ([contextos não seguros](#)). Um bloco do iterador sempre define um contexto seguro, mesmo quando sua declaração está aninhada em um contexto sem segurança.

## Listas de instruções

Uma \*lista de instruções \_ consiste em uma ou mais instruções escritas em sequência. As listas de instruções ocorrem em \_block \* s ([blocos](#)) e no *Switch\_block* s ([a instrução switch](#)).

```
antlr

statement_list
: statement+
;
```

Uma lista de instruções é executada transferindo o controle para a primeira instrução. Quando e se o controle atingir o ponto de extremidade de uma instrução, o controle será transferido para a próxima instrução. Quando e se o controle atingir o ponto de extremidade da última instrução, o controle será transferido para o ponto de extremidade da lista de instruções.

Uma instrução em uma lista de instruções pode ser acessada se pelo menos uma das seguintes opções for verdadeira:

- A instrução é a primeira instrução e a própria lista de instruções pode ser acessada.
- O ponto de extremidade da instrução anterior pode ser acessado.
- A instrução é uma instrução rotulada e o rótulo é referenciado por uma `goto` instrução alcançável.

O ponto de extremidade de uma lista de instruções pode ser acessado se o ponto final da última instrução na lista estiver acessível.

## A instrução vazia

Um *empty\_statement* não faz nada.

```
antlr
```

```
empty_statement
: ';'*
;
```

Uma instrução vazia é usada quando não há nenhuma operação a ser executada em um contexto em que uma instrução é necessária.

A execução de uma instrução vazia simplesmente transfere o controle para o ponto de extremidade da instrução. Assim, o ponto de extremidade de uma instrução vazia estará acessível se a instrução vazia puder ser acessada.

Uma instrução vazia pode ser usada ao gravar uma `while` instrução com um corpo nulo:

```
C#
```

```
bool ProcessMessage() {...}

void ProcessMessages() {
    while (ProcessMessage())
        ;
}
```

Além disso, uma instrução vazia pode ser usada para declarar um rótulo logo antes do fechamento "}" de um bloco:

```
C#
```

```
void F() {
    ...
    if (done) goto exit;
    ...
exit: ;
}
```

## Instruções rotuladas

Uma *labeled\_statement* permite que uma instrução seja prefixada por um rótulo.

Instruções rotuladas são permitidas em blocos, mas não são permitidas como instruções inseridas.

```
antlr
```

```
labeled_statement
: identifier `:' statement
;
```

Uma instrução rotulada declara um rótulo com o nome fornecido pelo *identificador*. O escopo de um rótulo é o bloco inteiro no qual o rótulo é declarado, incluindo qualquer bloco aninhado. É um erro de tempo de compilação para dois rótulos com o mesmo nome para ter escopos sobrepostos.

Um rótulo pode ser referenciado de `goto` instruções ([a instrução goto](#)) dentro do escopo do rótulo. Isso significa que as `goto` instruções podem transferir o controle dentro de blocos e fora de blocos, mas nunca em blocos.

Os rótulos têm seu próprio espaço de declaração e não interferem em outros identificadores. O exemplo

```
C#
int F(int x) {
    if (x >= 0) goto x;
    x = -x;
    x: return x;
}
```

é válido e usa o nome `x` como um parâmetro e um rótulo.

A execução de uma instrução rotulada corresponde exatamente à execução da instrução após o rótulo.

Além da acessibilidade fornecida pelo fluxo de controle normal, uma instrução rotulada pode ser acessada se o rótulo for referenciado por uma `goto` instrução alcançável. (Exceção: se uma `goto` instrução estiver dentro de um `try` que inclui um `finally` bloco, e a instrução rotulada estiver fora do `try`, e o ponto final do `finally` bloco estiver inacessível, a instrução rotulada não estará acessível a partir dessa `goto` instrução.)

## Instruções de declaração

Um `declaration_statement` declara uma variável ou constante local. Instruções de declaração são permitidas em blocos, mas não são permitidas como instruções inseridas.

```
declaration_statement
: local_variable_declaration ';'
| local_constant_declaration ';'
;
```

## Declarações de variáveis locais

Um *local\_variable\_declaration* declara uma ou mais variáveis locais.

antlr

```
local_variable_declaration
: local_variable_type local_variable_declarators
;

local_variable_type
: type
| 'var'
;

local_variable_declarators
: local_variable_declarator
| local_variable_declarators ',' local_variable_declarator
;

local_variable_declarator
: identifier
| identifier '=' local_variable_initializer
;

local_variable_initializer
: expression
| array_initializer
| local_variable_initializer_unsafe
;
```

A *local\_variable\_type* de uma *local\_variable\_declaration* especifica diretamente o tipo das variáveis introduzidas pela declaração ou indica com o identificador `var` que o tipo deve ser inferido com base em um inicializador. O tipo é seguido por uma lista de *local\_variable\_declarator*s, cada um deles apresentando uma nova variável. Uma *local\_variable\_declarator* consiste em um *identificador* que nomeia a variável, opcionalmente seguido por um `=` token "" e uma *local\_variable\_initializer* que fornece o valor inicial da variável.

No contexto de uma declaração de variável local, o identificador `var` age como uma palavra-chave contextual ([palavras-chave](#)). Quando o *local\_variable\_type* é especificado como `var` e nenhum tipo chamado `var` está no escopo, a declaração é uma **declaração**

*de variável local digitada implicitamente*, cujo tipo é inferido do tipo da expressão do inicializador associado. As declarações de variáveis locais digitadas implicitamente estão sujeitas às seguintes restrições:

- O *local\_variable\_declaration* não pode incluir vários *local\_variable\_declarator*s.
- O *local\_variable\_declarator* deve incluir um *local\_variable\_initializer*.
- O *local\_variable\_initializer* deve ser uma *expressão*.
- A *expressão* do inicializador deve ter um tipo de tempo de compilação.
- A *expressão* do inicializador não pode se referir à própria variável declarada

Veja a seguir exemplos de declarações de variáveis locais incorretas de tipo implícito:

C#

```
var x;           // Error, no initializer to infer type from
var y = {1, 2, 3}; // Error, array initializer not permitted
var z = null;    // Error, null does not have a type
var u = x => x + 1; // Error, anonymous functions do not have a type
var v = v++;     // Error, initializer cannot refer to variable itself
```

O valor de uma variável local é obtido em uma expressão usando um *Simple\_name* ([nomes simples](#)) e o valor de uma variável local é modificado usando uma *atribuição* ([operadores de atribuição](#)). Uma variável local deve ser definitivamente atribuída ([atribuição definitiva](#)) em cada local em que seu valor é obtido.

O escopo de uma variável local declarada em uma *local\_variable\_declaration* é o bloco no qual a declaração ocorre. É um erro fazer referência a uma variável local em uma posição textual que precede a *local\_variable\_declarator* da variável local. Dentro do escopo de uma variável local, é um erro de tempo de compilação para declarar outra variável local ou constante com o mesmo nome.

Uma declaração de variável local que declara várias variáveis é equivalente a várias declarações de variáveis únicas com o mesmo tipo. Além disso, um inicializador de variável em uma declaração de variável local corresponde exatamente a uma instrução de atribuição que é inserida imediatamente após a declaração.

O exemplo

C#

```
void F() {
    int x = 1, y, z = x * 2;
}
```

corresponde exatamente a

```
C#
```

```
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

Em uma declaração de variável local digitada implicitamente, o tipo da variável local que está sendo declarada é usado como sendo o mesmo que o tipo da expressão usada para inicializar a variável. Por exemplo:

```
C#
```

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

As declarações de variável local digitadas implicitamente acima são exatamente equivalentes às seguintes declarações tipadas explicitamente:

```
C#
```

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

## Declarações de constantes locais

Um *local\_constant\_declaration* declara uma ou mais constantes locais.

```
antlr
```

```
local_constant_declaration
: 'const' type constant_declarators
;

constant_declarators
: constant_declarator (',' constant_declarator)*
;

constant_declarator
```

```
: identifier '=' constant_expression  
;
```

O *tipo* de um *local\_constant\_declaration* especifica o tipo das constantes introduzidas pela declaração. O tipo é seguido por uma lista de *constant\_declarator*s, cada um dos quais introduz uma nova constante. Um *constant\_declarator* consiste em um *identificador* que nomeia a constante, seguido por um `=` token "", seguido por uma *constant\_expression* ([expressões constantes](#)) que fornece o valor da constante.

O *tipo* e *constant\_expression* de uma declaração de constante local devem seguir as mesmas regras que as de uma declaração de membro constante ([constantes](#)).

O valor de uma constante local é obtido em uma expressão usando um *Simple\_name* ([nomes simples](#)).

O escopo de uma constante local é o bloco no qual a declaração ocorre. É um erro fazer referência a uma constante local em uma posição textual que precede sua *constant\_declarator*. Dentro do escopo de uma constante local, é um erro de tempo de compilação para declarar outra variável local ou constante com o mesmo nome.

Uma declaração de constante local que declara várias constantes é equivalente a várias declarações de constantes únicas com o mesmo tipo.

## Instruções de expressão

Um *expression\_statement* avalia uma determinada expressão. O valor calculado pela expressão, se houver, é Descartado.

```
antlr  
  
expression_statement  
: statement_expression ';'  
;  
  
statement_expression  
: invocation_expression  
| null_conditional_invocation_expression  
| object_creation_expression  
| assignment  
| post_increment_expression  
| post_decrement_expression  
| pre_increment_expression  
| pre_decrement_expression  
| await_expression  
;
```

Nem todas as expressões são permitidas como instruções. Em particular, as expressões como `x + y` e `x == 1` que meramente computam um valor (que será Descartado) não são permitidas como instruções.

A execução de um *expression\_statement* avalia a expressão contida e, em seguida, transfere o controle para o ponto de extremidade da *expression\_statement*. O ponto de extremidade de um *expression\_statement* pode ser acessado se esse *expression\_statement* estiver acessível.

## Instruções de seleção

Instruções de seleção seleciona um número de possíveis instruções para execução com base no valor de alguma expressão.

```
antlr

selection_statement
: if_statement
| switch_statement
;
```

### Instrução if

A `if` instrução seleciona uma instrução para execução com base no valor de uma expressão booliana.

```
antlr

if_statement
: 'if' '(' boolean_expression ')' embedded_statement
| 'if' '(' boolean_expression ')' embedded_statement 'else'
embedded_statement
;
```

Uma `else` parte é associada ao precedentes lexicalmente mais próximo `if` que é permitido pela sintaxe. Portanto, uma `if` instrução do formulário

```
C#
if (x) if (y) F(); else G();
```

é equivalente a

C#

```
if (x) {
    if (y) {
        F();
    }
    else {
        G();
    }
}
```

Uma `if` instrução é executada da seguinte maneira:

- O *Boolean\_expression* ([expressões booleanas](#)) é avaliado.
- Se a expressão booliana produz `true`, o controle é transferido para a primeira instrução inserida. Quando e se o controle atingir o ponto final dessa instrução, o controle será transferido para o ponto de extremidade da `if` instrução.
- Se a expressão booliana produz `false` e se uma `else` parte estiver presente, o controle será transferido para a segunda instrução inserida. Quando e se o controle atingir o ponto final dessa instrução, o controle será transferido para o ponto de extremidade da `if` instrução.
- Se a expressão booliana produz `false` e se uma `else` parte não estiver presente, o controle será transferido para o ponto final da `if` instrução.

A primeira instrução inserida de uma `if` instrução pode ser acessada se a `if` instrução estiver acessível e a expressão booliana não tiver o valor constante `false`.

A segunda instrução inserida de uma `if` instrução, se presente, estará acessível se a `if` instrução estiver acessível e a expressão booliana não tiver o valor constante `true`.

O ponto de extremidade de uma `if` instrução pode ser acessado se o ponto de extremidade de pelo menos uma de suas instruções inseridas estiver acessível. Além disso, o ponto de extremidade de uma `if` instrução sem nenhuma `else` parte será acessível se a `if` instrução estiver acessível e a expressão booliana não tiver o valor constante `true`.

## A instrução switch

A instrução `switch` seleciona a execução de uma lista de instruções com um rótulo de comutador associado que corresponde ao valor da expressão `switch`.

```

switch_statement
: 'switch' '(' expression ')' switch_block
;

switch_block
: '{' switch_section* '}'
;

switch_section
: switch_label+ statement_list
;

switch_label
: 'case' constant_expression ':'
| 'default' ':'
;

```

Uma *switch\_statement* consiste na palavra-chave `switch`, seguida por uma expressão entre parênteses (chamada de expressão `switch`), seguida por uma *switch\_block*. O *switch\_block* consiste em zero ou mais *switch\_section*s, entre chaves. Cada *switch\_section* consiste em um ou mais *switch\_label*s seguidos por uma *statement\_list* ([listas de instruções](#)).

O *tipo de controle* de uma `switch` instrução é estabelecido pela expressão `switch`.

- Se o tipo da expressão `switch` for `sbyte` `byte` `short` `ushort` `int` `uint` `long` `ulong` `bool` `char`, `string` ou um `enum_type`, ou se for o tipo anulável correspondente a um desses tipos, esse será o tipo regulador da `switch` instrução.
- Caso contrário, exatamente uma conversão implícita definida pelo usuário ([conversões definidas pelo usuário](#)) deve existir a partir do tipo da expressão `switch` para um dos seguintes tipos de controle possíveis: `sbyte` `byte` `short` `ushort` `int` `uint` `long` `ulong`, `char`, `string` ou, um tipo anulável correspondente a um desses tipos.
- Caso contrário, se nenhuma conversão implícita existir, ou se houver mais de uma conversão implícita, ocorrerá um erro em tempo de compilação.

A expressão constante de cada `case` rótulo deve indicar um valor que é implicitamente conversível ([conversões implícitas](#)) para o tipo regulador da `switch` instrução. Um erro de tempo de compilação ocorre se dois ou mais `case` Rótulos na mesma `switch` instrução especificarem o mesmo valor de constante.

Pode haver no máximo um `default` rótulo em uma instrução `switch`.

Uma `switch` instrução é executada da seguinte maneira:

- A expressão switch é avaliada e convertida no tipo regulador.
- Se uma das constantes especificadas em um `case` rótulo na mesma `switch` instrução for igual ao valor da expressão switch, o controle será transferido para a lista de instruções após o `case` rótulo correspondente.
- Se nenhuma das constantes especificadas em `case` Rótulos na mesma `switch` instrução for igual ao valor da expressão switch e, se um `default` rótulo estiver presente, o controle será transferido para a lista de instruções após o `default` rótulo.
- Se nenhuma das constantes especificadas em `case` Rótulos na mesma `switch` instrução for igual ao valor da expressão switch e, se nenhum `default` rótulo estiver presente, o controle será transferido para o ponto final da `switch` instrução.

Se o ponto de extremidade da lista de instruções de uma seção de comutador estiver acessível, ocorrerá um erro em tempo de compilação. Isso é conhecido como a regra "não cair". O exemplo

```
C#  
  
switch (i) {  
    case 0:  
        CaseZero();  
        break;  
    case 1:  
        CaseOne();  
        break;  
    default:  
        CaseOthers();  
        break;  
}
```

é válido porque nenhuma seção de opção tem um ponto de extremidade acessível. Ao contrário de C e C++, a execução de uma seção de comutador não é permitida para "passar" para a próxima seção de switch e o exemplo

```
C#  
  
switch (i) {  
    case 0:  
        CaseZero();  
    case 1:  
        CaseZeroOrOne();  
    default:  
        CaseAny();  
}
```

resulta em um erro de tempo de compilação. Quando a execução de uma seção switch é seguida pela execução de outra seção switch, uma `goto case` instrução or explícita `goto default` deve ser usada:

```
C#  
  
switch (i) {  
    case 0:  
        CaseZero();  
        goto case 1;  
    case 1:  
        CaseZeroOrOne();  
        goto default;  
    default:  
        CaseAny();  
        break;  
}
```

Vários rótulos são permitidos em um *switch\_section*. O exemplo

```
C#  
  
switch (i) {  
    case 0:  
        CaseZero();  
        break;  
    case 1:  
        CaseOne();  
        break;  
    case 2:  
    default:  
        CaseTwo();  
        break;  
}
```

é válido. O exemplo não viola a regra "no enquadramento" porque os rótulos `case 2:` e `default:` fazem parte do mesmo *switch\_section*.

A regra "no enquadramento" impede uma classe comum de bugs que ocorrem em C e C++ quando as `break` instruções são acidentalmente omitidas. Além disso, devido a essa regra, as seções switch de uma `switch` instrução podem ser reorganizadas arbitrariamente sem afetar o comportamento da instrução. Por exemplo, as seções da `switch` instrução acima podem ser revertidas sem afetar o comportamento da instrução:

```
C#
```

```
switch (i) {
default:
    CaseAny();
    break;
case 1:
    CaseZeroOrOne();
    goto default;
case 0:
    CaseZero();
    goto case 1;
}
```

A lista de instruções de uma seção de alternância normalmente termina em uma `break`, `goto case` instrução, ou `goto default`, mas qualquer construção que renderiza o ponto final da lista de instrução inacessível é permitida. Por exemplo, uma `while` instrução controlada pela expressão booliana `true` é conhecida como nunca atingir seu ponto de extremidade. Da mesma forma, uma `throw` `return` instrução or sempre transfere o controle em outro lugar e nunca atinge seu ponto de extremidade. Portanto, o exemplo a seguir é válido:

```
C#

switch (i) {
case 0:
    while (true) F();
case 1:
    throw new ArgumentException();
case 2:
    return;
}
```

O tipo regulador de uma `switch` instrução pode ser o tipo `string`. Por exemplo:

```
C#

void DoCommand(string command) {
    switch (command.ToLower()) {
        case "run":
            DoRun();
            break;
        case "save":
            DoSave();
            break;
        case "quit":
            DoQuit();
            break;
        default:
            InvalidCommand(command);
    }
}
```

```
    break;  
}  
}
```

Como os operadores de igualdade de cadeia de caracteres ([operadores de igualdade de cadeia de caracteres](#)), a instrução diferencia `switch` maiúsculas de minúsculas e executará uma determinada seção de comutador somente se a cadeia de caracteres de expressão `switch` corresponder exatamente a uma `case` constante

Quando o tipo regulador de uma `switch` instrução é `string`, o valor `null` é permitido como uma constante de rótulo de caso.

O `statement_list`s de um `switch_block` pode conter instruções de declaração ([instruções de declaração](#)). O escopo de uma variável local ou constante declarada em um bloco `switch` é o bloco `switch`.

A lista de instruções de uma determinada seção de comutador pode ser acessada se a `switch` instrução estiver acessível e pelo menos uma das seguintes opções for verdadeira:

- A expressão `switch` é um valor não constante.
- A expressão `switch` é um valor constante que corresponde a um `case` rótulo na seção `switch`.
- A expressão `switch` é um valor constante que não corresponde a nenhum `case` rótulo e a seção `switch` contém o `default` rótulo.
- Um rótulo de `switch` da seção `switch` é referenciado por uma `goto case` instrução ou alcançável `goto default`.

O ponto de extremidade de uma `switch` instrução pode ser acessado se pelo menos uma das seguintes opções for verdadeira:

- A `switch` instrução contém uma `break` instrução alcançável que sai da `switch` instrução.
- A `switch` instrução está acessível, a expressão `switch` é um valor não constante e nenhum `default` rótulo está presente.
- A `switch` instrução está acessível, a expressão `switch` é um valor constante que não corresponde a nenhum `case` rótulo e nenhum `default` rótulo está presente.

## Instruções de iteração

Instruções de iteração executa repetidamente uma instrução inserida.

```
antlr
```

```
iteration_statement
: while_statement
| do_statement
| for_statement
| foreach_statement
;
```

## A instrução while

A `while` instrução Executa condicionalmente uma instrução inserida zero ou mais vezes.

```
antlr
```

```
while_statement
: 'while' '(' boolean_expression ')' embedded_statement
;
```

Uma `while` instrução é executada da seguinte maneira:

- O *Boolean\_expression* ([expressões booleanas](#)) é avaliado.
- Se a expressão booliana produz `true`, o controle é transferido para a instrução inserida. Quando e se o controle atingir o ponto de extremidade da instrução inserida (possivelmente da execução de uma `continue` instrução), o controle será transferido para o início da `while` instrução.
- Se a expressão booliana produz `false`, o controle é transferido para o ponto final da `while` instrução.

Dentro da instrução inserida de uma `while` instrução, uma `break` instrução ([a instrução break](#)) pode ser usada para transferir o controle para o ponto final da `while` instrução (encerrando, assim, a iteração da instrução incorporada) e uma `continue` instrução ([a instrução Continue](#)) pode ser usada para transferir o controle para o ponto final da instrução inserida (assim, executando outra iteração da `while` instrução).

A instrução inserida de uma `while` instrução pode ser acessada se a `while` instrução estiver acessível e a expressão booliana não tiver o valor constante `false`.

O ponto de extremidade de uma `while` instrução pode ser acessado se pelo menos uma das seguintes opções for verdadeira:

- A `while` instrução contém uma `break` instrução alcançável que sai da `while` instrução.

- A `while` instrução está acessível e a expressão booliana não tem o valor constante `true`.

## A instrução do

A `do` instrução Executa condicionalmente uma instrução inserida uma ou mais vezes.

```
antlr
do_statement
: 'do' embedded_statement 'while' '(' boolean_expression ')' ';'
```

Uma `do` instrução é executada da seguinte maneira:

- O controle é transferido para a instrução inserida.
- Quando e se o controle atingir o ponto de extremidade da instrução inserida (possivelmente da execução de uma `continue` instrução), a *Boolean\_expression* ([expressões booleanas](#)) será avaliada. Se a expressão booliana produz `true`, o controle é transferido para o início da `do` instrução. Caso contrário, o controle será transferido para o ponto de extremidade da `do` instrução.

Dentro da instrução inserida de uma `do` instrução, uma `break` instrução ([a instrução break](#)) pode ser usada para transferir o controle para o ponto final da `do` instrução (encerrando assim a iteração da instrução inserida) e uma `continue` instrução ([a instrução Continue](#)) pode ser usada para transferir o controle para o ponto final da instrução inserida.

A instrução inserida de uma `do` instrução pode ser acessada se a `do` instrução estiver acessível.

O ponto de extremidade de uma `do` instrução pode ser acessado se pelo menos uma das seguintes opções for verdadeira:

- A `do` instrução contém uma `break` instrução alcançável que sai da `do` instrução.
- O ponto de extremidade da instrução inserida pode ser acessado e a expressão booliana não tem o valor constante `true`.

## A instrução for

A `for` instrução avalia uma sequência de expressões de inicialização e, em seguida, enquanto uma condição é verdadeira, executa repetidamente uma instrução inserida e

avalia uma sequência de expressões de iteração.

antlr

```
for_statement
    : 'for' '(' for_initializer? ';' for_condition? ';' for_iterator? ')'
embedded_statement
;

for_initializer
: local_variable_declaration
| statement_expression_list
;

for_condition
: boolean_expression
;

for_iterator
: statement_expression_list
;

statement_expression_list
: statement_expression (',' statement_expression)*
;
```

O *for\_initializer*, se presente, consiste em uma *local\_variable\_declaration* ([declarações de variável local](#)) ou uma lista de *statement\_expression*s ([instruções de expressão](#)) separadas por vírgulas. O escopo de uma variável local declarada por um *for\_initializer* começa na *local\_variable\_declarator* da variável e se estende ao final da instrução inserida. O escopo inclui o *for\_condition* e o *for\_iterator*.

A *for\_condition*, se presente, deve ser uma *Boolean\_expression* ([expressões booleanas](#)).

A *for\_iterator*, se presente, consiste em uma lista de *statement\_expression*s ([instruções de expressão](#)) separadas por vírgulas.

Uma instrução for é executada da seguinte maneira:

- Se uma *for\_initializer* estiver presente, os inicializadores de variável ou as expressões de instrução serão executadas na ordem em que são gravadas. Esta etapa é executada apenas uma vez.
- Se uma *for\_condition* estiver presente, ela será avaliada.
- Se o *for\_condition* não estiver presente ou se a avaliação resultar `true`, o controle será transferido para a instrução inserida. Quando e se o controle atingir o ponto de extremidade da instrução inserida (possivelmente da execução de uma `continue` instrução), as expressões do *for\_iterator*, se houver, serão avaliadas em

sequência e outra iteração será executada, começando com a avaliação do *for\_condition* na etapa anterior.

- Se o *for\_condition* estiver presente e a avaliação resultar `false`, o controle será transferido para o ponto de extremidade da `for` instrução.

Dentro da instrução inserida de uma `for` instrução, uma `break` instrução ([a instrução break](#)) pode ser usada para transferir o controle para o ponto final da `for` instrução (encerrando assim a iteração da instrução inserida) e uma `continue` instrução ([a instrução Continue](#)) pode ser usada para transferir o controle para o ponto de extremidade da instrução inserida (executando o *for\_iterator* e executando outra iteração da `for` instrução, começando com o *for\_condition*).

A instrução inserida de uma `for` instrução pode ser acessada se uma das seguintes opções for verdadeira:

- A `for` instrução está acessível e nenhuma *for\_condition* está presente.
- A `for` instrução está acessível e um *for\_condition* está presente e não tem o valor constante `false`.

O ponto de extremidade de uma `for` instrução pode ser acessado se pelo menos uma das seguintes opções for verdadeira:

- A `for` instrução contém uma `break` instrução alcançável que sai da `for` instrução.
- A `for` instrução está acessível e um *for\_condition* está presente e não tem o valor constante `true`.

## A instrução foreach

A `foreach` instrução enumera os elementos de uma coleção, executando uma instrução incorporada para cada elemento da coleção.

```
antlr
```

```
foreach_statement
    : 'foreach' '(' local_variable_type identifier 'in' expression ')'
  embedded_statement
;
```

O *tipo* e o *identificador* de uma `foreach` instrução declaram a variável \* de **iteração \_** da instrução. Se o `var` identificador for fornecido como o `_local_variable_type` \* e nenhum tipo chamado `var` estiver no escopo, a variável de iteração será considerada uma **variável de iteração digitada implicitamente** e seu tipo será usado como sendo o tipo

de elemento da `foreach` instrução, conforme especificado abaixo. A variável de iteração corresponde a uma variável local somente leitura com um escopo que se estende pela instrução incorporada. Durante a execução de uma `foreach` instrução, a variável de iteração representa o elemento de coleção para o qual uma iteração está sendo executada no momento. Ocorrerá um erro de tempo de compilação se a instrução inserida tentar modificar a variável de iteração (por meio de atribuição ou os `++` `--` operadores e) ou passar a variável de iteração como um `ref` ou `out` parâmetro ou.

A seguir, para fins de brevidade,, `IEnumerable` `IEnumerator` `IEnumerable<T>` e `IEnumerator<T>` consulte os tipos correspondentes nos namespaces `System.Collections` e `System.Collections.Generic` .

O processamento em tempo de compilação de uma instrução `foreach` determina primeiro o *tipo de coleção* , *tipo de enumerador* e *tipo de elemento* da expressão. Essa determinação continua da seguinte maneira:

- Se o tipo `X` de *expressão* for um tipo de matriz, haverá uma conversão de referência implícita de `X` para a `IEnumerable` interface (desde que `System.Array` implementa essa interface). O *tipo de coleção* é a `IEnumerable` interface, o *tipo de enumerador* é a `IEnumerator` interface e o *tipo elemento* é o tipo de elemento do tipo de matriz `X` .
- Se o tipo `X` de *expressão* for `dynamic` , haverá uma conversão implícita de *expression* para a `IEnumerable` interface ([conversões dinâmicas implícitas](#)). O *tipo de coleção* é a `IEnumerable` interface e o *tipo de enumerador\** é a `IEnumerator` interface. Se o `var` identificador for fornecido como o `_local_variable_type` \*, o *tipo de elemento* será `dynamic` , caso contrário, será `object` .
- Caso contrário, determine se o tipo `X` tem um `GetEnumerator` método apropriado:
  - Executar pesquisa de membro no tipo `X` com identificador `GetEnumerator` e nenhum argumento de tipo. Se a pesquisa de membro não produzir uma correspondência ou se gerar uma ambiguidade ou produzir uma correspondência que não seja um grupo de métodos, verifique se há uma interface enumerável, conforme descrito abaixo. É recomendável que um aviso seja emitido se a pesquisa de membros produzir qualquer coisa, exceto um grupo de métodos ou nenhuma correspondência.
  - Execute a resolução de sobrecarga usando o grupo de métodos resultante e uma lista de argumentos vazia. Se a resolução de sobrecarga não resultar em métodos aplicáveis, resultar em uma ambiguidade ou resultar em um único método melhor, mas o método for estático ou não público, verifique se há uma interface enumerável, conforme descrito abaixo. É recomendável que um aviso

seja emitido se a resolução de sobrecarga produzir algo, exceto um método de instância pública não ambígua, ou nenhum dos métodos aplicáveis.

- Se o tipo `E` de retorno do `GetEnumerator` método não for uma classe, struct ou tipo de interface, um erro será produzido e nenhuma etapa adicional será executada.
  - A pesquisa de membros é executada em `E` com o identificador `Current` e nenhum argumento de tipo. Se a pesquisa de membro não produzir nenhuma correspondência, o resultado é um erro ou o resultado é qualquer coisa, exceto uma propriedade de instância pública que permite a leitura, um erro é produzido e nenhuma etapa adicional é executada.
  - A pesquisa de membros é executada em `E` com o identificador `MoveNext` e nenhum argumento de tipo. Se a pesquisa de membro não produzir nenhuma correspondência, o resultado é um erro ou o resultado é qualquer coisa, exceto um grupo de métodos, um erro é produzido e nenhuma etapa adicional é executada.
  - A resolução de sobrecarga é executada no grupo de métodos com uma lista de argumentos vazia. Se a resolução de sobrecarga não resultar em métodos aplicáveis, resultar em uma ambiguidade ou resultar em um único método melhor, mas esse método for estático ou não público, ou seu tipo de retorno não for `bool`, um erro será produzido e nenhuma etapa adicional será executada.
  - O *tipo de coleção\_* é `X`, o *tipo de enumerador\_* é `E` e o *tipo\_Element* é o tipo da `Current` propriedade.
- 
- Caso contrário, verifique se há uma interface enumerável:
    - Se entre todos os tipos `Ti` para os quais há uma conversão implícita de `X` para `IEnumerable<Ti>`, há um tipo exclusivo de `T` que `T` não é `dynamic` e para todos os outros `Ti` há uma conversão implícita de `IEnumerable<T>` para `IEnumerable<Ti>`, então o \* *tipo de coleção\_* é a interface `IEnumerable<T>`, o *tipo de enumerador* é a interface `IEnumerator<T>` e o *tipo\_elemento\** é `T`.
    - Caso contrário, se houver mais de um desses tipos `T`, um erro será produzido e nenhuma etapa adicional será executada.
    - Caso contrário, se houver uma conversão implícita de na `X` `System.Collections.IEnumerable` interface, o \* *tipo de coleção\_* é essa interface, o *tipo de enumerador* será a interface `System.Collections.IEnumerator` e o *tipo\_elemento\_\** será `object`.
    - Caso contrário, um erro é produzido e nenhuma etapa adicional é executada.

As etapas acima, se forem bem-sucedidas, produzirão um tipo de coleção `C`, tipo de enumerador e tipo de elemento de forma inequívoca `E T`. Uma instrução `foreach` do

## formulário

```
C#
```

```
foreach (V v in x) embedded_statement
```

é então expandido para:

```
C#
```

```
{  
    E e = ((C)(x)).GetEnumerator();  
    try {  
        while (e.MoveNext()) {  
            V v = (V)(T)e.Current;  
            embedded_statement  
        }  
    }  
    finally {  
        ... // Dispose e  
    }  
}
```

A variável `e` não é visível ou acessível à expressão ou à `x` instrução incorporada ou a qualquer outro código-fonte do programa. A variável `v` é somente leitura na instrução incorporada. Se não houver uma conversão explícita ([conversões explícitas](#)) de `T` (o tipo de elemento) para `v` (o *local\_variable\_type* na instrução `foreach`), um erro será produzido e nenhuma outra etapa será executada. Se `x` tiver o valor `null`, um `System.NullReferenceException` será lançado em tempo de execução.

Uma implementação tem permissão para implementar uma determinada instrução `foreach` de forma diferente, por exemplo, por motivos de desempenho, desde que o comportamento seja consistente com a expansão acima.

O posicionamento de `v` dentro do loop `while` é importante para a forma como ele é capturado por qualquer função anônima que ocorra na *embedded\_statement*.

Por exemplo:

```
C#
```

```
int[] values = { 7, 9, 13 };  
Action f = null;  
  
foreach (var value in values)  
{  
    if (f == null) f = () => Console.WriteLine("First value: " + value);  
}
```

```
}
```

```
f();
```

Se `v` foi declarado fora do loop `while`, ele seria compartilhado entre todas as iterações e seu valor após o loop `for` seria o valor final, `13`, que é o que a invocação `f` imprimiria. Em vez disso, como cada iteração tem sua própria variável `v`, aquela capturada pela `f` primeira iteração continuará a manter o valor `7`, que é o que será impresso.  
(Observação: versões anteriores do C# declaradas `v` fora do loop `While`.)

O corpo do bloco `finally` é construído de acordo com as seguintes etapas:

- Se houver uma conversão implícita do `e` para `System.IDisposable` interface, então
  - Se `e` for um tipo de valor não anulável, a cláusula `finally` será expandida para o equivalente semântico de:

```
C#
```

```
finally {
    ((System.IDisposable)e).Dispose();
}
```

- Caso contrário, a cláusula `finally` será expandida para o equivalente semântico de:

```
C#
```

```
finally {
    if (e != null) ((System.IDisposable)e).Dispose();
}
```

Exceto que se `e` for um tipo de valor ou um parâmetro de tipo instanciado para um tipo de valor, a conversão de `e` para `System.IDisposable` não fará com que a Boxing ocorra.

- Caso contrário, se `e` for um tipo lacrado, a cláusula `finally` será expandida para um bloco vazio:

```
C#
```

```
finally {  
}
```

- Caso contrário, a cláusula finally será expandida para:

C#

```
finally {
    System.IDisposable d = e as System.IDisposable;
    if (d != null) d.Dispose();
}
```

A variável local `d` não é visível ou acessível a nenhum código de usuário. Em particular, ele não entra em conflito com nenhuma outra variável cujo escopo inclui o bloco finally.

A ordem na qual `foreach` percorre os elementos de uma matriz é a seguinte: para os elementos de matrizes unidimensionais são atravessados em ordem de índice crescente, começando com index `0` e terminando com index `Length - 1`. Para matrizes multidimensionais, os elementos são percorridos de forma que os índices da dimensão mais à direita sejam aumentados primeiro, depois a dimensão à esquerda e assim por diante à esquerda.

O exemplo a seguir imprime cada valor em uma matriz bidimensional, na ordem do elemento:

C#

```
using System;

class Test
{
    static void Main() {
        double[,] values = {
            {1.2, 2.3, 3.4, 4.5},
            {5.6, 6.7, 7.8, 8.9}
        };

        foreach (double elementValue in values)
            Console.Write("{0} ", elementValue);

        Console.WriteLine();
    }
}
```

A saída produzida é a seguinte:

Console

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

No exemplo

C#

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers) Console.WriteLine(n);
```

o tipo de `n` é inferido como `int`, o tipo de elemento de `numbers`.

## Instruções de atalho

As instruções de salto transferem o controle incondicionalmente.

antlr

```
jump_statement
: break_statement
| continue_statement
| goto_statement
| return_statement
| throw_statement
;
```

O local para o qual uma instrução de salto transfere o controle é chamado de *destino* da instrução de salto.

Quando uma instrução de salto ocorre dentro de um bloco e o destino dessa instrução de salto está fora desse bloco, a instrução de salto é chamada para *sair* do bloco.

Embora uma instrução de salto possa transferir o controle para fora de um bloco, ela nunca pode transferir o controle para um bloco.

A execução de instruções de salto é complicada pela presença de instruções intermediárias `try`. Na ausência de tais `try` instruções, uma instrução de salto transfere incondicionalmente o controle da instrução de salto para seu destino. Na presença dessas instruções intermediárias `try`, a execução é mais complexa. Se a instrução de salto sair de um ou mais `try` blocos com `finally` blocos associados, o controle será transferido inicialmente para o `finally` bloco da `try` instrução mais interna. Quando o controle atingir o ponto de extremidade de um `finally` bloco, o controle será transferido para o `finally` bloco da próxima instrução de circunscrição `try`. Esse processo é repetido até que os `finally` blocos de todas as instruções intermediárias `try` tenham sido executados.

No exemplo

```
C#  
  
using System;  
  
class Test  
{  
    static void Main() {  
        while (true) {  
            try {  
                try {  
                    Console.WriteLine("Before break");  
                    break;  
                }  
                finally {  
                    Console.WriteLine("Innermost finally block");  
                }  
            }  
            finally {  
                Console.WriteLine("Outermost finally block");  
            }  
        }  
        Console.WriteLine("After break");  
    }  
}
```

os `finally` blocos associados a duas `try` instruções são executados antes de o controle ser transferido para o destino da instrução de salto.

A saída produzida é a seguinte:

```
Console  
  
Before break  
Innermost finally block  
Outermost finally block  
After break
```

## A instrução break

A `break` instrução sai da instrução,,, ou delimitadora mais próxima `switch` `while` `do` `for` `foreach` .

```
antlr  
  
break_statement  
: 'break' ' ';
```

```
;
```

O destino de uma `break` instrução é o ponto final da `switch` instrução,, `while` `do` , `for` ou delimitadora mais próxima `foreach` . Se uma `break` instrução não estiver entre uma `switch` instrução, `while` , `do` , `for` OU `foreach` , ocorrerá um erro de tempo de compilação.

Quando várias `switch` `while` instruções,, `do` , `for` ou `foreach` são aninhadas umas nas outras, uma `break` instrução aplica-se somente à instrução mais interna. Para transferir o controle entre vários níveis de aninhamento, uma `goto` instrução ([instrução goto](#)) deve ser usada.

Uma `break` instrução não pode sair de um `finally` bloco ([a instrução try](#)). Quando uma `break` instrução ocorre dentro de um `finally` bloco, o destino da `break` instrução deve estar dentro do mesmo `finally` bloco; caso contrário, ocorrerá um erro em tempo de compilação.

Uma `break` instrução é executada da seguinte maneira:

- Se a `break` instrução sair de um ou mais `try` blocos com `finally` blocos associados, o controle será transferido inicialmente para o `finally` bloco da instrução mais interna `try` . Quando e se o controle atingir o ponto de extremidade de um `finally` bloco, o controle será transferido para o `finally` bloco da próxima instrução de circunscrição `try` . Esse processo é repetido até que os `finally` blocos de todas as instruções intermediárias `try` tenham sido executados.
- O controle é transferido para o destino da `break` instrução.

Como uma `break` instrução transfere incondicionalmente o controle em outro lugar, o ponto de extremidade de uma `break` instrução nunca é acessível.

## A instrução Continue

A `continue` instrução inicia uma nova iteração da instrução,, ou delimitadora mais próxima `while` `do` `for` `foreach` .

```
antlr
```

```
continue_statement
: 'continue' ' ';
```

O destino de uma `continue` instrução é o ponto final da instrução inserida da `while` do instrução,, ou delimitadora mais próxima `for` `foreach` . Se uma `continue` instrução não estiver delimitada por `while` uma `do` instrução,, `for` ou `foreach` , ocorrerá um erro em tempo de compilação.

Quando várias `while` `do` instruções,, `for` ou `foreach` são aninhadas umas nas outras, uma `continue` instrução aplica-se somente à instrução mais interna. Para transferir o controle entre vários níveis de aninhamento, uma `goto` instrução ([instrução goto](#)) deve ser usada.

Uma `continue` instrução não pode sair de um `finally` bloco ([a instrução try](#)). Quando uma `continue` instrução ocorre dentro de um `finally` bloco, o destino da `continue` instrução deve estar dentro do mesmo `finally` bloco; caso contrário, ocorrerá um erro em tempo de compilação.

Uma `continue` instrução é executada da seguinte maneira:

- Se a `continue` instrução sair de um ou mais `try` blocos com `finally` blocos associados, o controle será transferido inicialmente para o `finally` bloco da instrução mais interna `try` . Quando e se o controle atingir o ponto de extremidade de um `finally` bloco, o controle será transferido para o `finally` bloco da próxima instrução de circunscrição `try` . Esse processo é repetido até que os `finally` blocos de todas as instruções intermediárias `try` tenham sido executados.
- O controle é transferido para o destino da `continue` instrução.

Como uma `continue` instrução transfere incondicionalmente o controle em outro lugar, o ponto de extremidade de uma `continue` instrução nunca é acessível.

## A instrução goto

A `goto` instrução transfere o controle para uma instrução que é marcada por um rótulo.

```
antlr

goto_statement
    : 'goto' identifier ';'
    | 'goto' 'case' constant_expression ';'
    | 'goto' 'default' ';'
    ;
```

O destino de uma `goto` instrução de *identificador* é a instrução rotulada com o rótulo fornecido. Se um rótulo com o nome fornecido não existir no membro da função atual ou se a `goto` instrução não estiver dentro do escopo do rótulo, ocorrerá um erro em tempo de compilação. Essa regra permite o uso de uma `goto` instrução para transferir o controle de um escopo aninhado, mas não para um escopo aninhado. No exemplo

C#

```
using System;

class Test
{
    static void Main(string[] args) {
        string[,] table = {
            {"Red", "Blue", "Green"},
            {"Monday", "Wednesday", "Friday"}
        };

        foreach (string str in args) {
            int row, colm;
            for (row = 0; row <= 1; ++row)
                for (colm = 0; colm <= 2; ++colm)
                    if (str == table[row,colm])
                        goto done;

            Console.WriteLine("{0} not found", str);
            continue;
        done:
            Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
        }
    }
}
```

uma `goto` instrução é usada para transferir o controle de um escopo aninhado.

O destino de uma `goto case` instrução é a lista de instruções na instrução de circunscrição imediatamente `switch` (a [instrução switch](#)), que contém um `case` rótulo com o valor constante fornecido. Se a `goto case` instrução não estiver delimitada por uma `switch` instrução, se a *constant\_expression* não for implicitamente conversível ([conversões implícitas](#)) para o tipo regulador da instrução de circunscrição mais próxima `switch`, ou se a instrução delimitadora mais próxima `switch` não contiver um `case` rótulo com o valor constante fornecido, ocorrerá um erro em tempo de compilação.

O destino de uma `goto default` instrução é a lista de instruções na instrução de circunscrição imediatamente `switch` (a [instrução switch](#)), que contém um `default` rótulo. Se a `goto default` instrução não estiver delimitada por uma `switch` instrução ou

se a instrução delimitadora mais próxima `switch` não contiver um `default` rótulo, ocorrerá um erro em tempo de compilação.

Uma `goto` instrução não pode sair de um `finally` bloco ([a instrução try](#)). Quando uma `goto` instrução ocorre dentro de um `finally` bloco, o destino da `goto` instrução deve estar dentro do mesmo `finally` bloco ou ocorre um erro de tempo de compilação.

Uma `goto` instrução é executada da seguinte maneira:

- Se a `goto` instrução sair de um ou mais `try` blocos com `finally` blocos associados, o controle será transferido inicialmente para o `finally` bloco da instrução mais interna `try`. Quando e se o controle atingir o ponto de extremidade de um `finally` bloco, o controle será transferido para o `finally` bloco da próxima instrução de circunscrição `try`. Esse processo é repetido até que os `finally` blocos de todas as instruções intermediárias `try` tenham sido executados.
- O controle é transferido para o destino da `goto` instrução.

Como uma `goto` instrução transfere incondicionalmente o controle em outro lugar, o ponto de extremidade de uma `goto` instrução nunca é acessível.

## A instrução return

A `return` instrução retorna o controle para o chamador atual da função na qual a `return` instrução é exibida.

```
antlr
return_statement
: 'return' expression? ';'
;
```

Uma `return` instrução sem expressão só pode ser usada em um membro de função que não Compute um valor, ou seja, um método com o tipo de resultado ([corpo do método](#)) `void`, o `set` acessador de uma propriedade ou um indexador, os `add` `remove` acessadores de um evento, um construtor de instância, um construtor estático ou um destruidor.

Uma `return` instrução com uma expressão só pode ser usada em um membro de função que computa um valor, ou seja, um método com um tipo de resultado não `void`, o `get` acessador de uma propriedade ou um indexador ou um operador definido pelo

usuário. Uma conversão implícita ([conversões implícitas](#)) deve existir do tipo da expressão para o tipo de retorno do membro da função que a contém.

Instruções de retorno também podem ser usadas no corpo de expressões de função anônimas ([expressões de função anônimas](#)) e participam da determinação de quais conversões existem para essas funções.

É um erro de tempo de compilação para `return` que uma instrução apareça em um `finally` bloco ([a instrução try](#)).

Uma `return` instrução é executada da seguinte maneira:

- Se a `return` instrução especificar uma expressão, a expressão será avaliada e o valor resultante será convertido no tipo de retorno da função que a contém por uma conversão implícita. O resultado da conversão se torna o valor de resultado produzido pela função.
- Se a `return` instrução estiver entre um ou mais `try` blocos ou `catch` com blocos associados `finally`, o controle será transferido inicialmente para o `finally` bloco da instrução mais interna `try`. Quando e se o controle atingir o ponto de extremidade de um `finally` bloco, o controle será transferido para o `finally` bloco da próxima instrução de circunscrição `try`. Esse processo é repetido até que os `finally` blocos de todas as instruções de circunscrição `try` tenham sido executados.
- Se a função que a contém não for uma função assíncrona, o controle será retornado para o chamador da função recipiente junto com o valor de resultado, se houver.
- Se a função que a contém for uma função assíncrona, o controle será retornado para o chamador atual e o valor de resultado, se houver, será registrado na tarefa de retorno conforme descrito em ([interfaces do enumerador](#)).

Como uma `return` instrução transfere incondicionalmente o controle em outro lugar, o ponto de extremidade de uma `return` instrução nunca é acessível.

## A instrução Throw

A instrução `throw` lança uma exceção.

antlr

```
throw_statement
    : 'throw' expression? ';'
    ;
```

Uma `throw` instrução com uma expressão lança o valor produzido avaliando a expressão. A expressão deve indicar um valor do tipo de classe `System.Exception`, de um tipo de classe que deriva de `System.Exception` ou de um tipo de parâmetro de tipo que tenha `System.Exception` (ou uma subclasse dele) como sua classe base efetiva. Se a avaliação da expressão produzir `null`, um `System.NullReferenceException` será lançado em vez disso.

Uma `throw` instrução sem expressão só pode ser usada em um `catch` bloco, caso em que a instrução relança a exceção que está sendo manipulada por esse bloco no momento `catch`.

Como uma `throw` instrução transfere incondicionalmente o controle em outro lugar, o ponto de extremidade de uma `throw` instrução nunca é acessível.

Quando uma exceção é lançada, o controle é transferido para a primeira `catch` cláusula em uma instrução delimitadora `try` que pode manipular a exceção. O processo que ocorre desde o ponto da exceção sendo gerada para o ponto de transferência do controle para um manipulador de exceção adequado é conhecido como **\*propagação de exceção\_**. A propagação de uma exceção consiste em avaliar repetidamente as etapas a seguir até que uma `catch` cláusula correspondente à exceção seja encontrada. Nesta descrição, o *\_ ponto de extensão\** é inicialmente o local no qual a exceção é lançada.

- No membro da função atual, cada `try` instrução que se aproxima do ponto de lançamento é examinada. Para cada instrução `S`, começando com a instrução mais interna `try` e terminando com a instrução mais externa `try`, as etapas a seguir são avaliadas:
  - Se o `try` bloco de `S` fechar o ponto de lançamento e se `S` tiver uma ou mais `catch` cláusulas, as `catch` cláusulas serão examinadas em ordem de aparência para localizar um manipulador adequado para a exceção, de acordo com as regras especificadas na seção [instrução try](#). Se uma `catch` cláusula correspondente estiver localizada, a propagação de exceção será concluída transferindo o controle para o bloco dessa `catch` cláusula.
  - Caso contrário, se o `try` bloco ou um `catch` bloco de `S` fechar o ponto de lançamento e se `S` tiver um `finally` bloco, o controle será transferido para o `finally` bloco. Se o `finally` bloco lançar outra exceção, o processamento da exceção atual será encerrado. Caso contrário, quando o controle atingir o ponto de extremidade do `finally` bloco, o processamento da exceção atual será continuado.

- Se um manipulador de exceção não estiver localizado na invocação de função atual, a invocação de função será encerrada e uma das seguintes situações ocorrerá:
  - Se a função atual for não Async, as etapas acima serão repetidas para o chamador da função com um ponto de geração correspondente à instrução da qual o membro da função foi invocado.
  - Se a função atual for Async e retornando Task, a exceção será registrada na tarefa de retorno, que será colocada em um estado falho ou cancelado, conforme descrito em [interfaces do enumerador](#).
  - Se a função atual for Async e void-retornando, o contexto de sincronização do thread atual será notificado conforme descrito em [interfaces enumeráveis](#).
- Se o processamento de exceções terminar todas as invocações de membro de função no thread atual, indicando que o thread não tem nenhum manipulador para a exceção, o thread será encerrado. O impacto dessa terminação é definido pela implementação.

## A instrução try

A `try` instrução fornece um mecanismo para capturar exceções que ocorrem durante a execução de um bloco. Além disso, a `try` instrução fornece a capacidade de especificar um bloco de código que é sempre executado quando o controle sai da `try` instrução.

```
antlr

try_statement
: 'try' block catch_clause+
| 'try' block finally_clause
| 'try' block catch_clause+ finally_clause
;

catch_clause
: 'catch' exception_specifier? exception_filter? block
;

exception_specifier
: '(' type identifier? ')'
;

exception_filter
: 'when' '(' expression ')'
;

finally_clause
```

```
: 'finally' block  
;
```

Há três formas de instruções possíveis `try`:

- Um `try` bloco seguido por um ou mais `catch` blocos.
- Um `try` bloco seguido por um `finally` bloco.
- Um `try` bloco seguido por um ou mais `catch` blocos seguidos por um `finally` bloco.

Quando uma `catch` cláusula Especifica um *exception\_specifier*, o tipo deve ser `System.Exception`, um tipo que deriva de `System.Exception` ou um tipo de parâmetro de tipo que tenha `System.Exception` (ou uma subclasse dele) como sua classe base em vigor.

Quando uma `catch` cláusula Especifica um *exception\_specifier* com um *identificador*, uma variável de exceção\* \_ do nome e tipo fornecido é declarada. A variável de exceção corresponde a uma variável local com um escopo que se estende pela `catch` cláusula. Durante a execução do `_exception_filter` \* e do `bloco`, a variável de exceção representa a exceção que está sendo manipulada no momento. Para fins de verificação de atribuição definitiva, a variável de exceção é considerada definitivamente atribuída em seu escopo inteiro.

A menos que uma `catch` cláusula inclua um nome de variável de exceção, é impossível acessar o objeto de exceção no filtro e no `catch` bloco.

Uma `catch` cláusula que não especifica um *exception\_specifier* é chamada de cláusula geral `catch`.

Algumas linguagens de programação podem dar suporte a exceções que não podem ser representes como um objeto derivado de `System.Exception`, embora essas exceções nunca pudessesem ser geradas pelo código C#. Uma `catch` cláusula geral pode ser usada para capturar essas exceções. Portanto, uma `catch` cláusula geral é semanticamente diferente de uma que especifica o tipo `System.Exception`, pois a primeira também pode detectar exceções de outras linguagens.

Para localizar um manipulador de uma exceção, as `catch` cláusulas são examinadas em ordem lexical. Se uma `catch` cláusula Especifica um tipo, mas nenhum filtro de exceção, é um erro de tempo de compilação para uma `catch` cláusula posterior na mesma `try` instrução para especificar um tipo que seja igual ou derivado desse tipo. Se uma `catch` cláusula não especificar nenhum tipo e nenhum filtro, ela deverá ser a última `catch` cláusula para essa `try` instrução.

Dentro de um `catch` bloco, uma `throw` instrução ([a instrução Throw](#)) sem expressão pode ser usada para relançar a exceção que foi detectada pelo `catch` bloco. As atribuições a uma variável de exceção não alteram a exceção gerada novamente.

No exemplo

```
C#  
  
using System;  
  
class Test  
{  
    static void F() {  
        try {  
            G();  
        }  
        catch (Exception e) {  
            Console.WriteLine("Exception in F: " + e.Message);  
            e = new Exception("F");  
            throw; // re-throw  
        }  
    }  
  
    static void G() {  
        throw new Exception("G");  
    }  
  
    static void Main() {  
        try {  
            F();  
        }  
        catch (Exception e) {  
            Console.WriteLine("Exception in Main: " + e.Message);  
        }  
    }  
}
```

o método `F` captura uma exceção, grava algumas informações de diagnóstico no console, altera a variável de exceção e lança novamente a exceção. A exceção gerada novamente é a exceção original, portanto, a saída produzida é:

```
Console  
  
Exception in F: G  
Exception in Main: G
```

Se o primeiro bloco `catch` tiver sido lançado `e` em vez de relançar a exceção atual, a saída produzida seria a seguinte:

## Console

```
Exception in F: G
Exception in Main: F
```

É um erro de tempo de compilação para uma `break` `continue` instrução, ou `goto` para transferir o controle de um `finally` bloco. Quando uma `break` `continue` instrução,, ou `goto` ocorre em um `finally` bloco, o destino da instrução deve estar dentro do mesmo `finally` bloco ou, caso contrário, ocorrerá um erro de tempo de compilação.

É um erro de tempo de compilação para `return` que uma instrução ocorra em um `finally` bloco.

Uma `try` instrução é executada da seguinte maneira:

- O controle é transferido para o `try` bloco.
- Quando e se o controle atingir o ponto de extremidade do `try` bloco:
  - Se a `try` instrução tiver um `finally` bloco, o `finally` bloco será executado.
  - O controle é transferido para o ponto de extremidade da `try` instrução.
- Se uma exceção for propagada para a `try` instrução durante a execução do `try` bloco:
  - As `catch` cláusulas, se houver, são examinadas em ordem de aparência para localizar um manipulador adequado para a exceção. Se uma `catch` cláusula não especificar um tipo, ou especificar o tipo de exceção ou um tipo base do tipo de exceção:
    - Se a `catch` cláusula declarar uma variável de exceção, o objeto de exceção será atribuído à variável de exceção.
    - Se a `catch` cláusula declarar um filtro de exceção, o filtro será avaliado. Se ele for avaliado como `false`, a cláusula `catch` não será uma correspondência e a pesquisa continuará em todas as `catch` cláusulas subsequentes para um manipulador adequado.
    - Caso contrário, a `catch` cláusula será considerada uma correspondência e o controle será transferido para o `catch` bloco correspondente.
    - Quando e se o controle atingir o ponto de extremidade do `catch` bloco:
      - Se a `try` instrução tiver um `finally` bloco, o `finally` bloco será executado.
      - O controle é transferido para o ponto de extremidade da `try` instrução.
    - Se uma exceção for propagada para a `try` instrução durante a execução do `catch` bloco:

- Se a `try` instrução tiver um `finally` bloco, o `finally` bloco será executado.
- A exceção é propagada para a próxima instrução de circunscrição `try`.
- Se a `try` instrução não tiver `catch` cláusulas ou se nenhuma `catch` cláusula corresponder à exceção:
  - Se a `try` instrução tiver um `finally` bloco, o `finally` bloco será executado.
  - A exceção é propagada para a próxima instrução de circunscrição `try`.

As instruções de um `finally` bloco são sempre executadas quando o controle sai de uma `try` instrução. Isso é verdadeiro se a transferência de controle ocorre como resultado da execução normal, como resultado da execução de uma `break` `continue` instrução,, `goto` ou `return`, ou como resultado da propagação de uma exceção fora da `try` instrução.

Se uma exceção for lançada durante a execução de um `finally` bloco e não for detectada no mesmo bloco `finally`, a exceção será propagada para a próxima instrução de circunscrição `try`. Se outra exceção estava no processo de ser propagada, essa exceção é perdida. O processo de propagação de uma exceção é abordado mais detalhadamente na descrição da `throw` instrução ([a instrução Throw](#)).

O `try` bloco de uma `try` instrução estará acessível se a `try` instrução puder ser acessada.

Um `catch` bloco de uma `try` instrução pode ser acessado se a `try` instrução estiver acessível.

O `finally` bloco de uma `try` instrução estará acessível se a `try` instrução puder ser acessada.

O ponto de extremidade de uma `try` instrução estará acessível se as seguintes opções forem verdadeiras:

- O ponto de extremidade do `try` bloco é acessível ou o ponto de extremidade de pelo menos um `catch` bloco está acessível.
- Se um `finally` bloco estiver presente, o ponto final do `finally` bloco estará acessível.

## As instruções marcadas e desmarcadas

As `checked` `unchecked` instruções e são usadas para controlar o *contexto de verificação de estouro* para operações aritméticas de tipo integral e conversões.

```
antlr
```

```
checked_statement
: 'checked' block
;

unchecked_statement
: 'unchecked' block
;
```

A `checked` instrução faz com que todas as expressões no *bloco* sejam avaliadas em um contexto selecionado, e a `unchecked` instrução faz com que todas as expressões no *bloco* sejam avaliadas em um contexto desmarcado.

As `checked` `unchecked` instruções e são precisamente equivalentes aos `checked` `unchecked` operadores e ([os operadores marcados e desmarcados](#)), exceto que operam em blocos em vez de expressões.

## A instrução lock

A `lock` instrução Obtém o bloqueio de mutual-exclusion para um determinado objeto, executa uma instrução e, em seguida, libera o bloqueio.

```
antlr
```

```
lock_statement
: 'lock' '(' expression ')' embedded_statement
;
```

A expressão de uma `lock` instrução deve indicar um valor de um tipo conhecido como um *reference\_type*. Nenhuma conversão de Boxing implícita ([conversões Boxing](#)) é executada para a expressão de uma `lock` instrução e, portanto, é um erro de tempo de compilação para a expressão denotar um valor de um *value\_type*.

Uma `lock` instrução do formulário

```
C#
```

```
lock (x) ...
```

em que `x` é uma expressão de uma *reference\_type*, é precisamente equivalente a

```
C#
```

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

exceto que `x` é avaliado apenas uma vez.

Enquanto um bloqueio de exclusão mútua é mantido, o código em execução no mesmo thread de execução também pode obter e liberar o bloqueio. No entanto, o código em execução em outros threads é impedido de obter o bloqueio até que o bloqueio seja liberado.

O bloqueio `System.Type` de objetos para sincronizar o acesso a dados estáticos não é recomendado. Outro código pode bloquear no mesmo tipo, o que pode resultar em deadlock. Uma abordagem melhor é sincronizar o acesso a dados estáticos bloqueando um objeto estático privado. Por exemplo:

C#

```
class Cache
{
    private static readonly object synchronizationObject = new object();

    public static void Add(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }

    public static void Remove(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }
}
```

## A instrução using

A `using` instrução Obtém um ou mais recursos, executa uma instrução e, em seguida, descarta o recurso.

```
using_statement
: 'using' '(' resource_acquisition ')' embedded_statement
;

resource_acquisition
: local_variable_declaration
| expression
;
```

Um **recurso** é uma classe ou struct que implementa `System.IDisposable`, que inclui um único método sem parâmetros chamado `Dispose`. O código que está usando um recurso pode chamar `Dispose` para indicar que o recurso não é mais necessário. Se `Dispose` não for chamado, a eliminação automática eventualmente ocorrerá como consequência da coleta de lixo.

Se a forma de `resource_acquisition` for `local_variable_declaration`, o tipo de `local_variable_declaration` deverá ser `dynamic` ou um tipo que possa ser convertido implicitamente em `System.IDisposable`. Se a forma de `resource_acquisition` for `expression`, essa expressão deverá ser conversível implicitamente `System.IDisposable`.

Variáveis locais declaradas em uma `resource_acquisition` são somente leitura e devem incluir um inicializador. Ocorrerá um erro de tempo de compilação se a instrução inserida tentar modificar essas variáveis locais (por meio de atribuição ou os `++` `--` operadores e), pegar o endereço delas ou passá-las como `ref` `out` parâmetros ou.

Uma `using` instrução é convertida em três partes: aquisição, uso e alienação. O uso do recurso é implicitamente incluído em uma `try` instrução que inclui uma `finally` cláusula. Essa `finally` cláusula descarta o recurso. Se um `null` recurso for adquirido, nenhuma chamada para `Dispose` será feita e nenhuma exceção será gerada. Se o recurso for do tipo `dynamic`, ele será convertido dinamicamente por meio de uma conversão dinâmica implícita ([conversões dinâmicas implícitas](#)) `IDisposable` durante a aquisição para garantir que a conversão seja bem-sucedida antes do uso e da alienação.

Uma `using` instrução do formulário

C#

```
using (ResourceType resource = expression) statement
```

corresponde a uma das três expansões possíveis. Quando `ResourceType` é um tipo de valor não anulável, a expansão é

C#

```
{  
    ResourceType resource = expression;  
    try {  
        statement;  
    }  
    finally {  
        ((IDisposable)resource).Dispose();  
    }  
}
```

Caso contrário, quando `ResourceType` for um tipo de valor anulável ou um tipo de referência diferente de `dynamic`, a expansão será

```
C#  
  
{  
    ResourceType resource = expression;  
    try {  
        statement;  
    }  
    finally {  
        if (resource != null) ((IDisposable)resource).Dispose();  
    }  
}
```

Caso contrário, quando `ResourceType` for `dynamic`, a expansão será

```
C#  
  
{  
    ResourceType resource = expression;  
    IDisposable d = (IDisposable)resource;  
    try {  
        statement;  
    }  
    finally {  
        if (d != null) d.Dispose();  
    }  
}
```

Em qualquer expansão, a `resource` variável é somente leitura na instrução incorporada, e a `d` variável é inacessível em, e invisível para, a instrução inserida.

Uma implementação tem permissão para implementar uma determinada instrução `using` de forma diferente, por exemplo, por motivos de desempenho, desde que o comportamento seja consistente com a expansão acima.

Uma `using` instrução do formulário

```
C#
```

```
using (expression) statement
```

tem as mesmas três expansões possíveis. Nesse caso `ResourceType`, é implicitamente o tipo de tempo de compilação do `expression`, se tiver um. Caso contrário, a `IDisposable` própria interface será usada como o `ResourceType`. A `resource` variável é inacessível em, e invisível para, a instrução inserida.

Quando um `resource_acquisition` usa a forma de um `local_variable_declaration`, é possível adquirir vários recursos de um determinado tipo. Uma `using` instrução do formulário

```
C#
```

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

é precisamente equivalente a uma sequência de instruções aninhadas `using`:

```
C#
```

```
using (ResourceType r1 = e1)
    using (ResourceType r2 = e2)
    ...
    using (ResourceType rN = eN)
        statement
```

O exemplo a seguir cria um arquivo chamado `log.txt` e grava duas linhas de texto no arquivo. Em seguida, o exemplo abre o mesmo arquivo para ler e copiar as linhas de texto contidas para o console.

```
C#
```

```
using System;
using System.IO;

class Test
{
    static void Main() {
        using (TextWriter w = File.CreateText("log.txt")) {
            w.WriteLine("This is line one");
            w.WriteLine("This is line two");
        }

        using (TextReader r = File.OpenText("log.txt")) {
```

```

        string s;
        while ((s = r.ReadLine()) != null) {
            Console.WriteLine(s);
        }
    }
}

```

Como as `TextWriter` `TextReader` classes e implementam a `IDisposable` interface, o exemplo pode usar `using` instruções para garantir que o arquivo subjacente seja fechado corretamente seguindo as operações de gravação ou leitura.

## A instrução `yield`

A `yield` instrução é usada em um bloco do iterador ([blocos](#)) para gerar um valor para o objeto do enumerador ([objetos do enumerador](#)) ou objeto enumerável ([objetos enumeráveis](#)) de um iterador ou para sinalizar o final da iteração.

```

antlr

yield_statement
: 'yield' 'return' expression ';'
| 'yield' 'break' ';'
;
```

`yield` Não é uma palavra reservada; Ele tem um significado especial somente quando usado imediatamente antes de uma `return` `break` palavra-chave ou. Em outros contextos, `yield` pode ser usado como um identificador.

Há várias restrições sobre onde uma `yield` instrução pode aparecer, conforme descrito a seguir.

- É um erro de tempo de compilação para uma `yield` instrução (de qualquer forma) aparecer fora de um `method_body`, `operator_body` ou `accessor_body`
- É um erro de tempo de compilação para uma `yield` instrução (de qualquer forma) aparecer dentro de uma função anônima.
- É um erro de tempo de compilação para uma `yield` instrução (de qualquer forma) aparecer na `finally` cláusula de uma `try` instrução.
- É um erro de tempo de compilação para uma `yield return` instrução aparecer em qualquer lugar em uma `try` instrução que contenha qualquer `catch` cláusula.

O exemplo a seguir mostra alguns usos válidos e inválidos de `yield` instruções.

C#

```
delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {
    try {
        yield return 1;          // Ok
        yield break;            // Ok
    }
    finally {
        yield return 2;          // Error, yield in finally
        yield break;            // Error, yield in finally
    }

    try {
        yield return 3;          // Error, yield return in try...catch
        yield break;            // Ok
    }
    catch {
        yield return 4;          // Error, yield return in try...catch
        yield break;            // Ok
    }

    D d = delegate {
        yield return 5;          // Error, yield in an anonymous function
    };
}

int MyMethod() {
    yield return 1;            // Error, wrong return type for an iterator
    block
}
```

Uma conversão implícita ([conversões implícitas](#)) deve existir do tipo da expressão na `yield return` instrução para o tipo `yield` ([tipo yield](#)) do iterador.

Uma `yield return` instrução é executada da seguinte maneira:

- A expressão fornecida na instrução é avaliada, implicitamente convertida para o tipo `yield` e atribuída à `Current` Propriedade do objeto `Enumerator`.
- A execução do bloco do iterador está suspensa. Se a `yield return` instrução estiver dentro de um ou mais `try` blocos, os `finally` blocos associados não serão executados neste momento.
- O `MoveNext` método do objeto enumerador retorna `true` para seu chamador, indicando que o objeto enumerador foi avançado com êxito para o próximo item.

A próxima chamada para o método do objeto enumerador `MoveNext` retoma a execução do bloco iterador de onde ele foi suspenso pela última vez.

Uma `yield break` instrução é executada da seguinte maneira:

- Se a `yield break` instrução estiver contida em um ou mais `try` blocos com `finally` blocos associados, o controle será transferido inicialmente para o `finally` bloco da instrução mais interna `try`. Quando e se o controle atingir o ponto de extremidade de um `finally` bloco, o controle será transferido para o `finally` bloco da próxima instrução de circunscrição `try`. Esse processo é repetido até que os `finally` blocos de todas as instruções de circunscrição `try` tenham sido executados.
- O controle é retornado para o chamador do bloco do iterador. Esse é o `MoveNext` método ou `Dispose` método do objeto enumerador.

Como uma `yield break` instrução transfere incondicionalmente o controle em outro lugar, o ponto de extremidade de uma `yield break` instrução nunca é acessível.

# Namespaces

Artigo • 16/09/2021

Os programas em C# são organizados usando namespaces. Os namespaces são usados como um sistema de organização "interno" para um programa e como um sistema de organização "externo" — uma maneira de apresentar os elementos do programa que são expostos a outros programas.

O uso de diretivas ([usando diretivas](#)) é fornecido para facilitar o uso de namespaces.

## Unidades de compilação

Uma *compilation\_unit* define a estrutura geral de um arquivo de origem. Uma unidade de compilação consiste em zero ou mais *using\_directive*s seguido por zero ou mais *global\_attributes* seguido por zero ou mais *namespace\_member\_declaration*s.

antlr

```
compilation_unit
    : extern_alias_directive* using_directive* global_attributes?
      namespace_member_declarati
        ;
```

Um programa C# consiste em uma ou mais unidades de compilação, cada uma contida em um arquivo de origem separado. Quando um programa em C# é compilado, todas as unidades de compilação são processadas em conjunto. Assim, as unidades de compilação podem depender umas das outras, possivelmente de maneira circular.

As *using\_directive*s de uma unidade de compilação afetam o *global\_attributes* e *namespace\_member\_declaration*s dessa unidade de compilação, mas não têm nenhum efeito em outras unidades de compilação.

O *global\_attributes* ([atributos](#)) de uma unidade de compilação permite a especificação de atributos para o assembly de destino e o módulo. Assemblies e módulos atuam como contêineres físicos para tipos. Um assembly pode consistir em vários módulos fisicamente separados.

As *namespace\_member\_declaration*s de cada unidade de compilação de um programa contribuem com membros para um único espaço de declaração chamado namespace global. Por exemplo:

Arquivo A.cs :

```
C#
```

```
class A {}
```

Arquivo `B.cs` :

```
C#
```

```
class B {}
```

As duas unidades de compilação contribuem para o namespace global único, neste caso, declarando duas classes com os nomes totalmente qualificados `A` e `B`. Como as duas unidades de compilação contribuem para o mesmo espaço de declaração, seria um erro se cada uma contivesse uma declaração de um membro com o mesmo nome.

## Declarações de namespace

Uma *namespace\_declaration* consiste na palavra-chave `namespace`, seguida por um nome de namespace e corpo, opcionalmente seguido por um ponto-e-vírgula.

```
antlr
```

```
namespace_declaration
    : 'namespace' qualified_identifier namespace_body ';'?
    ;

qualified_identifier
    : identifier ('.' identifier)*
    ;

namespace_body
    : '{' extern_alias_directive* using_directive*
    namespace_member_declaration* '}'
    ;
```

Um *namespace\_declaration* pode ocorrer como uma declaração de nível superior em uma *compilation\_unit* ou como uma declaração de membro dentro de outro *namespace\_declaration*. Quando um *namespace\_declaration* ocorre como uma declaração de nível superior em uma *compilation\_unit*, o namespace se torna um membro do namespace global. Quando um *namespace\_declaration* ocorre dentro de outro *namespace\_declaration*, o namespace interno se torna um membro do namespace externo. Em ambos os casos, o nome de um namespace deve ser exclusivo dentro do namespace que o contém.

Os namespaces são implicitamente `public` e a declaração de um namespace não pode incluir nenhum modificador de acesso.

Em um *namespace\_body*, os *using\_directive* opcionais importam os nomes de outros namespaces, tipos e membros, permitindo que eles sejam referenciados diretamente, em vez de nomes qualificados. Os *namespace\_member\_declaration*s opcionais contribuem com membros para o espaço de declaração do namespace. Observe que todos os *using\_directive*s devem aparecer antes de qualquer declaração de membro.

A *qualified\_identifier* de um *namespace\_declaration* pode ser um único identificador ou uma sequência de identificadores separados por "`.`" tokens. O último formulário permite que um programa defina um namespace aninhado sem aninhar lexicalmente várias declarações de namespace. Por exemplo,

```
C#  
  
namespace N1.N2  
{  
    class A {}  
  
    class B {}  
}
```

é semanticamente equivalente a

```
C#  
  
namespace N1  
{  
    namespace N2  
    {  
        class A {}  
  
        class B {}  
    }  
}
```

Os namespaces são abertos e duas declarações de namespace com o mesmo nome totalmente qualificado contribuem para o mesmo espaço de declaração ([declarações](#)). No exemplo

```
C#  
  
namespace N1.N2  
{  
    class A {}  
}
```

```
namespace N1.N2
{
    class B {}
}
```

as duas declarações de namespace acima contribuem para o mesmo espaço de declaração, neste caso, declarando duas classes com os nomes totalmente qualificados `N1.N2.A` e `N1.N2.B`. Como as duas declarações contribuem para o mesmo espaço de declaração, seria um erro se cada uma contivesse uma declaração de um membro com o mesmo nome.

## Aliases extern

Um `extern_alias_directive` introduz um identificador que serve como um alias para um namespace. A especificação do namespace com alias é externa ao código-fonte do programa e se aplica também a namespaces aninhados do namespace com alias.

antlr

```
extern_alias_directive
: 'extern' 'alias' identifier ';'
```

O escopo de uma `extern_alias_directive` estende-se pelas `using_directive`s, `global_attributes` e `namespace_member_declaration`s de sua própria unidade de compilação ou corpo de namespace.

Em uma unidade de compilação ou corpo de namespace que contém um `extern_alias_directive`, o identificador introduzido pelo `extern_alias_directive` pode ser usado para referenciar o namespace com alias. É um erro de tempo de compilação para o `identificador` ser a palavra `global`.

Um `extern_alias_directive` disponibiliza um alias em um corpo de namespace ou unidade de compilação em particular, mas não contribui com novos membros para o espaço de declaração subjacente. Em outras palavras, um `extern_alias_directive` não é transitiva, mas, em vez disso, afeta apenas a unidade de compilação ou o corpo do namespace no qual ele ocorre.

O programa a seguir declara e usa dois aliases extern, `X` e `Y` cada um deles representa a raiz de uma hierarquia de namespace distinta:

C#

```
extern alias X;
extern alias Y;

class Test
{
    X::N.A a;
    X::N.B b1;
    Y::N.B b2;
    Y::N.C c;
}
```

O programa declara a existência dos aliases extern `X` e `Y`, mas as definições reais dos aliases são externas ao programa. As classes nomeadas de `N.B` forma idêntica agora podem ser referenciadas como `X.N.B` e `Y.N.B`, ou usando o qualificador de alias de namespace `X::N.B` e `Y::N.B`. Ocorrerá um erro se um programa declarar um alias externo para o qual nenhuma definição externa é fornecida.

## Uso de diretivas

\*O uso de diretivas \_ facilita o uso de namespaces e tipos definidos em outros namespaces. O uso de diretivas afeta o processo de resolução de nomes de `_namespace_or_type_name` \* s ([namespace e nomes de tipo](#)) e `Simple_name` s ([nomes simples](#)), mas ao contrário de declarações, o uso de diretivas não contribuirá com novos membros para os espaços de declaração subjacentes das unidades de compilação ou namespaces nos quais eles são usados.

antlr

```
using_directive
  : using_alias_directive
  | using_namespace_directive
  | using_static_directive
  ;
```

Um `using_alias_directive` ([usando diretivas de alias](#)) apresenta um alias para um namespace ou tipo.

Um `using_namespace_directive` ([usando as diretivas de namespace](#)) importa os membros de tipo de um namespace.

Um `using_static_directive` ([usando diretivas estáticas](#)) importa os tipos aninhados e os membros estáticos de um tipo.

O escopo de uma `using_directive` estende-se pela `namespace_member_declaration` s de seu conteúdo de unidade de compilação ou corpo de namespace imediatamente

contido. O escopo de um *using directive* especificamente não inclui seu par *using directive*s. Assim, os pontos *using directive*s não afetam uns aos outros, e a ordem na qual eles são gravados é insignificante.

## Usando diretivas de alias

Um *using alias directive* introduz um identificador que serve como um alias para um namespace ou tipo dentro da unidade de compilação ou do corpo de namespace imediatamente delimitados.

antlr

```
using_alias_directive
: 'using' identifier '=' namespace_or_type_name ';'
;
```

Em declarações de membro em uma unidade de compilação ou corpo de namespace que contém um *using alias directive*, o identificador introduzido pelo *using alias directive* pode ser usado para referenciar o namespace ou tipo fornecido. Por exemplo:

C#

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;

    class B: A {}
}
```

Acima, em declarações de membro no `N3` namespace, `A` é um alias para `N1.N2.A` e, portanto, a classe `N3.B` deriva da classe `N1.N2.A`. O mesmo efeito pode ser obtido criando um alias `R` para `N1.N2` e, em seguida, fazendo referência a `R.A`:

C#

```
namespace N3
{
    using R = N1.N2;
```

```
    class B: R.A {}
}
```

O *identificador* de um *using\_alias\_directive* deve ser exclusivo dentro do espaço de declaração da unidade de compilação ou do namespace que contém imediatamente o *using\_alias\_directive*. Por exemplo:

```
C#  
  
namespace N3  
{  
    class A {}  
}  
  
namespace N3  
{  
    using A = N1.N2.A;           // Error, A already exists  
}
```

Acima, `N3` já contém um membro `A`, portanto, é um erro de tempo de compilação para um *using\_alias\_directive* usar esse identificador. Da mesma forma, é um erro de tempo de compilação para dois ou mais *using\_alias\_directive*s na mesma unidade de compilação ou corpo de namespace para declarar aliases com o mesmo nome.

Um *using\_alias\_directive* disponibiliza um alias em um corpo de namespace ou unidade de compilação em particular, mas não contribui com novos membros para o espaço de declaração subjacente. Em outras palavras, uma *using\_alias\_directive* não é transitiva, mas afeta apenas a unidade de compilação ou o corpo do namespace no qual ela ocorre. No exemplo

```
C#  
  
namespace N3  
{  
    using R = N1.N2;  
}  
  
namespace N3  
{  
    class B: R.A {}           // Error, R unknown  
}
```

o escopo do *using\_alias\_directive* que introduz `R` somente se estende a declarações de membro no corpo do namespace no qual ele está contido, portanto, `R` é desconhecido na segunda declaração de namespace. No entanto, colocar o *using\_alias\_directive* na

unidade de compilação que o contém faz com que o alias fique disponível nas duas declarações de namespace:

```
C#  
  
using R = N1.N2;  
  
namespace N3  
{  
    class B: R.A {}  
}  
  
namespace N3  
{  
    class C: R.A {}  
}
```

Assim como os membros regulares, os nomes apresentados por *using\_alias\_directive*s são ocultados por membros nomeados de forma semelhante em escopos aninhados. No exemplo

```
C#  
  
using R = N1.N2;  
  
namespace N3  
{  
    class R {}  
  
    class B: R.A {}      // Error, R has no member A  
}
```

a referência a `R.A` na declaração de `B` causa um erro de tempo de compilação porque `R` refere-se a `N3.R`, não `N1.N2`.

A ordem na qual os *using\_alias\_directive*s são gravados não tem significado, e a resolução do *namespace\_or\_type\_name* referenciado por um *using\_alias\_directive* não é afetada pelo *using\_alias\_directive* em si ou por outros *using\_directive*s no corpo da unidade de compilação ou do namespace imediatamente contido. Em outras palavras, a *namespace\_or\_type\_name* de um *using\_alias\_directive* é resolvida como se o corpo da unidade de compilação ou do namespace imediatamente contido não tivesse *using\_directive*s. No entanto, um *using\_alias\_directive* pode ser afetado por *extern\_alias\_directive*s no corpo da unidade de compilação ou do namespace imediatamente contido. No exemplo

```
C#
```

```

namespace N1.N2 {}

namespace N3
{
    extern alias E;

    using R1 = E.N;           // OK

    using R2 = N1;            // OK

    using R3 = N1.N2;          // OK

    using R4 = R2.N2;          // Error, R2 unknown
}

```

a última *using\_alias\_directive* resulta em um erro de tempo de compilação porque ele não é afetado pela primeira *using\_alias\_directive*. A primeira *using\_alias\_directive* não resulta em um erro, pois o escopo do alias externo `E` inclui o *using\_alias\_directive*.

Um *using\_alias\_directive* pode criar um alias para qualquer namespace ou tipo, incluindo o namespace no qual ele aparece e qualquer namespace ou tipo aninhado dentro desse namespace.

O acesso a um namespace ou tipo por meio de um alias produz exatamente o mesmo resultado que o acesso a esse namespace ou tipo por meio de seu nome declarado. Por exemplo, dada

```

C#

namespace N1.N2
{
    class A {}
}

namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;

    class B
    {
        N1.N2.A a;           // refers to N1.N2.A
        R1.N2.A b;           // refers to N1.N2.A
        R2.A c;              // refers to N1.N2.A
    }
}

```

os nomes `N1.N2.A`, `R1.N2.A`, e `R2.A` são equivalentes e todos referem-se à classe cujo nome totalmente qualificado é `N1.N2.A`.

O uso de aliases pode nomear um tipo construído fechado, mas não pode nomear uma declaração de tipo genérico não associado sem fornecer argumentos de tipo. Por exemplo:

```
C#  
  
namespace N1  
{  
    class A<T>  
    {  
        class B {}  
    }  
}  
  
namespace N2  
{  
    using W = N1.A;           // Error, cannot name unbound generic type  
  
    using X = N1.A.B;         // Error, cannot name unbound generic type  
  
    using Y = N1.A<int>;     // Ok, can name closed constructed type  
  
    using Z<T> = N1.A<T>;   // Error, using alias cannot have type  
parameters  
}
```

## Usando diretivas de namespace

Um *using\_namespace\_directive* importa os tipos contidos em um namespace para a unidade de compilação ou o corpo de namespace imediatamente delimitadores, habilitando o identificador de cada tipo a ser usado sem qualificação.

```
antlr  
  
using_namespace_directive  
: 'using' namespace_name ';' ;
```

Dentro de declarações de membro em uma unidade de compilação ou corpo de namespace que contém um *using\_namespace\_directive*, os tipos contidos no namespace fornecido podem ser referenciados diretamente. Por exemplo:

```
C#  
  
namespace N1.N2  
{  
    class A {}
```

```
}
```

```
namespace N3
{
    using N1.N2;

    class B: A {}
}
```

Acima, em declarações de membro no `N3` namespace, os membros de tipo de `N1.N2` estão diretamente disponíveis e, portanto, a classe `N3.B` deriva da classe `N1.N2.A`.

Um *using\_namespace\_directive* importa os tipos contidos no namespace fornecido, mas especificamente não importa namespaces aninhados. No exemplo

```
C#
```

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1;

    class B: N2.A {}           // Error, N2 unknown
}
```

o *using\_namespace\_directive* importa os tipos contidos em `N1`, mas não os namespaces aninhados em `N1`. Portanto, a referência a `N2.A` na declaração de `B` resulta em um erro de tempo de compilação porque nenhum membro nomeado `N2` está no escopo.

Ao contrário de um *using\_alias\_directive*, um *using\_namespace\_directive* pode importar tipos cujos identificadores já estão definidos dentro do corpo da unidade de compilação ou do namespace do delimitador. Na verdade, os nomes importados por um *using\_namespace\_directive* são ocultos por membros nomeados de forma semelhante no corpo da unidade de compilação ou do namespace de circunscrição. Por exemplo:

```
C#
```

```
namespace N1.N2
{
    class A {}

    class B {}
}
```

```
namespace N3
{
    using N1.N2;

    class A {}
}
```

Aqui, em declarações de membro no `N3` namespace, `A` refere-se ao `N3.A` em vez de `N1.N2.A`.

Quando mais de um namespace ou tipo importado por *using\_namespace\_directive*s ou *using\_static\_directive*s na mesma unidade de compilação ou corpo de namespace contiver tipos com o mesmo nome, as referências a esse nome como um *type\_name* serão consideradas ambíguas. No exemplo

```
C#

namespace N1
{
    class A {}
}

namespace N2
{
    class A {}
}

namespace N3
{
    using N1;

    using N2;

    class B: A           // Error, A is ambiguous
}
```

`N1` e `N2` contêm um membro `A`, e como `N3` as importações, a referência a `A` `N3` é um erro de tempo de compilação. Nessa situação, o conflito pode ser resolvido por meio da qualificação de referências para `A` ou introduzindo um *using\_alias\_directive* que escolha um específico `A`. Por exemplo:

```
C#

namespace N3
{
    using N1;

    using N2;
```

```

using A = N1.A;

class B: A           // A means N1.A
{

```

Além disso, quando mais de um namespace ou tipo importado por *using\_namespace\_directive*s ou *using\_static\_directive*s na mesma unidade de compilação ou corpo de namespace contém tipos ou membros com o mesmo nome, as referências a esse nome como um *Simple\_name* são consideradas ambíguas. No exemplo

C#

```

namespace N1
{
    class A {}
}

class C
{
    public static int A;
}

namespace N2
{
    using N1;
    using static C;

    class B
    {
        void M()
        {
            A a = new A();    // Ok, A is unambiguous as a type-name
            A.Equals(2);     // Error, A is ambiguous as a simple-name
        }
    }
}

```

`N1` contém um membro de tipo `A` e `C` contém um campo estático `A` e, como `N2` as importações, as referenciadas `A` como uma *Simple\_name* são ambíguas e um erro de tempo de compilação.

Como um *using\_alias\_directive*, um *using\_namespace\_directive* não contribui com novos membros para o espaço de declaração subjacente da unidade de compilação ou do namespace, mas, em vez disso, afeta apenas a unidade de compilação ou o corpo do namespace no qual ele aparece.

A *Namespace\_Name* referenciada por uma *using\_namespace\_directive* é resolvida da mesma maneira que a *namespace\_or\_type\_name* referenciada por um *using\_alias\_directive*. Portanto, *using\_namespace\_directive*s na mesma unidade de compilação ou corpo de namespace não afetam um ao outro e podem ser gravados em qualquer ordem.

## Usando diretivas estáticas

Um *using\_static\_directive* importa os tipos aninhados e os membros estáticos contidos diretamente em uma declaração de tipo para o corpo de namespace ou unidade de compilação imediatamente delimitadora, habilitando o identificador de cada membro e tipo a ser usado sem qualificação.

```
antlr
```

```
using_static_directive
: 'using' 'static' type_name ';'
;
```

Dentro de declarações de membro em uma unidade de compilação ou corpo de namespace que contém um *using\_static\_directive*, os tipos aninhados e membros estáticos acessíveis (exceto métodos de extensão) contidos diretamente na declaração do tipo fornecido podem ser referenciados diretamente. Por exemplo:

```
C#
```

```
namespace N1
{
    class A
    {
        public class B{}
        public static B M(){ return new B(); }
    }
}

namespace N2
{
    using static N1.A;
    class C
    {
        void N() { B b = M(); }
    }
}
```

Acima, dentro de declarações de membro no `N2` namespace, os membros estáticos e os tipos aninhados do `N1.A` estão diretamente disponíveis e, portanto, o método `N` é

capaz de referenciar tanto o `B` quanto os `M` membros de `N1.A`.

Um `using_static_directive` especificamente não importa métodos de extensão diretamente como métodos estáticos, mas os torna disponíveis para invocação de método de extensão ([invocações de método de extensão](#)). No exemplo

C#

```
namespace N1
{
    static class A
    {
        public static void M(this string s){}
    }
}

namespace N2
{
    using static N1.A;

    class B
    {
        void N()
        {
            M("A");          // Error, M unknown
            "B".M();         // Ok, M known as extension method
            N1.A.M("C");   // Ok, fully qualified
        }
    }
}
```

o `using_static_directive` importa o método de extensão `M` contido em `N1.A`, mas somente como um método de extensão. Portanto, a primeira referência a `M` no corpo de `B.N` resultados em um erro de tempo de compilação porque nenhum membro nomeado `M` está no escopo.

Uma `using_static_directive` importa somente membros e tipos declarados diretamente no tipo determinado, não membros e tipos declarados em classes base.

TODO: exemplo

As ambiguidades entre vários `using_namespace_directives` e `using_static_directives` são discutidas em [usando diretivas de namespace](#).

## Membros de namespace

Uma *namespace\_member\_declaration* é uma *namespace\_declaration* ([declarações de namespace](#)) ou uma *type\_declaration* ([declarações de tipo](#)).

antlr

```
namespace_member_declaration
: namespace_declaraction
| type_declaraction
;
```

Uma unidade de compilação ou um corpo de namespace pode conter *namespace\_member\_declaration*s e essas declarações contribuem com novos membros para o espaço de declaração subjacente da unidade de compilação ou do corpo do namespace que a contém.

## Declarações de tipo

Uma *type\_declaration* é uma *class\_declaration* ([declarações de classe](#)), uma *struct\_declaration* ([declarações struct](#)), uma *interface\_declaration* ([declarações de interface](#)), um *enum\_declaration* ([declarações enum](#)) ou uma *delegate\_declaration* ([declarações delegate](#)).

antlr

```
type_declaration
: class_declaraction
| struct_declaraction
| interface_declaraction
| enum_declaraction
| delegate_declaraction
;
```

Um *type\_declaration* pode ocorrer como uma declaração de nível superior em uma unidade de compilação ou como uma declaração de membro em um namespace, classe ou struct.

Quando uma declaração de tipo para um tipo *T* ocorre como uma declaração de nível superior em uma unidade de compilação, o nome totalmente qualificado do tipo declarado recentemente é simplesmente *T*. Quando uma declaração de tipo para um tipo *T* ocorre em um namespace, classe ou struct, o nome totalmente qualificado do tipo declarado recentemente é *N.T*, em que *N* é o nome totalmente qualificado do namespace, da classe ou do struct que o contém.

Um tipo declarado dentro de uma classe ou struct é chamado de tipo aninhado ([tipos aninhados](#)).

Os modificadores de acesso permitidos e o acesso padrão para uma declaração de tipo dependem do contexto no qual a declaração ocorre ([acessibilidade declarada](#)):

- Tipos declarados em unidades de compilação ou namespaces podem ter `public` ou `internal` acessar. O padrão é o `internal` acesso.
- Tipos declarados em classes podem ter `public`, `protected internal`, `protected`, `internal` ou `private` acesso. O padrão é o `private` acesso.
- Os tipos declarados em structs podem ter `public`, `internal` ou `private` Access. O padrão é o `private` acesso.

## Qualificadores alias de namespace

O *qualificador de alias de namespace* `::` torna possível garantir que as pesquisas de nome de tipo não sejam afetadas pela introdução de novos tipos e membros. O qualificador de alias de namespace sempre aparece entre dois identificadores referenciados como os identificadores à esquerda e à direita. Ao contrário do `.` qualificador regular, o identificador de lado esquerdo do `::` qualificador é pesquisado somente como um alias externo ou de uso.

Uma *qualified\_alias\_member* é definida da seguinte maneira:

```
antlr

qualified_alias_member
    : identifier '::' identifier type_argument_list?
    ;
```

Um *qualified\_alias\_member* pode ser usado como um *namespace\_or\_type\_name* ([namespace e nomes de tipo](#)) ou como o operando esquerdo em um *member\_access* ([acesso de membro](#)).

Uma *qualified\_alias\_member* tem uma das duas formas:

- `N::I<A1, ..., Ak>`, onde `N` e `I` representam identificadores e `<A1, ..., Ak>` é uma lista de argumentos de tipo. (`K` sempre é pelo menos uma.)
- `N::I`, onde `N` e `I` representam identificadores. (Nesse caso, `K` é considerado zero).

Usando essa notação, o significado de um *qualified\_alias\_member* é determinado da seguinte maneira:

- Se `N` for o identificador `global`, o namespace global será procurado `I`:
  - Se o namespace global contiver um namespace chamado `I` e `K` for zero, o *qualified\_alias\_member* se referirá a esse namespace.
  - Caso contrário, se o namespace global contiver um tipo não genérico chamado `I` e `K` for zero, o *qualified\_alias\_member* se referirá a esse tipo.
  - Caso contrário, se o namespace global contiver um tipo chamado `I` que tem `K` parâmetros de tipo, a *qualified\_alias\_member* se referirá a esse tipo construído com os argumentos de tipo fornecidos.
  - Caso contrário, o *qualified\_alias\_member* será indefinido e ocorrerá um erro em tempo de compilação.
- Caso contrário, começando com a declaração de namespace ([declarações de namespace](#)) imediatamente contendo a *qualified\_alias\_member* (se houver), continuando com cada declaração de namespace delimitadora (se houver) e terminando com a unidade de compilação que contém o *qualified\_alias\_member*, as etapas a seguir são avaliadas até que uma entidade seja localizada:
  - Se a declaração de namespace ou a unidade de compilação contiver um *using\_alias\_directive* que associa `N` um tipo, a *qualified\_alias\_member* será indefinida e ocorrerá um erro em tempo de compilação.
  - Caso contrário, se a declaração de namespace ou a unidade de compilação contiver um *extern\_alias\_directive* ou *using\_alias\_directive* que associe `N` com um namespace, então:
    - Se o namespace associado a `N` tiver um namespace chamado `I` e `K` for zero, o *qualified\_alias\_member* se referirá a esse namespace.
    - Caso contrário, se o namespace associado a `N` contiver um tipo não genérico chamado `I` e `K` for zero, o *qualified\_alias\_member* se referirá a esse tipo.
    - Caso contrário, se o namespace associado a `N` contiver um tipo chamado `I` que tem `K` parâmetros de tipo, a *qualified\_alias\_member* se referirá a esse tipo construído com os argumentos de tipo fornecidos.
    - Caso contrário, o *qualified\_alias\_member* será indefinido e ocorrerá um erro em tempo de compilação.
- Caso contrário, o *qualified\_alias\_member* será indefinido e ocorrerá um erro em tempo de compilação.

Observe que o uso do qualificador de alias de namespace com um alias que faz referência a um tipo causa um erro de tempo de compilação. Observe também que, se o identificador `N` for `global`, a pesquisa será executada no namespace global, mesmo que haja um alias de uso associado a `global` um tipo ou namespace.

## Exclusividade de aliases

Cada unidade de compilação e corpo de namespace tem um espaço de declaração separado para aliases extern e o uso de aliases. Portanto, embora o nome de um alias externo ou o uso de alias deva ser exclusivo dentro do conjunto de aliases extern e uso de aliases declarados no corpo da unidade de compilação ou do namespace imediatamente contido, um alias tem permissão para ter o mesmo nome que um tipo ou namespace, desde que ele seja usado somente com o `::` qualificador.

No exemplo

```
C#  
  
namespace N  
{  
    public class A {}  
  
    public class B {}  
}  
  
namespace N  
{  
    using A = System.IO;  
  
    class X  
    {  
        A.Stream s1;           // Error, A is ambiguous  
  
        A::Stream s2;         // Ok  
    }  
}
```

o nome `A` tem dois significados possíveis no segundo corpo de namespace porque a classe `A` e o alias de uso `A` estão no escopo. Por esse motivo, o uso de `A` no nome qualificado `A.Stream` é ambíguo e causa a ocorrência de um erro de tempo de compilação. No entanto, o uso de com o `::` qualificador não é um erro porque `A` é pesquisado somente como um alias de namespace.

# Classes

Artigo • 16/09/2021

Uma classe é uma estrutura de dados que pode conter membros de dados (constantes e campos), membros de função (métodos, propriedades, eventos, indexadores, operadores, construtores de instância, destruidores e construtores estáticos) e tipos aninhados. Os tipos de classe dão suporte à herança, um mecanismo no qual uma classe derivada pode estender e especializar uma classe base.

## Declarações de classe

Uma *class\_declaration* é uma *type\_declaration* ([declarações de tipo](#)) que declara uma nova classe.

antlr

```
class_declaration
    : attributes? class_modifier* 'partial'? 'class' identifier
      type_parameter_list?
        class_base? type_parameter_constraints_clause* class_body ';'?
```

Um *class\_declaration* consiste em um conjunto opcional de *atributos* ([atributos](#)), seguido por um conjunto opcional de *class\_modifier*s ([modificadores de classe](#)), seguido por um `partial` modificador opcional, seguido pela palavra-chave `class` e um *identificador* que nomeia a classe, seguido por um *type\_parameter\_list* opcional ([parâmetros de tipo](#)), seguido por uma especificação de *class\_base* opcional (especificação de [base de classe](#)), seguido por um conjunto opcional de *type\_parameter\_constraints\_clause*s (restrições de [parâmetro de tipo](#)), seguido por um *class\_body* ([corpo de classe](#)), opcionalmente seguido por um ponto e vírgula

Uma declaração de classe não pode fornecer *type\_parameter\_constraints\_clause*s, a menos que ele também forneça um *type\_parameter\_list*.

Uma declaração de classe que fornece uma *type\_parameter\_list* é uma **declaração de classe genérica**. Além disso, qualquer classe aninhada dentro de uma declaração de classe genérica ou uma declaração struct genérica é, em si, uma declaração de classe genérica, já que parâmetros de tipo para o tipo recipiente devem ser fornecidos para criar um tipo construído.

## Modificadores de classe

Um *class\_declaration* pode, opcionalmente, incluir uma sequência de modificadores de classe:

```
antlr

class_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'abstract'
| 'sealed'
| 'static'
| class_modifier_unsafe
;
```

É um erro de tempo de compilação para o mesmo modificador aparecer várias vezes em uma declaração de classe.

O `new` modificador é permitido em classes aninhadas. Ele especifica que a classe oculta um membro herdado com o mesmo nome, conforme descrito no [novo modificador](#). É um erro de tempo de compilação para `new` que o modificador apareça em uma declaração de classe que não seja uma declaração de classe aninhada.

Os `public` `protected` `internal` modificadores,, e `private` controlam a acessibilidade da classe. Dependendo do contexto no qual a declaração de classe ocorre, alguns desses modificadores podem não ser permitidos ([acessibilidade declarada](#)).

Os `abstract` `sealed` `static` modificadores e são discutidos nas seções a seguir.

## Classes abstratas

O `abstract` modificador é usado para indicar que uma classe está incompleta e que se destina a ser usada somente como uma classe base. Uma classe abstrata difere de uma classe não abstrata das seguintes maneiras:

- Uma classe abstrata não pode ser instanciada diretamente e é um erro de tempo de compilação para usar o `new` operador em uma classe abstrata. Embora seja possível ter variáveis e valores cujos tipos de tempo de compilação sejam abstratos, tais variáveis e valores serão necessariamente `null` ou contêm referências a instâncias de classes não abstratas derivadas dos tipos abstratos.
- Uma classe abstrata é permitida (mas não obrigatória) para conter membros abstratos.

- Uma classe abstrata não pode ser selada.

Quando uma classe não abstrata é derivada de uma classe abstrata, a classe não abstrata deve incluir implementações reais de todos os membros abstratos herdados, substituindo assim esses membros abstratos. No exemplo

```
C#  
  
abstract class A  
{  
    public abstract void F();  
}  
  
abstract class B: A  
{  
    public void G() {}  
}  
  
class C: B  
{  
    public override void F() {  
        // actual implementation of F  
    }  
}
```

a classe abstract `A` introduz um método abstract `F`. A classe `B` introduz um método adicional `G`, mas como não fornece uma implementação de `F`, `B` também deve ser declarada abstrata. As substituições `F` de classe e fornecem uma implementação real. Como não há membros abstratos no `C`, o `C` é permitido (mas não obrigatório) para ser não abstrato.

## Classes lacradas

O `sealed` modificador é usado para impedir a derivação de uma classe. Um erro em tempo de compilação ocorrerá se uma classe selada for especificada como a classe base de outra classe.

Uma classe selada também não pode ser uma classe abstrata.

O `sealed` modificador é usado principalmente para evitar derivação não intencional, mas também permite determinadas otimizações de tempo de execução. Em particular, como uma classe selada é conhecida por nunca ter classes derivadas, é possível transformar invocações de membro de função virtual em instâncias de classe seladas em invocações não virtuais.

## Classes estáticas

O `static` modificador é usado para marcar a classe que está sendo declarada como uma **classe estática**. Uma classe estática não pode ser instanciada, não pode ser usada como um tipo e pode conter somente membros estáticos. Somente uma classe estática pode conter declarações de métodos de extensão ([métodos de extensão](#)).

Uma declaração de classe estática está sujeita às seguintes restrições:

- Uma classe estática não pode incluir um `sealed abstract` modificador ou. No entanto, observe que, como uma classe estática não pode ser instanciada ou derivada de, ela se comporta como se fosse selada e abstrata.
- Uma classe estática não pode incluir uma especificação de `class_base` ([especificação de base de classe](#)) e não pode especificar explicitamente uma classe base ou uma lista de interfaces implementadas. Uma classe estática herda implicitamente do tipo `object`.
- Uma classe estática só pode conter membros estáticos ([membros estáticos e de instância](#)). Observe que as constantes e os tipos aninhados são classificados como membros estáticos.
- Uma classe estática não pode ter membros `protected` ou a `protected internal` acessibilidade declarada.

É um erro de tempo de compilação para violar qualquer uma dessas restrições.

Uma classe estática não tem construtores de instância. Não é possível declarar um construtor de instância em uma classe estática e nenhum construtor de instância padrão ([construtores padrão](#)) é fornecido para uma classe estática.

Os membros de uma classe estática não são estáticos automaticamente e as declarações de membro devem incluir explicitamente um `static` modificador (exceto para constantes e tipos aninhados). Quando uma classe é aninhada em uma classe externa estática, a classe aninhada não é uma classe estática, a menos que inclua explicitamente um `static` modificador.

### Referenciando tipos de classe estática

Um `namespace_or_type_name` ([namespace e nomes de tipo](#)) tem permissão para fazer referência a uma classe estática se

- O `namespace_or_type_name` é o `T` em uma `namespace_or_type_name` do formulário `T.I` ou
- O `namespace_or_type_name` é o `T` em uma `typeof_expression` ([lista de argumentos](#)) do formulário `typeof(T)`.

Um *primary\_expression* ([membros da função](#)) tem permissão para fazer referência a uma classe estática se

- O *primary\_expression* é o `E` em um *member\_access* ([verificação de tempo de compilação da resolução dinâmica de sobrecarga](#)) do formulário `E.I`.

Em qualquer outro contexto, é um erro de tempo de compilação para fazer referência a uma classe estática. Por exemplo, é um erro para uma classe estática ser usada como uma classe base, um tipo constituinte ([tipos aninhados](#)) de um membro, um argumento de tipo genérico ou uma restrição de parâmetro de tipo. Da mesma forma, uma classe estática não pode ser usada em um tipo de matriz, um tipo de ponteiro, uma expressão `new`, uma expressão de conversão, uma expressão, uma expressão `is` uma expressão `as` `sizeof` ou uma expressão de valor padrão.

## Modificador parcial

O `partial` modificador é usado para indicar que esse *class\_declaration* é uma declaração de tipo parcial. Várias declarações de tipo parcial com o mesmo nome dentro de um namespace delimitador ou declaração de tipo são combinadas para formar uma declaração de tipo, seguindo as regras especificadas em [tipos parciais](#).

Ter a declaração de uma classe distribuída em segmentos separados de texto do programa pode ser útil se esses segmentos são produzidos ou mantidos em diferentes contextos. Por exemplo, uma parte de uma declaração de classe pode ser gerada pela máquina, enquanto a outra é criada manualmente. A separação textual dos dois impede que as atualizações de um entrem em conflito com as atualizações do outro.

## Parâmetros de tipo

Um parâmetro de tipo é um identificador simples que denota um espaço reservado para um argumento de tipo fornecido para criar um tipo construído. Um parâmetro de tipo é um espaço reservado formal para um tipo que será fornecido posteriormente. Por outro lado, um argumento de tipo ([argumentos de tipo](#)) é o tipo real que é substituído pelo parâmetro de tipo quando um tipo construído é criado.

```
antlr

type_parameter_list
    : '<' type_parameters '>'
    ;
type_parameters
    : attributes? type_parameter
```

```
| type_parameters ',' attributes? type_parameter  
;  
  
type_parameter  
: identifier  
;
```

Cada parâmetro de tipo em uma declaração de classe define um nome no espaço de declaração ([declarações](#)) dessa classe. Portanto, ele não pode ter o mesmo nome que outro parâmetro de tipo ou um membro declarado nessa classe. Um parâmetro de tipo não pode ter o mesmo nome que o próprio tipo.

## Especificação de base de classe

Uma declaração de classe pode incluir uma especificação de *class\_base*, que define a classe base direta da classe e as interfaces ([interfaces](#)) implementadas diretamente pela classe.

```
antlr  
  
class_base  
: ':' class_type  
| ':' interface_type_list  
| ':' class_type ',' interface_type_list  
;  
  
interface_type_list  
: interface_type (',' interface_type)*  
;
```

A classe base especificada em uma declaração de classe pode ser um tipo de classe construído ([tipos construídos](#)). Uma classe base não pode ser um parâmetro de tipo por conta própria, embora possa envolver os parâmetros de tipo que estão no escopo.

C#

```
class Extend<V>: V {}           // Error, type parameter used as base class
```

## Classes base

Quando um *class\_type* é incluído no *class\_base*, ele especifica a classe base direta da classe que está sendo declarada. Se uma declaração de classe não tiver nenhum *class\_base*, ou se o *class\_base* listar apenas os tipos de interface, a classe base direta será

considerada `object`. Uma classe herda membros de sua classe base direta, conforme descrito em [herança](#).

No exemplo

```
C#  
  
class A {}  
  
class B: A {}
```

a classe `A` é considerada como a classe base direta de `B` e `B` é considerada derivada de `A`. Como `A` não especifica explicitamente uma classe base direta, sua classe base direta é implicitamente `object`.

Para um tipo de classe construído, se uma classe base for especificada na declaração de classe genérica, a classe base do tipo construído será obtida pela substituição, para cada *type\_parameter* na declaração de classe base, a *type\_argument* correspondente do tipo construído. Dadas as declarações de classe genéricas

```
C#  
  
class B<U,V> {...}  
  
class G<T>: B<string,T[]> {...}
```

a classe base do tipo construído `G<int>` seria `B<string,int[]>`.

A classe base direta de um tipo de classe deve ser pelo menos acessível como o próprio tipo de classe ([domínios de acessibilidade](#)). Por exemplo, é um erro de tempo de compilação para uma `public` classe derivar de uma `private` `internal` classe ou.

A classe base direta de um tipo de classe não deve ser um dos seguintes tipos:

`System.Array` ..., `System.Delegate` `System.MulticastDelegate` `System.Enum` ou `System.ValueType`. Além disso, uma declaração de classe genérica não pode usar `System.Attribute` como uma classe base direta ou indireta.

Ao determinar o significado da especificação de classe base direta `A` de uma classe `B`, a classe base direta de `B` é temporariamente considerada `object`. Intuitivamente, isso garante que o significado de uma especificação de classe base não possa depender recursivamente por si só. O exemplo:

```
C#
```

```
class A<T> {
    public class B {}
}

class C : A<C.B> {}
```

é um erro porque, na especificação da classe base `A<C.B>`, a classe base direta de `C` é considerada como sendo `object`, e portanto (pelos regras de [namespace e nomes de tipo](#)) `C` não é considerada como um membro `B`.

As classes base de um tipo de classe são a classe base direta e suas classes base. Em outras palavras, o conjunto de classes base é o fechamento transitivo da relação de classe base direta. Fazendo referência ao exemplo acima, as classes base do `B` são `A` e `object`. No exemplo

C#

```
class A {...}

class B<T>: A {...}

class C<T>: B<IComparable<T>> {...}

class D<T>: C<T[]> {...}
```

as classes base do `D<int>` são `C<int[]>`, `B<IComparable<int[]>>`, `A`, e `object`.

Exceto para classe `object`, cada tipo de classe tem exatamente uma classe base direta. A `object` classe não tem nenhuma classe base direta e é a classe base definitiva de todas as outras classes.

Quando uma classe `B` deriva de uma classe `A`, é um erro de tempo de compilação para o `A` qual depender `B`. Uma classe *depende diretamente* da sua classe base direta (se houver) e \_*depende diretamente\**\_ da classe na qual ela é imediatamente aninhada (se houver). Dada essa definição, o conjunto completo de classes sobre as quais uma classe depende é o fechamento reflexivo e transitivo da relação \_ *depende diretamente* de \*.

O exemplo

C#

```
class A: A {}
```

é errado porque a classe depende dele mesmo. Da mesma forma, o exemplo

C#

```
class A: B {}
class B: C {}
class C: A {}
```

é um erro porque as classes dependem circularmente. Por fim, o exemplo

C#

```
class A: B.C {}
class B: A
{
    public class C {}
}
```

resulta em um erro de tempo de compilação porque `A` depende de `B.C` (sua classe base direta), que depende `B` (sua classe de circunscrição imediatamente), que depende circularmente `A`.

Observe que uma classe não depende das classes que estão aninhadas dentro dela. No exemplo

C#

```
class A
{
    class B: A {}
}
```

`B` depende de `A` (porque `A` é sua classe base direta e sua classe imediatamente delimitadora), mas `A` não depende `B` (porque não `B` é uma classe base nem uma classe delimitadora `A`). Portanto, o exemplo é válido.

Não é possível derivar de uma `sealed` classe. No exemplo

C#

```
sealed class A {}

class B: A {}           // Error, cannot derive from a sealed class
```

`B` a classe está com erro porque tenta derivar da `sealed` classe `A`.

## Implementações de interfaces

Uma especificação de *class\_base* pode incluir uma lista de tipos de interface; nesse caso, a classe é mencionada para implementar diretamente os tipos de interface fornecidos. Implementações de interface são discutidas mais detalhadamente em [implementações de interface](#).

## Restrições de parâmetro de tipo

As declarações de tipo genérico e de método podem opcionalmente especificar restrições de parâmetro de tipo, incluindo *type\_parameter\_constraints\_clause*s.

```
antlr

type_parameter_constraints_clause
: 'where' type_parameter ':' type_parameter_constraints
;

type_parameter_constraints
: primary_constraint
| secondary_constraints
| constructor_constraint
| primary_constraint ',' secondary_constraints
| primary_constraint ',' constructor_constraint
| secondary_constraints ',' constructor_constraint
| primary_constraint ',' secondary_constraints ',' constructor_constraint
;
;

primary_constraint
: class_type
| 'class'
| 'struct'
;
;

secondary_constraints
: interface_type
| type_parameter
| secondary_constraints ',' interface_type
| secondary_constraints ',' type_parameter
;
;

constructor_constraint
: 'new' '(' ')'
;
;
```

Cada *type\_parameter\_constraints\_clause* consiste no token `where`, seguido pelo nome de um parâmetro de tipo, seguido por dois-pontos e pela lista de restrições para esse parâmetro de tipo. Pode haver no máximo uma `where` cláusula para cada parâmetro de

tipo e as `where` cláusulas podem ser listadas em qualquer ordem. Como os `get` `set` tokens e em um acessador de propriedade, o `where` token não é uma palavra-chave.

A lista de restrições fornecida em uma `where` cláusula pode incluir qualquer um dos seguintes componentes, nesta ordem: uma única restrição primária, uma ou mais restrições secundárias e a restrição de construtor, `new()`.

Uma restrição PRIMARY pode ser um tipo de classe ou a \* restrição de **tipo de referência \_ class** ou a *restrição de tipo de valor struct*. Uma restrição secundária pode ser uma `_type_parameter` \* ou *interface\_type*.

A restrição de tipo de referência especifica que um argumento de tipo usado para o parâmetro de tipo deve ser um tipo de referência. Todos os tipos de classe, tipos de interface, tipos delegados, tipos de matriz e parâmetros de tipo conhecidos como um tipo de referência (conforme definido abaixo) atendem a essa restrição.

A restrição de tipo de valor especifica que um argumento de tipo usado para o parâmetro de tipo deve ser um tipo de valor não anulável. Todos os tipos de `struct` não anuláveis, tipos de enumeração e parâmetros de tipo que têm a restrição de tipo de valor atendem a essa restrição. Observe que, embora seja classificado como um tipo de valor, um tipo anulável ([tipos anuláveis](#)) não satisfaz a restrição de tipo de valor. Um parâmetro de tipo com a restrição de tipo de valor também não pode ter a `constructor_constraint`.

Os tipos de ponteiro nunca podem ser argumentos de tipo e não são considerados para satisfazer as restrições de tipo de referência ou tipo de valor.

Se uma restrição for um tipo de classe, um tipo de interface ou um parâmetro de tipo, esse tipo especificará um "tipo base" mínimo que cada argumento de tipo usado para esse parâmetro de tipo deve dar suporte a. Sempre que um tipo construído ou um método genérico é usado, o argumento de tipo é verificado em relação às restrições no parâmetro de tipo em tempo de compilação. O argumento de tipo fornecido deve atender às condições descritas em [satisfazer restrições](#).

Uma restrição de `class_type` deve atender às seguintes regras:

- O tipo deve ser um tipo de classe.
- O tipo não deve ser `sealed`.
- O tipo não deve ser um dos seguintes tipos: `System.Array` , `System.Delegate` `System.Enum` OU `System.ValueType`.
- O tipo não deve ser `object`. Como todos os tipos derivam de `object`, essa restrição não teria nenhum efeito se fosse permitida.

- No máximo uma restrição para um determinado parâmetro de tipo pode ser um tipo de classe.

Um tipo especificado como uma restrição de *interface\_type* deve atender às seguintes regras:

- O tipo deve ser um tipo de interface.
- Um tipo não deve ser especificado mais de uma vez em uma determinada `where` cláusula.

Em ambos os casos, a restrição pode envolver qualquer um dos parâmetros de tipo do tipo associado ou declaração de método como parte de um tipo construído e pode envolver o tipo que está sendo declarado.

Qualquer classe ou tipo de interface especificado como uma restrição de parâmetro de tipo deve ser pelo menos como acessível ([restrições de acessibilidade](#)) como o tipo genérico ou o método que está sendo declarado.

Um tipo especificado como uma restrição de *type\_parameter* deve atender às seguintes regras:

- O tipo deve ser um parâmetro de tipo.
- Um tipo não deve ser especificado mais de uma vez em uma determinada `where` cláusula.

Além disso, não deve haver ciclos no grafo de dependência dos parâmetros de tipo, em que a dependência é uma relação transitiva definida por:

- Se um parâmetro de tipo `T` for usado como uma restrição para o parâmetro de tipo `S`, `S` **dependerá de** `T`.
- Se um parâmetro de tipo `S` depender de um parâmetro de tipo `T` e depender de `T` um parâmetro de tipo `U`, `S` **dependerá de** `U`.

Dada essa relação, é um erro de tempo de compilação para um parâmetro de tipo depender de si mesmo (direta ou indiretamente).

Todas as restrições devem ser consistentes entre os parâmetros de tipo dependentes. Se o parâmetro `S` de tipo depender do parâmetro de tipo `T`, então:

- `T` Não deve ter a restrição de tipo de valor. Caso contrário, `T` é efetivamente lacrado, portanto `S`, seria forçado a ser o mesmo tipo de `T`, eliminando a necessidade de dois parâmetros de tipo.
- Se `S` o tiver a restrição de tipo de valor, então `T` não deverá ter uma restrição *class\_type*.

- Se `s` o tiver uma restrição de `class_type A` e `T` tiver uma restrição de `class_type B`, deverá haver uma conversão de identidade ou conversão de referência implícita de `A` para `B` ou uma conversão de referência implícita de `B` para `A`.
- Se `s` também depender do parâmetro de tipo `U` e `U` tiver uma restrição de `class_type A` e `T` tiver uma restrição de `class_type B`, deverá haver uma conversão de `B` identidade ou conversão de referência implícita de `A` para `B` ou uma conversão de referência implícita de `B` para `A`.

É válido para que `s` o tenha a restrição de tipo de valor e `T` tenha a restrição de tipo de referência. Efetivamente, isso limita `T` os tipos `System.Object`, `System.ValueType`, `System.Enum` e qualquer tipo de interface.

Se a `where` cláusula de um parâmetro de tipo incluir uma restrição de Construtor (que tem o formulário `new()`), será possível usar o `new` operador para criar instâncias do tipo ([expressões de criação de objeto](#)). Qualquer argumento de tipo usado para um parâmetro de tipo com uma restrição de construtor deve ter um construtor público sem parâmetros (esse construtor existe implicitamente para qualquer tipo de valor) ou ser um parâmetro de tipo que tenha a restrição de tipo de valor ou restrição de Construtor (consulte [restrições de parâmetro de tipo](#) para obter detalhes).

Veja a seguir exemplos de restrições:

C#

```
interface IPrintable
{
    void Print();
}

interface IComparable<T>
{
    int CompareTo(T value);
}

interface IKeyProvider<T>
{
    T GetKey();
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}

class Dictionary<K,V>
    where K: IComparable<K>
    where V: IPrintable, IKeyProvider<K>, new()
{
```

```
    ...  
}
```

O exemplo a seguir está em erro porque causa uma circularidade no grafo de dependência dos parâmetros de tipo:

```
C#
```

```
class Circular<S,T>  
    where S: T  
    where T: S          // Error, circularity in dependency graph  
{  
    ...  
}
```

Os exemplos a seguir ilustram situações inválidas adicionais:

```
C#
```

```
class Sealed<S,T>  
    where S: T  
    where T: struct      // Error, T is sealed  
{  
    ...  
}  
  
class A {...}  
  
class B {...}  
  
class Incompat<S,T>  
    where S: A, T  
    where T: B          // Error, incompatible class-type constraints  
{  
    ...  
}  
  
class StructWithClass<S,T,U>  
    where S: struct, T  
    where T: U  
    where U: A          // Error, A incompatible with struct  
{  
    ...  
}
```

A **classe base efetiva** de um parâmetro de tipo `T` é definida da seguinte maneira:

- Se `T` o não tiver restrições primárias ou restrições de parâmetro de tipo, sua classe base efetiva será `object`.

- Se `T` tiver a restrição de tipo de valor, sua classe base efetiva será `System.ValueType`.
- Se `T` tiver uma restrição `class_type`, `C` mas sem restrições de `type_parameter`, sua classe base efetiva será `C`.
- Se `T` não tiver nenhuma restrição de `class_type`, mas tiver uma ou mais restrições de `type_parameter`, sua classe base efetiva será o tipo mais abrangendo ([operadores de conversão levantados](#)) no conjunto de classes base efetivas de suas restrições de `type_parameter`. As regras de consistência asseguram que exista um tipo mais abrangente.
- Se `T` o tiver uma restrição `class_type` e uma ou mais restrições `type_parameter`, sua classe base efetiva será o tipo mais abrangendo ([operadores de conversão levantados](#)) no conjunto que consiste na restrição de `class_type` de `T` e as classes base efetivas de suas restrições de `type_parameter`. As regras de consistência asseguram que exista um tipo mais abrangente.
- Se `T` o tiver a restrição de tipo de referência, mas não `class_type` restrições, sua classe base efetiva será `object`.

Para a finalidade dessas regras, se `T` tiver uma restrição `V` que seja uma `value_type`, use em vez disso, o tipo base mais específico `V` é um `class_type`. Isso nunca pode ocorrer em uma restrição explicitamente determinada, mas pode ocorrer quando as restrições de um método genérico são implicitamente herdadas por uma declaração de método de substituição ou uma implementação explícita de um método de interface.

Essas regras garantem que a classe base efetiva sempre seja uma `class_type`.

O **conjunto de interfaces efetivas** de um parâmetro de tipo `T` é definido da seguinte maneira:

- Se `T` não tiver nenhum `secondary_constraints`, seu conjunto de interface eficaz estará vazio.
- Se o `T` tiver `interface_type` restrições, mas nenhuma restrição de `type_parameter`, seu conjunto de interface eficaz será seu conjunto de restrições de `interface_type`.
- Se o `T` não tiver restrições de `interface_type`, mas tiver restrições de `type_parameter`, seu conjunto de interface eficaz será a União dos conjuntos de interface efetivos de suas restrições de `type_parameter`.
- Se `T` o tiver restrições `interface_type` e `type_parameter` restrições, seu conjunto de interface eficaz será a União de seu conjunto de restrições de `interface_type` e os conjuntos de interface efetivos de suas restrições de `type_parameter`.

Um parâmetro de tipo é **conhecido como um tipo de referência** se ele tiver a restrição de tipo de referência ou se sua classe base efetiva não for `object` ou `System.ValueType`.

Os valores de um tipo de parâmetro de tipo restrito podem ser usados para acessar os membros de instância implícitos pelas restrições. No exemplo

```
C#  
  
interface IPrintable  
{  
    void Print();  
}  
  
class Printer<T> where T: IPrintable  
{  
    void PrintOne(T x) {  
        x.Print();  
    }  
}
```

os métodos de `IPrintable` podem ser invocados diretamente no `x` porque o `T` é restrito a sempre implementar `IPrintable`.

## Corpo da classe

O *class\_body* de uma classe define os membros dessa classe.

```
antlr  
  
class_body  
: '{' class_member_declaraction* '}'  
;
```

## Tipos parciais

Uma declaração de tipo pode ser dividida em várias *declarações de tipo parciais*. A declaração de tipo é construída de suas partes seguindo as regras nesta seção, momento ela é tratada como uma única declaração durante o restante do processamento de tempo de compilação e tempo de execução do programa.

Um *class\_declaration*, *struct\_declaration* ou *interface\_declaration* representa uma declaração de tipo parcial se ele incluir um `partial` modificador. `partial` Não é uma palavra-chave e atua apenas como um modificador se aparecer imediatamente antes de uma das palavras-chave `class`, `struct` ou `interface` em uma declaração de tipo ou antes do tipo `void` em uma declaração de método. Em outros contextos, ele pode ser usado como um identificador normal.

Cada parte de uma declaração de tipo parcial deve incluir um `partial` modificador. Ele deve ter o mesmo nome e ser declarado no mesmo namespace ou em uma declaração de tipo que as outras partes. O `partial` modificador indica que partes adicionais da declaração de tipo podem existir em outro lugar, mas a existência dessas partes adicionais não é um requisito; ela é válida para um tipo com uma única declaração para incluir o `partial` modificador.

Todas as partes de um tipo parcial devem ser compiladas em conjunto, de modo que as partes possam ser mescladas em tempo de compilação em uma única declaração de tipo. Tipos parciais especificamente não permitem que tipos já compilados sejam estendidos.

Tipos aninhados podem ser declarados em várias partes usando o `partial` modificador. Normalmente, o tipo recipiente é declarado usando `partial` também, e cada parte do tipo aninhado é declarada em uma parte diferente do tipo recipiente.

O `partial` modificador não é permitido em declarações delegate ou enum.

## Atributos

Os atributos de um tipo parcial são determinados pela combinação, em uma ordem não especificada, os atributos de cada uma das partes. Se um atributo for colocado em várias partes, será equivalente a especificar o atributo várias vezes no tipo. Por exemplo, as duas partes:

```
C#  
  
[Attr1, Attr2("hello")]
partial class A {}  
  
[Attr3, Attr2("goodbye")]
partial class A {}
```

são equivalentes a uma declaração como:

```
C#  
  
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

Os atributos nos parâmetros de tipo são combinados de maneira semelhante.

## Modificadores

Quando uma declaração de tipo parcial inclui uma especificação de acessibilidade (os `public` `protected` modificadores,, `internal` e `private` ), ela deve concordar com todas as outras partes que incluem uma especificação de acessibilidade. Se nenhuma parte de um tipo parcial incluir uma especificação de acessibilidade, o tipo receberá a acessibilidade padrão apropriada ([acessibilidade declarada](#)).

Se uma ou mais declarações parciais de um tipo aninhado incluírem um `new` modificador, nenhum aviso será relatado se o tipo aninhado ocultar um membro herdado ([ocultando a herança](#)).

Se uma ou mais declarações parciais de uma classe incluírem um `abstract` modificador, a classe será considerada abstrata ([classes abstratas](#)). Caso contrário, a classe será considerada não abstrata.

Se uma ou mais declarações parciais de uma classe incluírem um `sealed` modificador, a classe será considerada selada ([classes seladas](#)). Caso contrário, a classe será considerada sem lacre.

Observe que uma classe não pode ser abstrata e selada.

Quando o `unsafe` modificador é usado em uma declaração de tipo parcial, somente essa parte específica é considerada um contexto não seguro ([contextos não seguros](#)).

## Parâmetros de tipo e restrições

Se um tipo genérico for declarado em várias partes, cada parte deverá declarar os parâmetros de tipo. Cada parte deve ter o mesmo número de parâmetros de tipo e o mesmo nome para cada parâmetro de tipo, em ordem.

Quando uma declaração de tipo genérico parcial inclui restrições ( `where` cláusulas), as restrições devem concordar com todas as outras partes que incluem restrições.

Especificamente, cada parte que inclui restrições deve ter restrições para o mesmo conjunto de parâmetros de tipo e, para cada parâmetro de tipo, os conjuntos de restrições primária, secundária e de construtor devem ser equivalentes. Dois conjuntos de restrições são equivalentes se contiverem os mesmos membros. Se nenhuma parte de um tipo genérico parcial especificar restrições de parâmetro de tipo, os parâmetros de tipo serão considerados irrestrito.

O exemplo

C#

```
partial class Dictionary<K,V>
    where K: IComparable<K>
```

```
    where V: IKeyProvider<K>, IPersistable
{
    ...
}

partial class Dictionary<K,V>
    where V: IPersistable, IKeyProvider<K>
    where K: IComparable<K>
{
    ...
}

partial class Dictionary<K,V>
{
    ...
}
```

está correto porque essas partes que incluem restrições (as duas primeiras) efetivamente especificam o mesmo conjunto de restrições primárias, secundárias e de construtor para o mesmo conjunto de parâmetros de tipo, respectivamente.

## Classe base

Quando uma declaração de classe parcial inclui uma especificação de classe base, ela deve concordar com todas as outras partes que incluem uma especificação de classe base. Se nenhuma parte de uma classe parcial incluir uma especificação de classe base, a classe base se tornará `System.Object` ([classes base](#)).

## Interfaces base

O conjunto de interfaces base para um tipo declarado em várias partes é a União das interfaces base especificadas em cada parte. Uma interface base específica só pode ser nomeada uma vez em cada parte, mas é permitida para várias partes nomear as mesmas interfaces base. Deve haver apenas uma implementação dos membros de qualquer interface base fornecida.

No exemplo

```
C#
partial class C: IA, IB {...}

partial class C: IC {...}

partial class C: IA, IB {...}
```

o conjunto de interfaces base para a classe C é IA , IB e IC .

Normalmente, cada parte fornece uma implementação das interfaces declaradas nessa parte; no entanto, isso não é um requisito. Uma parte pode fornecer a implementação para uma interface declarada em uma parte diferente:

C#

```
partial class X
{
    int IComparable.CompareTo(object o) {...}

partial class X: IComparable
{
    ...
}
```

## Membros

Com exceção dos métodos parciais ([métodos parciais](#)), o conjunto de membros de um tipo declarado em várias partes é simplesmente a União do conjunto de membros declarado em cada parte. Os corpos de todas as partes da declaração de tipo compartilham o mesmo espaço de declaração ([declarações](#)), e o escopo de cada membro ([escopos](#)) se estende aos corpos de todas as partes. O domínio de acessibilidade de qualquer membro sempre inclui todas as partes do tipo delimitador; um `private` Membro declarado em uma parte pode ser acessado livremente de outra parte. É um erro de tempo de compilação para declarar o mesmo membro em mais de uma parte do tipo, a menos que esse membro seja um tipo com o `partial` modificador.

C#

```
partial class A
{
    int x;                      // Error, cannot declare x more than once

    partial class Inner          // Ok, Inner is a partial type
    {
        int y;
    }
}

partial class A
{
    int x;                      // Error, cannot declare x more than once

    partial class Inner          // Ok, Inner is a partial type
```

```
{  
    int z;  
}
```

A ordenação de membros dentro de um tipo é raramente significativa para código C#, mas pode ser significativa ao fazer a interface com outras linguagens e ambientes. Nesses casos, a ordenação de membros dentro de um tipo declarado em várias partes é indefinida.

## Métodos parciais

Os métodos parciais podem ser definidos em uma parte de uma declaração de tipo e implementados em outro. A implementação é opcional; Se nenhuma parte implementar o método parcial, a declaração de método parcial e todas as chamadas para ele serão removidas da declaração de tipo resultante da combinação das partes.

Os métodos parciais não podem definir modificadores de acesso, mas são implicitamente `private`. O tipo de retorno deve ser `void`, e seus parâmetros não podem ter o `out` modificador. O identificador `partial` será reconhecido como uma palavra-chave especial em uma declaração de método somente se aparecer imediatamente antes do `void` tipo; caso contrário, ele poderá ser usado como um identificador normal. Um método parcial não pode implementar explicitamente métodos de interface.

Há dois tipos de declarações de método parciais: se o corpo da declaração do método for um ponto-e-vírgula, a declaração será um **\*definindo uma declaração de método parcial\***. Se o corpo for fornecido como um `_block` \*, a declaração será considerada uma declaração de ***método parcial de implementação***. Em todas as partes de uma declaração de tipo, pode haver apenas uma definição de declaração de método parcial com uma determinada assinatura, e pode haver apenas uma implementação de declaração de método parcial com uma determinada assinatura. Se uma declaração de método parcial de implementação for fornecida, uma declaração de método parcial de definição correspondente deverá existir e as declarações deverão corresponder conforme especificado no seguinte:

- As declarações devem ter os mesmos modificadores (embora não necessariamente na mesma ordem), nome do método, número de parâmetros de tipo e número de parâmetros.
- Os parâmetros correspondentes nas declarações devem ter os mesmos modificadores (embora não necessariamente na mesma ordem) e os mesmos tipos (diferenças de módulo em nomes de parâmetro de tipo).

- Os parâmetros de tipo correspondentes nas declarações devem ter as mesmas restrições (diferenças de módulo em nomes de parâmetro de tipo).

Uma declaração de método parcial de implementação pode aparecer na mesma parte da declaração de método parcial de definição correspondente.

Apenas um método parcial de definição participa da resolução de sobrecarga. Portanto, se uma declaração de implementação for ou não determinada, as expressões de invocação podem ser resolvidas para invocações do método parcial. Como um método parcial sempre retorna `void`, essas expressões de invocação sempre serão instruções de expressão. Além disso, como um método parcial é implicitamente `private`, essas instruções sempre ocorrerão dentro de uma das partes da declaração de tipo dentro da qual o método parcial é declarado.

Se nenhuma parte de uma declaração de tipo parcial contiver uma declaração de implementação para um determinado método parcial, qualquer instrução de expressão invocando-a será simplesmente removida da declaração de tipo combinado. Portanto, a expressão de invocação, incluindo quaisquer expressões constituintes, não tem nenhum efeito no tempo de execução. O próprio método parcial também é removido e não será um membro da declaração de tipo combinado.

Se existir uma declaração de implementação para um determinado método parcial, as invocações dos métodos parciais serão mantidas. O método parcial fornece a elevação para uma declaração de método semelhante à declaração de método parcial de implementação, exceto para o seguinte:

- O `partial` modificador não está incluído
- Os atributos na declaração de método resultante são os atributos combinados da definição e a declaração de método parcial de implementação em ordem não especificada. Duplicatas não são removidas.
- Os atributos nos parâmetros da declaração de método resultante são os atributos combinados dos parâmetros correspondentes da definição e a declaração de método parcial de implementação em ordem não especificada. Duplicatas não são removidas.

Se uma declaração de definição, mas não uma declaração de implementação, for fornecida para um método parcial M, as seguintes restrições serão aplicáveis:

- É um erro de tempo de compilação para criar um delegate para o método ([expressões de criação de delegado](#)).
- É um erro de tempo de compilação para fazer referência a M dentro de uma função anônima que é convertida em um tipo de árvore de expressão ([avaliação de conversões de função anônima para tipos de árvore de expressão](#)).

- As expressões que ocorrem como parte de uma invocação do não M afetam o estado de atribuição definido ([atribuição definitiva](#)), o que pode levar a erros em tempo de compilação.
- M Não pode ser o ponto de entrada para um aplicativo ([inicialização do aplicativo](#)).

Os métodos parciais são úteis para permitir que uma parte de uma declaração de tipo Personalize o comportamento de outra parte, por exemplo, uma que seja gerada por uma ferramenta. Considere a seguinte declaração de classe parcial:

```
C#  
  
partial class Customer  
{  
    string name;  
  
    public string Name {  
        get { return name; }  
        set {  
            OnNameChanging(value);  
            name = value;  
            OnNameChanged();  
        }  
    }  
  
    partial void OnNameChanging(string newName);  
  
    partial void OnNameChanged();  
}
```

Se essa classe for compilada sem nenhuma outra parte, a definição de declarações de método parcial e suas invocações será removida e a declaração de classe combinada resultante será equivalente à seguinte:

```
C#  
  
class Customer  
{  
    string name;  
  
    public string Name {  
        get { return name; }  
        set { name = value; }  
    }  
}
```

Suponha que outra parte seja fornecida, no entanto, que fornece declarações de implementação dos métodos parciais:

C#

```
partial class Customer
{
    partial void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    partial void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

Em seguida, a declaração de classe combinada resultante será equivalente à seguinte:

C#

```
class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }
    }

    void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

## Associação de nome

Embora cada parte de um tipo extensível deva ser declarada dentro do mesmo namespace, as partes normalmente são escritas em diferentes declarações de namespace. Portanto, `using` diretivas diferentes ([usando diretivas](#)) podem estar

presentes para cada parte. Ao interpretar nomes simples ([inferência de tipos](#)) dentro de uma parte, somente as `using` diretivas das declarações de namespace que envolvem essa parte são consideradas. Isso pode resultar no mesmo identificador com significados diferentes em diferentes partes:

```
C#  
  
namespace N  
{  
    using List = System.Collections.ArrayList;  
  
    partial class A  
    {  
        List x; // x has type System.Collections.ArrayList  
    }  
}  
  
namespace N  
{  
    using List = Widgets.LinkedList;  
  
    partial class A  
    {  
        List y; // y has type Widgets.LinkedList  
    }  
}
```

## Membros de classe

Os membros de uma classe consistem nos membros introduzidos por seu *class\_member\_declaration*s e os membros herdados da classe base direta.

```
antlr  
  
class_member_declaration  
: constant_declaraction  
| field_declaraction  
| method_declaraction  
| property_declaraction  
| event_declaraction  
| indexer_declaraction  
| operator_declaraction  
| constructor_declaraction  
| destructor_declaraction  
| static_constructor_declaraction  
| type_declaraction  
;
```

Os membros de um tipo de classe são divididos nas seguintes categorias:

- Constantes, que representam valores constantes associados à classe ([constants](#)).
- Campos, que são as variáveis da classe ([campos](#)).
- Métodos, que implementam os cálculos e as ações que podem ser executadas pela classe ([métodos](#)).
- Propriedades, que definem características nomeadas e as ações associadas à leitura e à gravação dessas características ([Propriedades](#)).
- Eventos, que definem as notificações que podem ser geradas pela classe ([eventos](#)).
- Indexadores, que permitem que as instâncias da classe sejam indexadas da mesma forma (sintaticamente) como matrizes ([indexadores](#)).
- Operadores, que definem os operadores de expressão que podem ser aplicados a instâncias da classe ([operadores](#)).
- Construtores de instância, que implementam as ações necessárias para inicializar instâncias da classe ([construtores de instância](#))
- Destruidores, que implementam as ações a serem executadas antes que as instâncias da classe sejam descartadas permanentemente ([destruidores](#)).
- Construtores estáticos, que implementam as ações necessárias para inicializar a própria classe ([construtores estáticos](#)).
- Tipos, que representam os tipos que são locais para a classe ([tipos aninhados](#)).

Membros que podem conter código executável são coletivamente conhecidos como *membros da função* do tipo de classe. Os membros da função de um tipo de classe são os métodos, as propriedades, os eventos, os indexadores, os operadores, os construtores de instância, os destruidores e os construtores estáticos desse tipo de classe.

Um *class\_declaration* cria um novo espaço de declaração ([declarações](#)) e os *class\_member\_declaration*s imediatamente contidos pelo *class\_declaration* introduzem novos membros nesse espaço de declaração. As regras a seguir se aplicam a *class\_member\_declaration*s:

- Construtores de instância, destruidores e construtores estáticos devem ter o mesmo nome que a classe de circunscrição imediata. Todos os outros membros devem ter nomes que diferem do nome da classe imediatamente delimitadora.
- O nome de uma constante, campo, propriedade, evento ou tipo deve ser diferente dos nomes de todos os outros membros declarados na mesma classe.
- O nome de um método deve ser diferente dos nomes de todos os outros não-métodos declarados na mesma classe. Além disso, a assinatura ([assinaturas e sobrecarga](#)) de um método deve ser diferente das assinaturas de todos os outros métodos declarados na mesma classe, e dois métodos declarados na mesma classe podem não ter assinaturas que diferem exclusivamente por `ref` and `out`.

- A assinatura de um construtor de instância deve ser diferente das assinaturas de todos os outros construtores de instância declarados na mesma classe, e dois construtores declarados na mesma classe podem não ter assinaturas que diferem exclusivamente pelo `ref` and `out`.
- A assinatura de um indexador deve ser diferente das assinaturas de todos os outros indexadores declarados na mesma classe.
- A assinatura de um operador deve ser diferente das assinaturas de todos os outros operadores declarados na mesma classe.

Os membros herdados de um tipo de classe ([herança](#)) não fazem parte do espaço de declaração de uma classe. Assim, uma classe derivada tem permissão para declarar um membro com o mesmo nome ou assinatura que um membro herdado (que em vigor oculta o membro herdado).

## O tipo de instância

Cada declaração de classe tem um tipo de associado associado ([tipos vinculados e desvinculados](#)), o *tipo de instância*. Para uma declaração de classe genérica, o tipo de instância é formado criando um tipo construído ([tipos construídos](#)) da declaração de tipo, com cada um dos argumentos de tipo fornecidos sendo o parâmetro de tipo correspondente. Como o tipo de instância usa os parâmetros de tipo, ele só pode ser usado quando os parâmetros de tipo estão no escopo; ou seja, dentro da declaração de classe. O tipo de instância é o tipo de `this` para código escrito dentro da declaração de classe. Para classes não genéricas, o tipo de instância é simplesmente a classe declarada. O exemplo a seguir mostra várias declarações de classe junto com seus tipos de instância:

C#

```
class A<T>                                // instance type: A<T>
{
    class B {}                            // instance type: A<T>.B
    class C<U> {}                      // instance type: A<T>.C<U>
}

class D {}                                // instance type: D
```

## Membros de tipos construídos

Os membros não herdados de um tipo construído são obtidos pela substituição, para cada *type\_parameter* na declaração de membro, o *type\_argument* correspondente do

tipo construído. O processo de substituição é baseado no significado semântico das declarações de tipo e não é simplesmente a substituição textual.

Por exemplo, considerando a declaração de classe genérica

```
C#  
  
class Gen<T,U>  
{  
    public T[,] a;  
    public void G(int i, T t, Gen<U,T> gt) {...}  
    public U Prop { get {...} set {...} }  
    public int H(double d) {...}  
}
```

o tipo construído `Gen<int[], IComparable<string>>` tem os seguintes membros:

```
C#  
  
public int[,][] a;  
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}  
public IComparable<string> Prop { get {...} set {...} }  
public int H(double d) {...}
```

O tipo do membro `a` na declaração de classe genérica `Gen` é "matriz bidimensional de `T`", portanto, o tipo do membro `a` no tipo construído acima é "matriz bidimensional de uma matriz unidimensional de `int`", ou `int[,][]`.

Em membros da função de instância, o tipo de `this` é o tipo de instância ([o tipo de instância](#)) da declaração que a contém.

Todos os membros de uma classe genérica podem usar parâmetros de tipo de qualquer classe delimitadora, seja diretamente ou como parte de um tipo construído. Quando um tipo construído específico fechado ([tipos abertos e fechados](#)) é usado em tempo de execução, cada uso de um parâmetro de tipo é substituído pelo argumento de tipo real fornecido para o tipo construído. Por exemplo:

```
C#  
  
class C<V>  
{  
    public V f1;  
    public C<V> f2 = null;  
  
    public C(V x) {  
        this.f1 = x;  
        this.f2 = this;
```

```

    }

class Application
{
    static void Main() {
        C<int> x1 = new C<int>(1);
        Console.WriteLine(x1.f1);           // Prints 1

        C<double> x2 = new C<double>(3.1415);
        Console.WriteLine(x2.f1);           // Prints 3.1415
    }
}

```

## Herança

Uma classe **herda** os membros de seu tipo de classe base direta. Herança significa que uma classe implicitamente contém todos os membros de seu tipo de classe base direta, exceto para construtores de instância, destruidores e construtores estáticos da classe base. Alguns aspectos importantes da herança são:

- A herança é transitiva. Se `C` é derivado de `B` e `B` é derivado de `A`, `C` herda os membros declarados em `B` bem como os membros declarados em `A`.
- Uma classe derivada estende sua classe base direta. Uma classe derivada pode adicionar novos membros aos que ela herda, mas ela não pode remover a definição de um membro herdado.
- Construtores de instância, destruidores e construtores estáticos não são herdados, mas todos os outros membros são, independentemente de sua acessibilidade declarada ([acesso de membro](#)). No entanto, dependendo da acessibilidade declarada, os membros herdados podem não estar acessíveis em uma classe derivada.
- Uma classe derivada pode **ocultar** membros herdados ([ocultando a herança](#)) declarando novos membros com o mesmo nome ou assinatura. No entanto, observe que ocultar um membro herdado não remove esse membro — ele simplesmente torna esse membro inacessível diretamente por meio da classe derivada.
- Uma instância de uma classe contém um conjunto de todos os campos de instância declarados na classe e suas classes base, e uma conversão implícita ([conversões de referência implícita](#)) existe de um tipo de classe derivada para qualquer um de seus tipos de classe base. Assim, uma referência a uma instância de uma classe derivada pode ser tratada como uma referência a uma instância de qualquer uma de suas classes base.

- Uma classe pode declarar métodos, propriedades e indexadores virtuais, e classes derivadas podem substituir a implementação desses membros da função. Isso permite que as classes exibam o comportamento polimórfico em que as ações executadas por uma invocação de membro de função variam dependendo do tipo de tempo de execução da instância por meio da qual esse membro de função é invocado.

O membro herdado de um tipo de classe construída são os membros do tipo de classe base imediato ([classes base](#)), que é encontrado substituindo os argumentos de tipo do tipo construído para cada ocorrência dos parâmetros de tipo correspondentes na especificação de *class\_base*. Esses membros, por sua vez, são transformados substituindo, para cada *type\_parameter* na declaração de membro, o *type\_argument* correspondente da especificação de *class\_base*.

C#

```
class B<U>
{
    public U F(long index) {...}

class D<T>: B<T[]>
{
    public T G(string s) {...}
}
```

No exemplo acima, o tipo construído `D<int>` tem um membro não herdado `public int G(string s)` obtido pela substituição do argumento de tipo `int` para o parâmetro de tipo `T`. `D<int>` também tem um membro herdado da declaração de classe `B`. Esse membro herdado é determinado pela primeira vez que determina o tipo de classe base `B<int[]>` de `D<int>` substituindo `int` para `T` na especificação de classe base `B<T[]>`. Em seguida, como um argumento de tipo para `B`, `int[]` é substituído por `U` no `public U F(long index)`, produzindo o membro herdado `public int[] F(long index)`.

## O novo modificador

Um *class\_member\_declaration* tem permissão para declarar um membro com o mesmo nome ou assinatura de um membro herdado. Quando isso ocorre, o membro da classe derivada é dito para *ocultar* o membro da classe base. Ocultar um membro herdado não é considerado um erro, mas faz com que o compilador emita um aviso. Para suprimir o aviso, a declaração do membro da classe derivada pode incluir um `new` modificador para indicar que o membro derivado deve ocultar o membro base. Este tópico é abordado mais detalhadamente em [ocultar por meio de herança](#).

Se um `new` modificador for incluído em uma declaração que não oculta um membro herdado, um aviso para esse efeito será emitido. Esse aviso é suprimido com a remoção do `new` modificador.

## Modificadores de acesso

Um *class\_member\_declaration* pode ter qualquer um dos cinco tipos possíveis de acessibilidade declarada ([acessibilidade declarada](#)): `public`, `protected internal`, `protected internal` ou `private`. Exceto para a `protected internal` combinação, é um erro de tempo de compilação para especificar mais de um modificador de acesso. Quando um *class\_member\_declaration* não inclui nenhum modificador de acesso, `private` é assumido.

## Tipos constituintes

Os tipos que são usados na declaração de um membro são chamados de tipos constituintes desse membro. Os tipos constituintes possíveis são o tipo de uma constante, campo, propriedade, evento ou indexador, o tipo de retorno de um método ou operador e os tipos de parâmetro de um construtor de método, indexador, operador ou instância. Os tipos constituintes de um membro devem ser pelo menos tão acessíveis quanto o próprio membro ([restrições de acessibilidade](#)).

## Membros estáticos e de instância

Os membros de uma classe são \*membros estáticos \_ ou \_ membros da instância \*. Em geral, é útil considerar os membros estáticos como pertencentes a tipos de classe e membros de instância como pertencentes a objetos (instâncias de tipos de classe).

Quando um campo, método, propriedade, evento, operador ou declaração de Construtor inclui um `static` modificador, ele declara um membro estático. Além disso, uma constante ou declaração de tipo declara implicitamente um membro estático. Os membros estáticos têm as seguintes características:

- Quando um membro estático `M` é referenciado em um *member\_access* ([acesso de membro](#)) do formulário, deve-se `E.M` indicar um tipo contendo `M`. É um erro de tempo de compilação para `E` indicar uma instância.
- Um campo estático identifica exatamente um local de armazenamento a ser compartilhado por todas as instâncias de um determinado tipo de classe fechado. Independentemente de quantas instâncias de um determinado tipo de classe fechada forem criadas, há apenas uma cópia de um campo estático.

- Um membro de função estática (método, propriedade, evento, operador ou Construtor) não opera em uma instância específica e é um erro de tempo de compilação para fazer referência a `this` esse membro de função.

Quando um campo, método, propriedade, evento, indexador, Construtor ou declaração de destruidor não inclui um `static` modificador, ele declara um membro de instância. (Um membro de instância é, às vezes, chamado de membro não estático.) Os membros da instância têm as seguintes características:

- Quando um membro `M` de instância é referenciado em um *member\_access* (acesso de membro) do formulário, deve-se `E.M` `E` indicar uma instância de um tipo que contém `M`. É um erro de tempo de ligação para `E` indicar um tipo.
- Cada instância de uma classe contém um conjunto separado de todos os campos de instância da classe.
- Um membro de função de instância (método, propriedade, indexador, Construtor de instância ou destruidor) opera em uma determinada instância da classe, e essa instância pode ser acessada como `this` (esse acesso).

O exemplo a seguir ilustra as regras para acessar membros estáticos e de instância:

C#

```
class Test
{
    int x;
    static int y;

    void F() {
        x = 1;          // Ok, same as this.x = 1
        y = 1;          // Ok, same as Test.y = 1
    }

    static void G() {
        x = 1;          // Error, cannot access this.x
        y = 1;          // Ok, same as Test.y = 1
    }

    static void Main() {
        Test t = new Test();
        t.x = 1;         // Ok
        t.y = 1;         // Error, cannot access static member through
instance
        Test.x = 1;       // Error, cannot access instance member through
type
        Test.y = 1;       // Ok
    }
}
```

O `F` método mostra que em um membro da função de instância, um *Simple\_name* ([nomes simples](#)) pode ser usado para acessar membros de instância e membros estáticos. O `G` método mostra que em um membro de função estática, é um erro de tempo de compilação para acessar um membro de instância por meio de um *Simple\_name*. O `Main` método mostra que em um *member\_access* ([acesso de membro](#)), os membros da instância devem ser acessados por meio de instâncias e os membros estáticos devem ser acessados por meio de tipos.

## Tipos aninhados

Um tipo declarado em uma declaração de classe ou struct é chamado de um **\*tipo aninhado\***. Um tipo declarado em uma unidade de compilação ou namespace é chamado de um tipo *não aninhado* \*.

No exemplo

```
C#  
  
using System;  
  
class A  
{  
    class B  
    {  
        static void F() {  
            Console.WriteLine("A.B.F");  
        }  
    }  
}
```

Class `B` é um tipo aninhado porque é declarado dentro de Class `A` e Class `A` é um tipo não aninhado porque é declarado em uma unidade de compilação.

## Nome totalmente qualificado

O nome totalmente qualificado ([nomes totalmente qualificados](#)) para um tipo aninhado é `S.N` onde `S` é o nome totalmente qualificado do tipo no qual o tipo `N` é declarado.

## Acessibilidade declarada

Tipos não aninhados podem ter `public` ou `internal` declarar acessibilidade e ter `internal` declarado acessibilidade por padrão. Os tipos aninhados também podem ter

esses formulários de acessibilidade declarados, além de uma ou mais formas adicionais de acessibilidade declarada, dependendo se o tipo recipiente é uma classe ou estrutura:

- Um tipo aninhado declarado em uma classe pode ter qualquer uma das cinco formas de acessibilidade declaradas ( „ public protected internal protected internal ou private ) e, como outros membros da classe, o padrão é private declarado como acessibilidade.
- Um tipo aninhado declarado em um struct pode ter qualquer uma das três formas de acessibilidade declarada ( public , internal ou private ) e, como outros membros de struct, usa private acessibilidade declarada por padrão.

O exemplo

C#

```
public class List
{
    // Private data structure
    private class Node
    {
        public object Data;
        public Node Next;

        public Node(object data, Node next) {
            this.Data = data;
            this.Next = next;
        }
    }

    private Node first = null;
    private Node last = null;

    // Public interface
    public void AddToFront(object o) {...}
    public void AddToBack(object o) {...}
    public object RemoveFromFront() {...}
    public object RemoveFromBack() {...}
    public int Count { get {...} }
}
```

declara uma classe aninhada privada Node .

## Ocultar

Um tipo aninhado pode ocultar (ocultar o nome) um membro base. O new modificador é permitido em declarações de tipo aninhadas para que a ocultação possa ser expressa explicitamente. O exemplo

C#

```
using System;

class Base
{
    public static void M() {
        Console.WriteLine("Base.M");
    }
}

class Derived: Base
{
    new public class M
    {
        public static void F() {
            Console.WriteLine("Derived.M.F");
        }
    }
}

class Test
{
    static void Main() {
        Derived.M.F();
    }
}
```

mostra uma classe aninhada `M` que oculta o método `M` definido em `Base`.

## Este acesso

Um tipo aninhado e seu tipo recipiente não têm uma relação especial com relação a `this_access` ([esse acesso](#)). Especificamente, `this` dentro de um tipo aninhado não pode ser usado para fazer referência a membros de instância do tipo recipiente. Nos casos em que um tipo aninhado precisa de acesso aos membros da instância de seu tipo recipiente, o acesso pode ser fornecido fornecendo o `this` para a instância do tipo recipiente como um argumento de construtor para o tipo aninhado. O exemplo a seguir

C#

```
using System;

class C
{
    int i = 123;

    public void F() {
        Nested n = new Nested(this);
```

```

        n.G();
    }

    public class Nested
    {
        C this_c;

        public Nested(C c) {
            this_c = c;
        }

        public void G() {
            Console.WriteLine(this_c.i);
        }
    }
}

class Test
{
    static void Main() {
        C c = new C();
        c.F();
    }
}

```

mostra essa técnica. Uma instância do `C` cria uma instância do `Nested` e passa seu próprio `this` construtor para o `Nested` para fornecer acesso subsequente aos `C` membros da instância do.

## Acesso a membros privados e protegidos do tipo recipiente

Um tipo aninhado tem acesso a todos os membros que são acessíveis para seu tipo recipiente, incluindo membros do tipo recipiente que têm `private` e têm `protected` acessibilidade declarada. O exemplo

```

C#

using System;

class C
{
    private static void F() {
        Console.WriteLine("C.F");
    }

    public class Nested
    {
        public static void G() {
            F();
        }
    }
}

```

```

        }
    }

class Test
{
    static void Main() {
        C.Nested.G();
    }
}

```

mostra uma classe `C` que contém uma classe aninhada `Nested`. Em `Nested`, o método `G` chama o método estático `F` definido em `C` e `F` tem acessibilidade definida como particular.

Um tipo aninhado também pode acessar membros protegidos definidos em um tipo base de seu tipo recipiente. No exemplo

```

C#

using System;

class Base
{
    protected void F() {
        Console.WriteLine("Base.F");
    }
}

class Derived: Base
{
    public class Nested
    {
        public void G() {
            Derived d = new Derived();
            d.F();           // ok
        }
    }
}

class Test
{
    static void Main() {
        Derived.Nested n = new Derived.Nested();
        n.G();
    }
}

```

a classe aninhada `Derived.Nested` acessa o método protegido `F` definido na `Derived` classe base do, `Base`, chamando por meio de uma instância do `Derived`.

## Tipos aninhados em classes genéricas

Uma declaração de classe genérica pode conter declarações de tipo aninhadas. Os parâmetros de tipo da classe delimitadora podem ser usados dentro dos tipos aninhados. Uma declaração de tipo aninhado pode conter parâmetros de tipo adicionais que se aplicam somente ao tipo aninhado.

Cada declaração de tipo contida em uma declaração de classe genérica é implicitamente uma declaração de tipo genérico. Ao gravar uma referência a um tipo aninhado em um tipo genérico, o tipo construído que o contém, incluindo seus argumentos de tipo, deve ser nomeado. No entanto, de dentro da classe externa, o tipo aninhado pode ser usado sem qualificação; o tipo de instância da classe externa pode ser usado implicitamente ao construir o tipo aninhado. O exemplo a seguir mostra três maneiras corretas diferentes de se referir a um tipo construído criado a partir de `Inner`; as duas primeiras são equivalentes:

C#

```
class Outer<T>
{
    class Inner<U>
    {
        public static void F(T t, U u) {...}

        static void F(T t) {
            Outer<T>.Inner<string>.F(t, "abc");           // These two statements
        have
            Inner<string>.F(t, "abc");                  // the same effect

            Outer<int>.Inner<string>.F(3, "abc");       // This type is different

            Outer.Inner<string>.F(t, "abc");           // Error, Outer needs type
        arg
        }
    }
}
```

Embora seja um estilo de programação inadequado, um parâmetro de tipo em um tipo aninhado pode ocultar um membro ou parâmetro de tipo declarado no tipo externo:

C#

```
class Outer<T>
{
    class Inner<T>          // Valid, hides Outer's T
    {
        public T t;          // Refers to Inner's T
    }
}
```

```
}
```

## Nomes de membros reservados

Para facilitar a implementação de tempo de execução C# subjacente, para cada declaração de membro de origem que seja uma propriedade, um evento ou um indexador, a implementação deve reservar duas assinaturas de método com base no tipo de declaração de membro, seu nome e seu tipo. É um erro de tempo de compilação para um programa declarar um membro cuja assinatura corresponde a uma dessas assinaturas reservadas, mesmo que a implementação de tempo de execução subjacente não faça uso dessas reservas.

Os nomes reservados não introduzem declarações, portanto, eles não participam da pesquisa de membros. No entanto, as assinaturas de método reservado associadas de uma declaração participam da herança ([herança](#)) e podem ser ocultadas com o `new` modificador ([o novo modificador](#)).

A reserva desses nomes serve para três finalidades:

- Para permitir que a implementação subjacente use um identificador comum como um nome de método para obter ou definir o acesso ao recurso de linguagem C#.
- Para permitir que outras linguagens interoperem usando um identificador comum como um nome de método para obter ou definir o acesso ao recurso de linguagem C#.
- Para ajudar a garantir que a origem aceita por um compilador em conformidade é aceita por outro, tornando-se as especificidades dos nomes de membro reservados consistentes em todas as implementações em C#.

A declaração de um destruidor ([destruidores](#)) também faz com que uma assinatura seja reservada ([nomes de membros reservados para destruidores](#)).

## Nomes de membro reservados para propriedades

Para uma propriedade `P` ([Propriedades](#)) do tipo `T`, as seguintes assinaturas são reservadas:

```
C#
```

```
T get_P();  
void set_P(T value);
```

Ambas as assinaturas são reservadas, mesmo que a propriedade seja somente leitura ou somente gravação.

No exemplo

```
C#  
  
using System;  
  
class A  
{  
    public int P {  
        get { return 123; }  
    }  
}  
  
class B: A  
{  
    new public int get_P() {  
        return 456;  
    }  
  
    new public void set_P(int value) {  
    }  
}  
  
class Test  
{  
    static void Main() {  
        B b = new B();  
        A a = b;  
        Console.WriteLine(a.P);  
        Console.WriteLine(b.P);  
        Console.WriteLine(b.get_P());  
    }  
}
```

uma classe `A` define uma propriedade somente leitura `P`, reservando, assim, assinaturas para os `get_P` `set_P` métodos e. Uma classe `B` deriva de `A` e oculta essas duas assinaturas reservadas. O exemplo produz a saída:

```
Console  
  
123  
123  
456
```

## Nomes de membro reservados para eventos

Para um evento `E` (eventos) do tipo delegate `T`, as seguintes assinaturas são reservadas:

C#

```
void add_E(T handler);
void remove_E(T handler);
```

## Nomes de membro reservados para indexadores

Para um indexador (indexadores) do tipo `T` com a lista de parâmetros `L`, as seguintes assinaturas são reservadas:

C#

```
T get_Item(L);
void set_Item(L, T value);
```

Ambas as assinaturas são reservadas, mesmo que o indexador seja somente leitura ou somente gravação.

Além disso, o nome do membro `Item` é reservado.

## Nomes de membro reservados para destruidores

Para uma classe que contém um destruidor (destruidores), a assinatura a seguir é reservada:

C#

```
void Finalize();
```

# Constantes

Uma \*constante \_ é um membro de classe que representa um valor constante: um valor que pode ser calculado em tempo de compilação. Um \_constant\_declaration \* introduz uma ou mais constantes de um determinado tipo.

antlr

```
constant_declaration
: attributes? constant_modifier* 'const' type constant_declarators ';'
```

```

;

constant_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
;

constant_declarators
: constant_declarator (',' constant_declarator)*
;

constant_declarator
: identifier '=' constant_expression
;

```

Um *constant\_declaration* pode incluir um conjunto de *atributos* ([atributos](#)), um `new` modificador ([o novo modificador](#)) e uma combinação válida dos quatro modificadores de acesso ([modificadores de acesso](#)). Os atributos e os modificadores se aplicam a todos os membros declarados pelo *constant\_declaration*. Embora constantes sejam consideradas membros estáticos, uma *constant\_declaration* não requer nem permite um `static` modificador. É um erro para que o mesmo modificador apareça várias vezes em uma declaração de constante.

O *tipo* de um *constant\_declaration* especifica o tipo dos membros introduzidos pela declaração. O tipo é seguido por uma lista de *constant\_declarator*s, cada um deles apresentando um novo membro. Uma *constant\_declarator* consiste em um *identificador* que nomeia o membro, seguido por um `=` token "", seguido por um *constant\_expression* ([expressões constantes](#)) que fornece o valor do membro.

O *tipo* especificado em uma declaração de constante deve ser `sbyte` `byte` `short` `ushort` `int` `uint` `long` `ulong` `char` `float` `double` `decimal` `bool` `string`, um *enum\_type* ou um *reference\_type*. Cada *constant\_expression* deve produzir um valor do tipo de destino ou de um tipo que possa ser convertido para o tipo de destino por uma conversão implícita ([conversões implícitas](#)).

O *tipo* de uma constante deve ser pelo menos tão acessível quanto a própria constante ([restrições de acessibilidade](#)).

O valor de uma constante é obtido em uma expressão usando um *Simple\_name* ([nomes simples](#)) ou uma *member\_access* ([acesso de membro](#)).

Uma constante pode participar de uma *constant\_expression*. Portanto, uma constante pode ser usada em qualquer construção que exija uma *constant\_expression*. Exemplos

dessas construções incluem `case` Rótulos, `goto case` instruções, declarações de `enum` membro, atributos e outras declarações de constantes.

Conforme descrito em [expressões constantes](#), um `constant_expression` é uma expressão que pode ser totalmente avaliada em tempo de compilação. Como a única maneira de criar um valor não nulo de um `reference_type` diferente de `string` é aplicar o `new` operador e, como o `new` operador não é permitido em um `constant_expression`, o único valor possível para constantes de `reference_type`s diferentes de `string` é `null`.

Quando é desejado um nome simbólico para um valor constante, mas quando o tipo desse valor não é permitido em uma declaração de constante, ou quando o valor não pode ser computado em tempo de compilação por um `constant_expression`, um `readonly` campo ([campos ReadOnly](#)) pode ser usado em vez disso.

Uma declaração constante que declara várias constantes é equivalente a várias declarações de constantes únicas com os mesmos atributos, modificadores e tipo. Por exemplo,

C#

```
class A
{
    public const double X = 1.0, Y = 2.0, Z = 3.0;
```

é equivalente a

C#

```
class A
{
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
}
```

As constantes podem depender de outras constantes no mesmo programa, contanto que as dependências não sejam de natureza circular. O compilador organiza automaticamente para avaliar as declarações constantes na ordem apropriada. No exemplo

C#

```
class A
{
```

```

    public const int X = B.Z + 1;
    public const int Y = 10;
}

class B
{
    public const int Z = A.Y + 1;
}

```

o compilador primeiro avalia `A.Y` depois avalia `B.Z` e, finalmente, avalia `A.X`, produzindo os valores `10`, `11` e `12`. Declarações constantes podem depender de constantes de outros programas, mas essas dependências só são possíveis em uma direção. Fazendo referência ao exemplo acima, se `A` e `B` foram declarados em programas separados, seria possível `A.X` depender `B.Z`, mas `B.Z`, em seguida, não depender simultaneamente `A.Y`.

## Campos

Um `*Field _` é um membro que representa uma variável associada a um objeto ou uma classe. Um `_field_declaration *` introduz um ou mais campos de um determinado tipo.

```

antlr

field_declaration
: attributes? field_modifier* type variable_declarators ';'*
;

field_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'static'
| 'readonly'
| 'volatile'
| field_modifier_unsafe
;

variable_declarators
: variable_declarator (',' variable_declarator)*
;

variable_declarator
: identifier ('=' variable_initializer)?
;

variable_initializer
: expression
;
```

```
| array_initializer  
| ;
```

Um *field\_declaration* pode incluir um conjunto de *atributos* ([atributos](#)), um `new` modificador ([o novo modificador](#)), uma combinação válida dos quatro modificadores de acesso ([modificadores de acesso](#)) e um `static` modificador ([campos estáticos e de instância](#)). Além disso, um *field\_declaration* pode incluir um `readonly` modificador ([campos somente leitura](#)) ou um `volatile` modificador ([campos voláteis](#)), mas não ambos. Os atributos e os modificadores se aplicam a todos os membros declarados pelo *field\_declaration*. É um erro para que o mesmo modificador apareça várias vezes em uma declaração de campo.

O *tipo* de um *field\_declaration* especifica o tipo dos membros introduzidos pela declaração. O tipo é seguido por uma lista de *variable\_declarator*s, cada um deles apresentando um novo membro. Uma *variable\_declarator* consiste em um *identificador* que nomeia esse membro, opcionalmente seguido por um `=` token "" e um *variable\_initializer* ([inicializadores de variável](#)) que fornece o valor inicial desse membro.

O *tipo* de um campo deve ser pelo menos tão acessível quanto o próprio campo ([restrições de acessibilidade](#)).

O valor de um campo é obtido em uma expressão usando um *Simple\_name* ([nomes simples](#)) ou um *member\_access* ([acesso de membro](#)). O valor de um campo não `ReadOnly` é modificado usando uma *atribuição* ([operadores de atribuição](#)). O valor de um campo não `ReadOnly` pode ser obtido e modificado usando o incremento de sufixo e os operadores de diminuição ([operadores de incremento e diminuição de sufixo](#)) e os operadores de incremento de prefixo e decréscimo ([incremento de prefixo e diminuição de operadores](#)).

Uma declaração de campo que declara vários campos é equivalente a várias declarações de campos únicos com os mesmos atributos, modificadores e tipo. Por exemplo,

```
C#  
  
class A  
{  
    public static int X = 1, Y, Z = 100;  
}
```

é equivalente a

```
C#
```

```
class A
{
    public static int X = 1;
    public static int Y;
    public static int Z = 100;
}
```

## Campos estáticos e de instância

Quando uma declaração de campo inclui um `static` modificador, os campos introduzidos pela declaração são **\*campos estáticos\***. Quando nenhum `static` modificador está presente, os campos introduzidos pela declaração são *campos de instância*. Campos estáticos e campos de instância são dois dos vários tipos de variáveis ([variáveis](#)) com suporte do C# e, às vezes, são chamados de *variáveis estáticas* e *variáveis de instância* \*, respectivamente.

Um campo estático não faz parte de uma instância específica; em vez disso, ele é compartilhado entre todas as instâncias de um tipo fechado ([tipos abertos e fechados](#)). Não importa quantas instâncias de um tipo de classe fechada são criadas, há apenas uma cópia de um campo estático para o domínio de aplicativo associado.

Por exemplo:

```
C#
class C<V>
{
    static int count = 0;

    public C() {
        count++;
    }

    public static int Count {
        get { return count; }
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>();
        Console.WriteLine(C<int>.Count);           // Prints 1

        C<double> x2 = new C<double>();
        Console.WriteLine(C<int>.Count);           // Prints 1

        C<int> x3 = new C<int>();
```

```
        Console.WriteLine(C<int>.Count);      // Prints 2
    }
}
```

Um campo de instância pertence a uma instância. Especificamente, cada instância de uma classe contém um conjunto separado de todos os campos de instância dessa classe.

Quando um campo é referenciado em um *member\_access* ([acesso de membro](#)) do formulário `E.M`, se `M` for um campo estático, `E` deverá indicar um tipo contendo `M`, e se `M` for um campo de instância, e deverá indicar uma instância de um tipo que contém `M`.

As diferenças entre os membros estático e de instância são discutidas mais detalhadamente em [membros estáticos e de instância](#).

## Campos somente leitura

Quando um *field\_declaration* inclui um `readonly` modificador, os campos introduzidos pela declaração são **campos somente leitura**. Atribuições diretas a campos `ReadOnly` só podem ocorrer como parte dessa declaração ou em um construtor de instância ou construtor estático na mesma classe. (Um campo `ReadOnly` pode ser atribuído a várias vezes nesses contextos.) Especificamente, as atribuições diretas a um `readonly` campo são permitidas apenas nos seguintes contextos:

- Na *variable\_declarator* que introduz o campo (incluindo um *variable\_initializer* na declaração).
- Para um campo de instância, nos construtores de instância da classe que contém a declaração de campo; para um campo estático, no construtor estático da classe que contém a declaração de campo. Esses são também os únicos contextos nos quais é válido passar um `readonly` campo como um `out` `ref` parâmetro ou.

A tentativa de atribuir a um `readonly` campo ou passá-lo como `out` um `ref` parâmetro ou em qualquer outro contexto é um erro em tempo de compilação.

## Usando campos somente leitura estáticos para constantes

Um `static readonly` campo é útil quando um nome simbólico para um valor constante é desejado, mas quando o tipo do valor não é permitido em uma `const` declaração ou quando o valor não pode ser computado em tempo de compilação. No exemplo

```

public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte red, green, blue;

    public Color(byte r, byte g, byte b) {
        red = r;
        green = g;
        blue = b;
    }
}

```

os `Black` Membros,, `White` `Red` , `Green` e `Blue` não podem ser declarados como `const` Membros porque seus valores não podem ser computados em tempo de compilação. No entanto, declará-las, `static readonly` em vez disso, tem muito o mesmo efeito.

## Controle de versão de constantes e campos somente leitura estáticos

Os campos `Constants` e `ReadOnly` têm semântica de versão binária diferente. Quando uma expressão faz referência a uma constante, o valor da constante é obtido em tempo de compilação, mas quando uma expressão faz referência a um campo `ReadOnly`, o valor do campo não é obtido até o tempo de execução. Considere um aplicativo que consiste em dois programas separados:

C#

```

using System;

namespace Program1
{
    public class Utils
    {
        public static readonly int X = 1;
    }
}

namespace Program2
{
    class Test
    {
        static void Main() {
            Console.WriteLine(Program1.Utils.X);
        }
    }
}

```

```
        }  
    }  
}
```

Os `Program1` `Program2` namespaces e denotam dois programas que são compilados separadamente. Como `Program1.Utils.X` é declarado como um campo somente leitura estático, a saída do valor pela `Console.WriteLine` instrução não é conhecida em tempo de compilação, mas é obtida em tempo de execução. Portanto, se o valor de `x` for alterado e `Program1` for recompilado, a `Console.WriteLine` instrução produzirá o novo valor, mesmo se `Program2` não for recompilado. No entanto, era `x` uma constante, o valor de `x` teria sido obtido no momento em `Program2` que foi compilado e permaneceria inalterado por alterações no `Program1` até que `Program2` seja recompilado.

## Campos voláteis

Quando um *field declaration* inclui um `volatile` modificador, os campos introduzidos por essa declaração são **campos voláteis**.

Para campos não voláteis, as técnicas de otimização que reordenam instruções podem levar a resultados inesperados e imprevisíveis em programas multi-threaded que acessam campos sem sincronização, como o fornecido pelo *lock statement* ([a instrução Lock](#)). Essas otimizações podem ser executadas pelo compilador, pelo sistema de tempo de execução ou por hardware. Para campos voláteis, essas otimizações de reordenação são restritas:

- Uma leitura de um campo volátil é chamada de **leitura volátil**. Uma leitura volátil tem "adquirir semântica"; ou seja, é garantido que ocorra antes de qualquer referência à memória que ocorra depois dela na sequência de instruções.
- Uma gravação de um campo volátil é chamada de **gravação volátil**. Uma gravação volátil tem "semântica de liberação"; ou seja, é garantido que ocorra após qualquer referência de memória antes da instrução de gravação na sequência de instruções.

Essas restrições garantem que todos os threads observem as gravações voláteis executadas por qualquer outro thread na ordem em que elas foram realizadas. Uma implementação de conformidade não é necessária para fornecer uma única ordenação total de gravações voláteis, como visto de todos os threads de execução. O tipo de um campo volátil deve ser um dos seguintes:

- Um *reference\_type*.
- O tipo `byte` `sbyte` `short` `ushort` `int` `uint` `char` `float` , `bool` , `System.IntPtr` ou `System.UIntPtr` .

- Um `enum_type` ter um tipo de base de enumeração de..., `byte` `sbyte` `short` `ushort` `int` ou `uint`.

O exemplo

C#

```
using System;
using System.Threading;

class Test
{
    public static int result;
    public static volatile bool finished;

    static void Thread2() {
        result = 143;
        finished = true;
    }

    static void Main() {
        finished = false;

        // Run Thread2() in a new thread
        new Thread(new ThreadStart(Thread2)).Start();

        // Wait for Thread2 to signal that it has a result by setting
        // finished to true.
        for (;;) {
            if (finished) {
                Console.WriteLine("result = {0}", result);
                return;
            }
        }
    }
}
```

produz a saída:

Console

```
result = 143
```

Neste exemplo, o método `Main` inicia um novo thread que executa o método `Thread2`. Esse método armazena um valor em um campo não volátil chamado `result` e, em seguida, armazena `true` no campo volátil `finished`. O thread principal aguarda que o campo `finished` seja definido como `true`, lê o campo `result`. Como `finished` foi declarado `volatile`, o thread principal deve ler o valor `143` do campo

`result`. Se o campo `finished` não tivesse sido declarado `volatile`, seria permitido que o repositório `result` fosse visível para o thread principal após o repositório `finished` e, portanto, para o thread principal ler o valor `0` do campo `result`. Declarar `finished` como um `volatile` campo impede qualquer inconsistência.

## Inicialização de campo

O valor inicial de um campo, seja um campo estático ou um campo de instância, é o valor padrão ([valores padrão](#)) do tipo do campo. Não é possível observar o valor de um campo antes que essa inicialização padrão tenha ocorrido, e um campo nunca é "não inicializado". O exemplo

C#

```
using System;

class Test
{
    static bool b;
    int i;

    static void Main() {
        Test t = new Test();
        Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}
```

produz a saída

Console

```
b = False, i = 0
```

Porque `b` e `i` são inicializados automaticamente para valores padrão.

## Inicializadores de variável

Declarações de campo podem incluir *variable\_initializer*s. Para campos estáticos, inicializadores de variáveis correspondem a instruções de atribuição que são executadas durante a inicialização da classe. Para campos de instância, inicializadores de variáveis correspondem a instruções de atribuição que são executadas quando uma instância da classe é criada.

O exemplo

C#

```
using System;

class Test
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";

    static void Main() {
        Test a = new Test();
        Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
    }
}
```

produz a saída

Console

```
x = 1.4142135623731, i = 100, s = Hello
```

Porque uma atribuição `x` ocorre quando inicializadores de campo estáticos são executados e atribuições e `i` `s` ocorrem quando os inicializadores de campo de instância são executados.

A inicialização de valor padrão descrita na [inicialização do campo](#) ocorre para todos os campos, incluindo campos que têm inicializadores variáveis. Assim, quando uma classe é inicializada, todos os campos estáticos nessa classe são inicializados primeiro com seus valores padrão e, em seguida, os inicializadores de campo estático são executados em ordem textual. Da mesma forma, quando uma instância de uma classe é criada, todos os campos de instância nessa instância são inicializados primeiro com seus valores padrão e, em seguida, os inicializadores de campo de instância são executados na ordem textual.

É possível que campos estáticos com inicializadores de variáveis sejam observados em seu estado de valor padrão. No entanto, isso é fortemente desencorajado como uma questão de estilo. O exemplo

C#

```
using System;

class Test
{
    static int a = b + 1;
    static int b = a + 1;
```

```
    static void Main() {
        Console.WriteLine("a = {0}, b = {1}", a, b);
    }
}
```

exibe esse comportamento. Apesar das definições circulares de `a` e `b`, o programa é válido. Isso resulta na saída

Console

```
a = 1, b = 2
```

Porque os campos estáticos `a` e `b` são inicializados para `0` (o valor padrão de `int`) antes de seus inicializadores serem executados. Quando o inicializador for `a` executado, o valor de `b` será zero e, portanto, `a` será inicializado para `1`. Quando o inicializador for `b` executado, o valor de `a` já será `1` e, portanto, `b` será inicializado para `2`.

## Inicialização de campo estático

Os inicializadores de variável de campo estático de uma classe correspondem a uma sequência de atribuições que são executadas na ordem textual na qual aparecem na declaração de classe. Se existir um construtor estático ([construtores estáticos](#)) na classe, a execução dos inicializadores de campo estáticos ocorrerá imediatamente antes da execução desse construtor estático. Caso contrário, os inicializadores de campo estáticos são executados em uma hora dependente da implementação antes do primeiro uso de um campo estático dessa classe. O exemplo

C#

```
using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}

class A
{
```

```
    public static int X = Test.F("Init A");
}

class B
{
    public static int Y = Test.F("Init B");
}
```

pode produzir a saída:

Console

```
Init A
Init B
1 1
```

ou a saída:

Console

```
Init B
Init A
1 1
```

como a execução do `X` inicializador e `Y` do inicializador do pode ocorrer em qualquer ordem; eles só são restritos a ocorrer antes das referências a esses campos. No entanto, no exemplo:

C#

```
using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}

class A
{
    static A() {}

    public static int X = Test.F("Init A");
```

```
}
```

```
class B
```

```
{
```

```
    static B() {}
```

```
    public static int Y = Test.F("Init B");
```

```
}
```

a saída deve ser:

Console

```
Init B  
Init A  
1 1
```

como as regras para quando os construtores estáticos são executados (conforme definido em [construtores estáticos](#)) fornecem esse `B` construtor estático (e, portanto, `B` inicializadores de campo estáticos) devem ser executados antes `A` do construtor estático e dos inicializadores de campo.

## Inicialização do campo de instância

Os inicializadores de variável de campo de instância de uma classe correspondem a uma sequência de atribuições que são executadas imediatamente após a entrada para qualquer um dos construtores de instância ([inicializadores de Construtor](#)) dessa classe. Os inicializadores de variável são executados na ordem textual em que aparecem na declaração de classe. O processo de criação e inicialização da instância de classe é descrito mais detalhadamente em [construtores de instância](#).

Um inicializador de variável para um campo de instância não pode fazer referência à instância que está sendo criada. Portanto, é um erro de tempo de compilação a ser referenciado `this` em um inicializador de variável, pois é um erro de tempo de compilação para um inicializador de variável referenciar qualquer membro de instância por meio de um *Simple\_name*. No exemplo

C#

```
class A
```

```
{
```

```
    int x = 1;
```

```
    int y = x + 1;           // Error, reference to instance member of this
```

```
}
```

o inicializador de variável para `y` resulta em um erro de tempo de compilação porque faz referência a um membro da instância que está sendo criada.

# Métodos

Um \*método\_\* é um membro que implementa uma computação ou ação que pode ser executada por um objeto ou uma classe. Os métodos são declarados usando \_method\_declaration\_\*'s:

```
antlr

method_declarator
: method_header method_body
;

method_header
: attributes? method_modifier* 'partial'? return_type member_name
type_parameter_list?
(' formal_parameter_list? ') type_parameter_constraints_clause*
;

method_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'static'
| 'virtual'
| 'sealed'
| 'override'
| 'abstract'
| 'extern'
| 'async'
| method_modifier_unsafe
;

return_type
: type
| 'void'
;

member_name
: identifier
| interface_type '.' identifier
;

method_body
: block
| '=>' expression ';'
```

```
| ';' ;
```

Uma *method\_declaration* pode incluir um conjunto de *atributos* (*atributos*) e uma combinação válida dos quatro modificadores de acesso (*modificadores de acesso*), o `new` (*o novo modificador*), `static` (*métodos estáticos e de instância*), `virtual` (*métodos virtuais*), (*métodos de override substituição*), (*métodos sealed lacrados*), os `abstract` modificadores (*métodos abstratos*) e `extern` (*métodos externos*).

Uma declaração tem uma combinação válida de modificadores se todas as seguintes opções forem verdadeiras:

- A declaração inclui uma combinação válida de modificadores de acesso (*modificadores de acesso*).
- A declaração não inclui o mesmo modificador várias vezes.
- A declaração inclui, no máximo, um dos seguintes modificadores: `static` , `virtual` e `override` .
- A declaração inclui, no máximo, um dos seguintes modificadores: `new` e `override` .
- Se a declaração incluir o `abstract` modificador, a declaração não incluirá nenhum dos seguintes modificadores: `static` , `virtual` `sealed` ou `extern` .
- Se a declaração incluir o `private` modificador, a declaração não incluirá nenhum dos seguintes modificadores: `virtual` , `override` ou `abstract` .
- Se a declaração incluir o `sealed` modificador, a declaração também incluirá o `override` modificador.
- Se a declaração incluir o `partial` modificador, ele não incluirá nenhum dos seguintes modificadores: `new` `public` `protected` `internal` `private` `virtual` `sealed` `override` `abstract` OU `extern` .

Um método que tem o `async` modificador é uma função assíncrona e segue as regras descritas em [funções assíncronas](#).

O *return\_type* de uma declaração de método especifica o tipo do valor calculado e retornado pelo método. O *return\_type* é `void` se o método não retornar um valor. Se a declaração incluir o `partial` modificador, o tipo de retorno deverá ser `void` .

O *member\_name* especifica o nome do método. A menos que o método seja uma implementação de membro de interface explícita ([implementações explícitas de membro de interface](#)), o *member\_name* é simplesmente um *identificador*. Para uma implementação de membro de interface explícita, a *member\_name* consiste em um *interface\_type* seguido por um " ." e um *identificador*.

O `type_parameter_list` opcional especifica os parâmetros de tipo do método ([parâmetros de tipo](#)). Se um `type_parameter_list` for especificado, o método será um [\\*método genérico](#). Se o método tiver um `extern` modificador, um `_type_parameter_list *` não poderá ser especificado.

O `formal_parameter_list` opcional especifica os parâmetros do método ([parâmetros do método](#)).

Os `type_parameter_constraints_clause` opcionais especificam restrições em parâmetros de tipo individuais ([restrições de parâmetro de tipo](#)) e só podem ser especificados se um `type_parameter_list` também for fornecido e o método não tiver um `override` modificador.

A `return_type` e cada um dos tipos referenciados na `formal_parameter_list` de um método devem ser pelo menos tão acessíveis quanto o próprio método ([restrições de acessibilidade](#)).

O `method_body` é um ponto-e-vírgula, um [\*\*corpo de instrução\*\*](#) ou um [\*\*corpo de expressão\*\*](#). Um corpo de instrução consiste em um `_block *`, que especifica as instruções a serem executadas quando o método é invocado. Um corpo de expressão consiste em `=>` seguido por uma `expressão` e um ponto e vírgula e denota uma única expressão a ser executada quando o método é invocado.

Para `abstract` `extern` métodos e, a `method_body` consiste apenas de um ponto e vírgula. Para `partial` métodos, a `method_body` pode consistir em um ponto e vírgula, um corpo de bloco ou um corpo de expressão. Para todos os outros métodos, o `method_body` é um corpo de bloco ou um corpo de expressão.

Se o `method_body` consiste em um ponto e vírgula, a declaração pode não incluir o `async` modificador.

O nome, a lista de parâmetros de tipo e a lista de parâmetros formais de um método definem a assinatura ([assinaturas e sobrecarga](#)) do método. Especificamente, a assinatura de um método consiste em seu nome, o número de parâmetros de tipo e o número, os modificadores e os tipos de seus parâmetros formais. Para essas finalidades, qualquer parâmetro de tipo do método que ocorre no tipo de um parâmetro formal é identificado não por seu nome, mas por sua posição ordinal na lista de argumentos de tipo do método. O tipo de retorno não faz parte da assinatura de um método, nem os nomes dos parâmetros de tipo ou dos parâmetros formais.

O nome de um método deve ser diferente dos nomes de todos os outros não-métodos declarados na mesma classe. Além disso, a assinatura de um método deve diferir das assinaturas de todos os outros métodos declarados na mesma classe, e dois métodos

declarados na mesma classe podem não ter assinaturas que diferem exclusivamente por `ref` e `out`.

Os `type_parameter`s do método estão no escopo durante o `method_declaration` e podem ser usados para formar tipos em todo esse escopo em `return_type`, `method_body` e `type_parameter_constraints_clause`s, mas não em `atributos`.

Todos os parâmetros formais e parâmetros de tipo devem ter nomes diferentes.

## Parâmetros do método

Os parâmetros de um método, se houver, são declarados pelo `formal_parameter_list` do método.

```
antlr

formal_parameter_list
    : fixed_parameters
    | fixed_parameters ',' parameter_array
    | parameter_array
    ;

fixed_parameters
    : fixed_parameter (',' fixed_parameter)*
    ;

fixed_parameter
    : attributes? parameter_modifier? type identifier default_argument?
    ;

default_argument
    : '=' expression
    ;

parameter_modifier
    : 'ref'
    | 'out'
    | 'this'
    ;

parameter_array
    : attributes? 'params' array_type identifier
    ;
```

A lista de parâmetros formais consiste em um ou mais parâmetros separados por vírgulas dos quais apenas o último pode ser um `parameter_array`.

Um *fixed\_parameter* consiste em um conjunto opcional de *atributos* ([atributos](#)), um opcional `ref`, `out` ou `this` modificador, um *tipo*, um *identificador* e um *default\_argument* opcional. Cada *fixed\_parameter* declara um parâmetro do tipo fornecido com o nome fornecido. O `this` modificador designa o método como um método de extensão e só é permitido no primeiro parâmetro de um método estático. Os métodos de extensão são descritos mais detalhadamente em [métodos de extensão](#).

Um *fixed\_parameter* com um *default\_argument* é conhecido como um **\*parâmetro opcional**, enquanto um *fixed\_parameter* \* sem um *default\_argument* é um **parâmetro \* obrigatório**. Um parâmetro necessário pode não aparecer após um parâmetro opcional em um `_formal_parameter_list` \*.

Um `ref` `out` parâmetro ou não pode ter um *default\_argument*. A expressão em uma *default\_argument* deve ser uma das seguintes:

- um *constant\_expression*
- uma expressão do formulário `new S()` em que `S` é um tipo de valor
- uma expressão do formulário `default(S)` em que `S` é um tipo de valor

A expressão deve ser conversível implicitamente por uma identidade ou conversão anulável para o tipo do parâmetro.

Se os parâmetros opcionais ocorrerem em uma declaração de método parcial de implementação ([métodos parciais](#)), uma implementação de membro de interface explícita ([implementações explícitas de membro de interface](#)) ou em uma declaração de indexador de parâmetro único ([indexadores](#)) o compilador deve dar um aviso, já que esses membros nunca podem ser invocados de forma a permitir que os argumentos sejam omitidos.

Um *parameter\_array* consiste em um conjunto opcional de *atributos* ([atributos](#)), um `params` modificador, um *array\_type* e um *identificador*. Uma matriz de parâmetros declara um único parâmetro do tipo de matriz fornecido com o nome fornecido. O *array\_type* de uma matriz de parâmetros deve ser um tipo de matriz de dimensão única ([tipos de matriz](#)). Em uma invocação de método, uma matriz de parâmetros permite que um único argumento do tipo de matriz fornecido seja especificado ou que zero ou mais argumentos do tipo de elemento de matriz sejam especificados. Matrizes de parâmetros são descritas em mais detalhes em [matrizes de parâmetros](#).

Um *parameter\_array* pode ocorrer após um parâmetro opcional, mas não pode ter um valor padrão – a omissão de argumentos para uma *parameter\_array* resultaria na criação de uma matriz vazia.

O exemplo a seguir ilustra os diferentes tipos de parâmetros:

C#

```
public void M(
    ref int      i,
    decimal      d,
    bool         b = false,
    bool?        n = false,
    string       s = "Hello",
    object       o = null,
    T            t = default(T),
    params int[] a
) { }
```

No *formal\_parameter\_list* para `M`, `i` é um parâmetro de referência necessário, `d` é um parâmetro de valor obrigatório, `b`, `s`, `o` e `t` são parâmetros de valor opcional e `a` é uma matriz de parâmetros.

Uma declaração de método cria um espaço de declaração separado para parâmetros, parâmetros de tipo e variáveis locais. Os nomes são introduzidos nesse espaço de declaração pela lista de parâmetros de tipo e pela lista de parâmetros formais do método e por declarações de variáveis locais no *bloco* do método. É um erro para dois membros de um espaço de declaração de método ter o mesmo nome. É um erro para o espaço de declaração de método e o espaço de declaração de variável local de um espaço de declaração aninhada para conter elementos com o mesmo nome.

Uma invocação de método ([invocações de método](#)) cria uma cópia, específica para essa invocação, dos parâmetros formais e das variáveis locais do método, e a lista de argumentos da invocação atribui valores ou referências de variáveis aos parâmetros formais recém-criados. Dentro do *bloco* de um método, os parâmetros formais podem ser referenciados por seus identificadores em *Simple\_name* expressões ([nomes simples](#)).

Há quatro tipos de parâmetros formais:

- Parâmetros de valor, que são declarados sem nenhum modificador.
- Parâmetros de referência, que são declarados com o `ref` modificador.
- Parâmetros de saída, que são declarados com o `out` modificador.
- Matrizes de parâmetros, que são declaradas com o `params` modificador.

Conforme descrito em [assinaturas e sobrecargas](#), os `ref` `out` modificadores e são parte da assinatura de um método, mas o `params` modificador não é.

## Parâmetros de valor

Um parâmetro declarado sem nenhum modificador é um parâmetro de valor. Um parâmetro de valor corresponde a uma variável local que obtém seu valor inicial do argumento correspondente fornecido na invocação do método.

Quando um parâmetro formal é um parâmetro de valor, o argumento correspondente em uma invocação de método deve ser uma expressão que é implicitamente conversível ([conversões implícitas](#)) para o tipo de parâmetro formal.

Um método tem permissão para atribuir novos valores a um parâmetro de valor. Essas atribuições afetam apenas o local de armazenamento local representado pelo parâmetro value — elas não têm nenhum efeito sobre o argumento real fornecido na invocação do método.

## Parâmetros de referência

Um parâmetro declarado com um `ref` modificador é um parâmetro de referência. Ao contrário de um parâmetro de valor, um parâmetro de referência não cria um novo local de armazenamento. Em vez disso, um parâmetro de referência representa o mesmo local de armazenamento que a variável fornecida como o argumento na invocação do método.

Quando um parâmetro formal é um parâmetro de referência, o argumento correspondente em uma invocação de método deve consistir na palavra-chave `ref` seguida por um *variable\_reference* ([regras precisas para determinar a atribuição definitiva](#)) do mesmo tipo que o parâmetro formal. Uma variável deve ser definitivamente atribuída antes de poder ser passada como um parâmetro de referência.

Dentro de um método, um parâmetro de referência é sempre considerado definitivamente atribuído.

Um método declarado como iterador ([iteradores](#)) não pode ter parâmetros de referência.

### O exemplo

C#

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
```

```

    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}

```

produz a saída

Console

```
i = 2, j = 1
```

Para a invocação de `Swap` em `Main`, `x` representa `i` e `y` representa `j`. Assim, a invocação tem o efeito de trocar os valores de `i` e `j`.

Em um método que usa parâmetros de referência, é possível que vários nomes representem o mesmo local de armazenamento. No exemplo

C#

```

class A
{
    string s;

    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }

    void G() {
        F(ref s, ref s);
    }
}

```

a invocação do `F` no `G` passa uma referência para `s` ou `a` e o `b`. Portanto, para essa invocação, os nomes `s`, `a` e `b` todos fazem referência ao mesmo local de armazenamento e as três atribuições modificam o campo de instância `s`.

## Parâmetros de saída

Um parâmetro declarado com um `out` modificador é um parâmetro de saída. Semelhante a um parâmetro de referência, um parâmetro de saída não cria um novo

local de armazenamento. Em vez disso, um parâmetro de saída representa o mesmo local de armazenamento que a variável fornecida como o argumento na invocação do método.

Quando um parâmetro formal é um parâmetro de saída, o argumento correspondente em uma invocação de método deve consistir na palavra-chave `out` seguida por um *variable\_reference* ([regras precisas para determinar a atribuição definitiva](#)) do mesmo tipo que o parâmetro formal. Uma variável não precisa ser definitivamente atribuída antes de poder ser passada como um parâmetro de saída, mas após uma invocação em que uma variável foi passada como um parâmetro de saída, a variável é considerada definitivamente atribuída.

Dentro de um método, assim como uma variável local, um parâmetro de saída é inicialmente considerado não atribuído e deve ser definitivamente atribuído antes de seu valor ser usado.

Cada parâmetro de saída de um método deve ser definitivamente atribuído antes do retorno do método.

Um método declarado como um método parcial ([métodos parciais](#)) ou um iterador ([iteradores](#)) não pode ter parâmetros de saída.

Os parâmetros de saída normalmente são usados em métodos que produzem vários valores de retorno. Por exemplo:

C#

```
using System;

class Test
{
    static void SplitPath(string path, out string dir, out string name) {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\' || ch == '/' || ch == ':') break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }

    static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        Console.WriteLine(dir);
        Console.WriteLine(name);
    }
}
```

```
    }  
}
```

O exemplo produz a saída:

Console

```
c:\Windows\System\  
hello.txt
```

Observe que as `dir` e `name` variáveis e podem ser desatribuídas antes de serem passadas para `SplitPath` e que elas sejam consideradas definitivamente atribuídas após a chamada.

## Matrizes de parâmetros

Um parâmetro declarado com um `params` modificador é uma matriz de parâmetros. Se uma lista de parâmetros formais incluir uma matriz de parâmetros, ela deverá ser o último parâmetro na lista e deverá ser de um tipo de matriz unidimensional. Por exemplo, os tipos `string[]` e `string[][]` podem ser usados como o tipo de uma matriz de parâmetros, mas o tipo `string[,]` não pode. Não é possível combinar o `params` modificador com os modificadores `ref` e `out`.

Uma matriz de parâmetros permite que argumentos sejam especificados de uma das duas maneiras em uma invocação de método:

- O argumento fornecido para uma matriz de parâmetros pode ser uma única expressão conversível implicitamente ([conversões implícitas](#)) para o tipo de matriz de parâmetros. Nesse caso, a matriz de parâmetros funciona precisamente como um parâmetro de valor.
- Como alternativa, a invocação pode especificar zero ou mais argumentos para a matriz de parâmetros, em que cada argumento é uma expressão que é implicitamente conversível ([conversões implícitas](#)) para o tipo de elemento da matriz de parâmetros. Nesse caso, a invocação cria uma instância do tipo de matriz de parâmetros com um comprimento correspondente ao número de argumentos, inicializa os elementos da instância de matriz com os valores de argumento fornecidos e usa a instância de matriz recém-criada como o argumento real.

Exceto para permitir um número variável de argumentos em uma invocação, uma matriz de parâmetros é precisamente equivalente a um parâmetro de valor ([parâmetros de valor](#)) do mesmo tipo.

## O exemplo

C#

```
using System;

class Test
{
    static void F(params int[] args) {
        Console.Write("Array contains {0} elements:", args.Length);
        foreach (int i in args)
            Console.Write(" {0}", i);
        Console.WriteLine();
    }

    static void Main() {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}
```

produz a saída

Console

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

A primeira invocação de `F` simplesmente passa a matriz `a` como um parâmetro de valor. A segunda invocação de `F` cria automaticamente um quatro elementos `int[]` com os valores de elemento fornecidos e passa essa instância de matriz como um parâmetro de valor. Da mesma forma, a terceira invocação de `F` cria um elemento zero `int[]` e passa essa instância como um parâmetro de valor. As duas últimas invocações são precisamente equivalentes a escrever:

C#

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

Ao executar a resolução de sobrecarga, um método com uma matriz de parâmetros pode ser aplicável em seu formato normal ou em sua forma expandida ([membro de função aplicável](#)). A forma expandida de um método só estará disponível se a forma

normal do método não for aplicável e somente se um método aplicável com a mesma assinatura do formulário expandido ainda não estiver declarado no mesmo tipo.

## O exemplo

C#

```
using System;

class Test
{
    static void F(params object[] a) {
        Console.WriteLine("F(object[])");
    }

    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object a0, object a1) {
        Console.WriteLine("F(object,object)");
    }

    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(1, 2, 3, 4);
    }
}
```

produz a saída

Console

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

No exemplo, duas das possíveis formas expandidas do método com uma matriz de parâmetros já estão incluídas na classe como métodos regulares. Portanto, esses formulários expandidos não são considerados ao executar a resolução de sobrecarga e a primeira e terceira invocações de método, portanto, selecionam os métodos regulares. Quando uma classe declara um método com uma matriz de parâmetros, não é incomum também incluir alguns dos formulários expandidos como métodos regulares.

Ao fazer isso, é possível evitar a alocação de uma instância de matriz que ocorre quando uma forma expandida de um método com uma matriz de parâmetros é invocada.

Quando o tipo de uma matriz de parâmetros é `object[]`, ocorre uma possível ambiguidade entre a forma normal do método e o formulário informado para um único `object` parâmetro. O motivo da ambiguidade é que um `object[]` é implicitamente conversível no tipo `object`. No entanto, a ambiguidade não apresenta nenhum problema, pois ela pode ser resolvida com a inserção de uma conversão, se necessário.

O exemplo

```
C#  
  
using System;  
  
class Test  
{  
    static void F(params object[] args) {  
        foreach (object o in args) {  
            Console.WriteLine(o.GetType().FullName);  
            Console.Write(" ");  
        }  
        Console.WriteLine();  
    }  
  
    static void Main() {  
        object[] a = {1, "Hello", 123.456};  
        object o = a;  
        F(a);  
        F((object)a);  
        F(o);  
        F((object[])o);  
    }  
}
```

produz a saída

```
Console  
  
System.Int32 System.String System.Double  
System.Object[]  
System.Object[]  
System.Int32 System.String System.Double
```

Na primeira e última invocações do `F`, a forma normal do `F` é aplicável porque existe uma conversão implícita do tipo de argumento para o tipo de parâmetro (ambos são do tipo `object[]`). Portanto, a resolução de sobrecarga seleciona a forma normal de `F` e o argumento é passado como um parâmetro de valor regular. Na segunda e terceira

invocações, a forma normal de `F` não é aplicável porque não existe nenhuma conversão implícita do tipo de argumento para o tipo de parâmetro (o tipo `object` não pode ser convertido implicitamente no tipo `object[]`). No entanto, a forma expandida do `F` é aplicável, portanto, é selecionada pela resolução de sobrecarga. Como resultado, um elemento One `object[]` é criado pela invocação, e o único elemento da matriz é inicializado com o valor do argumento fornecido (que, por sua vez, é uma referência a um `object[]`).

## Métodos estáticos e de instância

Quando uma declaração de método inclui um `static` modificador, esse método é considerado um método estático. Quando nenhum `static` modificador está presente, o método é considerado um método de instância.

Um método estático não funciona em uma instância específica e é um erro de tempo de compilação para fazer referência a `this` um método estático.

Um método de instância opera em uma determinada instância de uma classe, e essa instância pode ser acessada como `this` ([esse acesso](#)).

Quando um método é referenciado em um *member\_access* ([acesso de membro](#)) do formulário `E.M`, se `M` for um método estático, `E` deverá indicar um tipo contendo `M`, se `M` for um método de instância, `E` deverá indicar uma instância de um tipo que contém `M`.

As diferenças entre os membros estático e de instância são discutidas mais detalhadamente em [membros estáticos e de instância](#).

## Métodos virtuais

Quando uma declaração de método de instância inclui um `virtual` modificador, esse método é considerado um método virtual. Quando nenhum `virtual` modificador está presente, o método é considerado um método não virtual.

A implementação de um método não virtual é invariável: a implementação é a mesma se o método é invocado em uma instância da classe na qual é declarada ou uma instância de uma classe derivada. Por outro lado, a implementação de um método virtual pode ser substituída por classes derivadas. O processo de substituição da implementação de um método virtual herdado é conhecido como **substituição** desse método ([métodos de substituição](#)).

Em uma invocação de método virtual, o **tipo de tempo de execução\*** da instância para a qual a invocação ocorre determina a implementação do método real a ser invocado. Em uma invocação de método não virtual, o tipo \_ de *tempo de compilação\** da instância é o fator determinante. Em termos precisos, quando um método chamado `N` é invocado com uma lista `A` de argumentos em uma instância com um tipo de tempo de compilação `C` e um tipo de tempo de execução `R` (onde `R` é `C` ou uma classe derivada de `C`), a invocação é processada da seguinte maneira:

- Primeiro, a resolução de sobrecarga é aplicada a `C`, `N` e `A`, para selecionar um método específico `M` do conjunto de métodos declarado em e herdado por `C`. Isso é descrito em [invocações de método](#).
- Em seguida, se `M` for um método não virtual, `M` será invocado.
- Caso contrário, `M` é um método virtual e a implementação mais derivada de `M` com relação a `R` é invocada.

Para cada método virtual declarado em ou herdado por uma classe, existe uma **implementação mais derivada** do método em relação a essa classe. A implementação mais derivada de um método virtual `M` em relação a uma classe `R` é determinada da seguinte maneira:

- Se `R` contiver a `virtual` declaração de introdução de `M`, essa será a implementação mais derivada de `M`.
- Caso contrário, se `R` contiver um `override` de `M`, essa será a implementação mais derivada de `M`.
- Caso contrário, a implementação mais derivada de `M` em relação ao `R` é igual à implementação mais derivada de `M` em relação à classe base direta do `R`.

O exemplo a seguir ilustra as diferenças entre métodos virtuais e não virtuais:

C#

```
using System;

class A
{
    public void F() { Console.WriteLine("A.F"); }

    public virtual void G() { Console.WriteLine("A.G"); }
}

class B: A
{
    new public void F() { Console.WriteLine("B.F"); }

    public override void G() { Console.WriteLine("B.G"); }
```

```

}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        a.F();
        b.F();
        a.G();
        b.G();
    }
}

```

No exemplo, `A` apresenta um método não virtual `F` e um método virtual `G`. A classe `B` introduz um novo método não virtual `F`, ocultando o herdado `F` e também substitui o método herdado `G`. O exemplo produz a saída:

Console

```

A.F
B.F
B.G
B.G

```

Observe que a instrução `a.G()` invoca `B.G`, não `A.G`. Isso ocorre porque o tipo de tempo de execução da instância (que é `B`), não o tipo de tempo de compilação da instância (que é `A`), determina a implementação do método real a ser invocado.

Como os métodos têm permissão para ocultar os métodos herdados, é possível que uma classe contenha vários métodos virtuais com a mesma assinatura. Isso não apresenta um problema de ambiguidade, pois todos, exceto o método mais derivado, estão ocultos. No exemplo

C#

```

using System;

class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}

class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}

```

```

class C: B
{
    new public virtual void F() { Console.WriteLine("C.F"); }
}

class D: C
{
    public override void F() { Console.WriteLine("D.F"); }
}

class Test
{
    static void Main()
    {
        D d = new D();
        A a = d;
        B b = d;
        C c = d;
        a.F();
        b.F();
        c.F();
        d.F();
    }
}

```

as `C` classes e contêm dois métodos virtuais com a mesma assinatura: aquele introduzido pelo `A` e aquele introduzido pelo `C`. O método introduzido por `C` oculta o método herdado de `A`. Assim, a declaração de substituição em `D` substitui o método introduzido pelo `C`, e não é possível `D` substituir o método introduzido pelo `A`. O exemplo produz a saída:

The screenshot shows a terminal window titled "Console". The output consists of five lines of text: "B.F", "B.F", "D.F", "D.F", and "D.F". This output demonstrates that the method from class C is being called through the reference variable d, even though class D overrides it.

```

Console

B.F
B.F
D.F
D.F
D.F

```

Observe que é possível invocar o método virtual oculto acessando uma instância do `D` por meio de um tipo menos derivado no qual o método não está oculto.

## Métodos de substituição

Quando uma declaração de método de instância inclui um `override` modificador, o método é considerado um *método de substituição*. Um método override substitui um método virtual herdado pela mesma assinatura. Enquanto uma declaração de método virtual apresenta um novo método, uma declaração de método de substituição restringe um método virtual herdado existente fornecendo uma nova implementação do método.

O método substituído por uma `override` declaração é conhecido como **método base substituído**. Para um método `M` declarado em uma classe `C`, o método base substituído é determinado examinando cada tipo de classe base de `C`, começando com o tipo de classe base direta `C` e continuando com cada tipo de classe base direta sucessiva, até que em um determinado tipo de classe base seja localizado pelo menos um método acessível que tenha a mesma assinatura que `M` após a substituição dos argumentos de tipo. Para fins de localização do método base substituído, um método é considerado acessível se for, se for, `public` `protected` `protected internal` ou se for `internal` e declarado no mesmo programa que o `C`.

Um erro de tempo de compilação ocorre a menos que todos os itens a seguir sejam verdadeiros para uma declaração de substituição:

- Um método base substituído pode ser localizado conforme descrito acima.
- Há exatamente um desses métodos de base substituído. Essa restrição só terá efeito se o tipo de classe base for um tipo construído em que a substituição de argumentos de tipo torna a assinatura de dois métodos o mesmo.
- O método base substituído é um método `virtual`, `abstract` ou `override`. Em outras palavras, o método base substituído não pode ser `estático` ou não `virtual`.
- O método `override` e o método base substituído têm o mesmo tipo de retorno.
- A declaração de substituição e o método base substituído têm a mesma acessibilidade declarada. Em outras palavras, uma declaração de substituição não pode alterar a acessibilidade do método virtual. No entanto, se o método base substituído for protegido internamente e for declarado em um assembly diferente do assembly que contém o método `override`, a acessibilidade declarada do método de substituição deverá ser protegida.
- A declaração de substituição não especifica cláusulas de tipo-parâmetro-`Constraints`. Em vez disso, as restrições são herdadas do método base substituído. Observe que as restrições que são parâmetros de tipo no método substituído podem ser substituídas por argumentos de tipo na restrição herdada. Isso pode levar a restrições que não são legais quando explicitamente especificados, como tipos de valor ou tipos lacrados.

O exemplo a seguir demonstra como as regras de substituição funcionam para classes genéricas:

C#

```
abstract class C<T>
{
    public virtual T F() {...}
    public virtual C<T> G() {...}
```

```

    public virtual void H(C<T> x) {...}

}

class D: C<string>
{
    public override string F() {...} // Ok
    public override C<string> G() {...} // Ok
    public override void H(C<T> x) {...} // Error, should be
C<string>
}

class E<T,U>: C<U>
{
    public override U F() {...} // Ok
    public override C<U> G() {...} // Ok
    public override void H(C<T> x) {...} // Error, should be C<U>
}

```

Uma declaração de substituição pode acessar o método base substituído usando um *base\_access* ([acesso de base](#)). No exemplo

C#

```

class A
{
    int x;

    public virtual void PrintFields() {
        Console.WriteLine("x = {0}", x);
    }
}

class B: A
{
    int y;

    public override void PrintFields() {
        base.PrintFields();
        Console.WriteLine("y = {0}", y);
    }
}

```

a invocação `base.PrintFields()` no `B` invoca o `PrintFields` método declarado em `A`.

Um *base\_access* desabilita o mecanismo de invocação virtual e simplesmente trata o método base como um método não virtual. Se a invocação tivesse `B` sido escrita

`((A)this).PrintFields()`, ela invocaria recursivamente o `PrintFields` método declarado em `B`, e não o declarado em `A`, pois `PrintFields` é virtual e o tipo de tempo de execução de `((A)this)` é `B`.

Somente ao incluir um `override` modificador, um método pode substituir outro método. Em todos os outros casos, um método com a mesma assinatura que um método herdado simplesmente oculta o método herdado. No exemplo

```
C#  
  
class A  
{  
    public virtual void F() {}  
}  
  
class B: A  
{  
    public virtual void F() {}          // Warning, hiding inherited F()  
}
```

o `F` método no `B` não inclui um `override` modificador e, portanto, não substitui o `F` método em `A`. Em vez disso, o `F` método em `B` oculta o método em `A` e um aviso é relatado porque a declaração não inclui um `new` modificador.

No exemplo

```
C#  
  
class A  
{  
    public virtual void F() {}  
}  
  
class B: A  
{  
    new private void F() {}          // Hides A.F within body of B  
}  
  
class C: B  
{  
    public override void F() {}      // Ok, overrides A.F  
}
```

o `F` método em `B` oculta o método virtual `F` herdado de `A`. Como o novo `F` no `B` tem acesso privado, seu escopo inclui apenas o corpo da classe de `B` e não se estende para `C`. Portanto, a declaração de `F` no `C` tem permissão para substituir o `F` herdado de `A`.

## Métodos lacrados

Quando uma declaração de método de instância inclui um `sealed` modificador, esse método é considerado um *método lacrado*. Se uma declaração de método de instância incluir o `sealed` modificador, ele também deverá incluir o `override` modificador. O uso do `sealed` modificador impede que uma classe derivada substitua ainda mais o método.

No exemplo

C#

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }

    public virtual void G() {
        Console.WriteLine("A.G");
    }
}

class B: A
{
    sealed override public void F() {
        Console.WriteLine("B.F");
    }

    override public void G() {
        Console.WriteLine("B.G");
    }
}

class C: B
{
    override public void G() {
        Console.WriteLine("C.G");
    }
}
```

a classe `B` fornece dois métodos de substituição: um `F` método que tem o `sealed` modificador e um `G` método que não tem. O uso do `sealed` modificador impede a `C` substituição adicional `F`.

## Métodos abstratos

Quando uma declaração de método de instância inclui um `abstract` modificador, esse método é considerado um *método abstrato*. Embora um método `abstract` também seja

implicitamente um método virtual, ele não pode ter o modificador `virtual`.

Uma declaração de método `abstract` apresenta um novo método virtual, mas não fornece uma implementação desse método. Em vez disso, as classes derivadas não abstratas são necessárias para fornecer sua própria implementação substituindo esse método. Como um método `abstract` não fornece implementação real, a `method_body` de um método `abstract` simplesmente consiste em um ponto-e-vírgula.

Declarações de método `abstract` só são permitidas em classes abstratas ([classes abstratas](#)).

No exemplo

```
C#  
  
public abstract class Shape  
{  
    public abstract void Paint(Graphics g, Rectangle r);  
}  
  
public class Ellipse: Shape  
{  
    public override void Paint(Graphics g, Rectangle r) {  
        g.DrawEllipse(r);  
    }  
}  
  
public class Box: Shape  
{  
    public override void Paint(Graphics g, Rectangle r) {  
        g.DrawRect(r);  
    }  
}
```

a `Shape` classe define a noção abstrata de um objeto de forma geométrica que pode ser pintada. O `Paint` método é abstrato porque não há nenhuma implementação padrão significativa. As `Ellipse` `Box` classes e são `Shape` implementações concretas. Como essas classes não são abstratas, elas são necessárias para substituir o `Paint` método e fornecer uma implementação real.

É um erro de tempo de compilação para um `base_access` ([acesso de base](#)) para referenciar um método abstrato. No exemplo

```
C#  
  
abstract class A  
{  
    public abstract void F();
```

```
}
```

```
class B: A
{
    public override void F() {
        base.F();                                // Error, base.F is abstract
    }
}
```

um erro de tempo de compilação é relatado para a `base.F()` invocação porque faz referência a um método abstrato.

Uma declaração de método abstract tem permissão para substituir um método virtual. Isso permite que uma classe abstrata force a reimplementação do método em classes derivadas e torna a implementação original do método indisponível. No exemplo

```
C#
```

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
}

abstract class B: A
{
    public abstract override void F();
}

class C: B
{
    public override void F() {
        Console.WriteLine("C.F");
    }
}
```

classe `A` declara um método virtual, a classe `B` substitui esse método por um método abstrato e a classe `C` substitui o método abstract para fornecer sua própria implementação.

## Métodos externos

Quando uma declaração de método inclui um `extern` modificador, esse método é considerado um **\*método externo\***. Os métodos externos são implementados externamente, normalmente usando uma linguagem diferente de C#. Como uma

declaração de método externo não fornece implementação real, o `_method_body` \* de um método externo simplesmente consiste em um ponto-e-vírgula. Um método externo não pode ser genérico.

O `extern` modificador é normalmente usado em conjunto com um `DllImport` atributo ([interoperação com componentes COM e Win32](#)), permitindo que métodos externos sejam implementados por DLLs (bibliotecas de vínculo dinâmico). O ambiente de execução pode dar suporte a outros mecanismos nos quais implementações de métodos externos podem ser fornecidas.

Quando um método externo inclui um `DllImport` atributo, a declaração de método também deve incluir um `static` modificador. Este exemplo demonstra o uso do `extern` modificador e do `DllImport` atributo:

C#

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttribute sa);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);

    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}
```

## Métodos parciais (recapitulação)

Quando uma declaração de método inclui um `partial` modificador, esse método é considerado um *método parcial*. Os métodos parciais só podem ser declarados como membros de tipos parciais ([tipos parciais](#)) e estão sujeitos a várias restrições. Os métodos parciais são descritos mais detalhadamente em [métodos parciais](#).

## Métodos de extensão

Quando o primeiro parâmetro de um método inclui o `this` modificador, esse método é considerado um **método de extensão**. Os métodos de extensão só podem ser declarados em classes estáticas não genéricas e não aninhadas. O primeiro parâmetro de um método de extensão não pode ter nenhum modificador diferente de `this` e o tipo de parâmetro não pode ser um tipo de ponteiro.

Veja a seguir um exemplo de uma classe estática que declara dois métodos de extensão:

C#

```
public static class Extensions
{
    public static intToInt32(this string s) {
        return Int32.Parse(s);
    }

    public static T[] Slice<T>(this T[] source, int index, int count) {
        if (index < 0 || count < 0 || source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
        return result;
    }
}
```

Um método de extensão é um método estático regular. Além disso, onde sua classe estática delimitadora está no escopo, um método de extensão pode ser invocado usando a sintaxe de invocação do método de instância ([invocações de método de extensão](#)), usando a expressão Receiver como o primeiro argumento.

O programa a seguir usa os métodos de extensão declarados acima:

C#

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in strings.Slice(1, 2)) {
            Console.WriteLine(s.ToInt32());
        }
    }
}
```

O `slice` método está disponível no `string[]`, e o `ToInt32` método está disponível em `string`, porque eles foram declarados como métodos de extensão. O significado do programa é o mesmo que o seguinte, usando chamadas de método estáticos comuns:

C#

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in Extensions.Slice(strings, 1, 2)) {
            Console.WriteLine(Extensions.ToInt32(s));
        }
    }
}
```

## Corpo do método

O *method\_body* de uma declaração de método consiste em um corpo de bloco, um corpo de expressão ou um ponto-e-vírgula.

O *tipo de resultado* de um método é `void` se o tipo de retorno for `void` ou se o método for `Async` e o tipo de retorno for `System.Threading.Tasks.Task`. Caso contrário, o tipo de resultado de um método não assíncrono é seu tipo de retorno, e o tipo de resultado de um método assíncrono com tipo de retorno `System.Threading.Tasks.Task<T>` é `T`.

Quando um método tem um `void` tipo de resultado e um corpo de bloco, `return` as instruções (a instrução `return`) no bloco não têm permissão para especificar uma expressão. Se a execução do bloco de um método `void` for concluída normalmente (ou seja, o controle flui para fora do final do corpo do método), esse método simplesmente retorna ao seu chamador atual.

Quando um método tem um `void` resultado e um corpo de expressão, a expressão `E` deve ser uma *statement\_expression* e o corpo é exatamente equivalente a um corpo de bloco do formulário `{ E; }`.

Quando um método tem um tipo de resultado não `void` e um corpo de bloco, cada `return` instrução no bloco deve especificar uma expressão que é implicitamente conversível para o tipo de resultado. O ponto de extremidade de um corpo de bloco de um método de retorno de valor não deve ser acessível. Em outras palavras, em um método de retorno de valor com um corpo de bloco, o controle não tem permissão para fluir para fora do final do corpo do método.

Quando um método tem um tipo de resultado não `void` e um corpo de expressão, a expressão deve ser conversível implicitamente no tipo de resultado e o corpo é exatamente equivalente a um corpo de bloco do formulário `{ return E; }`.

No exemplo

C#

```
class A
{
    public int F() {} // Error, return value required

    public int G() {
        return 1;
    }

    public int H(bool b) {
        if (b) {
            return 1;
        }
        else {
            return 0;
        }
    }

    public int I(bool b) => b ? 1 : 0;
}
```

o método de retorno de valor `F` resulta em um erro de tempo de compilação porque o controle pode fluir para fora do final do corpo do método. Os `G` `H` métodos estão corretos porque todos os caminhos de execução possíveis terminam em uma instrução `Return` que especifica um valor de retorno. O `I` método está correto, pois seu corpo é equivalente a um bloco de instrução com apenas uma única instrução de retorno.

## Sobrecarga de método

As regras de resolução de sobrecarga de método são descritas em [inferência de tipos](#).

## Propriedades

Uma **\*Propriedade** é um membro que fornece acesso a uma característica de um objeto ou de uma classe. Exemplos de propriedades incluem o comprimento de uma cadeia de caracteres, o tamanho de uma fonte, a legenda de uma janela, o nome de um cliente e assim por diante. As propriedades são uma extensão natural dos campos – ambos são membros nomeados com tipos associados e a sintaxe para acessar campos e propriedades é a mesma. No entanto, diferentemente dos campos, as propriedades não denotam locais de armazenamento. Em vez disso, as propriedades têm **\_acessadores\*** que especificam as instruções a serem executadas quando seus valores são lidos ou gravados. Portanto, as propriedades fornecem um mecanismo para associar ações com

a leitura e gravação de atributos de um objeto; Além disso, eles permitem que esses atributos sejam computados.

As propriedades são declaradas usando *property\_declaration* s:

```
antlr

property_declaration
    : attributes? property_modifier* type member_name property_body
    ;

property_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | property_modifier_unsafe
    ;

property_body
    : '{' accessor_declarations '}' property_initializer?
    | '>' expression ';'
    ;

property_initializer
    : '=' variable_initializer ';'
    ;
```

Uma *property\_declaration* pode incluir um conjunto de *atributos* ([atributos](#)) e uma combinação válida dos quatro modificadores de acesso ([modificadores de acesso](#)), o `new` ([o novo modificador](#)), `static` ([métodos estáticos e de instância](#)), `virtual` ([métodos virtuais](#)), (métodos de `override` [substituição](#)), (métodos `sealed` [lacrados](#)), os `abstract` modificadores (métodos [abstratos](#)) e `extern` ([métodos externos](#)).

As declarações de propriedade estão sujeitas às mesmas regras que as declarações de método ([métodos](#)) em relação a combinações válidas de modificadores.

O *tipo* de uma declaração de propriedade especifica o tipo da propriedade introduzida pela declaração e o *member\_name* especifica o nome da propriedade. A menos que a propriedade seja uma implementação de membro de interface explícita, o *member\_name* é simplesmente um *identificador*. Para uma implementação explícita de

membro de interface ([implementações explícitas de membro de interface](#)), o *member\_name* consiste em um *interface\_type* seguido por um " . " e um *identificador*.

O *tipo* de uma propriedade deve ser pelo menos tão acessível quanto a própria propriedade ([restrições de acessibilidade](#)).

Uma *property\_body* pode consistir em um **corpo de acessador** ou um **corpo de expressão**. Em um corpo de acessador, *\_accessor\_declarations* \*, que deve ser incluído em { tokens "" e " } ", declare os acessadores ([acessadores](#)) da propriedade. Os acessadores especificam as instruções Executáveis associadas à leitura e gravação da propriedade.

Um corpo de expressão que consiste em => seguido por uma *expressão* E e um ponto-e-vírgula é exatamente equivalente ao corpo da instrução { get { return E; } } e, portanto, só pode ser usado para especificar propriedades somente getter em que o resultado do getter é fornecido por uma única expressão.

Um *property\_initializer* só pode ser fornecido para uma propriedade implementada automaticamente ([Propriedades implementadas automaticamente](#)) e causa a inicialização do campo subjacente de tais propriedades com o valor fornecido pela *expressão*.

Embora a sintaxe para acessar uma propriedade seja a mesma de um campo, uma propriedade não é classificada como uma variável. Portanto, não é possível passar uma propriedade como um *ref* *out* argumento ou.

Quando uma declaração de propriedade inclui um *extern* modificador, a propriedade é considerada uma **\*Propriedade externa**\_. Como uma declaração de propriedade externa não fornece implementação real, cada uma de suas *\_accessor\_declarations* \* consiste em um ponto e vírgula.

## Propriedades de instância e estática

Quando uma declaração de propriedade inclui um *static* modificador, a propriedade é considerada uma **\*propriedade estática**\_. Quando nenhum *static* modificador estiver presente, a propriedade será considerada uma *\_propriedade de instância* \*.

Uma propriedade estática não está associada a uma instância específica e é um erro de tempo de compilação para se referir aos *this* acessadores de uma propriedade estática.

Uma propriedade de instância é associada a uma determinada instância de uma classe, e essa instância pode ser acessada como *this* ([esse acesso](#)) nos acessadores dessa propriedade.

Quando uma propriedade é referenciada em um *member\_access* ([acesso de membro](#)) do formulário `E.M`, se `M` for uma propriedade estática, `E` deverá denotar um tipo contendo `M`, E se `M` for uma propriedade de instância, e deverá indicar uma instância de um tipo que contém `M`.

As diferenças entre os membros estático e de instância são discutidas mais detalhadamente em [membros estáticos e de instância](#).

## Acessadores

O *accessor\_declarations* de uma propriedade especifica as instruções Executáveis associadas à leitura e gravação dessa propriedade.

```
antlr

accessor_declarations
    : get_accessor_declaration set_accessor_declaration?
    | set_accessor_declaration get_accessor_declaration?
    ;
get_accessor_declaration
    : attributes? accessor_modifier? 'get' accessor_body
    ;
set_accessor_declaration
    : attributes? accessor_modifier? 'set' accessor_body
    ;
accessor_modifier
    : 'protected'
    | 'internal'
    | 'private'
    | 'protected' 'internal'
    | 'internal' 'protected'
    ;
accessor_body
    : block
    | ';'
    ;
```

As declarações de acessador consistem em um *get\_accessor\_declaration*, um *set\_accessor\_declaration* ou ambos. Cada declaração de acessador consiste no token `get` ou `set` seguido por um *accessor\_modifier* opcional e um *accessor\_body*.

O uso de *accessor\_modifier*s é regido pelas seguintes restrições:

- Um *accessor\_modifier* não pode ser usado em uma interface ou em uma implementação de membro de interface explícita.
- Para uma propriedade ou um indexador que não tenha nenhum `override` modificador, um *accessor\_modifier* só será permitido se a propriedade ou o indexador tiver um `get` e um `set` acessador e, em seguida, for permitido somente em um desses acessadores.
- Para uma propriedade ou um indexador que inclui um `override` modificador, um acessador deve corresponder ao *accessor\_modifier*, se houver, do acessador que está sendo substituído.
- O *accessor\_modifier* deve declarar uma acessibilidade que seja estritamente mais restritiva do que a acessibilidade declarada da propriedade ou do indexador em si. Para ser preciso:
  - Se a propriedade ou o indexador tiver uma acessibilidade declarada `public`, o *accessor\_modifier* poderá ser `protected internal`, `internal`, `protected` ou `private`.
  - Se a propriedade ou o indexador tiver uma acessibilidade declarada `protected internal`, o *accessor\_modifier* poderá ser `internal`, `protected` ou `private`.
  - Se a propriedade ou o indexador tiver uma acessibilidade declarada de `internal` ou `protected`, o *accessor\_modifier* deverá ser `private`.
  - Se a propriedade ou o indexador tiver uma acessibilidade declarada `private`, nenhuma *accessor\_modifier* poderá ser usada.

Para `abstract` `extern` Propriedades e, o *accessor\_body* para cada acessador especificado é simplesmente um ponto e vírgula. Uma propriedade não abstrata que não seja externa pode ter cada *accessor\_body* ser um ponto-e-vírgula; nesse caso, é uma **\*Propriedade automaticamente implementada \_ (Propriedades implementadas automaticamente)**. Uma propriedade implementada automaticamente deve ter pelo menos um acessador `get`. Para os acessadores de qualquer outra propriedade não abstrata e não externa, o `_accessor_body` \* é um bloco que especifica as instruções a serem executadas quando o acessador correspondente é invocado.

Um `get` acessador corresponde a um método sem parâmetros com um valor de retorno do tipo de propriedade. Exceto como o destino de uma atribuição, quando uma propriedade é referenciada em uma expressão, o `get` acessador da propriedade é invocado para calcular o valor da propriedade ([valores de expressões](#)). O corpo de um `get` acessador deve estar em conformidade com as regras para métodos de retorno de valor descritos no [corpo do método](#). Em particular, todas as `return` instruções no corpo de um `get` acessador devem especificar uma expressão que é implicitamente conversível para o tipo de propriedade. Além disso, o ponto de extremidade de um `get` acessador não deve ser acessível.

Um `set` acessador corresponde a um método com um parâmetro de valor único do tipo de propriedade e um `void` tipo de retorno. O parâmetro implícito de um `set` acessador é sempre denominado `value`. Quando uma propriedade é referenciada como o destino de uma atribuição ([operadores de atribuição](#)) ou como o operando de ou ([operadores de `++` `--`](#), [incremento e diminuição de sufixo](#), [incremento de prefixo](#) e [operadores de decréscimo](#)), o `set` acessador é invocado com um argumento (cujo valor é o lado direito da atribuição ou o operando do `++` `--` operador OR) que fornece o novo valor ([atribuição simples](#)). O corpo de um `set` acessador deve estar em conformidade com as regras para `void` métodos descritos no [corpo do método](#). Em particular, `return` as instruções no `set` corpo do acessador não têm permissão para especificar uma expressão. Como um `set` acessador implicitamente tem um parâmetro chamado `value`, ele é um erro de tempo de compilação para uma variável local ou uma declaração de constante em um `set` acessador para ter esse nome.

Com base na presença ou ausência de `get` `set` acessadores, uma propriedade é classificada da seguinte maneira:

- Uma propriedade que inclui um `get` acessador e um `set` acessador é considerada uma propriedade de ***leitura/gravação***.
- Uma propriedade que tem apenas um `get` acessador é considerada uma propriedade ***somente leitura***. É um erro de tempo de compilação para uma propriedade somente leitura ser o destino de uma atribuição.
- Uma propriedade que tem apenas um `set` acessador é considerada uma propriedade ***somente gravação***. Exceto como o destino de uma atribuição, é um erro de tempo de compilação para referenciar uma propriedade somente gravação em uma expressão.

No exemplo

C#

```
public class Button: Control
{
    private string caption;

    public string Caption {
        get {
            return caption;
        }
        set {
            if (caption != value) {
                caption = value;
                Repaint();
            }
        }
    }
}
```

```

        }

    public override void Paint(Graphics g, Rectangle r) {
        // Painting code goes here
    }
}

```

O `Button` controle declara uma propriedade pública `Caption`. O `get` acessador da `Caption` propriedade retorna a cadeia de caracteres armazenada no `caption` campo particular. O `set` acessador verifica se o novo valor é diferente do valor atual e, nesse caso, armazena o novo valor e redesenha o controle. As propriedades geralmente seguem o padrão mostrado acima: o `get` acessador simplesmente retorna um valor armazenado em um campo particular, e o `set` acessador modifica esse campo particular e, em seguida, executa quaisquer ações adicionais necessárias para atualizar totalmente o estado do objeto.

Considerando a `Button` classe acima, veja a seguir um exemplo de uso da `Caption` Propriedade:

C#

```

Button okButton = new Button();
okButton.Caption = "OK";           // Invokes set accessor
string s = okButton.Caption;      // Invokes get accessor

```

Aqui, o `set` acessador é invocado atribuindo um valor à propriedade, e o `get` acessador é invocado referenciando a propriedade em uma expressão.

O `get` e os `set` acessadores de uma propriedade não são membros distintos e não é possível declarar os acessadores de uma propriedade separadamente. Assim, não é possível que os dois acessadores de uma propriedade de leitura/gravação tenham acessibilidade diferente. O exemplo

C#

```

class A
{
    private string name;

    public string Name {           // Error, duplicate member name
        get { return name; }
    }

    public string Name {           // Error, duplicate member name
        set { name = value; }
    }
}

```

```
    }  
}
```

não declara uma única propriedade de leitura/gravação. Em vez disso, ele declara duas propriedades com o mesmo nome, uma somente leitura e uma somente gravação. Como dois membros declarados na mesma classe não podem ter o mesmo nome, o exemplo causa a ocorrência de um erro de tempo de compilação.

Quando uma classe derivada declara uma propriedade com o mesmo nome de uma propriedade herdada, a propriedade derivada oculta a propriedade herdada em relação à leitura e à gravação. No exemplo

```
C#  
  
class A  
{  
    public int P {  
        set {...}  
    }  
}  
  
class B: A  
{  
    new public int P {  
        get {...}  
    }  
}
```

a `P` propriedade em `B` oculta a `P` propriedade em `A` com relação à leitura e gravação. Portanto, nas instruções

```
C#  
  
B b = new B();  
b.P = 1;           // Error, B.P is read-only  
((A)b).P = 1;     // Ok, reference to A.P
```

a atribuição para `b.P` causa a reportação de um erro em tempo de compilação, pois a `P` propriedade somente leitura no `B` oculta a propriedade somente gravação `P` no `A`. Observe, no entanto, que uma conversão pode ser usada para acessar a `P` Propriedade Hidden.

Ao contrário dos campos públicos, as propriedades fornecem uma separação entre o estado interno de um objeto e sua interface pública. Considere o exemplo:

```
C#
```

```

class Label
{
    private int x, y;
    private string caption;

    public Label(int x, int y, string caption) {
        this.x = x;
        this.y = y;
        this.caption = caption;
    }

    public int X {
        get { return x; }
    }

    public int Y {
        get { return y; }
    }

    public Point Location {
        get { return new Point(x, y); }
    }

    public string Caption {
        get { return caption; }
    }
}

```

Aqui, a `Label` classe usa dois `int` campos `x` e `y`, para armazenar seu local. O local é exposto publicamente como um `X` e uma `Y` propriedade e como uma `Location` Propriedade do tipo `Point`. Se, em uma versão futura do `Label`, for mais conveniente armazenar o local como um `Point` internamente, a alteração poderá ser feita sem afetar a interface pública da classe:

C#

```

class Label
{
    private Point location;
    private string caption;

    public Label(int x, int y, string caption) {
        this.location = new Point(x, y);
        this.caption = caption;
    }

    public int X {
        get { return location.x; }
    }

    public int Y {

```

```

        get { return location.y; }

    }

    public Point Location {
        get { return location; }
    }

    public string Caption {
        get { return caption; }
    }
}

```

Tinha `x` e `y`, em vez de `public readonly` campos, seria impossível fazer essa alteração na `Label` classe.

Exportar o estado por meio de propriedades não é necessariamente qualquer menos eficiente do que exportar os campos diretamente. Em particular, quando uma propriedade é não virtual e contém apenas uma pequena quantidade de código, o ambiente de execução pode substituir chamadas para acessadores pelo código real dos acessadores. Esse processo é conhecido como *inlining* e torna o acesso à propriedade tão eficiente quanto o acesso ao campo, mas preserva a maior flexibilidade das propriedades.

Como invocar um `get` acessador é conceitualmente equivalente a ler o valor de um campo, ele é considerado um estilo de programação insatisfatório para `get` que os acessadores tenham efeitos colaterais observáveis. No exemplo

```
C#
class Counter
{
    private int next;

    public int Next {
        get { return next++; }
    }
}
```

o valor da `Next` propriedade depende do número de vezes que a propriedade foi acessada anteriormente. Portanto, o acesso à propriedade produz um efeito colateral observável, e a propriedade deve ser implementada como um método em vez disso.

A Convenção "sem efeitos colaterais" para `get` acessadores não significa que `get` os acessadores sempre devem ser escritos para simplesmente retornar valores armazenados em campos. Na verdade, `get` os acessadores geralmente calculam o valor de uma propriedade acessando vários campos ou invocando métodos. No entanto, um

get acessador projetado corretamente não executa nenhuma ação que cause alterações observáveis no estado do objeto.

As propriedades podem ser usadas para atrasar a inicialização de um recurso até o momento em que ele é referenciado pela primeira vez. Por exemplo:

C#

```
using System.IO;

public class Console
{
    private static TextReader reader;
    private static TextWriter writer;
    private static TextWriter error;

    public static TextReader In {
        get {
            if (reader == null) {
                reader = new StreamReader(Console.OpenStandardInput());
            }
            return reader;
        }
    }

    public static TextWriter Out {
        get {
            if (writer == null) {
                writer = new StreamWriter(Console.OpenStandardOutput());
            }
            return writer;
        }
    }

    public static TextWriter Error {
        get {
            if (error == null) {
                error = new StreamWriter(Console.OpenStandardError());
            }
            return error;
        }
    }
}
```

A `Console` classe contém três propriedades, `In`, `Out` e `Error`, que representam os dispositivos de entrada, saída e erro padrão, respectivamente. Ao expor esses membros como propriedades, a `Console` classe pode atrasar a inicialização até que elas sejam realmente usadas. Por exemplo, após a primeira referência à `Out` propriedade, como em

C#

```
Console.Out.WriteLine("hello, world");
```

o subjacente `TextWriter` para o dispositivo de saída é criado. Mas se o aplicativo não fizer nenhuma referência às `In` `Error` Propriedades e, nenhum objeto será criado para esses dispositivos.

## Propriedades implementadas automaticamente

Uma propriedade implementada automaticamente (ou *Propriedade automática* para curto), é uma propriedade não abstrata não abstraída com corpos de acessadores somente de ponto e vírgula. As propriedades automáticas devem ter um acessador get e, opcionalmente, podem ter um acessador set.

Quando uma propriedade é especificada como uma propriedade implementada automaticamente, um campo de apoio oculto fica automaticamente disponível para a propriedade e os acessadores são implementados para ler e gravar nesse campo de backup. Se a propriedade automática não tiver nenhum acessador set, o campo de backup será considerado `readonly` ([campos somente leitura](#)). Assim como um `readonly` campo, uma propriedade automática somente getter também pode ser atribuída ao corpo de um construtor da classe delimitadora. Tal atribuição atribui diretamente ao campo de apoio somente leitura da propriedade.

Uma propriedade automática pode, opcionalmente, ter um *property\_initializer*, que é aplicado diretamente ao campo de backup como um *variable\_initializer* ([inicializadores de variável](#)).

O exemplo a seguir:

C#

```
public class Point {  
    public int X { get; set; } = 0;  
    public int Y { get; set; } = 0;  
}
```

é equivalente à declaração a seguir:

C#

```
public class Point {  
    private int __x = 0;  
    private int __y = 0;  
    public int X { get { return __x; } set { __x = value; } }
```

```
    public int Y { get { return __y; } set { __y = value; } }
```

O exemplo a seguir:

C#

```
public class ReadOnlyPoint
{
    public int X { get; }
    public int Y { get; }
    public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

é equivalente à declaração a seguir:

C#

```
public class ReadOnlyPoint
{
    private readonly int __x;
    private readonly int __y;
    public int X { get { return __x; } }
    public int Y { get { return __y; } }
    public ReadOnlyPoint(int x, int y) { __x = x; __y = y; }
}
```

Observe que as atribuições para o campo `ReadOnly` são legais, pois elas ocorrem dentro do construtor.

## Acessibilidade

Se um acessador tiver um *accessor\_modifier*, o domínio de acessibilidade ([domínios de acessibilidade](#)) do acessador será determinado usando a acessibilidade declarada do *accessor\_modifier*. Se um acessador não tiver um *accessor\_modifier*, o domínio de acessibilidade do acessador será determinado da acessibilidade declarada da propriedade ou do indexador.

A presença de um *accessor\_modifier* nunca afeta a pesquisa de Membros ([operadores](#)) ou a resolução de sobrecarga ([resolução de sobrecarga](#)). Os modificadores na propriedade ou no indexador sempre determinam a qual propriedade ou indexador está associado, independentemente do contexto do acesso.

Depois que uma determinada propriedade ou um indexador tiver sido selecionado, os domínios de acessibilidade dos acessadores específicos envolvidos serão usados para

determinar se esse uso é válido:

- Se o uso for como um valor ([valores de expressões](#)), o `get` acessador deverá existir e estar acessível.
- Se o uso for como o destino de uma atribuição simples ([atribuição simples](#)), o `set` acessador deverá existir e estar acessível.
- Se o uso for como o destino da atribuição composta ([atribuição composta](#)) ou como o destino dos `++` `--` operadores ou (membros de [função](#). 9, [expressões de invocação](#)), os acessadores `get` e o `set` acessador deverão existir e estar acessíveis.

No exemplo a seguir, a propriedade `A.Text` é ocultada pela propriedade `B.Text`, mesmo em contextos onde apenas o `set` acessador é chamado. Por outro lado, a propriedade `B.Count` não é acessível para classe `M`, portanto, a propriedade acessível `A.Count` é usada em seu lugar.

C#

```
class A
{
    public string Text {
        get { return "hello"; }
        set { }
    }

    public int Count {
        get { return 5; }
        set { }
    }
}

class B: A
{
    private string text = "goodbye";
    private int count = 0;

    new public string Text {
        get { return text; }
        protected set { text = value; }
    }

    new protected int Count {
        get { return count; }
        set { count = value; }
    }
}

class M
{
```

```

static void Main() {
    B b = new B();
    b.Count = 12;           // Calls A.Count set accessor
    int i = b.Count;       // Calls A.Count get accessor
    b.Text = "howdy";      // Error, B.Text set accessor not
                           // accessible
    string s = b.Text;     // Calls B.Text get accessor
}

```

Um acessador que é usado para implementar uma interface pode não ter um *accessor\_modifier*. Se apenas um acessador for usado para implementar uma interface, o outro acessador poderá ser declarado com um *accessor\_modifier*:

C#

```

public interface I
{
    string Prop { get; }
}

public class C: I
{
    public string Prop {
        get { return "April"; }           // Must not have a modifier here
        internal set {...}             // Ok, because I.Prop has no set
accessor
    }
}

```

## Acessadores de propriedade virtual, sealed, override e abstract

Uma `virtual` declaração de propriedade especifica que os acessadores da propriedade são virtuais. O `virtual` modificador se aplica a ambos os acessadores de uma propriedade de leitura/gravação — não é possível que apenas um acessador de uma propriedade de leitura/gravação seja virtual.

Uma `abstract` declaração de propriedade especifica que os acessadores da propriedade são virtuais, mas não fornecem uma implementação real dos acessadores. Em vez disso, as classes derivadas não abstratas são necessárias para fornecer sua própria implementação para os acessadores, substituindo a propriedade. Como um acessador de uma declaração de propriedade abstrata não fornece implementação real, sua *accessor\_body* simplesmente consiste em um ponto-e-vírgula.

Uma declaração de propriedade que inclui os `abstract` `override` modificadores e especifica que a propriedade é abstrata e substitui uma propriedade base. Os acessadores de tal propriedade também são abstratos.

As declarações de propriedade abstratas só são permitidas em classes abstratas ([classes abstratas](#)). Os acessadores de uma propriedade virtual herdada podem ser substituídos em uma classe derivada, incluindo uma declaração de propriedade que especifica uma `override` diretiva. Isso é conhecido como uma **declaração de propriedade de substituição**. Uma declaração de propriedade de substituição não declara uma nova propriedade. Em vez disso, ele simplesmente especializa as implementações dos acessadores de uma propriedade virtual existente.

Uma declaração de propriedade de substituição deve especificar exatamente os mesmos modificadores de acessibilidade, tipo e nome que a propriedade herdada. Se a propriedade Inherited tiver apenas um único acessador (ou seja, se a propriedade herdada for somente leitura ou somente gravação), a propriedade de substituição deverá incluir somente esse acessador. Se a propriedade Inherited incluir os acessadores (ou seja, se a propriedade herdada for Read-Write), a propriedade de substituição poderá incluir um único acessador ou ambos os acessadores.

Uma declaração de propriedade de substituição pode incluir o `sealed` modificador. O uso desse modificador impede que uma classe derivada substitua ainda mais a propriedade. Os acessadores de uma propriedade selada também são lacrados.

Exceto pelas diferenças na sintaxe de declaração e invocação, os acessadores `virtual`, `sealed`, `override` e `abstract` se comportam exatamente como métodos virtuais, lacrados, de substituição e abstratos. Especificamente, as regras descritas em [métodos virtuais](#), [métodos de substituição](#), métodos [lacrados](#) e [métodos abstratos](#) se aplicam como se os acessadores fossem métodos de um formulário correspondente:

- Um `get` acessador corresponde a um método sem parâmetros com um valor de retorno do tipo de propriedade e os mesmos modificadores que a propriedade recipiente.
- Um `set` acessador corresponde a um método com um parâmetro de valor único do tipo de propriedade, um `void` tipo de retorno e os mesmos modificadores que a propriedade recipiente.

No exemplo

C#

```
abstract class A
{
    int y;
```

```

public virtual int X {
    get { return 0; }
}

public virtual int Y {
    get { return y; }
    set { y = value; }
}

public abstract int Z { get; set; }
}

```

`X` é uma propriedade somente leitura virtual, `Y` é uma propriedade de leitura/gravação virtual e `Z` é uma propriedade de leitura/gravação abstrata. Como `Z` é abstrato, a classe recipiente `A` também deve ser declarada abstrata.

Uma classe derivada de `A` é mostrada abaixo:

```

C#

class B: A
{
    int z;

    public override int X {
        get { return base.X + 1; }
    }

    public override int Y {
        set { base.Y = value < 0? 0: value; }
    }

    public override int Z {
        get { return z; }
        set { z = value; }
    }
}

```

Aqui, as declarações de `X`, `Y` e `Z` estão substituindo declarações de propriedade. Cada declaração de propriedade corresponde exatamente aos modificadores de acessibilidade, ao tipo e ao nome da propriedade herdada correspondente. O `get` acessador do `X` e o `set` acessador do `Y` usam a `base` palavra-chave para acessar os acessadores herdados. A declaração de `Z` substitui ambos os acessadores abstratos – portanto, não há nenhum membro de função abstrata pendente no `B` e `B` é permitido ser uma classe não abstrata.

Quando uma propriedade é declarada como um `override`, todos os acessadores substituídos devem ser acessíveis para o código de substituição. Além disso, a acessibilidade declarada da propriedade ou do indexador em si, e dos acessadores, deve corresponder à do membro e aos acessadores substituídos. Por exemplo:

```
C#  
  
public class B  
{  
    public virtual int P {  
        protected set {...}  
        get {...}  
    }  
}  
  
public class D: B  
{  
    public override int P {  
        protected set {...}           // Must specify protected here  
        get {...}                   // Must not have a modifier here  
    }  
}
```

## Eventos

Um `*evento*` é um membro que permite que um objeto ou classe forneça notificações. Os clientes podem anexar código executável para eventos fornecendo `_manipuladores de eventos_`.

Os eventos são declarados usando `event_declaration`s:

```
antlr  
  
event_declaration  
: attributes? event_modifier* 'event' type variable_declarators ';' | attributes? event_modifier* 'event' type member_name '{' event_accessor_declarations '}' ;  
  
event_modifier  
: 'new'  
| 'public'  
| 'protected'  
| 'internal'  
| 'private'  
| 'static'  
| 'virtual'  
| 'sealed'  
| 'override'
```

```

| 'abstract'
| 'extern'
| event_modifier_unsafe
;

event_accessor_declarations
: add_accessor_declaration remove_accessor_declaration
| remove_accessor_declaration add_accessor_declaration
;

add_accessor_declaration
: attributes? 'add' block
;

remove_accessor_declaration
: attributes? 'remove' block
;

```

Um *event\_declaration* pode incluir um conjunto de *atributos* ([atributos](#)) e uma combinação válida dos quatro modificadores de acesso ([modificadores de acesso](#)), o `new` ([o novo modificador](#)), `static` ([métodos estáticos e de instância](#)), `virtual` ([métodos virtuais](#)), ([métodos de `override` substituição](#)), ([métodos `sealed` lacrados](#)), os `abstract` modificadores ([métodos abstratos](#)) e `extern` ([métodos externos](#)).

As declarações de evento estão sujeitas às mesmas regras que as declarações de método ([métodos](#)) em relação a combinações válidas de modificadores.

O *tipo* de uma declaração de evento deve ser um *delegate\_type* ([tipos de referência](#)) e esse *delegate\_type* deve ser pelo menos acessível como o próprio evento ([restrições de acessibilidade](#)).

Uma declaração de evento pode incluir *event\_accessor\_declarations*. No entanto, se não for, para eventos não-extern e não abstratos, o compilador fornecerá esses itens automaticamente ([eventos semelhantes a campo](#)); para eventos extern, os acessadores são fornecidos externamente.

Uma declaração de evento que omite *event\_accessor\_declarations* define um ou mais eventos, um para cada um dos *variable\_declarator*s. Os atributos e os modificadores se aplicam a todos os membros declarados por tal *event\_declaration*.

É um erro de tempo de compilação para um *event\_declaration* incluir tanto o `abstract` modificador quanto o *event\_accessor\_declarations* delimitado por chaves.

Quando uma declaração de evento inclui um `extern` modificador, o evento é considerado um [\\*evento externo \\_](#). Como uma declaração de evento externo não

fornecer implementação real, é um erro para que ela inclua o `extern` modificador e o `_event_accessor_declarations *`.

É um erro de tempo de compilação para uma `variable_declarator` de uma declaração de evento com `abstract` um `external` modificador ou para incluir um `variable_initializer`.

Um evento pode ser usado como o operando esquerdo dos `+=` `-=` operadores e ([atribuição de evento](#)). Esses operadores são usados, respectivamente, para anexar manipuladores de eventos ou para remover manipuladores de eventos de um evento, e os modificadores de acesso do evento controlam os contextos nos quais essas operações são permitidas.

Como `+=` e `-=` são as únicas operações que são permitidas em um evento fora do tipo que declara o evento, o código externo pode adicionar e remover manipuladores para um evento, mas não pode obter ou modificar a lista subjacente de manipuladores de eventos.

Em uma operação do formulário `x += y` ou `x -= y`, quando `x` é um evento e a referência ocorre fora do tipo que contém a declaração de `x`, o resultado da operação tem o tipo `void` (em vez de ter o tipo de `x`, com o valor de `x` após a atribuição). Essa regra proíbe que o código externo examine indiretamente o delegado subjacente de um evento.

O exemplo a seguir mostra como os manipuladores de eventos são anexados a instâncias da `Button` classe:

C#

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;
}

public class LoginDialog: Form
{
    Button OkButton;
    Button CancelButton;

    public LoginDialog() {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
    }
}
```

```

    void OkButtonClick(object sender, EventArgs e) {
        // Handle OkButton.Click event
    }

    void CancelButtonClick(object sender, EventArgs e) {
        // Handle CancelButton.Click event
    }
}

```

Aqui, o `LoginDialog` Construtor de instância cria duas `Button` instâncias e anexa manipuladores de eventos aos `Click` eventos.

## Eventos do tipo campo

Dentro do texto do programa da classe ou struct que contém a declaração de um evento, determinados eventos podem ser usados como campos. Para ser usado dessa forma, um evento não deve ser `abstract` ou `extern`, e não deve incluir explicitamente `event_accessor_declarations`. Tal evento pode ser usado em qualquer contexto que permita um campo. O campo contém um delegado ([delegados](#)) que se refere à lista de manipuladores de eventos que foram adicionados ao evento. Se nenhum manipulador de eventos tiver sido adicionado, o campo conterá `null`.

No exemplo

```
C#
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;

    protected void OnClick(EventArgs e) {
        if (Click != null) Click(this, e);
    }

    public void Reset() {
        Click = null;
    }
}
```

`Click` é usado como um campo dentro da `Button` classe. Como demonstra o exemplo, o campo pode ser examinado, modificado e usado em expressões de invocação de delegado. O `OnClick` método na `Button` classe "gera" o `Click` evento. A noção de gerar um evento é precisamente equivalente a invocar o delegado representado pelo evento — assim, não há constructos de linguagem especial para gerar eventos. Observe que a

invocação de delegado é precedida por uma verificação que garante que o delegado não seja nulo.

Fora da declaração da `Button` classe, o `Click` membro só pode ser usado no lado esquerdo dos `+=` `-=` operadores e, como em

```
C#
```

```
b.Click += new EventHandler(...);
```

que acrescenta um delegado à lista de invocação do `Click` evento e

```
C#
```

```
b.Click -= new EventHandler(...);
```

que remove um delegado da lista de invocação do `Click` evento.

Ao compilar um evento do tipo campo, o compilador cria automaticamente o armazenamento para manter o delegado e cria acessadores para o evento que adiciona ou remove manipuladores de eventos ao campo delegado. As operações de adição e remoção são thread-safe, e pode (mas não é necessário) ser feito enquanto mantém o bloqueio ([a instrução Lock](#)) no objeto recipiente para um evento de instância ou o tipo `Object` ([expressões de criação de objeto anônimo](#)) para um evento estático.

Assim, uma declaração de evento de instância do formulário:

```
C#
```

```
class X
{
    public event D Ev;
}
```

será compilado para algo equivalente a:

```
C#
```

```
class X
{
    private D __Ev; // field to hold the delegate

    public event D Ev {
        add {
            /* add the delegate in a thread safe way */
        }
    }
}
```

```
remove {
    /* remove the delegate in a thread safe way */
}
}
```

Dentro da classe `X`, referências à `Ev` no lado esquerdo dos `+=` `-=` operadores e fazem com que os acessadores adicionar e remover sejam invocados. Todas as outras referências a `Ev` são compiladas para referenciar o campo oculto `_Ev` em vez disso ([acesso de membro](#)). O nome "`_Ev`" é arbitrário; o campo oculto pode ter qualquer nome ou nenhum nome.

## Acessadores de eventos

As declarações de evento normalmente omitem `event_accessor_declarations`, como no `Button` exemplo acima. Uma situação para isso envolve o caso em que o custo de armazenamento de um campo por evento não é aceitável. Nesses casos, uma classe pode incluir `event_accessor_declarations` e usar um mecanismo privado para armazenar a lista de manipuladores de eventos.

O `event_accessor_declarations` de um evento especifica as instruções Executáveis associadas à adição e remoção de manipuladores de eventos.

As declarações de acessador consistem em um `add_accessor_declaration` e um `remove_accessor_declaration`. Cada declaração de acessador consiste no token `add` ou `remove` seguido por um *bloco*. O *bloco* associado a um `add_accessor_declaration` especifica as instruções a serem executadas quando um manipulador de eventos é adicionado e o *bloco* associado a um `remove_accessor_declaration` especifica as instruções a serem executadas quando um manipulador de eventos é removido.

Cada `add_accessor_declaration` e `remove_accessor_declaration` corresponde a um método com um parâmetro de valor único do tipo de evento e um `void` tipo de retorno. O parâmetro implícito de um acessador de eventos é nomeado `value`. Quando um evento é usado em uma atribuição de evento, o acessador de evento apropriado é usado. Especificamente, se o operador de atribuição for `+=`, o acessador Add será usado e, se o operador de atribuição for `-=`, o acessador de remoção será usado. Em ambos os casos, o operando do lado direito do operador de atribuição é usado como o argumento para o acessador de eventos. O bloco de um `add_accessor_declaration` ou um `remove_accessor_declaration` deve estar em conformidade com as regras para `void` métodos descritos no [corpo do método](#). Em particular, `return` instruções nesse bloco não são permitidas para especificar uma expressão.

Como um acessador de evento implicitamente tem um parâmetro chamado `value`, ele é um erro de tempo de compilação para uma variável local ou constante declarada em um acessador de evento para ter esse nome.

No exemplo

```
C#  
  
class Control: Component  
{  
    // Unique keys for events  
    static readonly object mouseDownEventKey = new object();  
    static readonly object mouseUpEventKey = new object();  
  
    // Return event handler associated with key  
    protected Delegate GetEventHandler(object key) {...}  
  
    // Add event handler associated with key  
    protected void AddEventHandler(object key, Delegate handler) {...}  
  
    // Remove event handler associated with key  
    protected void RemoveEventHandler(object key, Delegate handler) {...}  
  
    // MouseDown event  
    public event MouseEventHandler MouseDown {  
        add { AddEventHandler(mouseDownEventKey, value); }  
        remove { RemoveEventHandler(mouseDownEventKey, value); }  
    }  
  
    // MouseUp event  
    public event MouseEventHandler MouseUp {  
        add { AddEventHandler(mouseUpEventKey, value); }  
        remove { RemoveEventHandler(mouseUpEventKey, value); }  
    }  
  
    // Invoke the MouseUp event  
    protected void OnMouseUp(MouseEventArgs args) {  
        MouseEventHandler handler;  
        handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);  
        if (handler != null)  
            handler(this, args);  
    }  
}
```

a `Control` classe implementa um mecanismo de armazenamento interno para eventos. O `AddEventHandler` método associa um valor delegado a uma chave, o `GetEventHandler` método retorna o delegado atualmente associado a uma chave e o `RemoveEventHandler` método Remove um delegado como um manipulador de eventos para o evento especificado. Supostamente, o mecanismo de armazenamento subjacente foi projetado

de forma que não há nenhum custo para associar um `null` valor delegado a uma chave e, portanto, eventos sem tratamento não consomem nenhum armazenamento.

## Eventos estáticos e de instância

Quando uma declaração de evento inclui um `static` modificador, o evento é considerado um *\*evento estático\**. Quando nenhum `static` modificador estiver presente, o evento será considerado como um *\*evento de instância\**.

Um evento estático não está associado a uma instância específica e é um erro de tempo de compilação para se referir aos `this` acessadores de um evento estático.

Um evento de instância é associado a uma determinada instância de uma classe, e essa instância pode ser acessada como `this` ([esse acesso](#)) nos acessadores desse evento.

Quando um evento é referenciado em um *member\_access* ([acesso de membro](#)) do formulário `E.M`, se `M` for um evento estático, deve-se `E` denotar um tipo contendo `M`, e se `M` for um evento de instância, e deve indicar uma instância de um tipo que contém `M`.

As diferenças entre os membros estático e de instância são discutidas mais detalhadamente em [membros estáticos e de instância](#).

## Acessadores de evento virtual, sealed, override e abstract

Uma `virtual` declaração de evento especifica que os acessadores desse evento são virtuais. O `virtual` modificador se aplica a ambos os acessadores de um evento.

Uma `abstract` declaração de evento especifica que os acessadores do evento são virtuais, mas não fornecem uma implementação real dos acessadores. Em vez disso, as classes derivadas não abstratas são necessárias para fornecer sua própria implementação para os acessadores, substituindo o evento. Como uma declaração de evento `abstract` não fornece implementação real, ela não pode fornecer *event\_accessor\_declarations* delimitado por chaves.

Uma declaração de evento que inclui os `abstract` `override` modificadores especifica que o evento é abstrato e substitui um evento base. Os acessadores de tal evento também são abstratos.

As declarações de evento `abstract` só são permitidas em classes abstratas ([classes abstratas](#)).

Os acessadores de um evento virtual herdado podem ser substituídos em uma classe derivada, incluindo uma declaração de evento que especifica um `override` modificador. Isso é conhecido como uma **declaração de evento de substituição**. Uma declaração de evento de substituição não declara um novo evento. Em vez disso, ele simplesmente especializa as implementações dos acessadores de um evento virtual existente.

Uma declaração de evento de substituição deve especificar exatamente os mesmos modificadores de acessibilidade, tipo e nome que o evento substituído.

Uma declaração de evento de substituição pode incluir o `sealed` modificador. O uso desse modificador impede que uma classe derivada substitua o evento. Os acessadores de um evento lacrado também são lacrados.

É um erro de tempo de compilação para uma declaração de evento de substituição para incluir um `new` modificador.

Exceto pelas diferenças na sintaxe de declaração e invocação, os acessadores virtual, sealed, override e abstract se comportam exatamente como métodos virtuais, lacrados, de substituição e abstratos. Especificamente, as regras descritas em [métodos virtuais](#), [métodos de substituição](#), métodos [lacrados](#) e [métodos abstratos](#) se aplicam como se os acessadores fossem métodos de um formulário correspondente. Cada acessador corresponde a um método com um parâmetro de valor único do tipo de evento, um `void` tipo de retorno e os mesmos modificadores que o evento recipiente.

## Indexadores

Um `*indexador_` é um membro que permite que um objeto seja indexado da mesma maneira que uma matriz. Os indexadores são declarados usando `_indexer_declarator *`:

```
antlr

indexer_declaration
: attributes? indexer_modifier* indexer_declarator indexer_body
;

indexer_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'virtual'
| 'sealed'
| 'override'
```

```

| 'abstract'
| 'extern'
| indexer_modifier_unsafe
;

indexer_declarator
: type 'this' '[' formal_parameter_list ']'
| type interface_type '.' 'this' '[' formal_parameter_list ']'
;

indexer_body
: '{' accessor_declarations '}'
| '=>' expression ';'
;

```

Um *indexer\_declaration* pode incluir um conjunto de *atributos* ([atributos](#)) e uma combinação válida dos quatro modificadores de acesso ([modificadores de acesso](#)), os `new` ([o novo modificador](#)), os modificadores (métodos virtuais), (métodos de `virtual` `override` substituição), `sealed` ([métodos lacrados](#)), `abstract` ([métodos abstratos](#)) e `extern` ([métodos externos](#)).

As declarações do indexador estão sujeitas às mesmas regras que as declarações de método ([métodos](#)) em relação a combinações válidas de modificadores, com a única exceção de que o modificador estático não é permitido em uma declaração de indexador.

Os modificadores `virtual`, `override` e `abstract` são mutuamente exclusivos, exceto em um caso. Os `abstract` `override` modificadores e podem ser usados juntos para que um indexador abstrato possa substituir um virtual.

O *tipo* de uma declaração de indexador especifica o tipo de elemento do indexador introduzido pela declaração. A menos que o indexador seja uma implementação de membro de interface explícita, o *tipo* é seguido pela palavra-chave `this`. Para uma implementação de membro de interface explícita, o *tipo* é seguido por um *interface\_type*, um `.` e a palavra-chave `this`. Ao contrário de outros membros, os indexadores não têm nomes definidos pelo usuário.

O *formal\_parameter\_list* especifica os parâmetros do indexador. A lista de parâmetros formais de um indexador corresponde ao de um método ([parâmetros de método](#)), exceto pelo menos um parâmetro que deve ser especificado e que os `ref` modificadores e de `out` parâmetro não são permitidos.

O *tipo* de um indexador e cada um dos tipos referenciados no *formal\_parameter\_list* devem ser pelo menos tão acessíveis quanto o próprio indexador ([restrições de acessibilidade](#)).

Uma `indexer_body` pode consistir em um **corpo de acessador** ou um **corpo de expressão**. Em um corpo de acessador, `_accessor_declarations` \*, que deve ser incluído em `{ tokens` `"" e " }` `"`, declare os acessadores ([acessadores](#)) da propriedade. Os acessadores especificam as instruções Executáveis associadas à leitura e gravação da propriedade.

Um corpo de expressão que consiste em " `=>` " seguido por uma expressão `E` e um ponto-e-vírgula é exatamente equivalente ao corpo da instrução `{ get { return E; } }` e, portanto, só pode ser usado para especificar indexadores somente getter em que o resultado do getter é fornecido por uma única expressão.

Embora a sintaxe para acessar um elemento do indexador seja a mesma que para um elemento de matriz, um elemento de indexador não é classificado como uma variável. Portanto, não é possível passar um elemento do indexador como um `ref` argumento ou `out`.

A lista de parâmetros formais de um indexador define a assinatura ([assinaturas e sobrecarga](#)) do indexador. Especificamente, a assinatura de um indexador consiste no número e tipos de seus parâmetros formais. O tipo de elemento e os nomes dos parâmetros formais não fazem parte da assinatura de um indexador.

A assinatura de um indexador deve ser diferente das assinaturas de todos os outros indexadores declarados na mesma classe.

Indexadores e propriedades são muito semelhantes em conceito, mas diferem das seguintes maneiras:

- Uma propriedade é identificada por seu nome, enquanto um indexador é identificado por sua assinatura.
- Uma propriedade é acessada por meio de um *Simple\_name* ([nomes simples](#)) ou uma *member\_access* ([acesso de membro](#)), enquanto um elemento de indexador é acessado por meio de um *element\_access* ([acesso do indexador](#)).
- Uma propriedade pode ser um `static` membro, enquanto um indexador é sempre um membro de instância.
- Um `get` acessador de uma propriedade corresponde a um método sem parâmetros, enquanto um `get` acessador de um indexador corresponde a um método com a mesma lista de parâmetros formais que o indexador.
- Um `set` acessador de uma propriedade corresponde a um método com um único parâmetro chamado `value`, enquanto `set` que um acessador de um indexador corresponde a um método com a mesma lista de parâmetros formais que o indexador, mais um parâmetro adicional chamado `value`.
- É um erro de tempo de compilação para um acessador de indexador declarar uma variável local com o mesmo nome que um parâmetro de indexador.

- Em uma declaração de propriedade de substituição, a propriedade herdada é acessada usando a sintaxe `base.P`, onde `P` é o nome da propriedade. Em uma declaração de substituição do indexador, o indexador herdado é acessado usando a sintaxe `base[E]`, em que `E` é uma lista separada por vírgulas de expressões.
- Não há nenhum conceito de "indexador implementado automaticamente". É um erro ter um indexador não-abstrato e não externo com acessadores de ponto e vírgula.

Além dessas diferenças, todas as regras definidas em [acessadores](#) e [Propriedades implementadas automaticamente](#) se aplicam a acessadores indexadores, bem como a acessadores de propriedade.

Quando uma declaração de indexador inclui um `extern` modificador, o indexador é considerado um `*indexador externo`. Como uma declaração externa do indexador não fornece nenhuma implementação real, cada uma de suas `_accessor_declarations` \* consiste em um ponto-e-vírgula.

O exemplo a seguir declara uma `BitArray` classe que implementa um indexador para acessar os bits individuais na matriz de bits.

C#

```
using System;

class BitArray
{
    int[] bits;
    int length;

    public BitArray(int length) {
        if (length < 0) throw new ArgumentException();
        bits = new int[((length - 1) >> 5) + 1];
        this.length = length;
    }

    public int Length {
        get { return length; }
    }

    public bool this[int index] {
        get {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            return (bits[index >> 5] & 1 << index) != 0;
        }
        set {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            bits[index >> 5] = bits[index >> 5] | (value & 1) << index;
        }
    }
}
```

```

        }
        if (value) {
            bits[index >> 5] |= 1 << index;
        }
        else {
            bits[index >> 5] &= ~(1 << index);
        }
    }
}

```

Uma instância da `BitArray` classe consome substancialmente menos memória do que uma correspondente `bool[]` (já que cada valor do primeiro ocupa apenas um bit em vez do último byte), mas permite as mesmas operações que um `bool[]`.

A classe a seguir `CountPrimes` usa um `BitArray` e o algoritmo clássico "sieve" para calcular o número de primos entre 1 e um determinado máximo:

C#

```

class CountPrimes
{
    static int Count(int max) {
        BitArray flags = new BitArray(max + 1);
        int count = 1;
        for (int i = 2; i <= max; i++) {
            if (!flags[i]) {
                for (int j = i * 2; j <= max; j += i) flags[j] = true;
                count++;
            }
        }
        return count;
    }

    static void Main(string[] args) {
        int max = int.Parse(args[0]);
        int count = Count(max);
        Console.WriteLine("Found {0} primes between 1 and {1}", count, max);
    }
}

```

Observe que a sintaxe para acessar elementos do `BitArray` é exatamente a mesma do para um `bool[]`.

O exemplo a seguir mostra uma classe de grade 26 \* 10 que tem um indexador com dois parâmetros. O primeiro parâmetro deve ser uma letra maiúscula ou minúscula no intervalo de A-Z e o segundo deve ser um número inteiro no intervalo de 0-9.

C#

```

using System;

class Grid
{
    const int NumRows = 26;
    const int NumCols = 10;

    int[,] cells = new int[NumRows, NumCols];

    public int this[char c, int col] {
        get {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            return cells[c - 'A', col];
        }

        set {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            cells[c - 'A', col] = value;
        }
    }
}

```

## Sobrecarga do indexador

As regras de resolução de sobrecarga do indexador são descritas em [inferência de tipos](#).

## Operadores

Um **\*Operator\_** é um membro que define o significado de um operador de expressão que pode ser aplicado a instâncias da classe. Os operadores são declarados usando `_operator_declaration * s:`

antlr

```

operator_declarator
: attributes? operator_modifier+ operator_declarator operator_body

```

```

;

operator_modifier
: 'public'
| 'static'
| 'extern'
| operator_modifier_unsafe
;

operator_declarator
: unary_operator_declarator
| binary_operator_declarator
| conversion_operator_declarator
;

unary_operator_declarator
: type 'operator' overloadable_unary_operator '(' type identifier ')'
;

overloadable_unary_operator
: '+' | '-' | '!' | '~' | '++' | '--' | 'true' | 'false'
;

binary_operator_declarator
: type 'operator' overloadable_binary_operator '(' type identifier ',', type identifier ')'
;

overloadable_binary_operator
: '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '<<'
| right_shift | '==' | '!='
| '>' | '<' | '>=' | '<='
;

conversion_operator_declarator
: 'implicit' 'operator' type '(' type identifier ')'
| 'explicit' 'operator' type '(' type identifier ')'
;

operator_body
: block
| '=>' expression ';'
| ';'
;

```

Há três categorias de operadores que podem ser sobrecarregados: operadores unários ([operadores unários](#)), operadores binários ([operadores binários](#)) e operadores de conversão ([operadores de conversão](#)).

O *operator\_body* é um ponto-e-vírgula, um *corpo de instrução* ou um *corpo de expressão*. Um corpo de instrução consiste em um \_block \*, que especifica as instruções a serem executadas quando o operador é invocado. O bloco deve estar em conformidade com as regras para métodos de retorno de valor descritos no [corpo do](#)

**método**. Um corpo de expressão consiste em `=>` seguido por uma expressão e um ponto e vírgula e denota uma única expressão a ser executada quando o operador é invocado.

Para `extern` operadores, a `operator_body` consiste apenas de um ponto e vírgula. Para todos os outros operadores, o `operator_body` é um corpo de bloco ou um corpo de expressão.

As regras a seguir se aplicam a todas as declarações de operador:

- Uma declaração de operador deve incluir um `public` e um `static` modificador.
- Os parâmetros de um operador devem ser parâmetros de valor (parâmetros `devalor`). É um erro de tempo de compilação para uma declaração de operador especificar `ref` ou `out` parâmetros.
- A assinatura de um operador ([operadores unários](#), [operadores binários](#), [operadores de conversão](#)) deve ser diferente das assinaturas de todos os outros operadores declarados na mesma classe.
- Todos os tipos referenciados em uma declaração de operador devem ser pelo menos tão acessíveis quanto o próprio operador ([restrições de acessibilidade](#)).
- É um erro para que o mesmo modificador apareça várias vezes em uma declaração de operador.

Cada categoria de operador impõe restrições adicionais, conforme descrito nas seções a seguir.

Como outros membros, os operadores declarados em uma classe base são herdados por classes derivadas. Como as declarações de operador sempre exigem a classe ou `struct` em que o operador é declarado para participar da assinatura do operador, não é possível que um operador declarado em uma classe derivada oculte um operador declarado em uma classe base. Assim, o `new` modificador nunca é necessário e, portanto, nunca é permitido em uma declaração de operador.

Informações adicionais sobre operadores unários e binários podem ser encontradas em [operadores](#).

Informações adicionais sobre operadores de conversão podem ser encontradas em [conversões definidas pelo usuário](#).

## Operadores unários

As regras a seguir se aplicam a declarações de operador unários, onde `T` denota o tipo de instância da classe ou `struct` que contém a declaração do operador:

- Um operador unário `+`, `-`, `!` ou `~` deve usar um único parâmetro do tipo `T` ou `T?` pode retornar qualquer tipo.
- Um `++` operador OR unário `--` deve usar um único parâmetro do tipo `T` ou `T?` deve retornar o mesmo tipo ou um tipo derivado dele.
- Um `true` operador OR unário `false` deve usar um único parâmetro do tipo `T` ou `T?` deve retornar o tipo `bool`.

A assinatura de um operador unário consiste no token do operador (`+, -, !, ~, ++, -`, `true` ou `false`) e no tipo do único parâmetro formal. O tipo de retorno não faz parte da assinatura de um operador unário, nem é o nome do parâmetro formal.

Os `true` `false` operadores e unários exigem declarações emparelhadas. Ocorrerá um erro em tempo de compilação se uma classe declarar um desses operadores sem também declarar o outro. Os `true` `false` operadores e são descritos mais detalhadamente em [operadores lógicos condicionais definidos pelo usuário](#) e [expressões booleanas](#).

O exemplo a seguir mostra uma implementação e o uso subsequente de `operator ++` para uma classe de vetor de inteiro:

C#

```
public class IntVector
{
    public IntVector(int length) {...}

    public int Length {...}                      // read-only property

    public int this[int index] {...}             // read-write indexer

    public static IntVector operator ++(IntVector iv) {
        IntVector temp = new IntVector(iv.Length);
        for (int i = 0; i < iv.Length; i++)
            temp[i] = iv[i] + 1;
        return temp;
    }
}

class Test
{
    static void Main() {
        IntVector iv1 = new IntVector(4);      // vector of 4 x 0
        IntVector iv2;

        iv2 = iv1++;    // iv2 contains 4 x 0, iv1 contains 4 x 1
        iv2 = ++iv1;    // iv2 contains 4 x 2, iv1 contains 4 x 2
    }
}
```

Observe como o método `Operator` retorna o valor produzido adicionando 1 ao operando, assim como os operadores de incremento de sufixo e decréscimo ([incrementos de sufixo e decremento](#)), e os operadores de incremento de prefixo e decrecimento ([incremento de prefixo e diminuição de operadores](#)). Ao contrário do C++, esse método não precisa modificar o valor de seu operando diretamente. Na verdade, modificar o valor do operando violaria a semântica padrão do operador de incremento de sufixo.

## Operadores binários

As regras a seguir se aplicam a declarações de operador binários, onde `T` denota o tipo de instância da classe ou struct que contém a declaração do operador:

- Um operador binário non-Shift deve usar dois parâmetros, pelo menos um dos quais deve ter tipo `T` ou `T?`, e pode retornar qualquer tipo.
- Um `<<` operador ou binário `>>` deve usar dois parâmetros, o primeiro deles deve ter o tipo `T` ou `T?` e o segundo deve ter o tipo `int` ou `int?`, e pode retornar qualquer tipo.

A assinatura de um operador binário consiste no token do operador (`..... + - * / % & | ^ << >> == != , > , < , >= ou <=`) e nos tipos dos dois parâmetros formais. O tipo de retorno e os nomes dos parâmetros formais não fazem parte da assinatura de um operador binário.

Determinados operadores binários exigem declarações emparelhadas. Para cada declaração de qualquer operador de um par, deve haver uma declaração correspondente do outro operador do par. Duas declarações de operador correspondem quando têm o mesmo tipo de retorno e o mesmo tipo para cada parâmetro. Os seguintes operadores exigem a declaração de pares:

- `operator ==` e `operator !=`
- `operator >` e `operator <`
- `operator >=` e `operator <=`

## Operadores de conversão

Uma declaração de operador de conversão apresenta uma **conversão definida pelo usuário** ([conversões definidas pelo usuário](#)) que aumenta as conversões implícitas e explícitas predefinidas.

Uma declaração de operador de conversão que inclui a `implicit` palavra-chave apresenta uma conversão implícita definida pelo usuário. Conversões implícitas podem ocorrer em várias situações, incluindo invocações de membro de função, expressões de conversão e atribuições. Isso é descrito mais detalhadamente em [conversões implícitas](#).

Uma declaração de operador de conversão que inclui a `explicit` palavra-chave apresenta uma conversão explícita definida pelo usuário. Conversões explícitas podem ocorrer em expressões de conversão e são descritas em [conversões explícitas](#).

Um operador de conversão converte de um tipo de origem, indicado pelo tipo de parâmetro do operador de conversão, em um tipo de destino, indicado pelo tipo de retorno do operador de conversão.

Para um determinado tipo de origem `S` e tipo `T` de destino, se `S` ou `T` forem tipos anuláveis, avise `s0` e `t0` faça referência aos seus tipos subjacentes, caso contrário, `s0` e `t0` sejam iguais a `S` e `T` respectivamente. Uma classe ou estrutura tem permissão para declarar uma conversão de um tipo de origem `S` para um tipo de destino `T` somente se todas as seguintes opções forem verdadeiras:

- `s0` e `t0` são tipos diferentes.
- `s0` ou `t0` é o tipo de classe ou struct no qual a declaração do operador ocorre.
- Nem `s0` nem `t0` é um *interface\_type*.
- Excluindo conversões definidas pelo usuário, uma conversão não existe de `S` ou para `T` `T` `S`.

Para os fins dessas regras, quaisquer parâmetros de tipo associados a `S` ou `T` são considerados como tipos exclusivos que não têm nenhuma relação de herança com outros tipos, e quaisquer restrições nesses parâmetros de tipo são ignoradas.

No exemplo

```
C#  
  
class C<T> {...}  
  
class D<T>: C<T>  
{  
    public static implicit operator C<int>(D<T> value) {...}      // Ok  
    public static implicit operator C<string>(D<T> value) {...}    // Ok  
    public static implicit operator C<T>(D<T> value) {...}        // Error  
}
```

as primeiras duas declarações de operador são permitidas porque, para fins de [indexadores](#)<sup>3</sup> e, `T` respectivamente, `int` `string` são consideradas tipos exclusivos sem

nenhuma relação. No entanto, o terceiro operador é um erro porque `C<T>` é a classe base de `D<T>`.

Da segunda regra, ela segue que um operador de conversão deve converter de ou para o tipo class ou struct no qual o operador é declarado. Por exemplo, é possível que um tipo de classe ou estrutura `C` defina uma conversão de `C` para `int` e de `int` para `C`, mas não de `int` para `bool`.

Não é possível redefinir diretamente uma conversão predefinida. Assim, os operadores de conversão não têm permissão para converter de ou para `object` porque as conversões implícitas e explícitas já existem entre o `object` e todos os outros tipos. Da mesma forma, nem os tipos de origem nem de destino de uma conversão podem ser um tipo base do outro, já que uma conversão já existirá.

No entanto, é possível declarar operadores em tipos genéricos que, para argumentos de tipo específicos, especificar conversões que já existem como conversões predefinidas. No exemplo

C#

```
struct Convertible<T>
{
    public static implicit operator Convertible<T>(T value) {...}
    public static explicit operator T(Convertible<T> value) {...}
}
```

Quando Type `object` é especificado como um argumento de tipo para `T`, o segundo operador declara uma conversão que já existe (um implícito e, portanto, também uma conversão explícita existe de qualquer tipo para tipo `object`).

Nos casos em que existe uma conversão predefinida entre dois tipos, todas as conversões definidas pelo usuário entre esses tipos serão ignoradas. Especificamente:

- Se uma conversão implícita predefinida ([conversões implícitas](#)) existir de tipo `S` para tipo `T`, todas as conversões definidas pelo usuário (implícitas ou explícitas) de `S` para `T` serão ignoradas.
- Se uma conversão explícita predefinida ([conversões explícitas](#)) existir de tipo `S` para tipo `T`, todas as conversões explícitas definidas pelo usuário de `S` para `T` serão ignoradas. Além disso:

Se `T` for um tipo de interface, conversões implícitas definidas pelo usuário de `S` para `T` serão ignoradas.

Caso contrário, conversões implícitas definidas pelo usuário de `s` para o `T` ainda serão consideradas.

Para todos os tipos, mas `object` os operadores declarados pelo `Convertible<T>` tipo acima não entram em conflito com conversões predefinidas. Por exemplo:

C#

```
void F(int i, Convertible<int> n) {
    i = n;                                // Error
    i = (int)n;                            // User-defined explicit conversion
    n = i;                                 // User-defined implicit conversion
    n = (Convertible<int>)i;               // User-defined implicit conversion
}
```

No entanto, para `object` conversões de tipo predefinidas, oculte as conversões definidas pelo usuário em todos os casos, exceto uma:

C#

```
void F(object o, Convertible<object> n) {
    o = n;                                // Pre-defined boxing conversion
    o = (object)n;                          // Pre-defined boxing conversion
    n = o;                                 // User-defined implicit conversion
    n = (Convertible<object>)o;            // Pre-defined unboxing conversion
}
```

As conversões definidas pelo usuário não têm permissão para converter de ou para *interface\_type*s. Em particular, essa restrição garante que nenhuma transformação definida pelo usuário ocorra durante a conversão para um *interface\_type* e que uma conversão em um *interface\_type* terá sucesso somente se o objeto que está sendo convertido realmente implementar o *interface\_type* especificado.

A assinatura de um operador de conversão consiste no tipo de origem e no tipo de destino. (Observe que essa é a única forma de membro para a qual o tipo de retorno participa na assinatura.) A `implicit` `explicit` classificação ou de um operador de conversão não faz parte da assinatura do operador. Assim, uma classe ou struct não pode declarar um `implicit` operador de `explicit` conversão e um com os mesmos tipos de origem e destino.

Em geral, as conversões implícitas definidas pelo usuário devem ser projetadas para nunca gerar exceções e nunca perderem informações. Se uma conversão definida pelo usuário puder dar origem às exceções (por exemplo, porque o argumento de origem está fora do intervalo) ou a perda de informações (como descartar bits de ordem superior), essa conversão deve ser definida como uma conversão explícita.

No exemplo

C#

```
using System;

public struct Digit
{
    byte value;

    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
        this.value = value;
    }

    public static implicit operator byte(Digit d) {
        return d.value;
    }

    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
}
```

a conversão de `Digit` para `byte` é implícita porque ela nunca gera exceções ou perde informações, mas a conversão de `byte` para `Digit` é explícita, pois `Digit` só pode representar um subconjunto dos possíveis valores de um `byte`.

## Construtores de instância

Um **\*Construtor de instância \*** é um membro que implementa as ações necessárias para inicializar uma instância de uma classe. Construtores de instância são declarados usando `_constructor_declaration` \* s:

antlr

```
constructor_declaration
    : attributes? constructor_modifier* constructor_declarator
constructor_body
    ;

constructor_modifier
    : 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'extern'
    | constructor_modifier_unsafe
    ;
```

```
constructor_declarator
: identifier '(' formal_parameter_list? ')' constructor_initializer?
;

constructor_initializer
: ':' 'base' '(' argument_list? ')'
| ':' 'this' '(' argument_list? ')'
;

constructor_body
: block
| ';'
;
```

Uma *constructor\_declaration* pode incluir um conjunto de *atributos* ([atributos](#)), uma combinação válida dos quatro modificadores de acesso ([modificadores de acesso](#)) e um `extern` modificador ([métodos externos](#)). Uma declaração de construtor não tem permissão para incluir o mesmo modificador várias vezes.

O *identificador* de um *constructor\_declarator* deve nomear a classe na qual o construtor de instância é declarado. Se qualquer outro nome for especificado, ocorrerá um erro em tempo de compilação.

O *formal\_parameter\_list* opcional de um construtor de instância está sujeito às mesmas regras que o *formal\_parameter\_list* de um método ([métodos](#)). A lista de parâmetros formais define a assinatura ([assinaturas e sobrecarga](#)) de um construtor de instância e governa o processo pelo qual a resolução de sobrecarga ([inferência de tipos](#)) seleciona um construtor de instância específico em uma invocação.

Cada um dos tipos referenciados no *formal\_parameter\_list* de um construtor de instância deve ser pelo menos acessível como o próprio Construtor ([restrições de acessibilidade](#)).

O *constructor\_initializer* opcional especifica outro construtor de instância para invocar antes de executar as instruções fornecidas no *constructor\_body* deste construtor de instância. Isso é descrito mais detalhadamente em [inicializadores de Construtor](#).

Quando uma declaração de Construtor inclui um `extern` modificador, o construtor é considerado um **\*Construtor externo**. Como uma declaração de Construtor externo não fornece nenhuma implementação real, seu *\_constructor\_body* \* consiste em um ponto-e-vírgula. Para todos os outros construtores, o *constructor\_body* consiste em um bloco que especifica as instruções para inicializar uma nova instância da classe. Isso corresponde exatamente ao bloco de um método de instância com um `void` tipo de retorno ([corpo do método](#)).

Construtores de instância não são herdados. Portanto, uma classe não tem construtores de instância diferentes daqueles realmente declarados na classe. Se uma classe não contiver nenhuma declaração de construtor de instância, um construtor de instância padrão será fornecido automaticamente ([construtores padrão](#)).

Os construtores de instância são invocados por *object\_creation\_expression s* ([expressões de criação de objeto](#)) e por meio de *constructor\_initializer s*.

## Inicializadores de construtores

Todos os construtores de instância (exceto aqueles para classe `object`) incluem implicitamente uma invocação de outro construtor de instância imediatamente antes da *constructor\_body*. O construtor para invocar implicitamente é determinado pelo *constructor\_initializer*:

- Um inicializador de construtor de instância do formulário `base(argument_list)` ou `base()` faz com que um construtor de instância da classe base direta seja invocado. Esse construtor é selecionado usando *argument\_list*, se presente, e as regras de resolução de sobrecarga da [resolução de sobrecarga](#). O conjunto de construtores de instância de candidato consiste em todos os construtores de instância acessíveis contidos na classe base direta ou no construtor padrão ([construtores padrão](#)), se nenhum construtor de instância for declarado na classe base direta. Se esse conjunto estiver vazio, ou se um único Construtor de instância recomendada não puder ser identificado, ocorrerá um erro em tempo de compilação.
- Um inicializador de construtor de instância do formulário `this(argument-list)` ou `this()` faz com que um construtor de instância da própria classe seja invocado. O construtor é selecionado usando *argument\_list*, se presente, e as regras de resolução de sobrecarga da [resolução de sobrecarga](#). O conjunto de construtores de instância de candidato consiste em todos os construtores de instância acessíveis declarados na própria classe. Se esse conjunto estiver vazio, ou se um único Construtor de instância recomendada não puder ser identificado, ocorrerá um erro em tempo de compilação. Se uma declaração de construtor de instância incluir um inicializador de construtor que invoca o Construtor em si, ocorrerá um erro em tempo de compilação.

Se um construtor de instância não tiver nenhum inicializador de construtor, um inicializador de construtor do formulário `base()` será fornecido implicitamente.

Portanto, uma declaração de construtor de instância do formulário

```
C(...){...}
```

é exatamente equivalente a

```
C#
```

```
C(...): base() {...}
```

O escopo dos parâmetros fornecidos pelo *formal\_parameter\_list* de uma declaração de construtor de instância inclui o inicializador de construtor dessa declaração. Assim, um inicializador de construtor tem permissão para acessar os parâmetros do construtor. Por exemplo:

```
C#
```

```
class A
{
    public A(int x, int y) {}
}

class B: A
{
    public B(int x, int y): base(x + y, x - y) {}
}
```

Um inicializador de construtor de instância não pode acessar a instância que está sendo criada. Portanto, é um erro de tempo de compilação a ser referenciado `this` em uma expressão de argumento do inicializador de construtor, pois é um erro de tempo de compilação para uma expressão de argumento referenciar qualquer membro de instância por meio de um *Simple\_name*.

## Inicializadores de variável de instância

Quando um construtor de instância não tem nenhum inicializador de construtor ou tem um inicializador de construtor do formulário `base(...)`, esse construtor executa implicitamente as inicializações especificadas pelo *variable\_initializer*s dos campos de instância declarados em sua classe. Isso corresponde a uma sequência de atribuições que são executadas imediatamente após a entrada para o construtor e antes da invocação implícita do construtor da classe base direta. Os inicializadores de variável são executados na ordem textual em que aparecem na declaração de classe.

## Execução do Construtor

Inicializadores de variáveis são transformados em instruções de atribuição, e essas instruções de atribuição são executadas antes da invocação do construtor da instância da classe base. Essa ordenação garante que todos os campos de instância sejam inicializados por seus inicializadores variáveis antes que qualquer instrução que tenha acesso a essa instância seja executada.

Dado o exemplo

```
C#  
  
using System;  
  
class A  
{  
    public A() {  
        PrintFields();  
    }  
  
    public virtual void PrintFields() {}  
}  
  
class B: A  
{  
    int x = 1;  
    int y;  
  
    public B() {  
        y = -1;  
    }  
  
    public override void PrintFields() {  
        Console.WriteLine("x = {0}, y = {1}", x, y);  
    }  
}
```

Quando `new B()` é usado para criar uma instância do `B`, a seguinte saída é produzida:

```
Console  
  
x = 1, y = 0
```

O valor de `x` é 1 porque o inicializador de variável é executado antes que o construtor da instância da classe base seja invocado. No entanto, o valor de `y` é 0 (o valor padrão de um `int`) porque a atribuição a `y` não é executada até que o construtor da classe base seja retornado.

É útil considerar inicializadores de variável de instância e inicializadores de construtor como instruções que são inseridas automaticamente antes da *constructor\_body*. O

## exemplo

```
C#  
  
using System;  
using System.Collections;  
  
class A  
{  
    int x = 1, y = -1, count;  
  
    public A() {  
        count = 0;  
    }  
  
    public A(int n) {  
        count = n;  
    }  
}  
  
class B: A  
{  
    double sqrt2 = Math.Sqrt(2.0);  
    ArrayList items = new ArrayList(100);  
    int max;  
  
    public B(): this(100) {  
        items.Add("default");  
    }  
  
    public B(int n): base(n - 1) {  
        max = n;  
    }  
}
```

contém vários inicializadores de variável; Ele também contém inicializadores de construtor de ambos os formulários ( `base` e `this` ). O exemplo corresponde ao código mostrado abaixo, onde cada comentário indica uma instrução inserida automaticamente (a sintaxe usada para as invocações de Construtor inseridas automaticamente não é válida, mas meramente serve para ilustrar o mecanismo).

```
C#  
  
using System.Collections;  
  
class A  
{  
    int x, y, count;  
  
    public A() {  
        x = 1; // Variable initializer
```

```

        y = -1;                                // Variable initializer
        object();                               // Invoke object() constructor
        count = 0;
    }

    public A(int n) {
        x = 1;                                // Variable initializer
        y = -1;                                // Variable initializer
        object();                               // Invoke object() constructor
        count = n;
    }
}

class B: A
{
    double sqrt2;
    ArrayList items;
    int max;

    public B(): this(100) {
        B(100);                             // Invoke B(int) constructor
        items.Add("default");
    }

    public B(int n): base(n - 1) {
        sqrt2 = Math.Sqrt(2.0);             // Variable initializer
        items = new ArrayList(100);          // Variable initializer
        A(n - 1);                           // Invoke A(int) constructor
        max = n;
    }
}

```

## Construtores padrão

Se uma classe não contiver nenhuma declaração de construtor de instância, um construtor de instância padrão será fornecido automaticamente. Esse construtor padrão simplesmente invoca o construtor sem parâmetros da classe base direta. Se a classe for abstrata, a acessibilidade declarada para o construtor padrão será protegida. Caso contrário, a acessibilidade declarada para o construtor padrão é pública. Portanto, o construtor padrão sempre está no formato

C#

```
protected C(): base() {}
```

ou

C#

```
public C(): base() {}
```

em que `C` é o nome da classe. Se a resolução de sobrecarga não puder determinar um melhor candidato exclusivo para o inicializador de construtor da classe base, ocorrerá um erro em tempo de compilação.

No exemplo

```
C#
```

```
class Message
{
    object sender;
    string text;
}
```

um construtor padrão é fornecido porque a classe não contém nenhuma declaração de construtor de instância. Portanto, o exemplo é precisamente equivalente a

```
C#
```

```
class Message
{
    object sender;
    string text;

    public Message(): base() {}
}
```

## Construtores particulares

Quando uma classe `T` declara apenas construtores de instância privada, não é possível para classes fora do texto do programa de `T` para derivar de `T` ou para criar instâncias diretamente do `T`. Portanto, se uma classe contiver apenas membros estáticos e não tiver a intenção de ser instanciada, a adição de um construtor de instância particular vazio impedirá a instanciação. Por exemplo:

```
C#
```

```
public class Trig
{
    private Trig() {}           // Prevent instantiation

    public const double PI = 3.14159265358979323846;
```

```
    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Tan(double x) {...}
}
```

A `Trig` classe agrupa métodos e constantes relacionados, mas não deve ser instanciada. Portanto, ele declara um único Construtor de instância particular vazio. Pelo menos um construtor de instância deve ser declarado para suprimir a geração automática de um construtor padrão.

## Parâmetros opcionais do construtor de instância

A `this(...)` forma de inicializador de construtor é comumente usada em conjunto com sobrecarga para implementar parâmetros de construtor de instância opcionais. No exemplo

```
C#
class Text
{
    public Text(): this(0, 0, null) {}

    public Text(int x, int y): this(x, y, null) {}

    public Text(int x, int y, string s) {
        // Actual constructor implementation
    }
}
```

os primeiros dois construtores de instância fornecem apenas os valores padrão para os argumentos ausentes. Ambos usam um `this(...)` inicializador de construtor para invocar o terceiro construtor de instância, que realmente faz o trabalho de inicializar a nova instância. O efeito é o dos parâmetros de Construtor opcionais:

```
C#
Text t1 = new Text();                      // Same as Text(0, 0, null)
Text t2 = new Text(5, 10);                  // Same as Text(5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

## Construtores estáticos

Um **\*construtor estático** é um membro que implementa as ações necessárias para inicializar um tipo de classe fechado. Construtores estáticos são declarados usando

\_static\_constructor\_declaration \* s:

antlr

```
static_constructor_declaration
    : attributes? static_constructor_modifiers identifier '(' ')'
static_constructor_body
    ;

static_constructor_modifiers
    : 'extern'? 'static'
    | 'static' 'extern'?
    | static_constructor_modifiers_unsafe
    ;

static_constructor_body
    : block
    | ';'
    ;
```

Uma *static\_constructor\_declaration* pode incluir um conjunto de *atributos* (atributos) e um `extern` modificador (métodos externos).

O *identificador* de um *static\_constructor\_declaration* deve nomear a classe na qual o construtor estático é declarado. Se qualquer outro nome for especificado, ocorrerá um erro em tempo de compilação.

Quando uma declaração de construtor estático inclui um `extern` modificador, o construtor estático é considerado um \*\*construtor estático externo\*\*. Como uma declaração de construtor estático externa não fornece nenhuma implementação real, seu `_static_constructor_body` \* consiste em um ponto e vírgula. Para todas as outras declarações de construtor estático, o *static\_constructor\_body* consiste em um *bloco* que especifica as instruções a serem executadas para inicializar a classe. Isso corresponde exatamente à *method\_body* de um método estático com um `void` tipo de retorno (corpo do método).

Construtores estáticos não são herdados e não podem ser chamados diretamente.

O construtor estático para um tipo de classe fechada é executado no máximo uma vez em um determinado domínio de aplicativo. A execução de um construtor estático é disparada pelo primeiro dos seguintes eventos para ocorrer dentro de um domínio de aplicativo:

- Uma instância do tipo de classe é criada.
- Qualquer um dos membros estáticos do tipo de classe é referenciado.

Se uma classe contiver o `Main` método ([inicialização do aplicativo](#)) no qual a execução começa, o construtor estático para aquela classe é executado antes de o `Main` método ser chamado.

Para inicializar um novo tipo de classe fechada, primeiro um novo conjunto de campos estáticos ([campos estáticos e de instância](#)) para esse tipo fechado específico é criado. Cada um dos campos estáticos é inicializado para seu valor padrão ([valores padrão](#)). Em seguida, os inicializadores de campo estáticos ([inicialização de campo estático](#)) são executados para esses campos estáticos. Por fim, o construtor estático é executado.

O exemplo

C#

```
using System;

class Test
{
    static void Main() {
        A.F();
        B.F();
    }
}

class A
{
    static A() {
        Console.WriteLine("Init A");
    }
    public static void F() {
        Console.WriteLine("A.F");
    }
}

class B
{
    static B() {
        Console.WriteLine("Init B");
    }
    public static void F() {
        Console.WriteLine("B.F");
    }
}
```

deve produzir a saída:

Console

```
Init A
A.F
```

```
Init B  
B.F
```

Porque a execução do A construtor estático do é disparada pela chamada para A.F , e a execução do B construtor estático do é disparada pela chamada para B.F .

É possível construir dependências circulares que permitem que campos estáticos com inicializadores de variáveis sejam observados em seu estado de valor padrão.

O exemplo

```
C#
```

```
using System;  
  
class A  
{  
    public static int X;  
  
    static A() {  
        X = B.Y + 1;  
    }  
}  
  
class B  
{  
    public static int Y = A.X + 1;  
  
    static B() {}  
  
    static void Main() {  
        Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);  
    }  
}
```

produz a saída

```
Console
```

```
X = 1, Y = 2
```

Para executar o Main método, o sistema primeiro executa o inicializador para B.Y , antes B do construtor estático da classe. O inicializador faz com que A o construtor estático seja executado porque o valor de A.X é referenciado. O construtor estático de A , por sua vez, prossegue para calcular o valor de X e, ao fazer isso, busca o valor padrão de Y , que é zero. A.X é, portanto, inicializado como 1. O processo de execução

de A inicializadores de campo estático e construtor estático é concluído, retornando ao cálculo do valor inicial de Y , o resultado é 2.

Como o construtor estático é executado exatamente uma vez para cada tipo de classe construída fechada, ele é um local conveniente para impor verificações em tempo de execução no parâmetro de tipo que não pode ser verificado em tempo de compilação via restrições ([restrições de parâmetro de tipo](#)). Por exemplo, o tipo a seguir usa um construtor estático para impor que o argumento de tipo seja uma enumeração:

```
C#
```

```
class Gen<T> where T: struct
{
    static Gen() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T must be an enum");
        }
    }
}
```

## Destruidores

Um **destruidor\*** é um membro que implementa as ações necessárias para destruir uma instância de uma classe. Um destruidor é declarado usando um `_destructor_declaration` \*:

```
antlr
```

```
destructor_declaracion
: attributes? 'extern'? '~' identifier '(' ')' destructor_body
| destructor_declaracion_unsafe
;

destructor_body
: block
| ';'
;
```

Um `destructor_declaration` pode incluir um conjunto de *atributos* ([atributos](#)).

O *identificador* de um `destructor_declaration` deve nomear a classe na qual o destruidor é declarado. Se qualquer outro nome for especificado, ocorrerá um erro em tempo de compilação.

Quando uma declaração de destruidor inclui um `extern` modificador, diz-se que o destruidor é um **\*destruidor externo**. Como uma declaração de destruidor externo não fornece implementação real, seu `_destructor_body` \* consiste em um ponto e vírgula. Para todos os outros destruidores, o `destructor_body` consiste em um bloco que especifica as instruções a serem executadas a fim de destruir uma instância da classe. Uma `destructor_body` corresponde exatamente à `method_body` de um método de instância com um `void` tipo de retorno ([corpo do método](#)).

Os destruidores não são herdados. Portanto, uma classe não tem destruidores além daquele que pode ser declarado nessa classe.

Como um destruidor é necessário para não ter parâmetros, ele não pode ser sobre carregado, portanto, uma classe pode ter, no máximo, um destruidor.

Os destruidores são invocados automaticamente e não podem ser invocados explicitamente. Uma instância fica qualificada para destruição quando não é mais possível que qualquer código use essa instância. A execução do destruidor para a instância pode ocorrer a qualquer momento depois que a instância for qualificada para destruição. Quando uma instância é destruída, os destruidores na cadeia de herança dessa instância são chamados, em ordem, da mais derivada para a menos derivada. Um destruidor pode ser executado em qualquer thread. Para obter mais informações sobre as regras que regem quando e como um destruidor é executado, consulte [gerenciamento automático de memória](#).

A saída do exemplo

```
C#  
  
using System;  
  
class A  
{  
    ~A() {  
        Console.WriteLine("A's destructor");  
    }  
}  
  
class B: A  
{  
    ~B() {  
        Console.WriteLine("B's destructor");  
    }  
}  
  
class Test  
{  
    static void Main()  
    {  
    }
```

```
B b = new B();
b = null;
GC.Collect();
GC.WaitForPendingFinalizers();
}
}
```

is

```
B's destructor
A's destructor
```

como os destruidores em uma cadeia de herança são chamados em ordem, da mais derivada para a menos derivada.

Os destruidores são implementados substituindo o método virtual `Finalize` em `System.Object`. Os programas em C# não têm permissão para substituir esse método ou chamá-lo (ou substituí-lo) diretamente. Por exemplo, o programa

```
C#
class A
{
    override protected void Finalize() {}      // error

    public void F() {
        this.Finalize();                      // error
    }
}
```

contém dois erros.

O compilador se comporta como se esse método e as substituições, não existem. Portanto, este programa:

```
C#
class A
{
    void Finalize() {}                      // permitted
}
```

é válido e o método mostrado oculta o `System.Object Finalize` método.

Para obter uma discussão sobre o comportamento quando uma exceção é gerada de um destruidor, consulte [como as exceções são tratadas](#).

## Iterators

Um membro de função ([membros da função](#)) implementado usando um bloco do iterador ([blocos](#)) é chamado de *iterador*.

Um bloco de iterador pode ser usado como o corpo de um membro da função, desde que o tipo de retorno do membro da função correspondente seja uma das interfaces do enumerador ([interfaces do enumerador](#)) ou uma das interfaces enumeráveis ([interfaces enumeráveis](#)). Isso pode ocorrer como um *method\_body*, *operator\_body* ou *accessor\_body*, enquanto eventos, construtores de instância, construtores estáticos e destruidores não podem ser implementados como iteradores.

Quando um membro de função é implementado usando um bloco iterador, ele é um erro de tempo de compilação para a lista de parâmetros formais do membro da função para especificar qualquer `ref` ou `out` parâmetro ou.

## Interfaces do enumerador

As *interfaces do enumerador* são a interface não genérica

`System.Collections.IEnumerator` e todas as instanciações da interface genérica `System.Collections.Generic.IEnumerator<T>`. Por questão de brevidade, neste capítulo, essas interfaces são referenciadas como `IEnumerator` e `IEnumerator<T>`, respectivamente.

## Interfaces enumeráveis

As *interfaces enumeráveis* são a interface não genérica `System.Collections.IEnumerable` e todas as instanciações da interface genérica `System.Collections.Generic.IEnumerable<T>`. Por questão de brevidade, neste capítulo, essas interfaces são referenciadas como `IEnumerable` e `IEnumerable<T>`, respectivamente.

## Tipo yield

Um iterador produz uma sequência de valores, todos do mesmo tipo. Esse tipo é chamado de *tipo yield* do iterador.

- O tipo yield de um iterador que retorna `IEnumerator` ou `IEnumerable` é `object`.
- O tipo yield de um iterador que retorna `IEnumerator<T>` ou `IEnumerable<T>` é `T`.

## Objetos do enumerador

Quando um membro de função que retorna um tipo de interface de enumerador é implementado usando um bloco de iterador, invocar o membro da função não executa imediatamente o código no bloco do iterador. Em vez disso, um **objeto enumerador** é criado e retornado. Esse objeto encapsula o código especificado no bloco do iterador e a execução do código no bloco do iterador ocorre quando o método do objeto enumerador `MoveNext` é invocado. Um objeto enumerador tem as seguintes características:

- Ele implementa `IEnumerator` e `IEnumerator<T>`, onde `T` é o tipo yield do iterador.
- Ele implementa `System.IDisposable`.
- Ele é inicializado com uma cópia dos valores de argumento (se houver) e o valor de instância passado para o membro da função.
- Ele tem quatro Estados potenciais, *antes de \_ , em execução\_ , suspenso e depois\**, e *initialmente está no estado \_ \*anterior\*\*., suspended, and after, and is initially in the \_ before\* state.*

Um objeto enumerador normalmente é uma instância de uma classe enumeradora gerada pelo compilador que encapsula o código no bloco iterador e implementa as interfaces do enumerador, mas outros métodos de implementação são possíveis. Se uma classe de enumerador for gerada pelo compilador, essa classe será aninhada, direta ou indiretamente, na classe que contém o membro da função, ela terá uma acessibilidade privada e terá um nome reservado para uso do compilador ([identificadores](#)).

Um objeto enumerador pode implementar mais interfaces do que aquelas especificadas acima.

As seções a seguir descrevem o comportamento exato dos `MoveNext` `Current` Membros, e, `Dispose` das `IEnumerable` `IEnumerable<T>` implementações de interface e fornecidas por um objeto enumerador.

Observe que os objetos do enumerador não oferecem suporte ao `IEnumerator.Reset` método. Invocar esse método faz com que um seja `System.NotSupportedException` gerado.

## O método MoveNext

O `MoveNext` método de um objeto de enumerador encapsula o código de um bloco de iterador. Invocar o `MoveNext` método executa o código no bloco do iterador e define a `Current` Propriedade do objeto enumerador conforme apropriado. A ação precisa executada `MoveNext` depende do estado do objeto do enumerador quando o `MoveNext` é invocado:

- Se o estado do objeto enumerador for *antes*, invocando `MoveNext` :
  - Altera o estado para *em execução*.
  - Inicializa os parâmetros (incluindo `this`) do bloco do iterador para os valores de argumento e o valor de instância salvos quando o objeto enumerador foi inicializado.
  - Executa o bloco de iteradores desde o início até que a execução seja interrompida (conforme descrito abaixo).
- Se o estado do objeto enumerador estiver *em execução*, o resultado da invocação `MoveNext` será não especificado.
- Se o estado do objeto do enumerador for *suspensão*, invocando `MoveNext` :
  - Altera o estado para *em execução*.
  - Restaura os valores de todas as variáveis e parâmetros locais (incluindo isso) para os valores salvos quando a execução do bloco do iterador foi suspensa pela última vez. Observe que o conteúdo de quaisquer objetos referenciados por essas variáveis pode ter sido alterado desde a chamada anterior para `MoveNext`.
  - Retoma a execução do bloco iterador imediatamente após a `yield return` instrução que causou a suspensão da execução e continua até que a execução seja interrompida (conforme descrito abaixo).
- Se o estado do objeto enumerador for *After*, invocando `MoveNext` retorna `false` .

Quando `MoveNext` o executa o bloco iterador, a execução pode ser interrompida de quatro maneiras: por uma `yield return` instrução, por uma `yield break` instrução, encontrando o final do bloco iterador e, por uma exceção sendo gerada e propagada do bloco iterador.

- Quando uma `yield return` instrução é encontrada ([a instrução yield](#)):
  - A expressão fornecida na instrução é avaliada, implicitamente convertida para o tipo `yield` e atribuída à `Current` Propriedade do objeto Enumerator.
  - A execução do corpo do iterador é suspensa. Os valores de todas as variáveis e parâmetros locais (incluindo `this`) são salvos, como é o local desta `yield return` instrução. Se a `yield return` instrução estiver dentro de um ou mais `try` blocos, os `finally` blocos associados não serão executados neste momento.
  - O estado do objeto do enumerador é alterado para *suspensão*.

- O `MoveNext` método retorna `true` ao chamador, indicando que a iteração foi avançada com êxito para o próximo valor.
- Quando uma `yield break` instrução é encontrada (a instrução `yield`):
  - Se a `yield break` instrução estiver dentro de um ou mais `try` blocos, os `finally` blocos associados serão executados.
  - O estado do objeto enumerador é alterado para ***After***.
  - O `MoveNext` método retorna `false` ao seu chamador, indicando que a iteração foi concluída.
- Quando o final do corpo do iterador for encontrado:
  - O estado do objeto enumerador é alterado para ***After***.
  - O `MoveNext` método retorna `false` ao seu chamador, indicando que a iteração foi concluída.
- Quando uma exceção é gerada e propagada para fora do bloco do iterador:
  - `finally` Os blocos apropriados no corpo do iterador serão executados pela propagação de exceção.
  - O estado do objeto enumerador é alterado para ***After***.
  - A propagação de exceção continua para o chamador do `MoveNext` método.

## A propriedade atual

A propriedade de um objeto enumerador `Current` é afetada por `yield return` instruções no bloco do iterador.

Quando um objeto enumerador está no estado \*suspenso \*, o valor de `Current` é o valor definido pela chamada anterior para `MoveNext`. Quando um objeto enumerador está nos Estados *anterior*, *running* ou \**After*\*\*, o resultado do acesso `Current` não é especificado.

Para um iterador com um tipo `yield` diferente de `object`, o resultado do acesso `Current` por meio da implementação do objeto enumerador `IEnumerable` corresponde ao acesso `Current` por meio da implementação do objeto enumerador `IEnumerator<T>` e da conversão do resultado em `object`.

## O método Dispose

O `Dispose` método é usado para limpar a iteração, trazendo o objeto enumerador para o estado ***After***.

- Se o estado do objeto do enumerador for \*antes de \*, invocar `Dispose` altera o estado para \_ *após* \*.

- Se o estado do objeto enumerador estiver *em execução*, o resultado da invocação `Dispose` será não especificado.
- Se o estado do objeto do enumerador for *suspensão*, invocando `Dispose` :
  - Altera o estado para *em execução*.
  - Executa qualquer bloco finally como se a última instrução executada `yield return` fosse uma `yield break` instrução. Se isso fizer com que uma exceção seja lançada e propagada do corpo do iterador, o estado do objeto enumerador será definido como *After* e a exceção será propagada para o chamador do `Dispose` método.
  - Altera o estado para *depois*.
- Se o estado do objeto enumerador for *After*, a invocação `Dispose` não terá nenhum efeito.

## Objetos enumeráveis

Quando um membro de função que retorna um tipo de interface enumerável é implementado usando um bloco de iterador, invocar o membro da função não executa imediatamente o código no bloco do iterador. Em vez disso, um *objeto enumerável* é criado e retornado. O método do objeto Enumerable `GetEnumerator` retorna um objeto enumerador que encapsula o código especificado no bloco do iterador e a execução do código no bloco do iterador ocorre quando o método do objeto enumerador `MoveNext` é invocado. Um objeto enumerável tem as seguintes características:

- Ele implementa `IEnumerable` e `IEnumerable<T>`, onde `T` é o tipo yield do iterador.
- Ele é inicializado com uma cópia dos valores de argumento (se houver) e o valor de instância passado para o membro da função.

Um objeto enumerável normalmente é uma instância de uma classe enumerável gerada pelo compilador que encapsula o código no bloco do iterador e implementa as interfaces enumeráveis, mas outros métodos de implementação são possíveis. Se uma classe enumerável for gerada pelo compilador, essa classe será aninhada, direta ou indiretamente, na classe que contém o membro da função, ela terá uma acessibilidade privada e terá um nome reservado para uso do compilador ([identificadores](#)).

Um objeto enumerável pode implementar mais interfaces do que aquelas especificadas acima. Em particular, um objeto enumerável também pode implementar `IEnumerator` e `IEnumerator<T>`, permitindo que ele sirva como um enumerável e um enumerador. Nesse tipo de implementação, na primeira vez que um método de objeto enumerável `GetEnumerator` é invocado, o próprio objeto enumerável é retornado. As invocações subsequentes do objeto Enumerable `GetEnumerator`, se houver, retornam uma cópia do

objeto Enumerable. Assim, cada enumerador retornado tem seu próprio Estado e as alterações em um enumerador não afetarão outra.

## O método GetEnumerator

Um objeto enumerável fornece uma implementação dos `GetEnumerator` métodos das `IEnumerable` interfaces e `IEnumerable<T>`. Os dois `GetEnumerator` métodos compartilham uma implementação comum que adquire e retorna um objeto enumerador disponível. O objeto enumerador é inicializado com os valores de argumento e o valor da instância salvos quando o objeto Enumerable foi inicializado, mas, caso contrário, o objeto enumerador funciona conforme descrito em [objetos do enumerador](#).

## Exemplo de implementação

Esta seção descreve uma possível implementação de iteradores em termos de construções C# padrão. A implementação descrita aqui se baseia nos mesmos princípios usados pelo compilador do Microsoft C#, mas não é, por isso, uma implementação obrigatória ou a única possível.

A classe a seguir `Stack<T>` implementa seu `GetEnumerator` método usando um iterador. O iterador enumera os elementos da pilha na ordem superior para a inferior.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>
{
    T[] items;
    int count;

    public void Push(T item) {
        if (items == null) {
            items = new T[4];
        }
        else if (items.Length == count) {
            T[] newItems = new T[count * 2];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
        items[count++] = item;
    }

    public T Pop() {
```

```

        T result = items[--count];
        items[count] = default(T);
        return result;
    }

    public Ienumerator<T> GetEnumerator() {
        for (int i = count - 1; i >= 0; --i) yield return items[i];
    }
}

```

O `GetEnumerator` método pode ser convertido em uma instanciação de uma classe de enumerador gerada pelo compilador que encapsula o código no bloco do iterador, conforme mostrado a seguir.

C#

```

class Stack<T>: IEnumerable<T>
{
    ...

    public Ienumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1: Ienumerator<T>, IEnumerator
    {
        int __state;
        T __current;
        Stack<T> __this;
        int i;

        public __Enumerator1(Stack<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
                case 1: goto __state1;
                case 2: goto __state2;
            }
            i = __this.count - 1;
            __loop:
            if (i < 0) goto __state2;
            __current = __this.items[i];
        }
    }
}

```

```

        __state = 1;
        return true;
    __state1:
        --i;
        goto __loop;
    __state2:
        __state = 2;
        return false;
    }

    public void Dispose() {
        __state = 2;
    }

    void IEnumerator.Reset() {
        throw new NotSupportedException();
    }
}
}

```

Na tradução anterior, o código no bloco do iterador é transformado em um computador de estado e colocado no `MoveNext` método da classe do enumerador. Além disso, a variável local `i` é transformada em um campo no objeto Enumerator para que ele possa continuar a existir em invocações de `MoveNext`.

O exemplo a seguir imprime uma tabela de multiplicação simples dos inteiros de 1 a 10. O `FromTo` método no exemplo retorna um objeto enumerável e é implementado usando um iterador.

C#

```

using System;
using System.Collections.Generic;

class Test
{
    static IEnumerable<int> FromTo(int from, int to) {
        while (from <= to) yield return from++;
    }

    static void Main() {
        IEnumerable<int> e = FromTo(1, 10);
        foreach (int x in e) {
            foreach (int y in e) {
                Console.Write("{0,3} ", x * y);
            }
            Console.WriteLine();
        }
    }
}

```

O `FromTo` método pode ser convertido em uma instanciação de uma classe enumerável gerada pelo compilador que encapsula o código no bloco do iterador, conforme mostrado a seguir.

C#

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;

class Test
{
    ...

    static IEnumerable<int> FromTo(int from, int to) {
        return new __Enumerable1(from, to);
    }

    class __Enumerable1:
        IEnumerable<int>, IEnumerable,
        IEnumerator<int>, Ienumerator
    {
        int __state;
        int __current;
        int __from;
        int from;
        int to;
        int i;

        public __Enumerable1(int __from, int to) {
            this.__from = __from;
            this.to = to;
        }

        public IEnumerator<int> GetEnumerator() {
            __Enumerable1 result = this;
            if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
                result = new __Enumerable1(__from, to);
                result.__state = 1;
            }
            result.from = result.__from;
            return result;
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return (IEnumerator)GetEnumerator();
        }

        public int Current {
            get { return __current; }
        }
    }
}
```

```

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
                case 1:
                    if (from > to) goto case 2;
                    __current = from++;
                    __state = 1;
                    return true;
                case 2:
                    __state = 2;
                    return false;
                default:
                    throw new InvalidOperationException();
            }
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}

```

A classe Enumerable implementa as interfaces enumeráveis e as interfaces do enumerador, permitindo que ele sirva como um enumerável e um enumerador. Na primeira vez que o `GetEnumerator` método é invocado, o próprio objeto Enumerable é retornado. As invocações subsequentes do objeto Enumerable `GetEnumerator`, se houver, retornam uma cópia do objeto Enumerable. Assim, cada enumerador retornado tem seu próprio Estado e as alterações em um enumerador não afetarão outra. O `Interlocked.CompareExchange` método é usado para garantir a operação thread-safe.

Os `from` parâmetros e são transformados `to` em campos na classe Enumerable. Como `from` é modificado no bloco do iterador, um `__from` campo adicional é introduzido para conter o valor inicial fornecido `from` para em cada enumerador.

O `MoveNext` método lançará um `InvalidOperationException` se for chamado quando `__state` for `0`. Isso protege contra o uso do objeto Enumerable como um objeto enumerador sem primeiro chamar `GetEnumerator`.

O exemplo a seguir mostra uma classe de árvore simples. A `Tree<T>` classe implementa seu `GetEnumerator` método usando um iterador. O iterador enumera os elementos da árvore em ordem de infixo.

C#

```
using System;
using System.Collections.Generic;

class Tree<T>: IEnumerable<T>
{
    T value;
    Tree<T> left;
    Tree<T> right;

    public Tree(T value, Tree<T> left, Tree<T> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    public IEnumerator<T> GetEnumerator() {
        if (left != null) foreach (T x in left) yield x;
        yield value;
        if (right != null) foreach (T x in right) yield x;
    }
}

class Program
{
    static Tree<T> MakeTree<T>(T[] items, int left, int right) {
        if (left > right) return null;
        int i = (left + right) / 2;
        return new Tree<T>(items[i],
            MakeTree(items, left, i - 1),
            MakeTree(items, i + 1, right));
    }

    static Tree<T> MakeTree<T>(params T[] items) {
        return MakeTree(items, 0, items.Length - 1);
    }

    // The output of the program is:
    // 1 2 3 4 5 6 7 8 9
    // Mon Tue Wed Thu Fri Sat Sun

    static void Main() {
        Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);
        foreach (int i in ints) Console.Write("{0} ", i);
        Console.WriteLine();

        Tree<string> strings = MakeTree(
            "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
        foreach (string s in strings) Console.Write("{0} ", s);
        Console.WriteLine();
    }
}
```

O `GetEnumerator` método pode ser convertido em uma instanciação de uma classe de enumerador gerada pelo compilador que encapsula o código no bloco do iterador, conforme mostrado a seguir.

C#

```
class Tree<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1 : IEnumerator<T>, IEnumerator
    {
        Node<T> __this;
        IEnumerator<T> __left, __right;
        int __state;
        T __current;

        public __Enumerator1(Node<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            try {
                switch (__state) {

                    case 0:
                        __state = -1;
                        if (__this.left == null) goto __yield_value;
                        __left = __this.left.GetEnumerator();
                        goto case 1;

                    case 1:
                        __state = -2;
                        if (!__left.MoveNext()) goto __left_Dispose;
                        __current = __left.Current;
                        __state = 1;
                        return true;

                    __left_Dispose:
                        __state = -1;
                        __left.Dispose();
                }
            }
        }
    }
}
```

```
    __yield_value:
        __current = __this.value;
        __state = 2;
        return true;

    case 2:
        __state = -1;
        if (__this.right == null) goto __end;
        __right = __this.right.GetEnumerator();
        goto case 3;

    case 3:
        __state = -3;
        if (!__right.MoveNext()) goto __right_dispose;
        __current = __right.Current;
        __state = 3;
        return true;

    __right_dispose:
        __state = -1;
        __right.Dispose();

    __end:
        __state = 4;
        break;

    }

}

finally {
    if (__state < 0) Dispose();
}
return false;
}

public void Dispose() {
    try {
        switch (__state) {

            case 1:
            case -2:
                __left.Dispose();
                break;

            case 3:
            case -3:
                __right.Dispose();
                break;

        }
    }
    finally {
        __state = 4;
    }
}
```

```
    void IEnumarator.Reset() {
        throw new NotSupportedException();
    }
}
```

O compilador gera temporaries usado nas `foreach` instruções é dividido nos `_left` `_right` campos e do objeto enumerador. O `_state` campo do objeto enumerador é cuidadosamente atualizado para que o `Dispose()` método correto seja chamado corretamente se uma exceção for lançada. Observe que não é possível escrever o código traduzido com instruções simples `foreach`.

## Funções assíncronas

Um método ([métodos](#)) ou uma função anônima ([expressões de função anônimas](#)) com o `async` modificador é chamado de uma **função \* Async**. Em geral, o termo `_Async*` é usado para descrever qualquer tipo de função que tenha o `async` modificador.

É um erro de tempo de compilação para a lista de parâmetros formais de uma função `Async` para especificar qualquer `ref` `out` parâmetro ou.

O `return_type` de um método assíncrono deve ser `void` um tipo de **tarefa** ou. Os tipos de tarefa são `System.Threading.Tasks.Task` e os tipos construídos a partir de `System.Threading.Tasks.Task<T>`. Para fins de brevidade, neste capítulo, esses tipos são referenciados como `Task` e `Task<T>`, respectivamente. Um método assíncrono que retorna um tipo de tarefa é considerado como sendo retornado por tarefa.

A definição exata dos tipos de tarefa é a implementação definida, mas, no ponto de vista do idioma, um tipo de tarefa está em um dos Estados incompletos, com êxito ou com falha. Uma tarefa com falha registra uma exceção pertinente. Um êxito `Task<T>` registra um resultado do tipo `T`. Os tipos de tarefa são `awaitable` e, portanto, podem ser os operandos das expressões `Await` ([Await expressões](#)).

Uma invocação de função assíncrona tem a capacidade de suspender a avaliação por meio de expressões `Await` ([expressões Await](#)) em seu corpo. Posteriormente, a avaliação pode ser retomada no ponto da expressão `Await` de suspensão por meio de um **delegado de continuação** de \*. O delegado de continuação é do tipo `System.Action` e, quando é invocado, a avaliação da invocação de função assíncrona retomará a partir da expressão `Await` onde ela parou. O `_chamador atual*` de uma invocação de função assíncrona é o chamador original se a invocação de função nunca foi suspensa ou o chamador mais recente do delegado de continuação, caso contrário.

## Avaliação de uma função assíncrona de retorno de tarefa

A invocação de uma função assíncrona de retorno de tarefa faz com que uma instância do tipo de tarefa retornada seja gerada. Isso é chamado de **tarefa de retorno** da função Async. A tarefa está inicialmente em um estado incompleto.

O corpo da função assíncrona é então avaliado até que seja suspenso (atingindo uma expressão Await) ou seja encerrado, no qual o controle de ponto é retornado ao chamador, junto com a tarefa de retorno.

Quando o corpo da função Async é encerrado, a tarefa de retorno é movida do estado incompleto:

- Se o corpo da função terminar como o resultado de atingir uma instrução de retorno ou o final do corpo, qualquer valor de resultado será registrado na tarefa de retorno, que será colocado em um estado de êxito.
- Se o corpo da função for encerrado como resultado de uma exceção não percebida ([a instrução Throw](#)), a exceção será registrada na tarefa de retorno que será colocada em um estado com falha.

## Avaliação de uma função assíncrona de retorno nulo

Se o tipo de retorno da função Async for `void`, a avaliação será diferente da seguinte maneira: como nenhuma tarefa é retornada, a função comunica a conclusão e as exceções ao **contexto de sincronização** do thread atual. A definição exata do contexto de sincronização é dependente da implementação, mas é uma representação de "onde" o thread atual está em execução. O contexto de sincronização é notificado quando a avaliação de uma função assíncrona de retorno nulo começa, é concluída com êxito ou faz com que uma exceção não percebida seja gerada.

Isso permite que o contexto Mantenha o controle de quantas funções assíncronas de retorno nulo estão em execução e para decidir como propagar exceções saindo delas.

# Estruturas

Artigo • 16/09/2021

As estruturas são semelhantes às classes, pois representam estruturas de dados que podem conter membros de dados e membros de função. No entanto, ao contrário das classes, as structs são tipos de valor e não exigem alocação de heap. Uma variável de um tipo struct contém diretamente os dados da estrutura, enquanto uma variável de um tipo de classe contém uma referência aos dados, o último conhecido como um objeto.

Os structs são particularmente úteis para estruturas de dados pequenas que têm semântica de valor. Números complexos, pontos em um sistema de coordenadas ou pares chave-valor em um dicionário são exemplos de structs. A chave para essas estruturas de dados é que elas têm poucos membros de dados, que não exigem o uso de herança ou identidade referencial, e que podem ser convenientemente implementadas usando semântica de valor em que a atribuição copia o valor em vez da referência.

Conforme descrito em [tipos simples](#), os tipos simples fornecidos pelo C#, como `int`, `double` e `bool`, são, na verdade, todos os tipos de struct. Assim como esses tipos predefinidos são structs, também é possível usar structs e sobrecarga de operador para implementar novos tipos "primitivos" na linguagem C#. Dois exemplos desses tipos são fornecidos no final deste capítulo ([exemplos de struct](#)).

## Declarações de struct

Uma *struct\_declaration* é uma *type\_declaration* ([declarações de tipo](#)) que declara uma nova struct:

```
antlr

struct_declaration
    : attributes? struct_modifier* 'partial'? 'struct' identifier
      type_parameter_list?
        struct_interfaces? type_parameter_constraints_clause* struct_body ';'?
```

Um *struct\_declaration* consiste em um conjunto opcional de *atributos* ([atributos](#)), seguido por um conjunto opcional de *struct\_modifier*s ([modificadores de struct](#)), seguido por um `partial` modificador opcional, seguido pela palavra-chave `struct` e um *identificador* que nomeia a struct, seguido por uma especificação opcional de *type\_parameter\_list* ([parâmetros de tipo](#)), seguido por uma especificação de

*struct\_interfaces* opcional ([modificador parcial](#)), seguido por uma especificação opcional de *type\_parameter\_constraints\_clause*s (restrições de [parâmetro de tipo](#)), seguido por um *struct\_body* ([corpo de struct](#)), opcionalmente seguido por um ponto e

## Modificadores de struct

Um *struct\_declaration* pode, opcionalmente, incluir uma sequência de modificadores de struct:

```
antlr

struct_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| struct_modifier_unsafe
;
```

É um erro de tempo de compilação para o mesmo modificador aparecer várias vezes em uma declaração de struct.

Os modificadores de uma declaração de struct têm o mesmo significado que os de uma declaração de classe ([declarações de classe](#)).

## Modificador parcial

O `partial` modificador indica que esse *struct\_declaration* é uma declaração de tipo parcial. Várias declarações de struct parciais com o mesmo nome dentro de um namespace delimitador ou declaração de tipo são combinadas para formar uma declaração de struct, seguindo as regras especificadas em [tipos parciais](#).

## Interfaces de struct

Uma declaração de struct pode incluir uma especificação de *struct\_interfaces*; nesse caso, o struct é dito para implementar diretamente os tipos de interface fornecidos.

```
antlr

struct_interfaces
: ':' interface_type_list
;
```

Implementações de interface são discutidas mais detalhadamente em [implementações de interface](#).

## Corpo da struct

A *struct\_body* de uma struct define os membros da estrutura.

```
antlr

struct_body
: '{' struct_member_declaraction* '}'
;
```

## Membros de struct

Os membros de uma struct consistem nos membros introduzidos por seu *struct\_member\_declaration*s e os membros herdados do tipo `System.ValueType`.

```
antlr

struct_member_declaraction
: constant_declaraction
| field_declaraction
| method_declaraction
| property_declaraction
| event_declaraction
| indexer_declaraction
| operator_declaraction
| constructor_declaraction
| static_constructor_declaraction
| type_declaraction
| struct_member_declaraction_unsafe
;
```

Exceto pelas diferenças observadas nas [diferenças de classe e estrutura](#), as descrições dos membros de classe fornecidos em [membros de classe](#) por meio de [funções assíncronas](#) também se aplicam a membros de struct.

## Diferenças de classe e struct

As estruturas diferem das classes de várias maneiras importantes:

- Structs são tipos de valor ([semântica de valor](#)).

- Todos os tipos de struct herdam implicitamente da classe `System.ValueType` ([herança](#)).
- A atribuição a uma variável de um tipo struct cria uma cópia do valor que está sendo atribuído ([atribuição](#)).
- O valor padrão de uma struct é o valor produzido pela configuração de todos os campos de tipo de valor para seu valor padrão e todos os campos de tipo de referência como `null` ([valores padrão](#)).
- As operações boxing e unboxing são usadas para converter entre um tipo struct e `object` ([boxing e unboxing](#)).
- O significado de `this` é diferente para structs ([esse acesso](#)).
- As declarações de campo de instância para um struct não têm permissão para incluir inicializadores de variável ([inicializadores de campo](#)).
- Um struct não tem permissão para declarar um construtor de instância sem parâmetros ([construtores](#)).
- Uma struct não tem permissão para declarar um destruidor ([destruidores](#)).

## Semântica de valor

Structs são tipos de valor ([tipos de valor](#)) e são considerados como semântica de valor. As classes, por outro lado, são tipos de referência ([tipos de referência](#)) e são consideradas semânticas de referência.

Uma variável de um tipo struct contém diretamente os dados da estrutura, enquanto uma variável de um tipo de classe contém uma referência aos dados, o último conhecido como um objeto. Quando uma struct `B` contém um campo de instância do tipo `A` e `A` é um tipo struct, é um erro de tempo de compilação para `A` depender `B` ou um tipo construído de `B`. Uma struct `X *` depende diretamente de `_struct Y` se `X` contiver um campo de instância do tipo `Y`. Dada essa definição, o conjunto completo de structs sobre o qual uma estrutura depende é o fechamento transitivo da relação `_depende diretamente de *`. Por exemplo,

C#

```
struct Node
{
    int data;
    Node next; // error, Node directly depends on itself
}
```

é um erro porque `Node` contém um campo de instância de seu próprio tipo. Outro exemplo

C#

```
struct A { B b; }

struct B { C c; }

struct C { A a; }
```

é um erro porque cada um dos tipos A , B e C depende uns dos outros.

Com classes, é possível que duas variáveis referenciem o mesmo objeto e, assim, possíveis para operações em uma variável afetem o objeto referenciado pela outra variável. Com structs, as variáveis têm sua própria cópia dos dados (exceto no caso de ref variáveis de parâmetro e out ), e não é possível que as operações em um afetem a outra. Além disso, como structs não são tipos de referência, não é possível que os valores de um tipo de struct sejam null .

Dada a declaração

C#

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

o fragmento de código

C#

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

gera o valor 10 . A atribuição de a para b cria uma cópia do valor e, b portanto, não é afetada pela atribuição para a.x . Em vez disso, tinha sido declarado como uma classe, a saída seria 100 porque a e b referenciaria o mesmo objeto.

## Herança

Todos os tipos de struct herdam implicitamente da classe `System.ValueType`, que, por sua vez, herda da classe `object`. Uma declaração struct pode especificar uma lista de interfaces implementadas, mas não é possível que uma declaração struct especifique uma classe base.

Os tipos de struct nunca são abstratos e sempre são lacrados implicitamente. Os `abstract` `sealed` modificadores e, portanto, não são permitidos em uma declaração struct.

Como a herança não tem suporte para structs, a acessibilidade declarada de um Membro struct não pode ser `protected` ou `protected internal`.

Membros de função em um struct não podem ser `abstract` ou `virtual`, e o `override` modificador é permitido apenas para substituir métodos herdados de `System.ValueType`.

## Atribuição

A atribuição a uma variável de um tipo struct cria uma cópia do valor que está sendo atribuído. Isso difere da atribuição a uma variável de um tipo de classe, que copia a referência, mas não o objeto identificado pela referência.

Semelhante a uma atribuição, quando uma struct é passada como um parâmetro de valor ou retornado como resultado de um membro de função, uma cópia da estrutura é criada. Uma struct pode ser passada por referência a um membro de função usando `ref` um `out` parâmetro ou.

Quando uma propriedade ou um indexador de uma struct é o destino de uma atribuição, a expressão de instância associada ao acesso de propriedade ou indexador deve ser classificada como uma variável. Se a expressão de instância for classificada como um valor, ocorrerá um erro em tempo de compilação. Isso é descrito em mais detalhes em [atribuição simples](#).

## Valores padrão

Conforme descrito em [valores padrão](#), vários tipos de variáveis são inicializados automaticamente para seu valor padrão quando eles são criados. Para variáveis de tipos de classe e outros tipos de referência, esse valor padrão é `null`. No entanto, como structs são tipos de valor que não podem ser `null`, o valor padrão de uma struct é o valor produzido pela configuração de todos os campos de tipo de valor para seu valor padrão e todos os campos de tipo de referência como `null`.

Referindo-se à `Point` estrutura declarada acima, o exemplo

```
C#
```

```
Point[] a = new Point[100];
```

Inicializa cada `Point` uma na matriz para o valor produzido Configurando `x` os `y` campos e como zero.

O valor padrão de uma struct corresponde ao valor retornado pelo construtor padrão da struct ([construtores padrão](#)). Ao contrário de uma classe, uma struct não tem permissão para declarar um construtor de instância sem parâmetros. Em vez disso, cada struct implicitamente tem um construtor de instância sem parâmetros que sempre retorna o valor resultante da definição de todos os campos de tipo de valor para seu valor padrão e todos os campos de tipo de referência como `null`.

As estruturas devem ser projetadas para considerar o estado de inicialização padrão um estado válido. No exemplo

```
C#
```

```
using System;

struct KeyValuePair
{
    string key;
    string value;

    public KeyValuePair(string key, string value) {
        if (key == null || value == null) throw new ArgumentException();
        this.key = key;
        this.value = value;
    }
}
```

o construtor de instância definido pelo usuário protege contra valores nulos somente quando ele é chamado explicitamente. Nos casos em que uma `KeyValuePair` variável está sujeita à inicialização de valor padrão, os `key` `value` campos e serão nulos e o struct deverá estar preparado para lidar com esse estado.

## Conversão boxing e unboxing

Um valor de um tipo de classe pode ser convertido em tipo `object` ou em um tipo de interface que é implementado pela classe simplesmente tratando a referência como

outro tipo em tempo de compilação. Da mesma forma, um valor do tipo `object` ou um valor de um tipo de interface pode ser convertido de volta para um tipo de classe sem alterar a referência (mas é claro que uma verificação de tipo em tempo de execução é necessária nesse caso).

Como structs não são tipos de referência, essas operações são implementadas de forma diferente para tipos struct. Quando um valor de um tipo struct é convertido em tipo `object` ou em um tipo de interface que é implementado pela estrutura, ocorre uma operação boxing. Da mesma forma, quando um valor do tipo `object` ou um valor de um tipo de interface é convertido de volta em um tipo struct, ocorre uma operação de não-boxing. Uma diferença importante das mesmas operações em tipos de classe é que a boxing e a unboxing copiam o valor de struct para dentro ou para fora da instância em caixa. Portanto, após uma operação Boxing ou unboxing, as alterações feitas no struct não-Boxed não são refletidas no struct boxed.

Quando um tipo de struct substitui um método virtual herdado de `System.Object` (como `Equals`, `GetHashCode` ou `ToString`), a invocação do método virtual por meio de uma instância do tipo struct não faz com que a Boxing ocorra. Isso é verdadeiro mesmo quando a struct é usada como um parâmetro de tipo e a invocação ocorre por meio de uma instância do tipo de parâmetro de tipo. Por exemplo:

```
C#  
  
using System;  
  
struct Counter  
{  
    int value;  
  
    public override string ToString() {  
        value++;  
        return value.ToString();  
    }  
}  
  
class Program  
{  
    static void Test<T>() where T: new() {  
        T x = new T();  
        Console.WriteLine(x.ToString());  
        Console.WriteLine(x.ToString());  
        Console.WriteLine(x.ToString());  
    }  
  
    static void Main() {  
        Test<Counter>();  
    }  
}
```

A saída do programa é:

Console

```
1  
2  
3
```

Embora seja um estilo inadequado para `ToString` ter efeitos colaterais, o exemplo demonstra que nenhuma Boxing ocorreu para as três invocações de `x.ToString()`.

Da mesma forma, a Boxing nunca ocorre implicitamente ao acessar um membro em um parâmetro de tipo restrito. Por exemplo, suponha que uma interface `ICounter` contenha um método `Increment` que possa ser usado para modificar um valor. Se `ICounter` é usado como uma restrição, a implementação do `Increment` método é chamada com uma referência à variável que `Increment` foi chamada on, nunca uma cópia em caixa.

C#

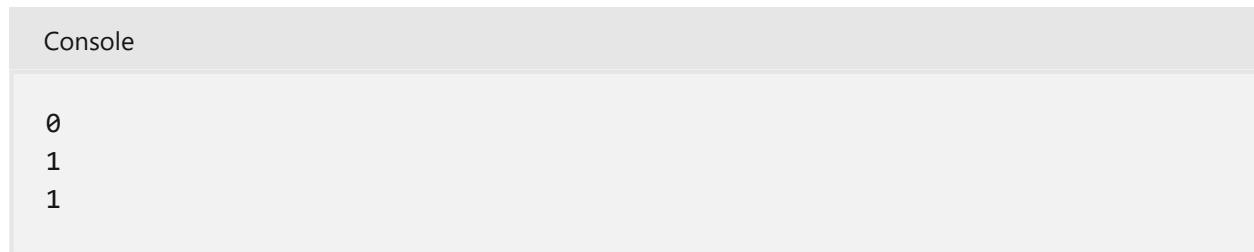
```
using System;  
  
interface ICounter  
{  
    void Increment();  
}  
  
struct Counter: ICounter  
{  
    int value;  
  
    public override string ToString() {  
        return value.ToString();  
    }  
  
    void ICounter.Increment() {  
        value++;  
    }  
}  
  
class Program  
{  
    static void Test<T>() where T: ICounter, new() {  
        T x = new T();  
        Console.WriteLine(x);  
        x.Increment();           // Modify x  
        Console.WriteLine(x);  
        ((ICounter)x).Increment(); // Modify boxed copy of x  
        Console.WriteLine(x);  
    }  
}
```

```
}

static void Main() {
    Test<Counter>();
}

}
```

A primeira chamada para `Increment` modifica o valor na variável `x`. Isso não é equivalente à segunda chamada para `Increment`, que modifica o valor em uma cópia em caixa de `x`. Assim, a saída do programa é:



```
Console

0
1
1
```

Para obter mais detalhes sobre boxing e unboxing, consulte [boxing e unboxing](#).

## Significado disso

Dentro de um construtor de instância ou membro de função de instância de uma classe, `this` é classificado como um valor. Portanto, embora `this` possa ser usado para fazer referência à instância para a qual o membro de função foi invocado, não é possível atribuir a `this` em um membro de função de uma classe.

Dentro de um construtor de instância de uma struct, `this` corresponde a um `out` parâmetro do tipo struct e, dentro de um membro da função de instância de uma struct, `this` corresponde a um `ref` parâmetro do tipo struct. Em ambos os casos, `this` é classificado como uma variável, e é possível modificar toda a estrutura para a qual o membro da função foi invocado, atribuindo a `this` ou passando isso como um `ref` `out` parâmetro ou.

## Inicializadores de campo

Conforme descrito em [valores padrão](#), o valor padrão de uma struct consiste no valor que resulta da definição de todos os campos de tipo de valor com o valor padrão e a todos os campos de tipo de referência como `null`. Por esse motivo, uma struct não permite que declarações de campo de instância incluam inicializadores de variável. Essa restrição se aplica somente a campos de instância. Os campos estáticos de uma estrutura têm permissão para incluir inicializadores de variável.

## O exemplo

C#

```
struct Point
{
    public int x = 1; // Error, initializer not permitted
    public int y = 1; // Error, initializer not permitted
}
```

está em erro porque as declarações de campo de instância incluem inicializadores de variável.

## Construtores

Ao contrário de uma classe, uma struct não tem permissão para declarar um construtor de instância sem parâmetros. Em vez disso, cada struct implicitamente tem um construtor de instância sem parâmetros que sempre retorna o valor resultante da definição de todos os campos de tipo de valor para seu valor padrão e todos os campos de tipo de referência como NULL ([construtores padrão](#)). Uma struct pode declarar construtores de instância com parâmetros. Por exemplo,

C#

```
struct Point
{
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Dada a declaração acima, as instruções

C#

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

Crie um `Point` com `x` e `y` inicializado como zero.

Um construtor de instância de struct não tem permissão para incluir um inicializador de construtor do formulário `base(...)`.

Se o construtor da instância de struct não especificar um inicializador de construtor, a `this` variável corresponderá a um `out` parâmetro do tipo struct e semelhante a um `out` parâmetro, `this` deverá ser definitivamente atribuída ([atribuição definida](#)) em cada local em que o Construtor retornar. Se o construtor da instância de struct especificar um inicializador de construtor, a `this` variável corresponderá a um `ref` parâmetro do tipo struct e, semelhante a um `ref` parâmetro, `this` será considerada definitivamente atribuída na entrada ao corpo do construtor. Considere a implementação do construtor de instância abaixo:

C#

```
struct Point
{
    int x, y;

    public int X {
        set { x = value; }
    }

    public int Y {
        set { y = value; }
    }

    public Point(int x, int y) {
        X = x;           // error, this is not yet definitely assigned
        Y = y;           // error, this is not yet definitely assigned
    }
}
```

Nenhuma função de membro de instância (incluindo os acessadores `set` para as propriedades `X` e `Y`) pode ser chamada até que todos os campos da estrutura que está sendo construída tenham sido definitivamente atribuídos. A única exceção envolve Propriedades implementadas automaticamente ([Propriedades implementadas automaticamente](#)). As regras de atribuição definidas ([expressões de atribuição simples](#)) especificamente isentam a atribuição a uma propriedade automática de um tipo struct dentro de um construtor de instância desse tipo struct: tal atribuição é considerada uma atribuição definitiva do campo de apoio oculto da propriedade automática. Assim, é permitido o seguinte:

C#

```
struct Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y) {
```

```
X = x;      // allowed, definitely assigns backing field
Y = y;      // allowed, definitely assigns backing field
}
```

## Destruidores

Um struct não tem permissão para declarar um destruidor.

## Construtores estáticos

Construtores estáticos para structs seguem a maioria das mesmas regras que para classes. A execução de um construtor estático para um tipo de struct é disparada pelo primeiro dos seguintes eventos para ocorrer dentro de um domínio de aplicativo:

- Um membro estático do tipo struct é referenciado.
- Um construtor declarado explicitamente do tipo struct é chamado.

A criação de valores padrão ([valores padrão](#)) de tipos de struct não dispara o construtor estático. (Um exemplo disso é o valor inicial dos elementos em uma matriz.)

## Exemplos de struct

O exemplo a seguir mostra dois exemplos significativos de `struct` como usar tipos para criar tipos que podem ser usados de forma semelhante aos tipos predefinidos da linguagem, mas com semânticas modificadas.

## Tipo de inteiro do banco de dados

A `DBInt` struct abaixo implementa um tipo inteiro que pode representar o conjunto completo de valores do `int` tipo, além de um estado adicional que indica um valor desconhecido. Um tipo com essas características é comumente usado em bancos de dados.

C#

```
using System;

public struct DBInt
{
    // The Null member represents an unknown DBInt value.

    public static readonly DBInt Null = new DBInt();
```

```
// When the defined field is true, this DBInt represents a known value
// which is stored in the value field. When the defined field is false,
// this DBInt represents an unknown value, and the value field is 0.

int value;
bool defined;

// Private instance constructor. Creates a DBInt with a known value.

DBInt(int value) {
    this.value = value;
    this.defined = true;
}

// The IsNull property is true if this DBInt represents an unknown
value.

public bool IsNull { get { return !defined; } }

// The Value property is the known value of this DBInt, or 0 if this
// DBInt represents an unknown value.

public int Value { get { return value; } }

// Implicit conversion from int to DBInt.

public static implicit operator DBInt(int x) {
    return new DBInt(x);
}

// Explicit conversion from DBInt to int. Throws an exception if the
// given DBInt represents an unknown value.

public static explicit operator int(DBInt x) {
    if (!x.defined) throw new InvalidOperationException();
    return x.value;
}

public static DBInt operator +(DBInt x) {
    return x;
}

public static DBInt operator -(DBInt x) {
    return x.defined ? -x.value : Null;
}

public static DBInt operator +(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value + y.value: Null;
}

public static DBInt operator -(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value - y.value: Null;
}

public static DBInt operator *(DBInt x, DBInt y) {
```

```

        return x.defined && y.defined? x.value * y.value: Null;
    }

    public static DBInt operator /(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value / y.value: Null;
    }

    public static DBInt operator %(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value % y.value: Null;
    }

    public static DBBool operator ==(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value == y.value: DBBool.Null;
    }

    public static DBBool operator !=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value != y.value: DBBool.Null;
    }

    public static DBBool operator >(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value > y.value: DBBool.Null;
    }

    public static DBBool operator <(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value < y.value: DBBool.Null;
    }

    public static DBBool operator >=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value >= y.value: DBBool.Null;
    }

    public static DBBool operator <=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value <= y.value: DBBool.Null;
    }

    public override bool Equals(object obj) {
        if (!(obj is DBInt)) return false;
        DBInt x = (DBInt)obj;
        return value == x.value && defined == x.defined;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        return defined? value.ToString(): "DBInt.Null";
    }
}

```

## Tipo booleano do banco de dados

A `DBBool` estrutura abaixo implementa um tipo lógico de três valores. Os valores possíveis desse tipo são `DBBool.True`, `DBBool.False` e `DBBool.Null`, em que o `Null` membro indica um valor desconhecido. Esses tipos lógicos de três valores são comumente usados em bancos de dados.

C#

```
using System;

public struct DBBool
{
    // The three possible DBBool values.

    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);

    // Private field that stores -1, 0, 1 for False, Null, True.

    sbyte value;

    // Private instance constructor. The value parameter must be -1, 0, or
    1.

    DBBool(int value) {
        this.value = (sbyte)value;
    }

    // Properties to examine the value of a DBBool. Return true if this
    // DBBool has the given value, false otherwise.

    public bool IsNull { get { return value == 0; } }

    public bool IsFalse { get { return value < 0; } }

    public bool IsTrue { get { return value > 0; } }

    // Implicit conversion from bool to DBBool. Maps true to DBBool.True and
    // false to DBBool.False.

    public static implicit operator DBBool(bool x) {
        return x? True: False;
    }

    // Explicit conversion from DBBool to bool. Throws an exception if the
    // given DBBool is Null, otherwise returns true or false.

    public static explicit operator bool(DBBool x) {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }

    // Equality operator. Returns Null if either operand is Null, otherwise
```

```
// returns True or False.

public static DBBool operator ==(DBBool x, DBBool y) {
    if (x.value == 0 || y.value == 0) return Null;
    return x.value == y.value? True: False;
}

// Inequality operator. Returns Null if either operand is Null,
otherwise
// returns True or False.

public static DBBool operator !=(DBBool x, DBBool y) {
    if (x.value == 0 || y.value == 0) return Null;
    return x.value != y.value? True: False;
}

// Logical negation operator. Returns True if the operand is False, Null
// if the operand is Null, or False if the operand is True.

public static DBBool operator !(DBBool x) {
    return new DBBool(-x.value);
}

// Logical AND operator. Returns False if either operand is False,
// otherwise Null if either operand is Null, otherwise True.

public static DBBool operator &(DBBool x, DBBool y) {
    return new DBBool(x.value < y.value? x.value: y.value);
}

// Logical OR operator. Returns True if either operand is True,
otherwise
// Null if either operand is Null, otherwise False.

public static DBBool operator |(DBBool x, DBBool y) {
    return new DBBool(x.value > y.value? x.value: y.value);
}

// Definitely true operator. Returns true if the operand is True, false
// otherwise.

public static bool operator true(DBBool x) {
    return x.value > 0;
}

// Definitely false operator. Returns true if the operand is False,
false
// otherwise.

public static bool operator false(DBBool x) {
    return x.value < 0;
}

public override bool Equals(object obj) {
    if (!(obj is DBBool)) return false;
```

```
        return value == ((DBBool)obj).value;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        if (value > 0) return "DBBool.True";
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}
```

# Matrizes

Artigo • 16/09/2021

Uma matriz é uma estrutura de dados que contém um número de variáveis que são acessadas por meio de índices computados. As variáveis contidas em uma matriz, também chamadas de elementos da matriz, são todas do mesmo tipo, e esse tipo é chamado de tipo de elemento da matriz.

Uma matriz tem uma classificação que determina o número de índices associados a cada elemento da matriz. A classificação de uma matriz também é conhecida como as dimensões da matriz. Uma matriz com uma classificação de uma é chamada de **\*matriz de dimensão única**. Uma matriz com uma classificação maior que uma é chamada de uma *matriz multidimensional*. Matrizes multidimensionais de tamanho específico geralmente são chamadas de matrizes bidimensionais, matrizes tridimensionais e assim por diante.

Cada dimensão de uma matriz tem um comprimento associado que é um número integral maior ou igual a zero. Os comprimentos de dimensão não fazem parte do tipo da matriz, mas, em vez disso, são estabelecidos quando uma instância do tipo de matriz é criada em tempo de execução. O comprimento de uma dimensão determina o intervalo válido de índices para essa dimensão: para uma dimensão de comprimento `N`, os índices podem variar de `0` para `N - 1` inclusivo. O número total de elementos em uma matriz é o produto dos comprimentos de cada dimensão na matriz. Se uma ou mais das dimensões de uma matriz tiver um comprimento igual a zero, a matriz será considerada vazia.

O tipo do elemento de uma matriz pode ser qualquer tipo, incluindo um tipo de matriz.

## Tipos de matriz

Um tipo de matriz é escrito como um *non\_array\_type* seguido por um ou mais *rankSpecifier*s:

```
antlr

array_type
: non_array_type rank_specifier+
;

non_array_type
: type
;
```

```
rank_specifier  
  : '[' dim_separator* ']'  
;  
  
dim_separator  
  : ','  
;
```

Um *non\_array\_type* é qualquer *tipo* que não seja um *array\_type*.

A classificação de um tipo de matriz é fornecida pela *rankSpecifier* mais à esquerda na *array\_type*: uma *rankSpecifier* indica que a matriz é uma matriz com uma classificação de um número de `,` tokens "" no *rankSpecifier*.

O tipo de elemento de um tipo de matriz é o tipo que resulta da exclusão da *rankSpecifier* mais à esquerda:

- Um tipo de matriz do formulário `T[R]` é uma matriz com classificação `R` e um tipo de elemento não matriz `T`.
- Um tipo de matriz do formulário `T[R][R1]...[Rn]` é uma matriz com classificação `R` e um tipo de elemento `T[R1]...[Rn]`.

Na verdade, as *rankSpecifier*s são lidas da esquerda para a direita antes do tipo de elemento não-matriz final. O tipo `int[][][,][,]` é uma matriz unidimensional de matrizes tridimensionais de matrizes bidimensionais do `int`.

Em tempo de execução, um valor de um tipo de matriz pode ser `null` ou uma referência a uma instância desse tipo de matriz.

## O tipo `System.Array`

O tipo `System.Array` é o tipo base abstrato de todos os tipos de matriz. Uma conversão de referência implícita ([conversões de referência implícita](#)) existe de qualquer tipo de matriz para `System.Array`, e uma conversão de referência explícita ([conversões de referência explícita](#)) existe em `System.Array` qualquer tipo de matriz. Observe que `System.Array` não é um *array\_type*. Em vez disso, é uma *class\_type* da qual todos os *array\_type*s são derivados.

Em tempo de execução, um valor do tipo `System.Array` pode ser `null` ou uma referência a uma instância de qualquer tipo de matriz.

## Matrizes e a interface `IList` genérica

Uma matriz unidimensional `T[]` implementa a interface `System.Collections.Generic.IList<T>` (`IList<T>` para abreviar) e suas interfaces base. Da mesma forma, há uma conversão implícita de `T[]` para `IList<T>` e suas interfaces base. Além disso, se houver uma conversão de referência implícita de `S` para `T` `S[]` implementações `IList<T>` e houver uma conversão de referência implícita de `S[]` para `IList<T>` e suas interfaces base ([conversões de referência implícitas](#)). Se houver uma conversão de referência explícita de `S` em `T`, haverá uma conversão de referência explícita de `S[]` para `IList<T>` e suas interfaces base ([conversões de referência explícitas](#)). Por exemplo:

C#

```
using System.Collections.Generic;

class Test
{
    static void Main()
    {
        string[] sa = new string[5];
        object[] oa1 = new object[5];
        object[] oa2 = sa;

        IList<string> lst1 = sa;                                // Ok
        IList<string> lst2 = oa1;                               // Error, cast needed
        IList<object> lst3 = sa;                               // Ok
        IList<object> lst4 = oa1;                               // Ok

        IList<string> lst5 = (IList<string>)oa1;           // Exception
        IList<string> lst6 = (IList<string>)oa2;           // Ok
    }
}
```

A atribuição `lst2 = oa1` gera um erro de tempo de compilação, pois a conversão de `object[]` para `IList<string>` é uma conversão explícita, não implícita. A conversão `(IList<string>)oa1` fará com que uma exceção seja lançada em tempo de execução, pois `oa1` faz referência a `object[]` e não a `string[]`. No entanto `(IList<string>)oa2`, a conversão não fará com que uma exceção seja gerada desde que `oa2` faça referência a `string[]`.

Sempre que houver uma conversão de referência implícita ou explícita de `S[]` para `IList<T>`, também há uma conversão de referência explícita de `IList<T>` e suas interfaces base para `S[]` ([conversões de referência explícitas](#)).

Quando um tipo de matriz `S[]` implementa `IList<T>`, alguns dos membros da interface implementada podem gerar exceções. O comportamento preciso da implementação da interface está além do escopo desta especificação.

# Criação de matriz

As instâncias de matriz são criadas por *array\_creation\_expressions* ([expressões de criação de matriz](#)) ou por declarações de campo ou de variável local que incluem um *array\_initializer* ([inicializadores de matriz](#)).

Quando uma instância de matriz é criada, a classificação e o comprimento de cada dimensão são estabelecidos e permanecem constantes para o tempo de vida inteiro da instância. Em outras palavras, não é possível alterar a classificação de uma instância de matriz existente, nem é possível redimensionar suas dimensões.

Uma instância de matriz é sempre de um tipo de matriz. O `System.Array` tipo é um tipo abstrato que não pode ser instanciado.

Os elementos das matrizes criados por *array\_creation\_expressions* são sempre inicializados para seu valor padrão ([valores padrão](#)).

## Acesso de elemento de matriz

Os elementos de matriz são acessados usando expressões de *element\_access* ([acesso de matriz](#)) do formulário `A[I1, I2, ..., In]`, em que `A` é uma expressão de um tipo de matriz e cada `Ix` é uma expressão de tipo `int` „ `uint` `long` `ulong` ou pode ser convertida implicitamente em um ou mais desses tipos. O resultado de um acesso de elemento de matriz é uma variável, ou seja, o elemento de matriz selecionado pelos índices.

Os elementos de uma matriz podem ser enumerados usando uma `foreach` instrução ([a instrução foreach](#)).

## Membros de matriz

Cada tipo de matriz herda os membros declarados pelo `System.Array` tipo.

## Covariância de matriz

Para quaisquer duas *reference\_types* `A` e `B`, se uma conversão de referência implícita ([conversões de referência implícita](#)) ou conversão de referência explícita ([conversões de referência explícita](#)) existir de `A` para `B`, a mesma conversão de referência também existirá no tipo de matriz `A[R]` para o tipo de matriz `B[R]`, em que `R` é qualquer *rank\_specifier* especificada (mas a mesma para ambos os tipos de matriz). Essa relação é

conhecida como **covariância de matriz**. A covariância de matriz em particular significa que um valor de um tipo de matriz `A[R]` pode, na verdade, ser uma referência a uma instância de um tipo de matriz `B[R]`, desde que exista uma conversão de referência implícita de `B` para `A`.

Devido à covariância da matriz, as atribuições a elementos de matrizes de tipo de referência incluem uma verificação de tempo de execução que garante que o valor que está sendo atribuído ao elemento da matriz seja, na verdade, de um tipo permitido ([atribuição simples](#)). Por exemplo:

C#

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
        for (int i = index; i < index + count; i++) array[i] = value;
    }

    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}
```

A atribuição para `array[i]` no `Fill` método inclui implicitamente uma verificação de tempo de execução que garante que o objeto referenciado por `value` seja `null` ou uma instância compatível com o tipo de elemento real de `array`. No `Main`, as duas primeiras invocações são `Fill` com sucesso, mas a terceira invocação faz com que uma `System.ArrayTypeMismatchException` seja gerada na execução da primeira atribuição para `array[i]`. A exceção ocorre porque um Boxed `int` não pode ser armazenado em uma `string` matriz.

A covariância de matrizes não se estende especificamente a matrizes de `value_type`s. Por exemplo, não existe nenhuma conversão que permita que um `int[]` seja tratado como um `object[]`.

## Inicializadores de matriz

Inicializadores de matriz podem ser especificados em declarações de campo ([campos](#)), declarações de variáveis locais ([declarações de variáveis locais](#)) e expressões de criação de matriz ([expressões de criação de matriz](#)):

```
antlr

array_initializer
: '{' variable_initializer_list? '}'
| '{' variable_initializer_list ',' '}'
;

variable_initializer_list
: variable_initializer (',' variable_initializer)*
;

variable_initializer
: expression
| array_initializer
;
```

Um inicializador de matriz consiste em uma sequência de inicializadores de variável, entre os `{` tokens "" e `}` e separados por `,`. Cada inicializador de variável é uma expressão ou, no caso de uma matriz multidimensional, um inicializador de matriz aninhada.

O contexto no qual um inicializador de matriz é usado determina o tipo da matriz que está sendo inicializada. Em uma expressão de criação de matriz, o tipo de matriz precede imediatamente o inicializador ou é inferido a partir das expressões no inicializador de matriz. Em uma declaração de campo ou variável, o tipo de matriz é o tipo do campo ou da variável que está sendo declarada. Quando um inicializador de matriz é usado em uma declaração de campo ou variável, como:

C#

```
int[] a = {0, 2, 4, 6, 8};
```

é simplesmente uma abreviação para uma expressão de criação de matriz equivalente:

C#

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

Para uma matriz unidimensional, o inicializador de matriz deve consistir em uma sequência de expressões que são compatíveis com o tipo de elemento da matriz. As expressões inicializam os elementos da matriz em ordem crescente, começando com o elemento no índice zero. O número de expressões no inicializador de matriz determina o comprimento da instância de matriz que está sendo criada. Por exemplo, o inicializador de matriz acima cria uma `int[]` instância de comprimento 5 e inicializa a instância com os seguintes valores:

C#

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

Para uma matriz multidimensional, o inicializador de matriz deve ter tantos níveis de aninhamento quanto há dimensões na matriz. O nível de aninhamento mais externo corresponde à dimensão mais à esquerda e o nível de aninhamento mais interno corresponde à dimensão mais à direita. O comprimento de cada dimensão da matriz é determinado pelo número de elementos no nível de aninhamento correspondente no inicializador de matriz. Para cada inicializador de matriz aninhada, o número de elementos deve ser o mesmo que os outros inicializadores de matriz no mesmo nível. O exemplo:

C#

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

Cria uma matriz bidimensional com um comprimento de cinco para a dimensão mais à esquerda e um comprimento de dois para a dimensão mais à direita:

C#

```
int[,] b = new int[5, 2];
```

e inicializa a instância de matriz com os seguintes valores:

C#

```
b[0, 0] = 0; b[0, 1] = 1;  
b[1, 0] = 2; b[1, 1] = 3;  
b[2, 0] = 4; b[2, 1] = 5;  
b[3, 0] = 6; b[3, 1] = 7;  
b[4, 0] = 8; b[4, 1] = 9;
```

Se uma dimensão que não seja a mais à direita for fornecida com o comprimento zero, as dimensões subsequentes serão consideradas também com comprimento zero. O exemplo:

C#

```
int[,] c = {};
```

Cria uma matriz bidimensional com um comprimento zero para a dimensão mais à esquerda e à extrema direita:

```
C#
```

```
int[,] c = new int[0, 0];
```

Quando uma expressão de criação de matriz inclui comprimentos de dimensão explícitos e um inicializador de matriz, os comprimentos devem ser expressões constantes e o número de elementos em cada nível de aninhamento deve corresponder ao comprimento de dimensão correspondente. Estes são alguns exemplos:

```
C#
```

```
int i = 3;
int[] x = new int[3] {0, 1, 2};           // OK
int[] y = new int[i] {0, 1, 2};           // Error, i not a constant
int[] z = new int[3] {0, 1, 2, 3};         // Error, length/initializer mismatch
```

Aqui, o inicializador `y` resulta em um erro de tempo de compilação porque a expressão de comprimento da dimensão não é uma constante e o inicializador para `z` resulta em um erro de tempo de compilação porque o comprimento e o número de elementos no inicializador não são aceitos.

# Interfaces

Artigo • 16/09/2021

Uma interface define um contrato. Uma classe ou struct que implementa uma interface deve aderir a seu contrato. Uma interface pode herdar de várias interfaces base e uma classe ou estrutura pode implementar várias interfaces.

As interfaces podem conter métodos, propriedades, eventos e indexadores. A própria interface não fornece implementações para os membros que ele define. A interface simplesmente especifica os membros que devem ser fornecidos por classes ou estruturas que implementam a interface.

## Declarações de interface

Uma *interface\_declaration* é uma *type\_declaration* ([declarações de tipo](#)) que declara um novo tipo de interface.

antlr

```
interface_declaration
: attributes? interface_modifier* 'partial'? 'interface'
  identifier variant_type_parameter_list? interface_base?
  type_parameter_constraints_clause* interface_body ';'?
```

Um *interface\_declaration* consiste em um conjunto opcional de *atributos* ([atributos](#)), seguido por um conjunto opcional de *interface\_modifier*s ([modificadores de interface](#)), seguido por um `partial` modificador opcional, seguido pela palavra-chave `interface` e um *identificador* que nomeia a interface, seguido por uma especificação opcional de *variant\_type\_parameter\_list* (listas de [parâmetros de tipo Variant](#)), seguido por uma especificação opcional de *Interface\_base* ([interfaces base](#)), seguida por uma especificação opcional de *type\_parameter\_constraints\_clause*s (restrições de [parâmetro de tipo](#)), seguido por um *interface\_body* ([corpo da interface](#)), opcionalmente seguido por um ponto

## Modificadores de interface

Um *interface\_declaration* pode, opcionalmente, incluir uma sequência de modificadores de interface:

antlr

```
interface_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| interface_modifier_unsafe
;
```

É um erro de tempo de compilação para o mesmo modificador aparecer várias vezes em uma declaração de interface.

O `new` modificador é permitido somente em interfaces definidas dentro de uma classe. Ele especifica que a interface oculta um membro herdado com o mesmo nome, conforme descrito no [novo modificador](#).

Os `public` `protected` `internal` modificadores,, e `private` controlam a acessibilidade da interface. Dependendo do contexto no qual a declaração de interface ocorre, somente alguns desses modificadores podem ser permitidos ([acessibilidade declarada](#)).

## Modificador parcial

O `partial` modificador indica que esse *interface\_declaration* é uma declaração de tipo parcial. Várias declarações de interface parciais com o mesmo nome dentro de um namespace delimitador ou declaração de tipo são combinadas para formar uma declaração de interface, seguindo as regras especificadas em [tipos parciais](#).

## Listas de parâmetros de tipo Variant

Listas de parâmetros de tipo Variant só podem ocorrer em tipos de interface e delegados. A diferença entre as *type\_parameter\_list* comuns é a *variance\_annotation* opcional em cada parâmetro de tipo.

```
antlr
```

```
variant_type_parameter_list
: '<' variant_type_parameters '>'
;

variant_type_parameters
: attributes? variance_annotation? type_parameter
| variant_type_parameters ',' attributes? variance_annotation?
type_parameter
;

variance_annotation
```

```
: 'in'  
| 'out'  
;
```

Se a anotação de variação for `out`, o parâmetro de tipo é considerado *\*covariable*\_. Se a anotação de variação for `in`, o parâmetro de tipo será considerado *contravariant*. Se não houver nenhuma anotação de variação, o parâmetro de tipo será considerado \_*invariável*\_.

No exemplo

C#

```
interface C<out X, in Y, Z>  
{  
    X M(Y y);  
    Z P { get; set; }  
}
```

`X` é covariant, `Y` é contravariant e `Z` é invariável.

## Segurança de variação

A ocorrência de anotações de variação na lista de parâmetros de tipo de um tipo restringe os locais em que os tipos podem ocorrer dentro da declaração de tipo.

Um tipo `T` é de *saída-não seguro* se uma das seguintes isenções:

- `T` é um parâmetro de tipo contravariant
- `T` é um tipo de matriz com um tipo de elemento de saída não seguro
- `T` é uma interface ou tipo delegado `S<A1, ..., Ak>` construído a partir de um tipo genérico `S<X1, ..., Xk>`, em que pelo menos uma `Ai` das seguintes isenções:
  - `Xi` é covariant ou invariável e `Ai` é de saída-não segura.
  - `Xi` é contravariant ou invariável e `Ai` é de entrada segura.

Um tipo `T` é *de entrada-não seguro* se uma das seguintes isenções:

- `T` é um parâmetro de tipo covariant
- `T` é um tipo de matriz com um tipo de elemento de entrada não seguro
- `T` é uma interface ou tipo delegado `S<A1, ..., Ak>` construído a partir de um tipo genérico `S<X1, ..., Xk>`, em que pelo menos uma `Ai` das seguintes isenções:
  - `Xi` é covariant ou invariável e `Ai` é de entrada-não segura.
  - `Xi` é contravariant ou invariável e `Ai` é de saída-não segura.

Intuitivamente, um tipo de saída não segura é proibido em uma posição de saída e um tipo de entrada não segura é proibido em uma posição de entrada.

Um tipo é *saída-seguro* se não for de saída-não seguro e é *de entrada-seguro* se não for de entrada-não segura.

## Conversão de variância

A finalidade das anotações de variação é fornecer conversões de brandas (mas ainda de tipo seguro) aos tipos de interface e delegado. Para esse fim, as definições de conversões implícitas ([conversões implícitas](#)) e explícitas ([conversões explícitas](#)) fazem uso da noção de variância-convertibilidade, que é definida da seguinte maneira:

Um tipo  $T<A_1, \dots, A_n>$  é conversível de variação em um tipo  $T<B_1, \dots, B_n>$  se  $T$  for uma interface ou um tipo delegado declarado com os parâmetros de tipo Variant  $T<X_1, \dots, X_n>$  e para cada parâmetro de tipo Variant,  $x_i$  uma das seguintes isenções:

- $x_i$  é covariant e uma referência implícita ou conversão de identidade existe de  $A_i$  para  $B_i$
- $x_i$  é contravariant e uma referência implícita ou conversão de identidade existe em  $B_i$  para  $A_i$
- $x_i$  é invariável e existe uma conversão de identidade de  $A_i$  para  $B_i$

## Interfaces base

Uma interface pode herdar de zero ou mais tipos de interface, que são chamados de *interfaces base explícitas* da interface. Quando uma interface tem uma ou mais interfaces base explícitas, na declaração dessa interface, o identificador de interface é seguido por dois-pontos e uma lista separada por vírgulas de tipos de interface base.

```
antlr

interface_base
    : ':' interface_type_list
    ;
```

Para um tipo de interface construída, as interfaces base explícitas são formadas por meio das declarações de interface base explícitas na declaração de tipo genérico e a substituição, para cada *type\_parameter* na declaração de interface base, a *type\_argument* correspondente do tipo construído.

As interfaces base explícitas de uma interface devem ser pelo menos tão acessíveis quanto a própria interface ([restrições de acessibilidade](#)). Por exemplo, é um erro de tempo de compilação para especificar uma `private` `internal` interface ou no `interface_base` de uma `public` interface.

É um erro de tempo de compilação para uma interface herdar direta ou indiretamente de si mesma.

As **interfaces base** de uma interface são as interfaces base explícitas e suas interfaces base. Em outras palavras, o conjunto de interfaces base é o fechamento transitivo completo das interfaces base explícitas, suas interfaces base explícitas e assim por diante. Uma interface herda todos os membros de suas interfaces base. No exemplo

C#

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

as interfaces base do `IComboBox` são `IControl` , `ITextBox` e `IListBox` .

Em outras palavras, a `IComboBox` interface acima herda membros `SetText` e `SetItems` também `Paint` .

Cada interface base de uma interface deve ser de saída segura ([segurança de variação](#)). Uma classe ou struct que implementa uma interface também implementa implicitamente todas as interfaces base da interface.

## Corpo da interface

A *interface\_body* de uma interface define os membros da interface.

antlr

```
interface_body
: '{' interface_member_declaraction* '}'
;
```

## Membros da interface

Os membros de uma interface são os membros herdados das interfaces base e os membros declarados pela própria interface.

antlr

```
interface_member_declaraction
: interface_method_declaraction
| interface_property_declaraction
| interface_event_declaraction
| interface_indexer_declaraction
;
```

Uma declaração de interface pode declarar zero ou mais membros. Os membros de uma interface devem ser métodos, propriedades, eventos ou indexadores. Uma interface não pode conter constantes, campos, operadores, construtores de instância, destruidores ou tipos, nem uma interface que contenha membros estáticos de qualquer tipo.

Todos os membros de interface implicitamente têm acesso público. É um erro de tempo de compilação para declarações de membro de interface para incluir qualquer modificador. Em particular, os membros de interfaces não podem ser declarados com os modificadores `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override` ou `static`.

O exemplo

C#

```
public delegate void StringListEvent(IStringList sender);

public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

declara uma interface que contém um dos tipos de membros possíveis: um método, uma propriedade, um evento e um indexador.

Um *interface\_declaration* cria um novo espaço de declaração ([declarações](#)) e os *interface\_member\_declaration*s imediatamente contidos pelo *interface\_declaration* introduzem novos membros nesse espaço de declaração. As regras a seguir se aplicam a *interface\_member\_declaration*s:

- O nome de um método deve ser diferente dos nomes de todas as propriedades e eventos declarados na mesma interface. Além disso, a assinatura ([assinaturas e sobrecarga](#)) de um método deve ser diferente das assinaturas de todos os outros métodos declarados na mesma interface, e dois métodos declarados na mesma interface podem não ter assinaturas que diferem exclusivamente pelo `ref` e `out`.
- O nome de uma propriedade ou evento deve ser diferente dos nomes de todos os outros membros declarados na mesma interface.
- A assinatura de um indexador deve ser diferente das assinaturas de todos os outros indexadores declarados na mesma interface.

Os membros herdados de uma interface não fazem parte especificamente do espaço de declaração da interface. Assim, uma interface tem permissão para declarar um membro com o mesmo nome ou assinatura que um membro herdado. Quando isso ocorre, o membro da interface derivada é dito para ocultar o membro da interface base. Ocultar um membro herdado não é considerado um erro, mas faz com que o compilador emita um aviso. Para suprimir o aviso, a declaração do membro de interface derivada deve incluir um `new` modificador para indicar que o membro derivado destina-se a ocultar o membro base. Este tópico é abordado mais detalhadamente em [ocultar por meio de herança](#).

Se um `new` modificador for incluído em uma declaração que não oculta um membro herdado, um aviso será emitido para esse efeito. Esse aviso é suprimido com a remoção do `new` modificador.

Observe que os membros da classe `object` não são, estritamente falando, membros de qualquer interface ([membros da interface](#)). No entanto, os membros da classe `object` estão disponíveis por meio de pesquisa de membros em qualquer tipo de interface ([pesquisa de membros](#)).

## Métodos de interface

Métodos de interface são declarados usando *interface\_method\_declaration*s:

```
interface_method_declaration
: attributes? 'new'? return_type identifier type_parameter_list
  '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
;
```

Os *atributos*, *return\_type*, *identificador* e *formal\_parameter\_list* de uma declaração de método de interface têm o mesmo significado que os de uma declaração de método em uma classe ([métodos](#)). Uma declaração de método de interface não tem permissão para especificar um corpo de método, e a declaração, portanto, sempre termina com um ponto e vírgula.

Cada tipo de parâmetro formal de um método de interface deve ser de entrada segura ([segurança de variação](#)) e o tipo de retorno deve ser `void` ou de saída seguro. Além disso, cada restrição de tipo de classe, restrição de tipo de interface e restrição de parâmetro de tipo em qualquer parâmetro de tipo do método deve ser de entrada segura.

Essas regras garantem que qualquer uso covariante ou contravariant da interface permaneça de tipo seguro. Por exemplo,

C#

```
interface I<out T> { void M<U>() where U : T; }
```

é ilegal porque o uso de `T` como uma restrição de parâmetro de tipo em `U` não é de entrada segura.

Essa restrição não estava em vigor, seria possível violar a segurança do tipo da seguinte maneira:

C#

```
class B {}
class D : B{}
class E : B {}
class C : I<D> { public void M<U>() {...} }
...
I<B> b = new C();
b.M<E>();
```

Essa é, na verdade, uma chamada para `C.M<E>`. Mas essa chamada exige que `E` derive de `D`, portanto, a segurança do tipo seria violada aqui.

## Propriedades de interface

As propriedades da interface são declaradas usando *interface\_property\_declaration* s:

```
antlr

interface_property_declaration
: attributes? 'new'? type identifier '{' interface_accessors '}'
;

interface_accessors
: attributes? 'get' ';'
| attributes? 'set' ';'
| attributes? 'get' ';' attributes? 'set' ';'
| attributes? 'set' ';' attributes? 'get' ';'
;
```

Os *atributos*, o *tipo* e o *identificador* de uma declaração de propriedade de interface têm o mesmo significado que os de uma declaração de propriedade em uma classe ([Propriedades](#)).

Os acessadores de uma declaração de propriedade de interface correspondem aos acessadores de uma declaração de propriedade de classe ([acessadores](#)), exceto que o corpo do acessador sempre deve ser um ponto e vírgula. Portanto, os acessadores simplesmente indicam se a propriedade é de leitura/gravação, somente leitura ou somente gravação.

O tipo de uma propriedade de interface deve ser de saída segura se houver um acessador get e deve ser de entrada segura se houver um acessador set.

## Eventos de interface

Eventos de interface são declarados usando *interface\_event\_declaration* s:

```
antlr

interface_event_declaration
: attributes? 'new'? 'event' type identifier ';'
;
```

Os *atributos*, o *tipo* e o *identificador* de uma declaração de evento de interface têm o mesmo significado que os de uma declaração de evento em uma classe ([eventos](#)).

O tipo de um evento de interface deve ser de entrada segura.

## Indexadores de interface

Indexadores de interface são declarados usando *interface\_indexer\_declaration* s:

```
antlr
```

```
interface_indexer_declaration
    : attributes? 'new'? type 'this' '[' formal_parameter_list ']' '{'
    interface_accessors '}'
    ;
```

Os *atributos*, o *tipo* e a *formal\_parameter\_list* de uma declaração de indexador de interface têm o mesmo significado que os de uma declaração de indexador em uma classe ([indexadores](#)).

Os acessadores de uma declaração de indexador de interface correspondem aos acessadores de uma declaração de indexador de classe ([indexadores](#)), exceto que o corpo do acessador sempre deve ser um ponto e vírgula. Assim, os acessadores simplesmente indicam se o indexador é de leitura/gravação, somente leitura ou somente gravação.

Todos os tipos de parâmetro formais de um indexador de interface devem ser de entrada segura. Além disso, qualquer `out` `ref` tipo de parâmetro ou formal também deve ser de saída segura. Observe que até mesmo `out` os parâmetros devem ser de entrada segura, devido a uma limitação da plataforma de execução subjacente.

O tipo de um indexador de interface deve ser seguro para saída se houver um acessador `get` e deve ser de entrada segura se houver um acessador `set`.

## Acesso de membro de interface

Os membros da interface são acessados por meio de acesso de membro ([acesso de membro](#)) e expressões de acesso do indexador (acesso ao[indexador](#)) do formulário `I.M` e `I[A]`, em que `I` é um tipo de interface, `M` é um método, propriedade ou evento desse tipo de interface e `A` é uma lista de argumentos do indexador.

Para interfaces que são estritamente herança única (cada interface na cadeia de herança tem exatamente zero ou uma interface de base direta), os efeitos da pesquisa de Membros ([pesquisa de membros](#)), da invocação de método ([invocações de método](#)) e das regras de acesso do indexador ([acesso ao indexador](#)) são exatamente iguais aos de classes e structs: Membros mais derivados ocultam Membros derivados menores com o mesmo nome ou assinatura. No entanto, para interfaces de várias heranças, as ambiguidades podem ocorrer quando duas ou mais interfaces base não relacionadas declaram Membros com o mesmo nome ou assinatura. Esta seção mostra vários

exemplos dessas situações. Em todos os casos, as conversões explícitas podem ser usadas para resolver as ambiguidades.

No exemplo

```
C#  
  
interface IList  
{  
    int Count { get; set; }  
}  
  
interface ICounter  
{  
    void Count(int i);  
}  
  
interface IListCounter: IList, ICounter {}  
  
class C  
{  
    void Test(IListCounter x) {  
        x.Count(1);                // Error  
        x.Count = 1;                // Error  
        ((IList)x).Count = 1;       // Ok, invokes IList.Count.set  
        ((ICounter)x).Count(1);    // Ok, invokes ICounter.Count  
    }  
}
```

as duas primeiras instruções causam erros de tempo de compilação porque a pesquisa de membro ([pesquisa de membro](#)) de `Count` no `IListCounter` é ambígua. Conforme ilustrado pelo exemplo, a ambiguidade é resolvida pela conversão `x` para o tipo de interface base apropriado. Tais conversões não têm custos de tempo de execução — elas simplesmente consistem em exibir a instância como um tipo menos derivado em tempo de compilação.

No exemplo

```
C#  
  
interface IInteger  
{  
    void Add(int i);  
}  
  
interface IDouble  
{  
    void Add(double d);  
}
```

```

interface INumber: IInteger, IDouble {}

class C
{
    void Test(INumber n) {
        n.Add(1);                      // Invokes IInteger.Add
        n.Add(1.0);                    // Only IDouble.Add is applicable
        ((IInteger)n).Add(1);          // Only IInteger.Add is a candidate
        ((IDouble)n).Add(1);          // Only IDouble.Add is a candidate
    }
}

```

a invocação `n.Add(1)` seleciona `IInteger.Add` aplicando as regras de resolução de sobrecarga da [resolução de sobrelocação](#). Da mesma forma, a invocação é `n.Add(1.0)` selecionada `IDouble.Add`. Quando as conversões explícitas são inseridas, há apenas um método candidato e, portanto, nenhuma ambiguidade.

No exemplo

```

C#

interface IBase
{
    void F(int i);
}

interface ILeft: IBase
{
    new void F(int i);
}

interface IRight: IBase
{
    void G();
}

interface IDerived: ILeft, IRight {}

class A
{
    void Test(IDerived d) {
        d.F(1);                      // Invokes ILeft.F
        ((IBase)d).F(1);            // Invokes IBase.F
        ((ILeft)d).F(1);           // Invokes ILeft.F
        ((IRight)d).F(1);          // Invokes IBase.F
    }
}

```

O `IBase.F` membro é oculto pelo `ILeft.F` membro. A invocação `d.F(1)`, portanto `ILeft.F`, é selecionada, embora `IBase.F` pareça não estar oculta no caminho de acesso

que leva `IRight` .

A regra intuitiva para ocultar em interfaces de várias heranças é simplesmente: se um membro estiver oculto em qualquer caminho de acesso, ele ficará oculto em todos os caminhos de acesso. Como o caminho de acesso de `IDerived` para `ILeft` para `IBase` oculta `IBase.F`, o membro também fica oculto no caminho de acesso de `IDerived` para `IRight` `IBase` .

## Nomes de membros de interface totalmente qualificados

Um membro de interface, às vezes, é referido por seu *nome totalmente qualificado*. O nome totalmente qualificado de um membro de interface consiste no nome da interface na qual o membro é declarado, seguido por um ponto, seguido pelo nome do membro. O nome totalmente qualificado de um membro faz referência à interface na qual o membro é declarado. Por exemplo, considerando as declarações

```
C#  
  
interface IControl  
{  
    void Paint();  
}  
  
interface ITextBox: IControl  
{  
    void SetText(string text);  
}
```

o nome totalmente qualificado de `Paint` é `IControl.Paint` e o nome totalmente qualificado de `SetText` é `ITextBox.SetText` .

No exemplo acima, não é possível fazer referência a `Paint` as `ITextBox.Paint` .

Quando uma interface faz parte de um namespace, o nome totalmente qualificado de um membro de interface inclui o nome do namespace. Por exemplo,

```
C#  
  
namespace System  
{  
    public interface ICloneable  
    {  
        object Clone();  
    }  
}
```

```
    }  
}
```

Aqui, o nome totalmente qualificado do `Clone` método é `System.ICloneable.Clone`.

## Implementações de interfaces

As interfaces podem ser implementadas por classes e estruturas. Para indicar que uma classe ou struct implementa diretamente uma interface, o identificador de interface é incluído na lista de classes base da classe ou estrutura. Por exemplo:

```
C#  
  
interface ICloneable  
{  
    object Clone();  
}  
  
interface IComparable  
{  
    int CompareTo(object other);  
}  
  
class ListEntry: ICloneable, IComparable  
{  
    public object Clone() {...}  
    public int CompareTo(object other) {...}  
}
```

Uma classe ou estrutura que implementa diretamente uma interface também implementa diretamente todas as interfaces base da interface implicitamente. Isso é verdadeiro mesmo que a classe ou struct não liste explicitamente todas as interfaces base na lista de classes base. Por exemplo:

```
C#  
  
interface IControl  
{  
    void Paint();  
}  
  
interface ITextBox: IControl  
{  
    void SetText(string text);  
}  
  
class TextBox: ITextBox  
{
```

```
    public void Paint() {...}
    public void SetText(string text) {...}
}
```

Aqui, a classe `TextBox` implementa `IControl` e `ITextBox`.

Quando uma classe `C` implementa diretamente uma interface, todas as classes derivadas de `C` também implementam a interface implicitamente. As interfaces base especificadas em uma declaração de classe podem ser tipos de interface construídos ([tipos construídos](#)). Uma interface base não pode ser um parâmetro de tipo por conta própria, embora possa envolver os parâmetros de tipo que estão no escopo. O código a seguir ilustra como uma classe pode implementar e estender tipos construídos:

C#

```
class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}
```

As interfaces base de uma declaração de classe genérica devem satisfazer a regra de exclusividade descrita em [exclusividade das interfaces implementadas](#).

## Implementações explícitas de membro de interface

Para fins de implementação de interfaces, uma classe ou estrutura pode declarar **implementações explícitas de membro de interface**. Uma implementação de membro de interface explícita é uma declaração de método, propriedade, evento ou indexador que faz referência a um nome de membro de interface totalmente qualificado. Por exemplo,

C#

```
interface IList<T>
{
    T[] GetElements();
}

interface IDictionary<K,V>
{
    V this[K key];
    void Add(K key, V value);
}
```

```
class List<T>: IList<T>, IDictionary<int,T>
{
    T[] IList<T>.GetElements() {...}
    T IDictionary<int,T>.this[int index] {...}
    void IDictionary<int,T>.Add(int index, T value) {...}
}
```

Aqui `IDictionary<int,T>.this` `IDictionary<int,T>.Add` estão as implementações explícitas de membros de interface.

Em alguns casos, o nome de um membro de interface pode não ser apropriado para a classe de implementação; nesse caso, o membro de interface pode ser implementado usando a implementação de membro de interface explícita. Uma classe que implementa uma abstração de arquivo, por exemplo, provavelmente implementaria uma `Close` função de membro que tem o efeito de liberar o recurso de arquivo e implementaria o `Dispose` método da `IDisposable` interface usando a implementação de membro de interface explícita:

C#

```
interface IDisposable
{
    void Dispose();
}

class MyFile: IDisposable
{
    void IDisposable.Dispose() {
        Close();
    }

    public void Close() {
        // Do what's necessary to close the file
        System.GC.SuppressFinalize(this);
    }
}
```

Não é possível acessar uma implementação de membro de interface explícita por meio de seu nome totalmente qualificado em uma invocação de método, acesso de propriedade ou acesso de indexador. Uma implementação de membro de interface explícita só pode ser acessada por meio de uma instância de interface e, nesse caso, é referenciada simplesmente pelo seu nome de membro.

É um erro de tempo de compilação para uma implementação de membro de interface explícita para incluir modificadores de acesso e é um erro de tempo de compilação para incluir os modificadores `abstract`, `virtual`, `override` ou `static`.

Implementações explícitas de membros de interface têm características de acessibilidade diferentes de outros membros. Como implementações explícitas de membros de interface nunca são acessíveis por meio de seu nome totalmente qualificado em uma invocação de método ou um acesso de propriedade, elas estão em um aspecto particular. No entanto, como eles podem ser acessados por meio de uma instância de interface, eles também são públicos.

Implementações explícitas de membro de interface atendem a duas finalidades principais:

- Como as implementações explícitas de membro de interface não são acessíveis por meio de instâncias de classe ou struct, elas permitem que as implementações de interface sejam excluídas da interface pública de uma classe ou estrutura. Isso é particularmente útil quando uma classe ou estrutura implementa uma interface interna que não é de interesse de um consumidor dessa classe ou struct.
- Implementações explícitas de membros de interface permitem a desambiguidade de membros de interface com a mesma assinatura. Sem implementações explícitas de membros de interface, seria impossível que uma classe ou estrutura tenha diferentes implementações de membros de interface com a mesma assinatura e tipo de retorno, como seria impossível para uma classe ou estrutura ter qualquer implementação em todos os membros da interface com a mesma assinatura, mas com tipos de retorno diferentes.

Para que uma implementação de membro de interface explícita seja válida, a classe ou struct deve nomear uma interface em sua lista de classes base que contém um membro cujos nomes totalmente qualificados, tipos, e tipo de parâmetro correspondem exatamente aos da implementação de membro de interface explícita. Portanto, na seguinte classe

```
C#  
  
class Shape: ICloneable  
{  
    object ICloneable.Clone() {...}  
    int IComparable.CompareTo(object other) {...}    // invalid  
}
```

a declaração de `IComparable.CompareTo` resulta em um erro de tempo de compilação porque `IComparable` não está listado na lista de classes base de `Shape` e não é uma interface base do `ICloneable`. Da mesma forma, nas declarações

```
C#
```

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
}

class Ellipse: Shape
{
    object ICloneable.Clone() {...} // invalid
}
```

a declaração do `ICloneable.Clone` nos `Ellipse` resulta em um erro de tempo de compilação porque `ICloneable` não está explicitamente listado na lista de classes base do `Ellipse`.

O nome totalmente qualificado de um membro de interface deve referenciar a interface na qual o membro foi declarado. Portanto, nas declarações

C#

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

a implementação explícita de membro de interface de `Paint` deve ser escrita como `IControl.Paint`.

## Exclusividade das interfaces implementadas

As interfaces implementadas por uma declaração de tipo genérico devem permanecer exclusivas para todos os tipos construídos possíveis. Sem essa regra, seria impossível determinar o método correto a ser chamado para determinados tipos construídos. Por exemplo, suponha que uma declaração de classe genérica tenha permissão para ser escrita da seguinte maneira:

C#

```
interface I<T>
{
    void F();
}

class X<U,V>: I<U>, I<V>           // Error: I<U> and I<V> conflict
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
```

Isso era permitido, seria impossível determinar qual código deve ser executado no seguinte caso:

C#

```
I<int> x = new X<int,int>();
x.F();
```

Para determinar se a lista de interfaces de uma declaração de tipo genérico é válida, as etapas a seguir são executadas:

- Permite que a lista de interfaces seja especificada diretamente em uma declaração de classe, struct ou interface genérica .
- Adicione a qualquer interface base das interfaces já existentes no .
- Remova as duplicatas de .
- Se qualquer tipo construído possível criado a partir de , depois que os argumentos de tipo forem substituídos , fazer com que duas interfaces sejam idênticas; a declaração de é inválida. Declarações de restrição não são consideradas ao determinar todos os tipos construídos possíveis.

Na declaração de classe X acima, a lista de interface consiste em I<U> e I<V> . A declaração é inválida porque qualquer tipo construído com U e V sendo o mesmo tipo faria com que essas duas interfaces fossem tipos idênticos.

É possível que as interfaces especificadas em diferentes níveis de herança sejam unificadas:

C#

```
interface I<T>
{
    void F();
}
```

```

class Base<U>: I<U>
{
    void I<U>.F() {...}
}

class Derived<U,V>: Base<U>, I<V> // Ok
{
    void I<V>.F() {...}
}

```

Esse código é válido mesmo que `Derived<U,V>` implemente o `I<U>` e o `I<V>`. O código

C#

```

I<int> x = new Derived<int,int>();
x.F();

```

invoca o método no `Derived`, desde que `Derived<int,int>` efetivamente Reimplemente `I<int>` ([reimplementação de interface](#)).

## Implementação de métodos genéricos

Quando um método genérico implementa implicitamente um método de interface, as restrições dadas para cada parâmetro de tipo de método devem ser equivalentes em ambas as declarações (depois que quaisquer parâmetros de tipo de interface são substituídos pelos argumentos de tipo apropriados), em que os parâmetros de tipo de método são identificados por posições ordinais, da esquerda para a direita.

No entanto, quando um método genérico implementa explicitamente um método de interface, nenhuma restrição é permitida no método de implementação. Em vez disso, as restrições são herdadas do método de interface

C#

```

interface I<A,B,C>
{
    void F<T>(T t) where T: A;
    void G<T>(T t) where T: B;
    void H<T>(T t) where T: C;
}

class C: I<object,C,string>
{
    public void F<T>(T t) {...} // Ok
    public void G<T>(T t) where T: C {...} // Ok
}

```

```
    public void H<T>(T t) where T: string {...} // Error
}
```

O método `C.F<T>` implementa implicitamente `I<object,C,string>.F<T>`. Nesse caso, `C.F<T>` o não é necessário (nem permitido) para especificar a restrição `T:object`, pois `object` é uma restrição implícita em todos os parâmetros de tipo. O método `C.G<T>` implementa implicitamente `I<object,C,string>.G<T>` porque as restrições correspondem àquelas na interface, depois que os parâmetros de tipo de interface são substituídos pelos argumentos de tipo correspondentes. A restrição para `C.H<T>` o método é um erro porque os tipos lacrados (`string` nesse caso) não podem ser usados como restrições. Omitir a restrição também seria um erro, já que restrições de implementações do método de interface implícita são necessárias para corresponder. Portanto, é impossível implementar implicitamente `I<object,C,string>.H<T>`. Este método de interface só pode ser implementado usando uma implementação de membro de interface explícita:

```
C#
class C: I<object,C,string>
{
    ...
    public void H<U>(U u) where U: class {...}

    void I<object,C,string>.H<T>(T t) {
        string s = t;      // Ok
        H<T>(t);
    }
}
```

Neste exemplo, a implementação de membro de interface explícita invoca um método público com restrições estritamente mais fracas. Observe que a atribuição de `t` para `s` é válida desde que `T` herde uma restrição de `T:string`, mesmo que essa restrição não seja expressa no código-fonte.

## Mapeamento de interface

Uma classe ou estrutura deve fornecer implementações de todos os membros das interfaces que estão listadas na lista classe base da classe ou estrutura. O processo de localizar implementações de membros de interface em uma classe de implementação ou struct é conhecido como *mapeamento de interface*.

O mapeamento de interface para uma classe ou struct `C` localiza uma implementação para cada membro de cada interface especificada na lista de classes base do `C`. A implementação de um membro de interface específico `I.M`, em que `I` é a interface na qual o membro `M` é declarado, é determinada examinando cada classe ou struct `S`, começando com `C` e repetindo cada classe base sucessiva de `C`, até que uma correspondência seja localizada:

- Se `S` contiver uma declaração de uma implementação de membro de interface explícita que corresponda `I M` a `e`, esse membro será a implementação de `I.M`.
- Caso contrário, se `S` contiver uma declaração de um membro público não estático que corresponda `M`, esse membro será a implementação de `I.M`. Se mais de um membro corresponder, ele não especificará qual membro é a implementação de `I.M`. Essa situação só pode ocorrer se `S` for um tipo construído em que os dois membros como declarados no tipo genérico tenham assinaturas diferentes, mas os argumentos de tipo tornam suas assinaturas idênticas.

Ocorrerá um erro de tempo de compilação se não for possível localizar implementações para todos os membros de todas as interfaces especificadas na lista de classes base de `C`. Observe que os membros de uma interface incluem os membros que são herdados das interfaces base.

Para fins de mapeamento de interface, um membro de classe `A` corresponde a um membro de interface `B` quando:

- `A` e `B` são métodos, e as listas de parâmetro Name, Type e formal de `A` e `B` são idênticas.
- `A` e `B` são propriedades, o nome e o tipo de `A` e `B` são idênticos e `A` têm os mesmos acessadores que (tem `B` `A` permissão para ter acessadores adicionais se não for uma implementação de membro de interface explícita).
- `A` e `B` são eventos, e o nome e o tipo de `A` e `B` são idênticos.
- `A` e `B` são indexadores, o tipo e as listas de parâmetros formais de `A` e `B` são idênticos e `A` têm os mesmos acessadores como `B` (`A` é permitido ter acessadores adicionais se não for uma implementação de membro de interface explícita).

As implicações notáveis do algoritmo de mapeamento de interface são:

- Implementações explícitas de membros de interface têm precedência sobre outros membros na mesma classe ou struct ao determinar o membro Class ou struct que implementa um membro de interface.

- Nem membros não públicos nem estáticos participam do mapeamento de interface.

No exemplo

C#

```
interface ICloneable
{
    object Clone();
}

class C: ICloneable
{
    object ICloneable.Clone() {...}
    public object Clone() {...}
}
```

O `ICloneable.Clone` membro de `C` se torna a implementação do `Clone` no `ICloneable` porque as implementações explícitas de membros de interface têm precedência sobre outros membros.

Se uma classe ou struct implementa duas ou mais interfaces que contêm um membro com os mesmos tipos de nome, tipo e parâmetro, é possível mapear cada um desses membros de interface em um único membro de classe ou struct. Por exemplo,

C#

```
interface IControl
{
    void Paint();
}

interface IForm
{
    void Paint();
}

class Page: IControl, IForm
{
    public void Paint() {...}
}
```

Aqui, os `Paint` métodos de `IControl` e `IForm` são mapeados para o `Paint` método no `Page`. É claro que também é possível ter implementações de membros de interface explícita separadas para os dois métodos.

Se uma classe ou estrutura implementar uma interface que contenha Membros ocultos, alguns membros deverão ser necessariamente implementados por meio de implementações de membro de interface explícita. Por exemplo,

```
C#  
  
interface IBase  
{  
    int P { get; }  
}  
  
interface IDerived: IBase  
{  
    new int P();  
}
```

Uma implementação dessa interface exigiria pelo menos uma implementação de membro de interface explícita e usaria um dos seguintes formulários

```
C#  
  
class C: IDerived  
{  
    int IBase.P { get {...} }  
    int IDerived.P() {...}  
}  
  
class C: IDerived  
{  
    public int P { get {...} }  
    int IDerived.P() {...}  
}  
  
class C: IDerived  
{  
    int IBase.P { get {...} }  
    public int P() {...}  
}
```

Quando uma classe implementa várias interfaces que têm a mesma interface base, pode haver apenas uma implementação da interface base. No exemplo

```
C#  
  
interface IControl  
{  
    void Paint();  
}
```

```

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

class ComboBox: IControl, ITextBox, IListBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
    void IListBox.SetItems(string[] items) {...}
}

```

Não é possível ter implementações separadas para o `IControl` nome na lista de classes base, o `IControl` herdado por `ITextBox` e o `IControl` herdado por `IListBox`. Na verdade, não há nenhuma noção de uma identidade separada para essas interfaces. Em vez disso, as implementações de `ITextBox` e `IListBox` compartilham a mesma implementação de `IControl`, e `ComboBox` são simplesmente consideradas para implementar três interfaces, `IControl`, `ITextBox` e `IListBox`.

Os membros de uma classe base participam do mapeamento de interface. No exemplo

```

C#

interface Interface1
{
    void F();
}

class Class1
{
    public void F() {}
    public void G() {}
}

class Class2: Class1, Interface1
{
    new public void G() {}
}

```

o método `F` no `Class1` é usado na `Class2` implementação de `Interface1`.

## Herança de implementação de interface

Uma classe herda todas as implementações de interface fornecidas por suas classes base.

Sem a **reimplementação** explícita de uma interface, uma classe derivada não pode alterar os mapeamentos de interface herdados de suas classes base. Por exemplo, nas declarações

```
C#  
  
interface IControl  
{  
    void Paint();  
}  
  
class Control: IControl  
{  
    public void Paint() {...}  
}  
  
class TextBox: Control  
{  
    new public void Paint() {...}  
}
```

O `Paint` método em `TextBox` oculta o `Paint` método em `Control`, mas não altera o mapeamento de `Control.Paint` para e as `IControl.Paint` chamadas para as instâncias de `Paint` classe e instâncias de interface terão os seguintes efeitos

```
C#  
  
Control c = new Control();  
TextBox t = new TextBox();  
IControl ic = c;  
IControl it = t;  
c.Paint();           // invokes Control.Paint();  
t.Paint();           // invokes TextBox.Paint();  
ic.Paint();          // invokes Control.Paint();  
it.Paint();          // invokes Control.Paint();
```

No entanto, quando um método de interface é mapeado em um método virtual em uma classe, é possível que as classes derivadas substituam o método virtual e alterem a implementação da interface. Por exemplo, reescrever as declarações acima para

```
C#  
  
interface IControl  
{  
    void Paint();  
}
```

```
}

class Control: IControl
{
    public virtual void Paint() {...}
}

class TextBox: Control
{
    public override void Paint() {...}
}
```

os efeitos a seguir agora serão observados

C#

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();           // invokes Control.Paint();
t.Paint();           // invokes TextBox.Paint();
ic.Paint();          // invokes Control.Paint();
it.Paint();          // invokes TextBox.Paint();
```

Como as implementações explícitas de membro de interface não podem ser declaradas como virtuais, não é possível substituir uma implementação de membro de interface explícita. No entanto, é perfeitamente válido para uma implementação de membro de interface explícita chamar outro método, e esse outro método pode ser declarado como virtual para permitir que as classes derivadas a substituam. Por exemplo,

C#

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() { PaintControl(); }
    protected virtual void PaintControl() {...}
}

class TextBox: Control
{
    protected override void PaintControl() {...}
}
```

Aqui, as classes derivadas de `Control` podem especializar a implementação do `IControl.Paint` substituindo o `PaintControl` método.

## Reimplementação da interface

Uma classe que herda uma implementação de interface tem permissão para implementar a interface **novamente**, incluindo-a na lista de classes base.

Uma nova implementação de uma interface segue exatamente as mesmas regras de mapeamento de interface que uma implementação inicial de uma interface. Portanto, o mapeamento de interface herdado não tem nenhum efeito no mapeamento de interface estabelecido para a reimplementação da interface. Por exemplo, nas declarações

```
C#  
  
interface IControl  
{  
    void Paint();  
}  
  
class Control: IControl  
{  
    void IControl.Paint() {...}  
}  
  
class MyControl: Control, IControl  
{  
    public void Paint() {}  
}
```

o fato de que `Control` mapas `IControl.Paint` em `Control.IControl.Paint` não afeta a reimplementação no `MyControl`, que mapeia `IControl.Paint` para `MyControl.Paint`.

As declarações de membro público herdadas e as declarações de membro de interface explícita herdadas participam do processo de mapeamento de interface para interfaces reimplementadas. Por exemplo,

```
C#  
  
interface IMethods  
{  
    void F();  
    void G();  
    void H();  
    void I();  
}
```

```

class Base: IMethods
{
    void IMethods.F() {}
    void IMethods.G() {}
    public void H() {}
    public void I() {}
}

class Derived: Base, IMethods
{
    public void F() {}
    void IMethods.H() {}
}

```

Aqui, a implementação do `IMethods` no `Derived` mapeia os métodos de interface para `Derived.F`, `Base.IMethods.G`, `Derived.IMETHODS.H` e `Base.I`.

Quando uma classe implementa uma interface, ela também implementa implicitamente todas as interfaces base dessa interface. Da mesma forma, uma reimplementação de uma interface também é implicitamente uma reimplementação de todas as interfaces base da interface. Por exemplo,

C#

```

interface IBase
{
    void F();
}

interface IDerived: IBase
{
    void G();
}

class C: IDerived
{
    void IBase.F() {...}
    void IDerived.G() {...}
}

class D: C, IDerived
{
    public void F() {...}
    public void G() {...}
}

```

Aqui, a nova implementação de `IDerived` também reimplementa `IBase`, mapeamento `IBase.F` para `D.F`.

## Interfaces e classes abstratas

Como uma classe não abstrata, uma classe abstrata deve fornecer implementações de todos os membros das interfaces que estão listadas na lista classe base da classe. No entanto, uma classe abstrata é permitida para mapear métodos de interface em métodos abstratos. Por exemplo,

C#

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    public abstract void F();
    public abstract void G();
}
```

Aqui, a implementação de `IMethods` mapas `F` e `G` em métodos abstratos, que deve ser substituída em classes não abstratas que derivam de `C`.

Observe que as implementações explícitas de membros de interface não podem ser abstratas, mas implementações explícitas de membros de interface, obviamente, têm permissão para chamar métodos abstract. Por exemplo,

C#

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    void IMethods.F() { FF(); }
    void IMethods.G() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}
```

Aqui, as classes não abstratas que derivam de `C` seriam necessárias para substituir `FF` e `GG`, portanto, fornecer a implementação real de `IMethods`.

# Enumerações

Artigo • 16/09/2021

Um *tipo de enumeração* é um tipo de valor distinto ([tipos de valor](#)) que declara um conjunto de constantes nomeadas.

O exemplo

C#

```
enum Color
{
    Red,
    Green,
    Blue
}
```

declara um tipo enum chamado `Color` com members `Red` , `Green` e `Blue` .

## Declarações de enum

Uma declaração enum declara um novo tipo enum. Uma declaração enum começa com a palavra-chave `enum` e define o nome, a acessibilidade, o tipo subjacente e os membros da enumeração.

antlr

```
enum_declaration
    : attributes? enum_modifier* 'enum' identifier enum_base? enum_body ';'?
;

enum_base
    : ':' integral_type
;

enum_body
    : '{' enum_member_declarations? '}'
    | '{' enum_member_declarations ',' '}'
;
```

Cada tipo de enumeração tem um tipo integral correspondente chamado de *tipo subjacente* do tipo de enumeração. Esse tipo subjacente deve ser capaz de representar todos os valores de enumerador definidos na enumeração. Uma declaração de enumeração pode declarar explicitamente um tipo subjacente de,,, `byte` `sbyte` `short`

`ushort` `int` `uint` `long` ou `ulong`. Observe que `char` não pode ser usado como um tipo subjacente. Uma declaração de enumeração que não declara explicitamente um tipo subjacente tem um tipo subjacente de `int`.

O exemplo

C#

```
enum Color: long
{
    Red,
    Green,
    Blue
}
```

declara um enum com um tipo subjacente de `long`. Um desenvolvedor pode optar por usar um tipo subjacente de `long`, como no exemplo, para habilitar o uso de valores que estão no intervalo de, `long` mas não no intervalo de `int`, ou para preservar essa opção para o futuro.

## Modificadores de enum

Um *enum\_declaration* pode, opcionalmente, incluir uma sequência de modificadores de enumeração:

antlr

```
enum_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
;
```

É um erro de tempo de compilação para o mesmo modificador aparecer várias vezes em uma declaração de enumeração.

Os modificadores de uma declaração enum têm o mesmo significado que os de uma declaração de classe ([modificadores de classe](#)). Observe, no entanto, que os `abstract` `sealed` modificadores e não são permitidos em uma declaração enum. Enums não podem ser `abstract` e não permitem derivação.

# Membros de enum

O corpo de uma declaração de tipo enum define zero ou mais membros enum, que são as constantes nomeadas do tipo enum. Dois membros enum não podem ter o mesmo nome.

```
antlr
```

```
enum_member_declarations
: enum_member_declaration (',' enum_member_declaration)*
;

enum_member_declaration
: attributes? identifier ('=' constant_expression)?
;
```

Cada membro enum tem um valor constante associado. O tipo desse valor é o tipo subjacente para a enumeração que o contém. O valor constante para cada membro de enumeração deve estar no intervalo do tipo subjacente para a enumeração. O exemplo

```
C#
```

```
enum Color: uint
{
    Red = -1,
    Green = -2,
    Blue = -3
}
```

resulta em um erro de tempo de compilação porque os valores de constante `-1`, `-2` e `-3` não estão no intervalo do tipo integral subjacente `uint`.

Vários membros enum podem compartilhar o mesmo valor associado. O exemplo

```
C#
```

```
enum Color
{
    Red,
    Green,
    Blue,
    Max = Blue
}
```

mostra uma enumeração na qual dois membros de enum-- `Blue` e `Max`--têm o mesmo valor associado.

O valor associado de um membro enum é atribuído implicitamente ou explicitamente. Se a declaração do membro enum tiver um inicializador *constant\_expression*, o valor dessa expressão constante, implicitamente convertido no tipo subjacente da enumeração, será o valor associado do membro enum. Se a declaração do membro enum não tiver nenhum inicializador, seu valor associado será definido implicitamente, da seguinte maneira:

- Se o membro enum for o primeiro membro enum declarado no tipo enum, seu valor associado será zero.
- Caso contrário, o valor associado do membro de enumeração será obtido aumentando o valor associado do membro de enumeração textualmente anterior por um. Esse valor maior deve estar dentro do intervalo de valores que podem ser representados pelo tipo subjacente. caso contrário, ocorrerá um erro de tempo de compilação.

O exemplo

```
C#  
  
using System;  
  
enum Color  
{  
    Red,  
    Green = 10,  
    Blue  
}  
  
class Test  
{  
    static void Main() {  
        Console.WriteLine(StringFromColor(Color.Red));  
        Console.WriteLine(StringFromColor(Color.Green));  
        Console.WriteLine(StringFromColor(Color.Blue));  
    }  
  
    static string StringFromColor(Color c) {  
        switch (c) {  
            case Color.Red:  
                return String.Format("Red = {0}", (int) c);  
  
            case Color.Green:  
                return String.Format("Green = {0}", (int) c);  
  
            case Color.Blue:  
                return String.Format("Blue = {0}", (int) c);  
        }  
    }  
}
```

```
        default:
            return "Invalid color";
    }
}
```

imprime os nomes de membro de enumeração e seus valores associados. A saída é:

Console

```
Red = 0
Green = 10
Blue = 11
```

pelos seguintes motivos:

- o membro de enumeração `Red` recebe automaticamente o valor zero (já que ele não tem nenhum inicializador e é o primeiro membro de enumeração);
- o membro de enumeração `Green` recebe explicitamente o valor `10`;
- e o membro enum `Blue` recebe automaticamente o valor um maior que o membro que o precede de uma vez.

O valor associado de um membro enum pode não ser, direta ou indiretamente, usar o valor de seu próprio membro enum associado. Além dessa restrição de circularidade, os inicializadores de membro de enumeração podem se referir livremente a outros inicializadores de membro de enumeração, independentemente da sua posição textual. Dentro de um inicializador de membro de enumeração, os valores de outros membros de enum são sempre tratados como tendo o tipo de seu tipo subjacente, de modo que as conversões não são necessárias ao fazer referência a outros membros de enumeração.

O exemplo

C#

```
enum Circular
{
    A = B,
    B
}
```

resulta em um erro de tempo de compilação porque as declarações de `A` e `B` são circulares. `A` depende `B` explicitamente e `B` depende `A` implicitamente.

Os membros de enumeração são nomeados e têm o escopo definido de forma exatamente análoga aos campos dentro das classes. O escopo de um membro de enumeração é o corpo de seu tipo de enumeração que o contém. Dentro desse escopo, os membros de enumeração podem ser referenciados por seu nome simples. De todos os outros códigos, o nome de um membro de enumeração deve ser qualificado com o nome de seu tipo de enumeração. Os membros de enumeração não têm nenhuma acessibilidade declarada--um membro de enumeração é acessível se seu tipo de enumeração que o contém está acessível.

## O tipo de System.Enum

O tipo `System.Enum` é a classe base abstrata de todos os tipos enum (isso é distinto e diferente do tipo subjacente do tipo enum) e os membros herdados de `System.Enum` estão disponíveis em qualquer tipo de enumeração. Uma conversão boxing ([conversões Boxing](#)) existe de qualquer tipo enum para `System.Enum`, e uma conversão unboxing ([conversões unboxing](#)) existe a partir de `System.Enum` qualquer tipo enum.

Observe que `System.Enum` não é um *enum\_type*. Em vez disso, é uma *class\_type* da qual todos os *enum\_type* são derivados. O tipo `System.Enum` herda do tipo `System.ValueType` ([o tipo System.ValueType](#)), que, por sua vez, herda do tipo `object`. Em tempo de execução, um valor do tipo `System.Enum` pode ser `null` ou uma referência a um valor em caixa de qualquer tipo de enumeração.

## Operações e valores de enum

Cada tipo de enumeração define um tipo distinto; uma conversão de enumeração explícita ([conversões de enumeração explícitas](#)) é necessária para converter entre um tipo de enumeração e um tipo integral ou entre dois tipos de enumeração. O conjunto de valores que um tipo de enumeração pode assumir não é limitado por seus membros enum. Em particular, qualquer valor do tipo subjacente de uma enumeração pode ser convertido para o tipo de enumeração e é um valor válido distinto desse tipo de enumeração.

Os membros de enumeração têm o tipo de tipo de enumeração contendo (exceto em outros inicializadores de membro de enumeração: consulte [membros de enumeração](#)). O valor de um membro enum declarado no tipo enum `E` com o valor associado `v` é `(E)v`.

Os operadores a seguir podem ser usados em valores de tipos de enumeração:,,, == != < > <= >= ([operadores de comparação de enumeração](#)), binary + (operador

de adição), binário - (operador de subtração), ^ & , | (operadores lógicos de enumeração), ~ (operador de complemento de byte do) ++ e -- (incremento de sufixo e diminuição de operadores e incrementos de prefixo e diminuir

Cada tipo de enumeração deriva automaticamente da classe `System.Enum` (que, por sua vez, deriva de `System.ValueType` e `object`). Assim, os métodos herdados e as propriedades dessa classe podem ser usados em valores de um tipo enum.

# Delegados

Artigo • 16/09/2021

Os delegados habilitam cenários que outras linguagens, como C++, Pascal e modula, foram abordadas com ponteiros de função. No entanto, ao contrário dos ponteiros de função do C++, os delegados são totalmente orientados a objeto e, ao contrário de ponteiros do C++ para funções de membro, os delegados encapsulam uma instância de objeto e um

Uma declaração delegate define uma classe que é derivada da classe `System.Delegate`. Uma instância de delegado encapsula uma lista de invocação, que é uma lista de um ou mais métodos, cada um dos quais é conhecido como uma entidade que possa ser chamada. Para métodos de instância, uma entidade que possa ser chamada consiste em uma instância e um método nessa instância. Para métodos estáticos, uma entidade que possa ser chamada consiste apenas em um método. Invocar uma instância de delegado com um conjunto apropriado de argumentos faz com que cada uma das entidades chamáveis do delegado seja invocada com o conjunto de argumentos fornecido.

Uma propriedade interessante e útil de uma instância de delegado é que ela não sabe ou se preocupa com as classes dos métodos que encapsula; Tudo o que importa é que esses métodos sejam compatíveis ([delegar declarações](#)) com o tipo do delegado. Isso torna os delegados perfeitamente adequados para invocação "anônima".

## Declarações de delegado

Uma `delegate_declaration` é uma `type_declaration` ([declarações de tipo](#)) que declara um novo tipo de delegado.

```
antlr

delegate_declaration
: attributes? delegate_modifier* 'delegate' return_type
  identifier variant_type_parameter_list?
  '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
;

delegate_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| delegate_modifier_unsafe
;
```

É um erro de tempo de compilação para o mesmo modificador aparecer várias vezes em uma declaração de delegado.

O `new` modificador só é permitido em delegados declarados dentro de outro tipo; nesse caso, ele especifica que um delegado de cada vez oculta um membro herdado com o mesmo nome, conforme descrito no [novo modificador](#).

Os `public` `protected` `internal` modificadores, e `private` controlam a acessibilidade do tipo delegado. Dependendo do contexto no qual a declaração delegate ocorre, alguns desses modificadores podem não ser permitidos ([acessibilidade declarada](#)).

O nome do tipo do delegado é *identificador*.

O *formal\_parameter\_list* opcional especifica os parâmetros do delegado e *return\_type* indica o tipo de retorno do delegado.

As *variant\_type\_parameter\_list* opcionais ([listas de parâmetros de tipo Variant](#)) especificam os parâmetros de tipo para o próprio delegado.

O tipo de retorno de um tipo delegado deve ser `void` ou de saída (segurança devariação).

Todos os tipos de parâmetro formais de um tipo delegado devem ser de entrada segura. Além disso, `out` qualquer `ref` tipo de parâmetro ou também deve ser de saída seguro. Observe que até mesmo `out` os parâmetros devem ser de entrada segura, devido a uma limitação da plataforma de execução subjacente.

Tipos delegados em C# são equivalentes a nome, não são estruturalmente equivalentes. Especificamente, dois tipos delegados diferentes que têm as mesmas listas de parâmetros e tipos de retorno são considerados tipos delegados diferentes. No entanto, as instâncias de dois tipos delegados distintos, mas estruturalmente equivalentes podem ser comparadas como iguais ([operadores de igualdade de delegado](#)).

Por exemplo:

```
C#  
  
delegate int D1(int i, double d);  
  
class A  
{  
    public static int M1(int a, double b) {...}  
}  
  
class B
```

```
{  
    delegate int D2(int c, double d);  
    public static int M1(int f, double g) {...}  
    public static void M2(int k, double l) {...}  
    public static int M3(int g) {...}  
    public static void M4(int g) {...}  
}
```

Os métodos `A.M1` e `B.M1` são compatíveis com os tipos delegados `D1` e `D2`, como têm o mesmo tipo de retorno e lista de parâmetros; no entanto, esses tipos delegados são dois tipos diferentes, portanto, não são intercambiáveis. Os métodos `B.M2`, `B.M3`, e `B.M4` são incompatíveis com os tipos delegados `D1` e `D2`, como têm diferentes tipos de retorno ou listas de parâmetros.

Assim como outras declarações de tipo genérico, argumentos de tipo devem ser fornecidos para criar um tipo de delegado construído. Os tipos de parâmetro e o tipo de retorno de um tipo de delegado construído são criados pela substituição, para cada parâmetro de tipo na declaração de delegado, o argumento de tipo correspondente do tipo de delegado construído. O tipo de retorno resultante e os tipos de parâmetro são usados para determinar quais métodos são compatíveis com um tipo de delegado construído. Por exemplo:

```
C#  
  
delegate bool Predicate<T>(T value);  
  
class X  
{  
    static bool F(int i) {...}  
    static bool G(string s) {...}  
}
```

O método `X.F` é compatível com o tipo delegado `Predicate<int>` e o método `X.G` é compatível com o tipo delegado `Predicate<string>`.

A única maneira de declarar um tipo delegado é por meio de um *delegate declaration*. Um tipo delegado é um tipo de classe que é derivado de `System.Delegate`. Os tipos delegados são implicitamente `sealed`, portanto, não é permitido derivar qualquer tipo de um tipo delegado. Também não é permitido derivar um tipo de classe não-delegado de `System.Delegate`. Observe que `System.Delegate` não é, em si, um tipo delegado; é um tipo de classe do qual todos os tipos delegados são derivados.

O C# fornece uma sintaxe especial para a instanciação e a invocação de delegado. Exceto para instanciação, qualquer operação que possa ser aplicada a uma classe ou

instância de classe também pode ser aplicada a uma classe ou instância de delegado, respectivamente. Em particular, é possível acessar os membros do `System.Delegate` tipo por meio da sintaxe de acesso de membro usual.

O conjunto de métodos encapsulado por uma instância de delegado é chamado de lista de invocação. Quando uma instância de delegado é criada ([compatibilidade de delegado](#)) de um único método, ela encapsula esse método e sua lista de invocação contém apenas uma entrada. No entanto, quando duas instâncias delegadas não nulas são combinadas, suas listas de invocação são concatenadas – no operando de ordem esquerda e, em seguida, operando à direita – para formar uma nova lista de invocação, que contém duas ou mais entradas.

Os delegados são combinados usando o `+` operador binary ([adição](#)) e os `+=` operadores ([atribuição composta](#)). Um delegado pode ser removido de uma combinação de delegados, usando o binário `-` ([operador de subtração](#)) e `-=` operadores ([atribuição composta](#)). Os delegados podem ser comparados para igualdade ([operadores de igualdade de delegado](#)).

O exemplo a seguir mostra a instanciação de vários delegados e suas listas de invocação correspondentes:

```
C#  
  
delegate void D(int x);  
  
class C  
{  
    public static void M1(int i) {...}  
    public static void M2(int i) {...}  
  
}  
  
class Test  
{  
    static void Main()  
    {  
        D cd1 = new D(C.M1);           // M1  
        D cd2 = new D(C.M2);           // M2  
        D cd3 = cd1 + cd2;             // M1 + M2  
        D cd4 = cd3 + cd1;             // M1 + M2 + M1  
        D cd5 = cd4 + cd3;             // M1 + M2 + M1 + M1 + M2  
    }  
}
```

Quando `cd1` e `cd2` são instanciados, cada um encapsula um método. Quando `cd3` é instanciado, ele tem uma lista de invocação de dois métodos `M1` e `M2`, nessa ordem.

`cd4` a lista de invocação do contém `M1` , `M2` e `M1` , nessa ordem. Finalmente, `cd5` a lista de invocação do contém,,, `M1 M2 M1 M1` e `M2` , nessa ordem. Para obter mais exemplos de como combinar (bem como remover) delegados, consulte [delegar invocação](#).

## Compatibilidade de delegado

Um método ou delegado `M` é *compatível* com um tipo `D` de delegado se todos os itens a seguir forem verdadeiros:

- `D` e `M` têm o mesmo número de parâmetros, e cada parâmetro no `D` tem os mesmos `ref` ou `out` modificadores que o parâmetro correspondente no `M` .
- Para cada parâmetro de valor (um parâmetro sem `ref` `out` modificador ou), existe uma conversão de identidade ([conversão de identidade](#)) ou conversão de referência implícita ([conversões de referência implícitas](#)) do tipo de parâmetro em `D` para o tipo de parâmetro correspondente no `M` .
- Para cada `ref` `out` parâmetro ou, o tipo de parâmetro no `D` é o mesmo que o tipo de parâmetro em `M` .
- Existe uma conversão de identidade ou de referência implícita do tipo de retorno de `M` para o tipo de retorno de `D` .

## Instanciação de delegado

Uma instância de um delegado é criada por um *delegate\_creation\_expression* ([delegar expressões de criação](#)) ou uma conversão para um tipo delegado. A instância delegada recém-criada então se refere a:

- O método estático referenciado no *delegate\_creation\_expression*, ou
- O objeto de destino (que não pode ser `null` ) e o método de instância referenciados no *delegate\_creation\_expression* ou
- Outro delegado.

Por exemplo:

C#

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public void M2(int i) {...}
}
```

```
class Test
{
    static void Main() {
        D cd1 = new D(C.M1);           // static method
        C t = new C();
        D cd2 = new D(t.M2);          // instance method
        D cd3 = new D(cd2);           // another delegate
    }
}
```

Depois de instanciadas, as instâncias delegadas sempre se referem ao mesmo objeto e método de destino. Lembre-se de que, quando dois delegados são combinados, ou um é removido de outro, um novo delegado resulta em sua própria lista de invocação; as listas de invocação dos delegados combinados ou removidos permanecem inalteradas.

## Invocação de delegado

O C# fornece uma sintaxe especial para invocar um delegado. Quando uma instância de delegado não nula cuja lista de invocação contém uma entrada é invocada, ela invoca um método com os mesmos argumentos que foi fornecido e retorna o mesmo valor que o método referenciado. (Confira as [invocações de representante](#) para obter informações detalhadas sobre a invocação de delegado.) Se ocorrer uma exceção durante a invocação de tal delegado, e essa exceção não for detectada dentro do método que foi invocado, a pesquisa de uma cláusula catch de exceção continuará no método que chamou o delegado, como se esse método tivesse chamado diretamente do método para o qual o delegado foi referenciado.

A invocação de uma instância delegada cuja lista de invocação contém várias entradas continua invocando cada um dos métodos na lista de invocação, de forma síncrona, na ordem. Cada método chamado é passado como o mesmo conjunto de argumentos, como foi dado à instância delegada. Se essa invocação de delegado incluir parâmetros de referência ([parâmetros de referência](#)), cada invocação de método ocorrerá com uma referência à mesma variável; as alterações nessa variável por um método na lista de invocação ficarão visíveis para os métodos mais adiante na lista de invocação. Se a invocação de delegado incluir parâmetros de saída ou um valor de retorno, o valor final será proveniente da invocação do último delegado na lista.

Se ocorrer uma exceção durante o processamento da invocação de tal delegado, e essa exceção não for detectada dentro do método que foi invocado, a pesquisa de uma cláusula catch de exceção continuará no método que chamou o delegado, e todos os métodos mais abaixo da lista de invocação não serão invocados.

A tentativa de invocar uma instância delegada cujo valor é nulo resulta em uma exceção do tipo `System.NullReferenceException`.

O exemplo a seguir mostra como criar uma instância, combinar, remover e invocar delegados:

C#

```
using System;

delegate void D(int x);

class C
{
    public static void M1(int i) {
        Console.WriteLine("C.M1: " + i);
    }

    public static void M2(int i) {
        Console.WriteLine("C.M2: " + i);
    }

    public void M3(int i) {
        Console.WriteLine("C.M3: " + i);
    }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        cd1(-1); // call M1

        D cd2 = new D(C.M2);
        cd2(-2); // call M2

        D cd3 = cd1 + cd2;
        cd3(10); // call M1 then M2

        cd3 += cd1;
        cd3(20); // call M1, M2, then M1

        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
        cd3(30); // call M1, M2, M1, then M3

        cd3 -= cd1; // remove last M1
        cd3(40); // call M1, M2, then M3

        cd3 -= cd4;
        cd3(50); // call M1 then M2
    }
}
```

```

        cd3 -= cd2;
        cd3(60);           // call M1

        cd3 -= cd2;        // impossible removal is benign
        cd3(60);           // call M1

        cd3 -= cd1;        // invocation list is empty so cd3 is null

        cd3(70);           // System.NullReferenceException thrown

        cd3 -= cd1;        // impossible removal is benign
    }
}

```

Conforme mostrado na instrução `cd3 += cd1;`, um delegado pode estar presente em uma lista de invocação várias vezes. Nesse caso, ele é simplesmente invocado uma vez por ocorrência. Em uma lista de invocação como essa, quando esse delegado é removido, a última ocorrência na lista de invocação é a que foi realmente removida.

Imediatamente antes da execução da instrução final, `cd3 -= cd1;`, o delegado `cd3` se refere a uma lista de invocação vazia. A tentativa de remover um delegado de uma lista vazia (ou de remover um delegado não existente de uma lista não vazia) não é um erro.

A saída produzida é:

Console

```

C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M2: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60

```

# Exceções

Artigo • 16/09/2021

As exceções em C# fornecem uma maneira estruturada, uniforme e de tipo seguro de lidar com as condições de erro no nível do sistema e do aplicativo. O mecanismo de exceção no C# é bastante semelhante ao do C++, com algumas diferenças importantes:

- Em C#, todas as exceções devem ser representadas por uma instância de um tipo de classe derivada de `System.Exception`. Em C++, qualquer valor de qualquer tipo pode ser usado para representar uma exceção.
- No C#, um bloco `finally` ([a instrução try](#)) pode ser usado para gravar o código de encerramento que é executado na execução normal e em condições excepcionais. Esse código é difícil de escrever em C++ sem o código de duplicação.
- No C#, exceções no nível do sistema como estouro, divisão por zero e desreferências nulas têm classes de exceção bem definidas e estão em um par com condições de erro em nível de aplicativo.

## Causas de exceções

A exceção pode ser lançada de duas maneiras diferentes.

- Uma `throw` instrução ([a instrução Throw](#)) gera uma exceção imediatamente e incondicionalmente. O controle nunca atinge a instrução imediatamente após o `throw`.
- Determinadas condições excepcionais que surgem durante o processamento de instruções e expressões C# causam uma exceção em determinadas circunstâncias quando a operação não pode ser concluída normalmente. Por exemplo, uma operação de divisão de inteiro ([operador de divisão](#)) gera um `System.DivideByZeroException` se o denominador for zero. Consulte [classe de exceção comuns](#) para obter uma lista das várias exceções que podem ocorrer dessa maneira.

## A classe `System.Exception`

A `System.Exception` classe é o tipo base de todas as exceções. Essa classe tem algumas propriedades notáveis que todas as exceções compartilham:

- `Message` é uma propriedade somente leitura do tipo `string` que contém uma descrição legível do motivo para a exceção.

- `InnerException` é uma propriedade somente leitura do tipo `Exception`. Se seu valor for não nulo, ele se refere à exceção que causou a exceção atual, ou seja, a exceção atual foi gerada em um bloco catch que manipula o `InnerException`. Caso contrário, seu valor será NULL, indicando que essa exceção não foi causada por outra exceção. O número de objetos de exceção encadeados dessa maneira pode ser arbitrário.

O valor dessas propriedades pode ser especificado em chamadas para o construtor de instância para `System.Exception`.

## Como as exceções são tratadas

As exceções são tratadas por uma `try` instrução ([a instrução try](#)).

Quando ocorre uma exceção, o sistema pesquisa a cláusula mais próxima `catch` que pode manipular a exceção, conforme determinado pelo tipo de tempo de execução da exceção. Primeiro, o método atual é procurado por uma instrução de circunscrição lexical `try`, e as cláusulas catch associadas da instrução `try` são consideradas na ordem. Se isso falhar, o método que chamou o método atual será pesquisado em busca de uma instrução de circunscrição lexical `try` que indique o ponto da chamada para o método atual. Essa pesquisa continua até que `catch` seja encontrada uma cláusula que possa manipular a exceção atual, nomeando uma classe de exceção que seja da mesma classe ou uma classe base, do tipo de tempo de execução da exceção que está sendo gerada. Uma `catch` cláusula que não nomeie uma classe de exceção pode manipular qualquer exceção.

Depois que uma cláusula `catch` correspondente é encontrada, o sistema se prepara para transferir o controle para a primeira instrução da cláusula `catch`. Antes da execução da cláusula `catch` começar, o sistema primeiro executa, em ordem, todas as `finally` cláusulas que foram associadas às instruções Try mais aninhadas que aquela que capturou a exceção.

Se nenhuma cláusula `catch` correspondente for encontrada, ocorrerá uma das duas coisas:

- Se a pesquisa de uma cláusula `catch` correspondente atingir um construtor estático ([construtores estáticos](#)) ou inicializador de campo estático, um `System.TypeInitializationException` será lançado no ponto que disparou a invocação do construtor estático. A exceção interna do `System.TypeInitializationException` contém a exceção que foi lançada originalmente.

- Se a pesquisa de cláusulas catch correspondentes atingir o código que iniciou inicialmente o thread, a execução do thread será encerrada. O impacto dessa terminação é definido pela implementação.

Exceções que ocorrem durante a execução do destruidor valem a pena mencionar. Se ocorrer uma exceção durante a execução do destruidor e essa exceção não for detectada, a execução desse destruidor será encerrada e o destruidor da classe base (se houver) será chamado. Se não houver nenhuma classe base (como no caso do `object` tipo) ou se não houver nenhum destruidor de classe base, a exceção será descartada.

## Classes de exceção comuns

As exceções a seguir são geradas por determinadas operações do C#.

<code>System.ArithmeticException</code>	Uma classe base para exceções que ocorrem durante operações aritméticas, tais como <code>System.DivideByZeroException</code> e <code>System.OverflowException</code> .
<code>System.ArrayTypeMismatchException</code>	Gerado quando um repositório em uma matriz falha, porque o tipo real do elemento armazenado é incompatível com o tipo real da matriz.
<code>System.DivideByZeroException</code>	Gerado quando ocorre uma tentativa de dividir um valor integral por zero.
<code>System.IndexOutOfRangeException</code>	Gerado quando uma tentativa de indexar uma matriz por meio de um índice que é menor que zero ou fora dos limites da matriz.
<code>System.InvalidCastException</code>	Gerado quando uma conversão explícita de um tipo base ou interface para um tipo derivado falha em tempo de execução.
<code>System.NullReferenceException</code>	Gerado quando uma <code>null</code> referência é usada de uma maneira que faz com que o objeto referenciado seja necessário.
<code>System.OutOfMemoryException</code>	Gerado quando uma tentativa de alocar memória (via <code>new</code> ) falha.
<code>System.OverflowException</code>	Lançada quando uma operação aritmética em um contexto <code>checked</code> estoura.

<code>System.StackOverflowException</code>	Gerado quando a pilha de execução é esgotada com o excesso de chamadas de método pendentes; normalmente indica uma recursão muito profunda ou não associada.
<code>System.TypeInitializationException</code>	Gerado quando um construtor estático gera uma exceção e nenhuma <code>catch</code> cláusula existe para capturá-la.

# Atributos

Artigo • 16/09/2021

Grande parte da linguagem C# permite que o programador especifique informações declarativas sobre as entidades definidas no programa. Por exemplo, a acessibilidade de um método em uma classe é especificada decorando-o com o *method\_modifier*s `public`, `protected`, `internal`, e `private`.

O C# permite que os programadores inventem novos tipos de informações declarativas, chamados **atributos**. Os programadores podem anexar atributos a várias entidades de programa e recuperar informações de atributo em um ambiente de tempo de execução. Por exemplo, uma estrutura pode definir um `HelpAttribute` atributo que pode ser colocado em determinados elementos do programa (como classes e métodos) para fornecer um mapeamento desses elementos de programa para sua documentação.

Os atributos são definidos por meio da declaração de classes de atributo ([classes de atributo](#)), que podem ter parâmetros posicionais e nomeados ([parâmetros posicionais e nomeados](#)). Os atributos são anexados a entidades em um programa em C# usando especificações de atributo ([especificação de atributo](#)) e podem ser recuperados em tempo de execução como instâncias de atributo ([instâncias de atributo](#)).

## Classes de atributos

Uma classe que deriva da classe abstrata `System.Attribute`, direta ou indiretamente, é uma **classe de atributo\***. A declaração de uma classe de atributo define um novo tipo de `_Attribute*` que pode ser colocado em uma declaração. Por convenção, as classes de atributo são nomeadas com um sufixo de `Attribute`. O uso de um atributo pode incluir ou omitir esse sufixo.

## Uso de atributo

O atributo `AttributeUsage` (o atributo `AttributeUsage`) é usado para descrever como uma classe de atributo pode ser usada.

`AttributeUsage` tem um parâmetro posicional ([parâmetros posicionais e nomeados](#)) que permite que uma classe de atributo especifique os tipos de declarações nas quais ele pode ser usado. O exemplo

C#

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
    ...
}
```

define uma classe de atributo chamada `SimpleAttribute` que pode ser colocada somente em *class\_declaration*s e *interface\_declaration*s. O exemplo

```
C#

[Simple] class Class1 {...}

[Simple] interface Interface1 {...}
```

mostra vários usos do `Simple` atributo. Embora esse atributo seja definido com o nome `SimpleAttribute`, quando esse atributo é usado, o `Attribute` sufixo pode ser omitido, resultando no nome curto `Simple`. Portanto, o exemplo acima é semanticamente equivalente ao seguinte:

```
C#

[SimpleAttribute] class Class1 {...}

[SimpleAttribute] interface Interface1 {...}
```

`AttributeUsage` tem um parâmetro nomeado ([parâmetros posicionais e nomeados](#)) chamado `AllowMultiple`, que indica se o atributo pode ser especificado mais de uma vez para uma determinada entidade. Se `AllowMultiple` para uma classe de atributo for true, essa classe de atributo será uma **classe de atributo \* multi-uso\*** e poderá ser especificada mais de uma vez em uma entidade. Se `AllowMultiple` for uma classe de atributo for false ou não for especificada, essa classe de atributo será uma classe de **atributo \_ de uso único\*** e poderá ser especificada no máximo uma vez em uma entidade.

O exemplo

```
C# 

using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
```

```
public class AuthorAttribute: Attribute
{
    private string name;

    public AuthorAttribute(string name) {
        this.name = name;
    }

    public string Name {
        get { return name; }
    }
}
```

define uma classe de atributo de uso múltiplo chamada `AuthorAttribute`. O exemplo

C#

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
    ...
}
```

mostra uma declaração de classe com dois usos do `Author` atributo.

`AttributeUsage` tem outro parâmetro nomeado chamado `Inherited`, que indica se o atributo, quando especificado em uma classe base, também é herdado por classes que derivam dessa classe base. Se `Inherited` para uma classe de atributo for true, esse atributo será herdado. Se `Inherited` for uma classe de atributo for false, esse atributo não será herdado. Se não for especificado, seu valor padrão será true.

Uma classe `X` de atributo que não tem um `AttributeUsage` atributo anexado a ela, como em

C#

```
using System;

class X: Attribute {...}
```

é equivalente ao seguinte:

C#

```
using System;

[AttributeUsage(
```

```
    AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)
]
class X: Attribute {...}
```

## Parâmetros posicionais e nomeados

As classes de atributo podem ter *\*parâmetros posicionais \_ e \_ parâmetros nomeados \**. Cada Construtor de instância pública para uma classe de atributo define uma sequência válida de parâmetros posicionais para essa classe de atributo. Cada campo de leitura/gravação público não estático e propriedade para uma classe de atributo define um parâmetro nomeado para a classe de atributo.

O exemplo

C#

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) {           // Positional parameter
        ...
    }

    public string Topic {                      // Named parameter
        get {...}
        set {...}
    }

    public string Url {
        get {...}
    }
}
```

define uma classe de atributo chamada `HelpAttribute` que tem um parâmetro positional, `url` e um parâmetro nomeado, `Topic` . Embora seja não estático e público, a propriedade não `Url` define um parâmetro nomeado, pois não é de leitura/gravação.

Essa classe de atributo pode ser usada da seguinte maneira:

C#

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class Class1
{
```

```
...
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
    ...
}
```

## Tipos de parâmetro de atributo

Os tipos de parâmetros posicionais e nomeados para uma classe de atributo são limitados aos **tipos de parâmetro de atributo**, que são:

- Um dos seguintes tipos:,,, `bool` `byte` `char` `double` `float` `int` `long` , `sbyte` ,,,  
`short` `string` `uint` `ulong` `ushort` .
- O tipo `object`.
- O tipo `System.Type`.
- Um tipo de enumeração, desde que ele tenha acessibilidade pública e os tipos em que ele está aninhado (se houver) também têm acessibilidade pública ([especificação de atributo](#)).
- Matrizes unidimensionais dos tipos acima.
- Um argumento de construtor ou campo público que não tem um desses tipos não pode ser usado como um parâmetro posicional ou nomeado em uma especificação de atributo.

## Especificação de atributo

\*A **especificação de atributo** é o aplicativo de um atributo previamente definido para uma declaração. Um atributo é uma parte das informações declarativas adicionais que são especificadas para uma declaração. Os atributos podem ser especificados no escopo global (para especificar atributos no assembly ou módulo que contém) e para `_type_declarator * s` ([declarações de tipo](#)), `class_member_declarator s` (restrições de [parâmetro de tipo](#)), `interface_member_declarator s` ([membros da interface](#)), `struct_member_declarator s` ([membros de struct](#)), `enum_member_declarator s` ([membros de enumeração](#)), `Accessor_declarations` ([acessadores](#)), `event_accessor_declarations` ([eventos de Campo](#)) e `formal_parameter_list s` ([parâmetros de método](#)).

Os atributos são especificados em **seções de atributo**. Uma seção de atributo consiste em um par de colchetes, que envolve uma lista separada por vírgulas de um ou mais atributos. A ordem na qual os atributos são especificados em uma lista como essa, e a

ordem em que as seções anexadas à mesma entidade do programa são organizadas, não é significativa. Por exemplo, as especificações de atributo [A][B] , [B][A] , [A,B] e [B,A] são equivalentes.

antlr

```
global_attributes
: global_attribute_section+
;

global_attribute_section
: '[' global_attribute_target_specifier attribute_list ']'
| '[' global_attribute_target_specifier attribute_list ',' ']'
;

global_attribute_target_specifier
: global_attribute_target ':'
;

global_attribute_target
: 'assembly'
| 'module'
;

attributes
: attribute_section+
;

attribute_section
: '[' attribute_target_specifier? attribute_list ']'
| '[' attribute_target_specifier? attribute_list ',' ']'
;

attribute_target_specifier
: attribute_target ':'
;

attribute_target
: 'field'
| 'event'
| 'method'
| 'param'
| 'property'
| 'return'
| 'type'
;

attribute_list
: attribute (',' attribute)*
;

attribute
: attribute_name attribute_arguments?
;
```

```

;

attribute_name
: type_name
;

attribute_arguments
: '(' positional_argument_list? ')'
| '(' positional_argument_list ',' named_argument_list ')'
| '(' named_argument_list ')'
;

positional_argument_list
: positional_argument (',' positional_argument)*
;

positional_argument
: attribute_argument_expression
;

named_argument_list
: named_argument (',' named_argument)*
;

named_argument
: identifier '=' attribute_argument_expression
;

attribute_argument_expression
: expression
;

```

Um atributo consiste em um *ATTRIBUTE\_NAME* e uma lista opcional de argumentos posicionais e nomeados. Os argumentos posicionais (se houver) precedem os argumentos nomeados. Um argumento posicional consiste em um *attribute\_argument\_expression*; um argumento nomeado consiste em um nome, seguido por um sinal de igual, seguido por um *attribute\_argument\_expression*, que, juntos, são restritos pelas mesmas regras que a atribuição simples. A ordem dos argumentos nomeados não é significativa.

O *ATTRIBUTE\_NAME* identifica uma classe de atributo. Se a forma de *ATTRIBUTE\_NAME* for *type\_name*, esse nome deverá se referir a uma classe de atributo. Caso contrário, ocorrerá um erro de tempo de compilação. O exemplo

C#

```

class Class1 {}

[Class1] class Class2 {}    // Error

```

resulta em um erro de tempo de compilação porque ele tenta usar `Class1` como uma classe de atributo quando `Class1` não é uma classe de atributo.

Determinados contextos permitem a especificação de um atributo em mais de um destino. Um programa pode especificar explicitamente o destino incluindo um *attribute\_target\_specifier*. Quando um atributo é colocado no nível global, um *global\_attribute\_target\_specifier* é necessário. Em todos os outros locais, um padrão razoável é aplicado, mas um *attribute\_target\_specifier* pode ser usado para afirmar ou substituir o padrão em determinados casos ambíguos (ou apenas para afirmar o padrão em casos não ambíguos). Portanto, normalmente, *attribute\_target\_specifier*s pode ser omitido, exceto no nível global. Os contextos potencialmente ambíguos são resolvidos da seguinte maneira:

- Um atributo especificado em escopo global pode ser aplicado ao `assembly` de destino ou ao módulo de destino. Não existe nenhum padrão para esse contexto, portanto, um *attribute\_target\_specifier* sempre é necessário neste contexto. A presença do `assembly attribute_target_specifier` indica que o atributo se aplica ao assembly de destino; a presença do `module attribute_target_specifier` indica que o atributo se aplica ao módulo de destino.
- Um atributo especificado em uma declaração `delegate` pode ser aplicado ao delegado que está sendo declarado ou ao valor de retorno. Na ausência de um *attribute\_target\_specifier*, o atributo se aplica ao delegado. A presença do `type attribute_target_specifier` indica que o atributo se aplica ao delegado; a presença do `return attribute_target_specifier` indica que o atributo se aplica ao valor de retorno.
- Um atributo especificado em uma declaração de método pode ser aplicado ao método que está sendo declarado ou ao valor de retorno. Na ausência de um *attribute\_target\_specifier*, o atributo se aplica ao método. A presença do `method attribute_target_specifier` indica que o atributo se aplica ao método; a presença do `return attribute_target_specifier` indica que o atributo se aplica ao valor de retorno.
- Um atributo especificado em uma declaração de operador pode ser aplicado ao operador que está sendo declarado ou ao valor de retorno. Na ausência de um *attribute\_target\_specifier*, o atributo se aplica ao operador. A presença do `method attribute_target_specifier` indica que o atributo se aplica ao operador; a presença do `return attribute_target_specifier` indica que o atributo se aplica ao valor de retorno.
- Um atributo especificado em uma declaração de evento que omite acessadores de eventos pode se aplicar ao evento que está sendo declarado, ao campo associado (se o evento não for abstrato) ou aos métodos adicionar e remover associados. Na ausência de um *attribute\_target\_specifier*, o atributo se aplica ao evento. A presença do `event attribute_target_specifier` indica que o atributo se aplica ao evento; a presença do `field attribute_target_specifier` indica que o atributo se

aplica ao campo; e a presença do `method attribute_targetSpecifier` indica que o atributo se aplica aos métodos.

- Um atributo especificado em uma Declaração Get do acessador para uma propriedade ou declaração do indexador pode se aplicar ao método associado ou ao valor de retorno. Na ausência de um `attribute_targetSpecifier`, o atributo se aplica ao método. A presença do `method attribute_targetSpecifier` indica que o atributo se aplica ao método; a presença do `return attribute_targetSpecifier` indica que o atributo se aplica ao valor de retorno.
- Um atributo especificado em um acessador set para uma declaração de propriedade ou de indexador pode se aplicar ao método associado ou ao seu parâmetro implícito solitário. Na ausência de um `attribute_targetSpecifier`, o atributo se aplica ao método. A presença do `method attribute_targetSpecifier` indica que o atributo se aplica ao método; a presença do `param attribute_targetSpecifier` indica que o atributo se aplica ao parâmetro; a presença do `return attribute_targetSpecifier` indica que o atributo se aplica ao valor de retorno.
- Um atributo especificado em uma declaração de acessador Add ou remove para uma declaração de evento pode ser aplicado ao método associado ou ao seu parâmetro solitário. Na ausência de um `attribute_targetSpecifier`, o atributo se aplica ao método. A presença do `method attribute_targetSpecifier` indica que o atributo se aplica ao método; a presença do `param attribute_targetSpecifier` indica que o atributo se aplica ao parâmetro; a presença do `return attribute_targetSpecifier` indica que o atributo se aplica ao valor de retorno.

Em outros contextos, a inclusão de um `attribute_targetSpecifier` é permitida, mas desnecessária. Por exemplo, uma declaração de classe pode incluir ou omitir o especificador `type`:

```
C#  
  
[type: Author("Brian Kernighan")]
class Class1 {}  
  
[Author("Dennis Ritchie")]
class Class2 {}
```

É um erro especificar um `attribute_targetSpecifier` inválido. Por exemplo, o especificador `param` não pode ser usado em uma declaração de classe:

```
C#
```

```
[param: Author("Brian Kernighan")]           // Error
class Class1 {}
```

Por convenção, as classes de atributo são nomeadas com um sufixo de `Attribute`. Uma `ATTRIBUTE_NAME` do formulário `type_name` pode incluir ou omitir esse sufixo. Se uma classe de atributo for encontrada com e sem esse sufixo, uma ambiguidade estará presente e ocorrerá um erro de tempo de compilação. Se o `ATTRIBUTE_NAME` for escrito de forma que seu *identificador* mais à direita seja um identificador textual ([identificadores](#)), somente um atributo sem um sufixo será correspondido, permitindo assim que essa ambiguidade seja resolvida. O exemplo

C#

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                         // Error: ambiguity
class Class1 {}

[XAttribute]                 // Refers to XAttribute
class Class2 {}

[@X]                         // Refers to X
class Class3 {}

[@XAttribute]                // Refers to XAttribute
class Class4 {}
```

mostra duas classes de atributo chamadas `X` e `XAttribute`. O atributo `[X]` é ambíguo, pois ele pode se referir a um `X` ou `XAttribute`. O uso de um identificador textual permite que a intenção exata seja especificada nesses casos raros. O atributo `[XAttribute]` não é ambíguo (embora seja uma classe de atributo chamada `XAttributeAttribute`!). Se a declaração para a classe `X` for removida, ambos os atributos referem-se à classe de atributo denominada `XAttribute`, da seguinte maneira:

C#

```
using System;
```

```
[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                                     // Refers to XAttribute
class Class1 {}

[XAttribute]                            // Refers to XAttribute
class Class2 {}

[@X]                                    // Error: no attribute named "X"
class Class3 {}
```

É um erro de tempo de compilação para usar uma classe de atributo de uso único mais de uma vez na mesma entidade. O exemplo

C#

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
    string value;

    public HelpStringAttribute(string value) {
        this.value = value;
    }

    public string Value {
        get {...}
    }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

resulta em um erro de tempo de compilação porque ele tenta usar `HelpString`, que é uma classe de atributo de uso único, mais de uma vez na declaração de `Class1`.

Uma expressão `E` é uma *attribute\_argument\_expression* se todas as seguintes instruções forem verdadeiras:

- O tipo de `E` é um tipo de parâmetro de atributo ([tipos de parâmetro de atributo](#)).
- Em tempo de compilação, o valor de `E` pode ser resolvido para um dos seguintes:
  - Um valor constante.
  - Um objeto `System.Type`.
  - Uma matriz unidimensional de *attribute\_argument\_expression*s.

Por exemplo:

```
C#  
  
using System;  
  
[AttributeUsage(AttributeTargets.Class)]  
public class TestAttribute: Attribute  
{  
    public int P1 {  
        get {...}  
        set {...}  
    }  
  
    public Type P2 {  
        get {...}  
        set {...}  
    }  
  
    public object P3 {  
        get {...}  
        set {...}  
    }  
}  
  
[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]  
class MyClass {}
```

Um *typeof\_expression* (o operador `typeof`) usado como uma expressão de argumento de atributo pode referenciar um tipo não genérico, um tipo construído fechado ou um tipo genérico não associado, mas não pode fazer referência a um tipo aberto. Isso é para garantir que a expressão possa ser resolvida em tempo de compilação.

```
C#  
  
class A: Attribute  
{  
    public A(Type t) {...}  
}  
  
class G<T>  
{  
    [A(typeof(T))] T t;           // Error, open type in attribute  
}  
  
class X  
{  
    [A(typeof(List<int>))] int x;      // Ok, closed constructed type  
    [A(typeof(List<>))] int y;          // Ok, unbound generic type  
}
```

# Instâncias de atributos

Uma *instância de atributo* é uma instância que representa um atributo em tempo de execução. Um atributo é definido com uma classe de atributo, argumentos posicionais e argumentos nomeados. Uma instância de atributo é uma instância da classe de atributo que é inicializada com os argumentos posicionais e nomeados.

A recuperação de uma instância de atributo envolve o processamento em tempo de compilação e em tempo de execução, conforme descrito nas seções a seguir.

## Compilação de um atributo

A compilação de um *atributo* com classe de atributo  $T$ , *positional\_argument\_list*  $P$  e *named\_argument\_list*  $N$ , consiste nas seguintes etapas:

- Siga as etapas de processamento em tempo de compilação para compilar uma *object\_creation\_expression* do formulário `new T(P)`. Essas etapas resultam em um erro de tempo de compilação ou determinam um construtor  $C$  de instância no  $T$  que pode ser invocado em tempo de execução.
- Se  $C$  o não tiver uma acessibilidade pública, ocorrerá um erro de tempo de compilação.
- Para cada *named\_argument*  $Arg$  em  $N$  :
  - Deixe  $Name$  o *identificador* da *named\_argument*  $Arg$  .
  - $Name$  deve identificar um campo ou propriedade pública de leitura/gravação não estática em  $T$  . Se  $T$  não tiver tal campo ou propriedade, ocorrerá um erro de tempo de compilação.
- Mantenha as seguintes informações para instanciação de tempo de execução do atributo: a classe de atributo  $T$  , o construtor de instância  $C$  em  $T$  , o *positional\_argument\_list*  $P$  e o *named\_argument\_list*  $N$  .

## Recuperação de tempo de execução de uma instância de atributo

A compilação de um *atributo* produz uma classe de atributo  $T$  , um construtor de instância  $C$  em  $T$  , um *positional\_argument\_list*  $P$  e um *named\_argument\_list*  $N$  . Dadas essas informações, uma instância de atributo pode ser recuperada em tempo de execução usando as seguintes etapas:

- Siga as etapas de processamento em tempo de execução para executar uma *object\_creation\_expression* do formulário `new T(P)` , usando o construtor de

instância `C` como determinado em tempo de compilação. Essas etapas resultam em uma exceção ou produzem uma instância `O` do `T`.

- Para cada `named_argument Arg` em `N`, em ordem:
  - Deixe `Name` o *identificador* da `named_argument Arg`. Se `Name` o não identificar um campo ou uma propriedade de leitura/gravação pública não estática em `O`, uma exceção será lançada.
  - Vamos `Value` ser o resultado da avaliação da `attribute_argument_expression` de `Arg`.
  - Se `Name` o identificar um campo em `O`, defina esse campo como `Value`.
  - Caso contrário, `Name` identifica uma propriedade em `O`. Defina esta propriedade como `Value`.
  - O resultado é `O` uma instância da classe de atributo `T` que foi inicializada com o `positional_argument_list P` e o `named_argument_list N`.

## Atributos reservados

Um pequeno número de atributos afeta o idioma de alguma forma. Esses atributos incluem:

- `System.AttributeUsageAttribute` ([O atributo AttributeUsage](#)), que é usado para descrever as maneiras em que uma classe de atributo pode ser usada.
- `System.Diagnostics.ConditionalAttribute` ([O atributo Conditional](#)), que é usado para definir métodos condicionais.
- `System.ObsoleteAttribute` ([O atributo obsoleto](#)), que é usado para marcar um membro como obsoleto.
- `System.Runtime.CompilerServices.CallerLineNumberAttribute` `` `System.Runtime.CompilerServices.CallerFilePathAttribute`   
`System.Runtime.CompilerServices.CallerMemberNameAttribute` (atributos de [informações do chamador](#)), que são usados para fornecer informações sobre o contexto de chamada para parâmetros opcionais.

## O atributo AttributeUsage

O atributo `AttributeUsage` é usado para descrever a maneira como a classe de atributo pode ser usada.

Uma classe que é decorada com o `AttributeUsage` atributo deve derivar de `System.Attribute`, direta ou indiretamente. Caso contrário, ocorrerá um erro de tempo de compilação.

C#

```
namespace System
{
    [AttributeUsage(AttributeTargets.Class)]
    public class AttributeUsageAttribute: Attribute
    {
        public AttributeUsageAttribute(AttributeTargets validOn) {...}
        public virtual bool AllowMultiple { get {...} set {...} }
        public virtual bool Inherited { get {...} set {...} }
        public virtual AttributeTargets ValidOn { get {...} }
    }

    public enum AttributeTargets
    {
        Assembly      = 0x0001,
        Module        = 0x0002,
        Class         = 0x0004,
        Struct         = 0x0008,
        Enum           = 0x0010,
        Constructor   = 0x0020,
        Method         = 0x0040,
        Property       = 0x0080,
        Field          = 0x0100,
        Event          = 0x0200,
        Interface     = 0x0400,
        Parameter      = 0x0800,
        Delegate       = 0x1000,
        ReturnValue    = 0x2000,

        All = Assembly | Module | Class | Struct | Enum | Constructor |
              Method | Property | Field | Event | Interface | Parameter |
              Delegate | ReturnValue
    }
}
```

## O atributo Conditional

O atributo `Conditional` habilita a definição de \*métodos condicionais \_ e \_ classes de atributo condicional\*.

C#

```
namespace System.Diagnostics
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
    AllowMultiple = true)]
    public class ConditionalAttribute: Attribute
    {
        public ConditionalAttribute(string conditionString) {...}
        public string ConditionString { get {...} }
```

```
    }  
}
```

## Métodos condicionais

Um método decorado com o `Conditional` atributo é um método condicional. O `Conditional` atributo indica uma condição testando um símbolo de compilação condicional. Chamadas para um método condicional são incluídas ou omitidas dependendo se esse símbolo é definido no ponto da chamada. Se o símbolo for definido, a chamada será incluída; caso contrário, a chamada (incluindo a avaliação do receptor e dos parâmetros da chamada) é omitida.

Um método condicional está sujeito às seguintes restrições:

- O método condicional deve ser um método em um `class_declaration` ou `struct_declaration`. Ocorrerá um erro de tempo de compilação se o `Conditional` atributo for especificado em um método em uma declaração de interface.
- O método condicional deve ter um tipo de retorno de `void`.
- O método condicional não deve ser marcado com o `override` modificador. Um método condicional pode ser marcado com o `virtual` modificador, no entanto. As substituições desse método são implicitamente condicionais e não devem ser marcadas explicitamente com um `Conditional` atributo.
- O método condicional não deve ser uma implementação de um método de interface. Caso contrário, ocorrerá um erro de tempo de compilação.

Além disso, ocorrerá um erro de tempo de compilação se um método condicional for usado em um `delegate_creation_expression`. O exemplo

C#

```
#define DEBUG  
  
using System;  
using System.Diagnostics;  
  
class Class1  
{  
    [Conditional("DEBUG")]  
    public static void M() {  
        Console.WriteLine("Executed Class1.M");  
    }  
}  
  
class Class2  
{
```

```
public static void Test() {
    Class1.M();
}
```

declara `Class1.M` como um método condicional. `Class2.Test` o método do chama esse método. Como o símbolo de compilação condicional `DEBUG` é definido, se `Class2.Test` for chamado, ele chamará `M`. Se o símbolo `DEBUG` não tivesse sido definido, então `Class2.Test` não chamaría `Class1.M`.

É importante observar que a inclusão ou a exclusão de uma chamada para um método condicional é controlada pelos símbolos de compilação condicional no ponto da chamada. No exemplo

Arquivo `class1.cs` :

```
C#  
  
using System.Diagnostics;  
  
class Class1  
{  
    [Conditional("DEBUG")]
    public static void F() {
        Console.WriteLine("Executed Class1.F");
    }
}
```

Arquivo `class2.cs` :

```
C#  
  
#define DEBUG  
  
class Class2
{
    public static void G() {
        Class1.F();           // F is called
    }
}
```

Arquivo `class3.cs` :

```
C#  
  
#undef DEBUG
```

```
class Class3
{
    public static void H() {
        Class1.F(); // F is not called
    }
}
```

as classes `Class2` e `Class3` cada uma contém chamadas para o método condicional `Class1.F`, que é condicional com base em se `DEBUG` está definido ou não. Como esse símbolo é definido no contexto de, `Class2` mas não `Class3`, a chamada para `F` no `Class2` é incluída, enquanto a chamada para `F` in `Class3` é omitida.

O uso de métodos condicionais em uma cadeia de herança pode ser confuso. As chamadas feitas a um método condicional `base`, do formulário `base.M`, estão sujeitas às regras de chamada de método condicional normais. No exemplo

Arquivo `class1.cs` :

```
C#
using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public virtual void M() {
        Console.WriteLine("Class1.M executed");
    }
}
```

Arquivo `class2.cs` :

```
C#
using System;

class Class2: Class1
{
    public override void M() {
        Console.WriteLine("Class2.M executed");
        base.M(); // base.M is not called!
    }
}
```

Arquivo `class3.cs` :

C#

```
#define DEBUG

using System;

class Class3
{
    public static void Test() {
        Class2 c = new Class2();
        c.M();                                // M is called
    }
}
```

`Class2` inclui uma chamada para o `M` definido em sua classe base. Essa chamada é omitida porque o método base é condicional com base na presença do símbolo `DEBUG`, que é indefinido. Portanto, o método grava somente no console "`Class2.M executed`". O uso criterioso de *pp\_declaration*s pode eliminar esses problemas.

## Classes de atributo condicional

Uma classe de atributo ([classes de atributo](#)) decorada com um ou mais `Conditional` atributos é uma classe de **atributo condicional**. Uma classe de atributo condicional é, portanto, associada aos símbolos de compilação condicional declarados em seus `Conditional` atributos. Neste exemplo:

C#

```
using System;
using System.Diagnostics;
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

declara `TestAttribute` como uma classe de atributo condicional associada aos símbolos de compilações condicionais `ALPHA` e `BETA`.

As especificações de atributo ([especificação de atributo](#)) de um atributo condicional são incluídas se um ou mais de seus símbolos de compilação condicional associados estiverem definidos no ponto de especificação, caso contrário, a especificação de atributo será omitida.

É importante observar que a inclusão ou exclusão de uma especificação de atributo de uma classe de atributo condicional é controlada pelos símbolos de compilação condicional no ponto da especificação. No exemplo

Arquivo `test.cs` :

```
C#  
  
using System;  
using System.Diagnostics;  
  
[Conditional("DEBUG")]  
  
public class TestAttribute : Attribute {}
```

Arquivo `class1.cs` :

```
C#  
  
#define DEBUG  
  
[Test] // TestAttribute is specified  
  
class Class1 {}
```

Arquivo `class2.cs` :

```
C#  
  
#undef DEBUG  
  
[Test] // TestAttribute is not specified  
  
class Class2 {}
```

as classes `Class1` e `Class2` são cada uma decorada com atributo `Test`, que é condicional com base em se `DEBUG` está definida ou não. Como esse símbolo é definido no contexto de, `Class1` mas não `Class2`, a especificação do `Test` atributo em `Class1` é incluída, enquanto a especificação do `Test` atributo em `Class2` é omitida.

## O atributo obsoleto

O atributo `Obsolete` é usado para marcar tipos e membros de tipos que não devem mais ser usados.

```
C#  
  
namespace System  
{  
    [AttributeUsage(
```

```

        AttributeTargets.Class |
        AttributeTargets.Struct |
        AttributeTargets.Enum |
        AttributeTargets.Interface |
        AttributeTargets.Delegate |
        AttributeTargets.Method |
        AttributeTargets.Constructor |
        AttributeTargets.Property |
        AttributeTargets.Field |
        AttributeTargets.Event,
        Inherited = false)
    ]
public class ObsoleteAttribute: Attribute
{
    public ObsoleteAttribute() {...}
    public ObsoleteAttribute(string message) {...}
    public ObsoleteAttribute(string message, bool error) {...}
    public string Message { get {...} }
    public bool IsError { get {...} }
}
}

```

Se um programa usa um tipo ou membro que é decorado com o `Obsolete` atributo, o compilador emite um aviso ou um erro. Especificamente, o compilador emitirá um aviso se nenhum parâmetro de erro for fornecido ou se o parâmetro de erro for fornecido e tiver o valor `false`. O compilador emitirá um erro se o parâmetro de erro for especificado e tiver o valor `true`.

No exemplo

C#

```

[Obsolete("This class is obsolete; use class B instead")]
class A
{
    public void F() {}
}

class B
{
    public void F() {}
}

class Test
{
    static void Main() {
        A a = new A();           // Warning
        a.F();
    }
}

```

a classe A é decorada com o `Obsolete` atributo. Cada uso de A em Main resulta em um aviso que inclui a mensagem especificada, "esta classe é obsoleta; em vez disso, use a classe B. "

## Atributos de informações do chamador

Para fins como registro em log e relatórios, às vezes é útil para um membro de função obter determinadas informações de tempo de compilação sobre o código de chamada. Os atributos de informações do chamador fornecem uma maneira de passar essas informações de forma transparente.

Quando um parâmetro opcional é anotado com um dos atributos de informações do chamador, omitir o argumento correspondente em uma chamada não faz necessariamente com que o valor do parâmetro padrão seja substituído. Em vez disso, se as informações especificadas sobre o contexto de chamada estiverem disponíveis, essas informações serão passadas como o valor do argumento.

Por exemplo:

```
C#  
  
using System.Runtime.CompilerServices  
  
...  
  
public void Log(  
    [CallerLineNumber] int line = -1,  
    [CallerFilePath] string path = null,  
    [CallerMemberName] string name = null  
)  
{  
    Console.WriteLine((line < 0) ? "No line" : "Line " + line);  
    Console.WriteLine((path == null) ? "No file path" : path);  
    Console.WriteLine((name == null) ? "No member name" : name);  
}
```

Uma chamada para sem `Log()` argumentos imprimiria o número de linha e o caminho do arquivo da chamada, bem como o nome do membro no qual a chamada ocorreu.

Atributos de informações do chamador podem ocorrer em parâmetros opcionais em qualquer lugar, incluindo declarações de delegado. No entanto, os atributos específicos de informações do chamador têm restrições sobre os tipos dos parâmetros que eles podem atribuir, de modo que sempre haverá uma conversão implícita de um valor substituído para o tipo de parâmetro.

É um erro ter o mesmo atributo de informações do chamador em um parâmetro de definição e implementação de parte de uma declaração de método parcial. Somente os atributos de informações do chamador na parte de definição são aplicados, enquanto os atributos de informações do chamador que ocorrem somente na parte de implementação são ignorados.

As informações do chamador não afetam a resolução de sobrecarga. Como os parâmetros opcionais atribuídos ainda são omitidos do código-fonte do chamador, a resolução de sobrecarga ignora esses parâmetros da mesma maneira que ignora outros parâmetros opcionais omitidos ([resolução de sobrecarga](#)).

As informações do chamador só são substituídas quando uma função é invocada explicitamente no código-fonte. Invocações implícitas, como chamadas de Construtor pai implícitas, não têm um local de origem e não substituirão as informações do chamador. Além disso, as chamadas que são vinculadas dinamicamente não substituirão as informações do chamador. Quando um parâmetro de informações de chamador atribuído é omitido nesses casos, o valor padrão especificado do parâmetro é usado em vez disso.

Uma exceção é Query-Expressions. Essas são consideradas expansões sintáticas e, se as chamadas forem expandidas para omitir parâmetros opcionais com atributos de informações do chamador, as informações do chamador serão substituídas. O local usado é o local da cláusula de consulta da qual a chamada foi gerada.

Se mais de um atributo de informações do chamador for especificado em um determinado parâmetro, eles serão preferidos na seguinte ordem: `CallerLineNumber` , `CallerFilePath` , `CallerMemberName` .

## O atributo `CallerLineNumber`

O `System.Runtime.CompilerServices.CallerLineNumberAttribute` é permitido em parâmetros opcionais quando há uma conversão implícita padrão ([conversões implícitas padrão](#)) do valor constante `int.MaxValue` para o tipo do parâmetro. Isso garante que qualquer número de linha não negativo até esse valor possa ser passado sem erro.

Se uma invocação de função de um local no código-fonte omitir um parâmetro opcional com o `CallerLineNumberAttribute` , um literal numérico representando o número de linha desse local será usado como um argumento para a invocação em vez do valor de parâmetro padrão.

Se a invocação abrange várias linhas, a linha escolhida será dependente de implementação.

Observe que o número da linha pode ser afetado por `#line` Diretivas ([diretivas de linha](#)).

## O atributo CallerFilePath

O `System.Runtime.CompilerServices.CallerFilePathAttribute` é permitido em parâmetros opcionais quando há uma conversão implícita padrão ([conversões implícitas padrão](#)) de `string` para o tipo do parâmetro.

Se uma invocação de função de um local no código-fonte omitir um parâmetro opcional com o `CallerFilePathAttribute`, um literal de cadeia de caracteres representando o caminho do arquivo desse local será usado como um argumento para a invocação em vez do valor de parâmetro padrão.

O formato do caminho do arquivo é dependente de implementação.

Observe que o caminho do arquivo pode ser afetado por `#line` Diretivas ([diretivas de linha](#)).

## O atributo CallerMemberName

O `System.Runtime.CompilerServices.CallerMemberNameAttribute` é permitido em parâmetros opcionais quando há uma conversão implícita padrão ([conversões implícitas padrão](#)) de `string` para o tipo do parâmetro.

Se uma invocação de função de um local dentro do corpo de um membro de função ou dentro de um atributo aplicado ao próprio membro da função ou seu tipo de retorno, parâmetros ou parâmetros de tipo no código-fonte omitir um parâmetro opcional com o `CallerMemberNameAttribute`, um literal de cadeia de caracteres representando o nome desse membro será usado como um argumento para a invocação em vez do valor de parâmetro padrão.

Para invocações que ocorrem em métodos genéricos, somente o nome do método em si é usado, sem a lista de parâmetros de tipo.

Para invocações que ocorrem em implementações de membro de interface explícitas, somente o nome do método em si é usado, sem a qualificação da interface anterior.

Para invocações que ocorrem em acessadores de propriedade ou evento, o nome de membro usado é o próprio evento ou propriedade.

Para invocações que ocorrem dentro de acessadores do indexador, o nome do membro usado é fornecido por um `IndexerNameAttribute` ([o atributo IndexerName](#)) no membro

do indexador, se presente, ou o nome padrão `Item` também.

Para invocações que ocorrem em declarações de construtores de instância, construtores estáticos, destruidores e operadores, o nome de membro usado é dependente de implementação.

## Atributos para interoperabilidade

Observação: Esta seção é aplicável somente à implementação de Microsoft .NET do C#.

### Interoperabilidade com componentes COM e Win32

O tempo de execução do .NET fornece um grande número de atributos que permitem que programas em C# interoperem com componentes escritos usando DLLs COM e Win32. Por exemplo, o `DllImport` atributo pode ser usado em um `static extern` método para indicar que a implementação do método deve ser encontrada em uma DLL Win32. Esses atributos são encontrados no `System.Runtime.InteropServices` namespace e a documentação detalhada para esses atributos é encontrada na documentação do tempo de execução do .net.

### Interoperabilidade com outras linguagens .NET

#### O atributo `IndexerName`

Os indexadores são implementados no .NET usando propriedades indexadas e têm um nome nos metadados do .NET. Se nenhum `IndexerName` atributo estiver presente para um indexador, o nome `Item` será usado por padrão. O `IndexerName` atributo permite que um desenvolvedor substitua esse padrão e especifique um nome diferente.

C#

```
namespace System.Runtime.CompilerServices.CSharp
{
    [AttributeUsage(AttributeTargets.Property)]
    public class IndexerNameAttribute: Attribute
    {
        public IndexerNameAttribute(string indexerName) {...}
        public string Value { get {...} }
    }
}
```

# Código não seguro

Artigo • 16/09/2021

A linguagem C# principal, conforme definido nos capítulos anteriores, difere notavelmente de C e C++ em sua omissão de ponteiros como um tipo de dados. Em vez disso, o C# fornece referências e a capacidade de criar objetos gerenciados por um coletor de lixo. Esse design, juntamente com outros recursos, torna o C# uma linguagem muito mais segura do que C ou C++. Na linguagem C# principal, simplesmente não é possível ter uma variável não inicializada, um ponteiro "pendente" ou uma expressão que indexe uma matriz além de seus limites. Categorias inteiras de bugs que, rotineiramente, os programas C e C++ são eliminados.

Embora praticamente todas as construções de tipo de ponteiro em C ou C++ tenham um tipo de referência equivalente em C#, no entanto, há situações em que o acesso a tipos de ponteiro se torna uma necessidade. Por exemplo, a interface com o sistema operacional subjacente, o acesso a um dispositivo mapeado por memória ou a implementação de um algoritmo de tempo crítico pode não ser possível ou prática sem acesso a ponteiros. Para atender a essa necessidade, o C# fornece a capacidade de escrever *código não seguro*.

Em código não seguro, é possível declarar e operar em ponteiros, executar conversões entre ponteiros e tipos inteiros, para obter o endereço de variáveis e assim por diante. De certa forma, escrever código inseguro é muito parecido com escrever código C em um programa em C#.

O código não seguro é, de fato, um recurso "seguro" da perspectiva de desenvolvedores e usuários. O código não seguro deve ser claramente marcado com o modificador `unsafe`, de modo que os desenvolvedores não podem possivelmente usar recursos não seguros acidentalmente, e o mecanismo de execução funciona para garantir que o código não seguro não possa ser executado em um ambiente não confiável.

## Contextos não seguros

Os recursos não seguros do C# estão disponíveis apenas em contextos não seguros. Um contexto não seguro é introduzido por meio da inclusão de um `unsafe` modificador na declaração de um tipo ou membro, ou ao empregar um `unsafe_statement`:

- Uma declaração de Class, struct, interface ou delegate pode incluir um `unsafe` modificador; nesse caso, toda a extensão textual dessa declaração de tipo

(incluindo o corpo da classe, struct ou interface) é considerada um contexto não seguro.

- Uma declaração de um campo, método, propriedade, evento, indexador, operador, Construtor de instância, destruidor ou construtor estático pode incluir um `unsafe` modificador; nesse caso, toda a extensão textual dessa declaração de membro é considerada um contexto não seguro.
- Um *unsafe\_statement* permite o uso de um contexto sem segurança dentro de um *bloco*. Toda a extensão textual do *bloco* associado é considerada um contexto não seguro.

As produções gramaticais associadas são mostradas abaixo.

antlr

```
class_modifier_unsafe
: 'unsafe'
;

struct_modifier_unsafe
: 'unsafe'
;

interface_modifier_unsafe
: 'unsafe'
;

delegate_modifier_unsafe
: 'unsafe'
;

field_modifier_unsafe
: 'unsafe'
;

method_modifier_unsafe
: 'unsafe'
;

property_modifier_unsafe
: 'unsafe'
;

event_modifier_unsafe
: 'unsafe'
;

indexer_modifier_unsafe
: 'unsafe'
;

operator_modifier_unsafe
```

```

        : 'unsafe'
;

constructor_modifier_unsafe
: 'unsafe'
;

destructor_declaration_unsafe
: attributes? 'extern'? 'unsafe'? '~' identifier '(' ')' destructor_body
| attributes? 'unsafe'? 'extern'? '~' identifier '(' ')' destructor_body
;

static_constructor_modifiers_unsafe
: 'extern'? 'unsafe'? 'static'
| 'unsafe'? 'extern'? 'static'
| 'extern'? 'static' 'unsafe'?
| 'unsafe'? 'static' 'extern'?
| 'static' 'extern'? 'unsafe'?
| 'static' 'unsafe'? 'extern'?
;
;

embedded_statement_unsafe
: unsafe_statement
| fixed_statement
;
;

unsafe_statement
: 'unsafe' block
;
;
```

No exemplo

C#

```

public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```

O `unsafe` modificador especificado na declaração `struct` faz com que toda a extensão textual da declaração `struct` se torne um contexto não seguro. Portanto, é possível declarar os `Left` `Right` campos e como sendo de um tipo de ponteiro. O exemplo acima também poderia ser escrito

C#

```

public struct Node
{
```

```
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

Aqui, os `unsafe` modificadores nas declarações de campo fazem com que essas declarações sejam consideradas contextos não seguros.

Além de estabelecer um contexto não seguro, permitindo, portanto, o uso de tipos de ponteiro, o `unsafe` modificador não tem nenhum efeito sobre um tipo ou um membro. No exemplo

```
C#  
  
public class A  
{  
    public unsafe virtual void F() {  
        char* p;  
        ...  
    }  
}  
  
public class B: A  
{  
    public override void F() {  
        base.F();  
        ...  
    }  
}
```

O `unsafe` modificador no `F` método em `A` simplesmente faz com que a extensão textual `F` se torne um contexto não seguro no qual os recursos não seguros do idioma podem ser usados. Na substituição de `F` em, não há `B` necessidade de especificar novamente o `unsafe` modificador, a menos que, é claro, o `F` método em `B` si precise de acesso a recursos não seguros.

A situação é um pouco diferente quando um tipo de ponteiro faz parte da assinatura do método

```
C#  
  
public unsafe class A  
{  
    public virtual void F(char* p) {...}  
}  
  
public class B: A  
{
```

```
    public unsafe override void F(char* p) {...}  
}
```

Aqui, como `F` a assinatura do inclui um tipo de ponteiro, ela só pode ser gravada em um contexto sem segurança. No entanto, o contexto não seguro pode ser introduzido, tornando a classe inteira insegura, como é o caso no `A`, ou incluindo um `unsafe` modificador na declaração do método, como é o caso no `B`.

## Tipos de ponteiro

Em um contexto não seguro, um *tipo* (*tipos*) pode ser um *pointer\_type*, bem como um *value\_type* ou um *reference\_type*. No entanto, um *pointer\_type* também pode ser usado em uma `typeof` expressão ([expressões de criação de objeto anônimo](#)) fora de um contexto sem segurança, pois tal uso não é seguro.

```
antlr  
  
type_unsafe  
: pointer_type  
;
```

Uma *pointer\_type* é escrita como uma *unmanaged\_type* ou a palavra-chave `void`, seguida por um `*` token:

```
antlr  
  
pointer_type  
: unmanaged_type '*'  
| 'void' '*'  
;  
  
unmanaged_type  
: type  
;
```

O tipo especificado antes do `*` em um tipo de ponteiro é chamado de ***tipo Referent*** do tipo de ponteiro. Representa o tipo da variável para a qual um valor do tipo de ponteiro aponta.

Ao contrário das referências (valores de tipos de referência), os ponteiros não são rastreados pelo coletor de lixo – o coletor de lixo não tem conhecimento de ponteiros e dos dados para os quais eles apontam. Por esse motivo, um ponteiro não tem

permissão para apontar para uma referência ou para uma struct que contém referências, e o tipo Referent de um ponteiro deve ser um *unmanaged\_type*.

Um *unmanaged\_type* é qualquer tipo que não seja um tipo *reference\_type* ou construído e não contenha *reference\_type* ou campos de tipo construído em qualquer nível de aninhamento. Em outras palavras, um *unmanaged\_type* é um dos seguintes:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` ou `bool`.
- Qualquer *enum\_type*.
- Qualquer *pointer\_type*.
- Qualquer *struct\_type* definida pelo usuário que não seja um tipo construído e contém campos somente *unmanaged\_type*s.

A regra intuitiva para a combinação de ponteiros e referências é que referents de referências (objetos) têm permissão para conter ponteiros, mas referents de ponteiros não tem permissão para conter referências.

Alguns exemplos de tipos de ponteiro são fornecidos na tabela a seguir:

Exemplo	Descrição
<code>byte*</code>	Ponteiro para <code>byte</code>
<code>char*</code>	Ponteiro para <code>char</code>
<code>int**</code>	Ponteiro para ponteiro para <code>int</code>
<code>int*[]</code>	Matriz unidimensional de ponteiros para <code>int</code>
<code>void*</code>	Ponteiro para tipo desconhecido

Para uma determinada implementação, todos os tipos de ponteiro devem ter o mesmo tamanho e representação.

Ao contrário de C e C++, quando vários ponteiros são declarados na mesma declaração, em C#, o `*` é escrito junto com o tipo subjacente apenas, não como um pontuador de prefixo em cada nome de ponteiro. Por exemplo,

C#

```
int* pi, pj; // NOT as int *pi, *pj;
```

O valor de um ponteiro com tipo `T*` representa o endereço de uma variável do tipo `T`. O operador de indireção de ponteiro `*` ([indireção de ponteiro](#)) pode ser usado para

acessar essa variável. Por exemplo, dada uma variável `P` do tipo `int*`, a expressão `*P` denota a `int` variável encontrada no endereço contido em `P`.

Como uma referência de objeto, um ponteiro pode ser `null`. A aplicação do operador de indireção a um `null` ponteiro resulta em comportamento definido pela implementação. Um ponteiro com valor `null` é representado por todos os bits-zero.

O `void*` tipo representa um ponteiro para um tipo desconhecido. Como o tipo Referent é desconhecido, o operador de indireção não pode ser aplicado a um ponteiro do tipo `void*`, nem qualquer aritmética pode ser executada nesse ponteiro. No entanto, um ponteiro do tipo `void*` pode ser convertido em qualquer outro tipo de ponteiro (e vice-versa).

Os tipos de ponteiro são uma categoria separada de tipos. Diferentemente dos tipos de referência e tipos de valor, os tipos de ponteiro não herdam de `object` e nenhuma conversões existem entre os tipos de ponteiro e `object`. Em particular, boxing e unboxing ([boxing e unboxing](#)) não têm suporte para ponteiros. No entanto, as conversões são permitidas entre diferentes tipos de ponteiro e entre os tipos de ponteiro e os tipos integrais. Isso é descrito em [conversões de ponteiro](#).

Um `pointer_type` não pode ser usado como um argumento de tipo ([tipos construídos](#)), e a inferência de tipos ([inferência de tipos](#)) falha em chamadas de método genérico que teriam inferido um argumento de tipo para ser um tipo de ponteiro.

Um `pointer_type` pode ser usado como o tipo de um campo volátil ([campos voláteis](#)).

Embora os ponteiros possam ser passados como `ref` `out` parâmetros ou, fazer isso pode causar um comportamento indefinido, já que o ponteiro pode ser bem definido para apontar para uma variável local que não existe mais quando o método chamado retorna, ou o objeto fixo ao qual ele costumava apontar, não é mais fixo. Por exemplo:

C#

```
using System;

class Test
{
    static int value = 20;

    unsafe static void F(out int* pi1, ref int* pi2) {
        int i = 10;
        pi1 = &i;

        fixed (int* pj = &value) {
            // ...
            pi2 = pj;
        }
    }
}
```

```

    }

    static void Main() {
        int i = 10;
        unsafe {
            int* px1;
            int* px2 = &i;

            F(out px1, ref px2);

            Console.WriteLine("*px1 = {0}, *px2 = {1}",
                *px1, *px2);      // undefined behavior
        }
    }
}

```

Um método pode retornar um valor de algum tipo, e esse tipo pode ser um ponteiro. Por exemplo, quando um ponteiro é fornecido para uma sequência contígua de `int`s, a contagem de elementos dessa sequência e algum outro `int` valor, o método a seguir retorna o endereço desse valor nessa sequência, se ocorrer uma correspondência; caso contrário, retornará `null`:

C#

```

unsafe static int* Find(int* pi, int size, int value) {
    for (int i = 0; i < size; ++i) {
        if (*pi == value)
            return pi;
        ++pi;
    }
    return null;
}

```

Em um contexto não seguro, várias construções estão disponíveis para operar em ponteiros:

- O `*` operador pode ser usado para executar o direcionamento de ponteiro ([indireção de ponteiro](#)).
- O `->` operador pode ser usado para acessar um membro de uma struct por meio de um ponteiro ([acesso de membro de ponteiro](#)).
- O `[]` operador pode ser usado para indexar um ponteiro ([acesso de elemento de ponteiro](#)).
- O `&` operador pode ser usado para obter o endereço de uma variável ([o operador address-of](#)).

- Os `++` `--` operadores e podem ser usados para incrementar e decrementar ponteiros ([incremento de ponteiro e decréscimo](#)).
- Os `+` `-` operadores e podem ser usados para executar aritmética de ponteiro ([aritmética de ponteiro](#)).
- Os `==` operadores, `!=`, `<`, `,`, `>`, `<=` e `>=` podem ser usados para comparar ponteiros ([comparação de ponteiros](#)).
- O `stackalloc` operador pode ser usado para alocar memória da pilha de chamadas ([buffers de tamanho fixo](#)).
- A `fixed` instrução pode ser usada para corrigir temporariamente uma variável para que seu endereço possa ser obtido ([a instrução Fixed](#)).

## Variáveis fixas e móveis

O operador address-of ([o operador address-of](#)) e a `fixed` instrução ([a instrução Fixed](#)) dividem variáveis em duas categorias: *\*variáveis fixas \_ e \_ variáveis móveis \**.

As variáveis fixas residem em locais de armazenamento que não são afetados pela operação do coletor de lixo. (Exemplos de variáveis fixas incluem variáveis locais, parâmetros de valor e variáveis criadas pela desreferenciação de ponteiros.) Por outro lado, as variáveis móveis residem em locais de armazenamento que estão sujeitos a realocação ou descarte pelo coletor de lixo. (Exemplos de variáveis móveis incluem campos em objetos e elementos de matrizes.)

O `&` operador ([o operador address-of](#)) permite que o endereço de uma variável fixa seja obtido sem restrições. No entanto, como uma variável móvel está sujeita a realocação ou alienação pelo coletor de lixo, o endereço de uma variável móvel só pode ser obtido usando uma `fixed` instrução ([a instrução fixa](#)) e esse endereço permanece válido somente pela duração dessa `fixed` instrução.

Em termos precisos, uma variável fixa é uma das seguintes:

- Uma variável resultante de um *Simple\_name* ([nomes simples](#)) que se refere a uma variável local ou a um parâmetro de valor, a menos que a variável seja capturada por uma função anônima.
- Uma variável resultante de uma *member\_access* ([acesso de membro](#)) do formulário `v.I`, em que `v` é uma variável fixa de um *struct\_type*.
- Uma variável resultante de uma *pointer\_indirection\_expression* ([indireção de ponteiro](#)) do formulário `*P`, uma *pointer\_member\_access* (acesso de [membro de ponteiro](#)) do formulário `P->I` ou uma *pointer\_element\_access* (acesso de [elemento de ponteiro](#)) do formulário `P[E]`.

Todas as outras variáveis são classificadas como variáveis móveis.

Observe que um campo estático é classificado como uma variável móvel. Observe também que um `ref` ou `out` parâmetro ou é classificado como uma variável móvel, mesmo que o argumento fornecido para o parâmetro seja uma variável fixa. Por fim, observe que uma variável produzida por desreferenciar um ponteiro é sempre classificada como uma variável fixa.

## Conversões de ponteiro

Em um contexto sem segurança, o conjunto de conversões implícitas disponíveis ([conversões implícitas](#)) é estendido para incluir as seguintes conversões de ponteiro implícitas:

- De qualquer *pointer\_type* para o tipo `void*`.
- Do `null` literal para qualquer *pointer\_type*.

Além disso, em um contexto não seguro, o conjunto de conversões explícitas disponíveis ([conversões explícitas](#)) é estendido para incluir as seguintes conversões de ponteiro explícitas:

- De qualquer *pointer\_type* a qualquer outro *pointer\_type*.
- De `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` ou `ulong` para qualquer *pointer\_type*.
- De qualquer *pointer\_type* para `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` ou `ulong`.

Por fim, em um contexto não seguro, o conjunto de conversões implícitas padrão ([conversões implícitas padrão](#)) inclui a seguinte conversão de ponteiro:

- De qualquer *pointer\_type* para o tipo `void*`.

As conversões entre dois tipos de ponteiro nunca alteram o valor real do ponteiro. Em outras palavras, uma conversão de um tipo de ponteiro para outro não tem nenhum efeito sobre o endereço subjacente fornecido pelo ponteiro.

Quando um tipo de ponteiro é convertido em outro, se o ponteiro resultante não estiver alinhado corretamente para o tipo apontado, o comportamento será indefinido se o resultado for de referência. Em geral, o conceito "alinhado corretamente" é transitivo: se um ponteiro para o tipo `A` estiver alinhado corretamente para um ponteiro para tipo `B`, que, por sua vez, está alinhado corretamente para que um ponteiro seja digitado `C`, um ponteiro para tipo `A` é alinhado corretamente para que um ponteiro seja digitado `C`.

Considere o seguinte caso em que uma variável com um tipo é acessada por meio de um ponteiro para um tipo diferente:

```
C#  
  
char c = 'A';  
char* pc = &c;  
void* pv = pc;  
int* pi = (int*)pv;  
int i = *pi;           // undefined  
*pi = 123456;         // undefined
```

Quando um tipo de ponteiro é convertido em um ponteiro para byte, o resultado aponta para o byte de endereçamento mais baixo da variável. Incrementos sucessivos do resultado, até o tamanho da variável, geram ponteiros para os bytes restantes dessa variável. Por exemplo, o método a seguir exibe cada um dos oito bytes em um duplo como um valor hexadecimal:

```
C#  
  
using System;  
  
class Test  
{  
    unsafe static void Main()  
    {  
        double d = 123.456e23;  
        unsafe {  
            byte* pb = (byte*)&d;  
            for (int i = 0; i < sizeof(double); ++i)  
                Console.Write("{0:X2} ", *pb++);  
            Console.WriteLine();  
        }  
    }  
}
```

É claro que a saída produzida depende da ordenação.

Os mapeamentos entre ponteiros e inteiros são definidos pela implementação. No entanto, em arquiteturas de CPU de 32 \* e 64 bits com um espaço de endereço linear, conversões de ponteiros para ou de tipos integrais normalmente se comportam exatamente como conversões de `uint` `ulong` valores ou, respectivamente, de ou para esses tipos integral.

## Matrizes de ponteiro

Em um contexto não seguro, matrizes de ponteiros podem ser construídas. Somente algumas das conversões que se aplicam a outros tipos de matriz são permitidas em matrizes de ponteiro:

- A conversão de referência implícita ([conversões de referência implícitas](#)) de qualquer *array\_type* para `System.Array` e as interfaces que ele implementa também se aplica a matrizes de ponteiro. No entanto, qualquer tentativa de acessar os elementos da matriz por meio `System.Array` do ou das interfaces que ele implementa resultará em uma exceção em tempo de execução, pois os tipos de ponteiro não são conversíveis `object`.
- As conversões de referência implícita e explícita ([conversões de referência implícita](#), [conversões de referência explícitas](#)) de um tipo de matriz unidimensional `S[]` para `System.Collections.Generic.IList<T>` e suas interfaces base genéricas nunca se aplicam a matrizes de ponteiro, pois tipos de ponteiro não podem ser usados como argumentos de tipo, e não há conversões de tipos de ponteiro para tipos que não sejam ponteiros.
- A conversão de referência explícita ([conversões de referência explícitas](#)) `System.Array` e as interfaces implementadas para qualquer *array\_type* se aplica a matrizes de ponteiro.
- As conversões de referência explícita ([conversões de referência explícitas](#)) de `System.Collections.Generic.IList<S>` e suas interfaces base para um tipo de matriz unidimensional `T[]` nunca se aplicam a matrizes de ponteiro, já que tipos de ponteiro não podem ser usados como argumentos de tipo, e não há conversões de tipos de ponteiro para tipos de não ponteiro.

Essas restrições significam que a expansão para a `foreach` instrução sobre matrizes descritas na [instrução foreach](#) não pode ser aplicada a matrizes de ponteiro. Em vez disso, uma instrução foreach do formulário

C#

```
foreach (V v in x) embedded_statement
```

onde o tipo de `x` é um tipo de matriz do formulário `T[, , ..., ]`, `N` é o número de dimensões menos 1 e `T` ou `V` é um tipo de ponteiro, é expandido usando loops aninhados da seguinte maneira:

C#

```
{  
    T[, , ..., ] a = x;  
    for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)
```

```

for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
...
for (int iN = a.GetLowerBound(N); iN <= a.GetUpperBound(N); iN++) {
    V v = (V)a.GetValue(i0,i1,...,iN);
    embedded_statement
}
}

```

As variáveis `a`, `i0`, `i1`, ..., `iN` não são visíveis ou acessíveis para `x` ou `o` ou o *embedded\_statement* ou qualquer outro código-fonte do programa. A variável `v` é somente leitura na instrução incorporada. Se não houver uma conversão explícita ([conversões de ponteiro](#)) de `T` (o tipo de elemento) para `v`, um erro será produzido e nenhuma etapa adicional será executada. Se `x` tiver o valor `null`, um `System.NullReferenceException` será lançado em tempo de execução.

## Ponteiros em expressões

Em um contexto não seguro, uma expressão pode gerar um resultado de um tipo de ponteiro, mas fora de um contexto sem segurança, é um erro de tempo de compilação para uma expressão ser de um tipo de ponteiro. Em termos precisos, fora de um contexto sem segurança, um erro de tempo de compilação ocorrerá se qualquer *Simple\_name* ([nomes simples](#)), *member\_access* ([acesso de membro](#)), *invocation\_expression* ([expressões de invocação](#)) ou *element\_access* ([acesso ao elemento](#)) for de um tipo de ponteiro.

Em um contexto sem segurança, as produções *primary\_no\_array\_creation\_expression* ([expressões primárias](#)) e *unary\_expression* ([operadores unários](#)) permitem as seguintes construções adicionais:

```

antlr

primary_no_array_creation_expression_unsafe
: pointer_member_access
| pointer_element_access
| sizeof_expression
;

unary_expression_unsafe
: pointer_indirection_expression
| addressof_expression
;

```

Essas construções são descritas nas seções a seguir. A precedência e a Associação dos operadores inseguros é implícita pela gramática.

## Indireção de ponteiro

Um *pointer\_indirection\_expression* consiste em um asterisco (`*`) seguido por um *unary\_expression*.

```
antlr

pointer_indirection_expression
: '*' unary_expression
;
```

O operador unário `*` denota indireção de ponteiro e é usado para obter a variável para a qual um ponteiro aponta. O resultado da avaliação `*P`, em que `P` é uma expressão de um tipo de ponteiro `T*`, é uma variável do tipo `T`. É um erro de tempo de compilação para aplicar o operador unário `*` a uma expressão do tipo `void*` ou a uma expressão que não é de um tipo de ponteiro.

O efeito de aplicar o operador unário `*` a um `null` ponteiro é definido pela implementação. Em particular, não há nenhuma garantia de que essa operação gere um `System.NullReferenceException`.

Se um valor inválido tiver sido atribuído ao ponteiro, o comportamento do operador unário `*` será indefinido. Entre os valores inválidos para desreferenciar um ponteiro pelo operador unário `*` estão um endereço incorretamente alinhado para o tipo apontado (consulte o exemplo em [conversões de ponteiro](#)) e o endereço de uma variável após o término de seu tempo de vida.

Para fins de análise de atribuição definitiva, uma variável produzida pela avaliação de uma expressão do formulário `*P` é considerada inicialmente atribuída ([variáveis inicialmente atribuídas](#)).

## Acesso de membro do ponteiro

Um *pointer\_member\_access* consiste em um *primary\_expression*, seguido por um `->` token "", seguido por um *identificador* e um *type\_argument\_list* opcional.

```
antlr

pointer_member_access
: primary_expression '-' identifier
;
```

Em um membro de ponteiro acesso do formulário `P->I`, `P` deve ser uma expressão de um tipo de ponteiro diferente de `void*` e `I` deve indicar um membro acessível do tipo para o qual os `P` pontos.

Um acesso de membro de ponteiro do formulário `P->I` é avaliado exatamente como `(*P).I`. Para obter uma descrição do operador de indireção de ponteiro (`*`), consulte [indireção de ponteiro](#). Para obter uma descrição do operador de acesso de membro (`.`), consulte [acesso de membro](#).

No exemplo

C#

```
using System;

struct Point
{
    public int x;
    public int y;

    public override string ToString() {
        return "(" + x + "," + y + ")";
    }
}

class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

o `->` operador é usado para acessar campos e invocar um método de uma struct por meio de um ponteiro. Como a operação `P->I` é precisamente equivalente a `(*P).I`, o `Main` método poderia igualmente ter sido escrito:

C#

```
class Test
{
    static void Main() {
        Point point;
        unsafe {
```

```

        Point* p = &point;
        (*p).x = 10;
        (*p).y = 20;
        Console.WriteLine((*p).ToString());
    }
}

```

## Acesso de elemento do ponteiro

Um *pointer\_element\_access* consiste em um *primary\_no\_array\_creation\_expression* seguido por uma expressão colocada em " [ " e " ] ".

antlr

```

pointer_element_access
: primary_no_array_creation_expression '[' expression ']'
;

```

Em um elemento de ponteiro, o acesso ao formulário `P[E]` `P` deve ser uma expressão de um tipo de ponteiro diferente de `void*` e `E` deve ser uma expressão que possa ser convertida implicitamente em `int`, `uint`, `long` ou `ulong`.

Um elemento de ponteiro acesso ao formulário `P[E]` é avaliado exatamente como `*(P + E)`. Para obter uma descrição do operador de indireção de ponteiro (`*`), consulte [indireção de ponteiro](#). Para obter uma descrição do operador de adição de ponteiro (`+`), consulte [aritmética de ponteiro](#).

No exemplo

C#

```

class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}

```

um acesso de elemento de ponteiro é usado para inicializar o buffer de caracteres em um `for` loop. Como a operação `P[E]` é precisamente equivalente a `*(P + E)`, o exemplo poderia igualmente ter sido escrito:

C#

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}
```

O operador de acesso de elemento de ponteiro não verifica erros fora de limites e o comportamento ao acessar um elemento fora dos limites é indefinido. Isso é o mesmo que C e C++.

## O operador address-of

Um *addressof\_expression* consiste em um e comercial ( `&` ) seguido por um *unary\_expression*.

antlr

```
addressof_expression
: '&' unary_expression
;
```

Dada uma expressão `E` que é de um tipo `T` e é classificada como uma variável fixa ([variáveis fixas e móveis](#)), a construção `&E` computa o endereço da variável fornecida por `E`. O tipo do resultado é `T*` e é classificado como um valor. Ocorrerá um erro de tempo de compilação se `E` não for classificado como uma variável, se `E` for classificado como uma variável local somente leitura, ou se for `E` indicado uma variável móvel. No último caso, uma instrução fixa ([a instrução Fixed](#)) pode ser usada para "corrigir" temporariamente a variável antes de obter seu endereço. Conforme indicado em [acesso de membro](#), fora de um construtor de instância ou construtor estático para uma struct ou classe que define um `readonly` campo, esse campo é considerado um valor, não uma variável. Como tal, seu endereço não pode ser obtido. Da mesma forma, o endereço de uma constante não pode ser obtido.

O `&` operador não exige que seu argumento seja definitivamente atribuído, mas após uma `&` operação, a variável à qual o operador é aplicado é considerada definitivamente atribuída no caminho de execução no qual a operação ocorre. É de responsabilidade do

programador garantir que a inicialização correta da variável realmente ocorra nessa situação.

No exemplo

```
C#  
  
using System;  
  
class Test  
{  
    static void Main() {  
        int i;  
        unsafe {  
            int* p = &i;  
            *p = 123;  
        }  
        Console.WriteLine(i);  
    }  
}
```

`i` é considerado definitivamente atribuído após a `&i` operação usada para inicializar `p`. A atribuição a `*p` no efeito é inicializada `i`, mas a inclusão dessa inicialização é responsabilidade do programador e nenhum erro de tempo de compilação ocorrerá se a atribuição tiver sido removida.

As regras de atribuição definitiva para o `&` operador existem de modo que a inicialização redundante de variáveis locais possa ser evitada. Por exemplo, muitas APIs externas usam um ponteiro para uma estrutura que é preenchida pela API. Chamadas para essas APIs normalmente passam o endereço de uma variável de struct local e sem a regra, a inicialização redundante da variável de struct seria necessária.

## Incrementar e decrementar ponteiros

Em um contexto sem segurança, os `++` `--` operadores e ([incremento de sufixo e diminuição de operadores](#) e os operadores de [incremento e diminuição de prefixo](#)) podem ser aplicados a variáveis de ponteiro de todos os tipos, exceto `void*`. Portanto, para cada tipo de ponteiro `T*`, os seguintes operadores são definidos implicitamente:

```
C#  
  
T* operator +(T* x);  
T* operator -(T* x);
```

Os operadores produzem os mesmos resultados de `x + 1` e `x - 1`, respectivamente ([aritmética de ponteiro](#)). Em outras palavras, para uma variável de ponteiro do tipo `T*`, o `++` operador adiciona `sizeof(T)` ao endereço contido na variável e o `--` operador subtrai `sizeof(T)` do endereço contido na variável.

Se uma operação de incremento ou diminuição de ponteiro estourar o domínio do tipo de ponteiro, o resultado será definido pela implementação, mas nenhuma exceção será produzida.

## Aritmética do ponteiro

Em um contexto sem segurança, os `+` operadores and (operador de `- adição` e `subtração`) podem ser aplicados a valores de todos os tipos de ponteiro, exceto `void*`. Portanto, para cada tipo de ponteiro `T*`, os seguintes operadores são definidos implicitamente:

C#

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);

long operator -(T* x, T* y);
```

Dada uma expressão `P` de um tipo de ponteiro `T*` e uma expressão `N` de tipo `int`, `uint`, `long` ou `ulong`, as expressões `P + N` e `N + P` computar o valor do ponteiro do tipo `T*` que resulta da adição `N * sizeof(T)` ao endereço fornecido por `P`. Da mesma forma, a expressão `P - N` computa o valor de ponteiro do tipo `T*` que resulta da subtração `N * sizeof(T)` do endereço fornecido por `P`.

Dadas duas expressões `P` e `Q`, de um tipo de ponteiro `T*`, a expressão `P - Q` computa a diferença entre os endereços fornecidos pelo `P` e `Q`, em seguida, divide essa

diferença por `sizeof(T)` . O tipo do resultado é sempre `long` . Na verdade, `P - Q` é calculado como `((long)(P) - (long)(Q)) / sizeof(T)` .

Por exemplo:

```
C#  
  
using System;  
  
class Test  
{  
    static void Main()  
    {  
        unsafe {  
            int* values = stackalloc int[20];  
            int* p = &values[1];  
            int* q = &values[15];  
            Console.WriteLine("p - q = {0}", p - q);  
            Console.WriteLine("q - p = {0}", q - p);  
        }  
    }  
}
```

que produz a saída:

```
Console  
  
p - q = -14  
q - p = 14
```

Se uma operação aritmética de ponteiro estoura o domínio do tipo de ponteiro, o resultado é truncado de maneira definida pela implementação, mas nenhuma exceção é produzida.

## Comparação de ponteiros

Em um contexto não seguro, os `==`, `!=`, `<`, `>`, `<=` e `>=` (operadores relacionais e de teste de tipo) podem ser aplicados aos valores de todos os tipos de ponteiro. Os operadores de comparação de ponteiro são:

```
C#  
  
bool operator ==(void* x, void* y);  
bool operator !=(void* x, void* y);  
bool operator <(void* x, void* y);  
bool operator >(void* x, void* y);  
bool operator <=(void* x, void* y);  
bool operator >=(void* x, void* y);
```

Como existe uma conversão implícita de qualquer tipo de ponteiro para o `void*` tipo, operandos de qualquer tipo de ponteiro podem ser comparados usando esses operadores. Os operadores de comparação comparam os endereços fornecidos pelos dois operandos como se fossem inteiros sem sinal.

## O operador `sizeof`

O operador `sizeof` retorna o número de bytes ocupados por uma variável de um determinado tipo. O tipo especificado como um operando `sizeof` deve ser um *unmanaged\_type* ([tipos de ponteiro](#)).

antlr

```
sizeof_expression
  : 'sizeof' '(' unmanaged_type ')'
  ;
```

O resultado do `sizeof` operador é um valor do tipo `int`. Para determinados tipos predefinidos, o `sizeof` operador produz um valor constante, conforme mostrado na tabela a seguir.

Expression	Resultado
<code>sizeof(sbyte)</code>	1
<code>sizeof(byte)</code>	1
<code>sizeof(short)</code>	2
<code>sizeof(ushort)</code>	2
<code>sizeof(int)</code>	4
<code>sizeof(uint)</code>	4
<code>sizeof(long)</code>	8
<code>sizeof(ulong)</code>	8
<code>sizeof(char)</code>	2
<code>sizeof(float)</code>	4
<code>sizeof(double)</code>	8
<code>sizeof(bool)</code>	1

Para todos os outros tipos, o resultado do `sizeof` operador é definido pela implementação e é classificado como um valor, não uma constante.

A ordem na qual os membros são empacotados em um struct não é especificada.

Para fins de alinhamento, pode haver um preenchimento sem nome no início de uma struct, dentro de uma struct e no final da estrutura. O conteúdo dos bits usados como preenchimento é indeterminado.

Quando aplicado a um operando que tem o tipo struct, o resultado é o número total de bytes em uma variável desse tipo, incluindo qualquer preenchimento.

## A instrução `fixed`

Em um contexto sem segurança, a produção `embedded_statement` ([instruções](#)) permite um constructo adicional, a `fixed` instrução, que é usada para "corrigir" uma variável móvel, de modo que seu endereço permaneça constante durante a instrução.

```
antlr

fixed_statement
    : 'fixed' '(' pointer_type fixed_pointer_declarators ')'
embedded_statement
;

fixed_pointer_declarators
: fixed_pointer_declarator (',' fixed_pointer_declarator)*
;

fixed_pointer_declarator
: identifier '=' fixed_pointer_initializer
;

fixed_pointer_initializer
: '&' variable_reference
| expression
;
```

Cada `fixed_pointer_declarator` declara uma variável local do `pointer_type` especificado e inicializa essa variável local com o endereço computado pelo `fixed_pointer_initializer` correspondente. Uma variável local declarada em uma `fixed` instrução pode ser acessada em qualquer `fixed_pointer_initializer`s que ocorrem à direita da declaração dessa variável e na `embedded_statement` da `fixed` instrução. Uma variável local declarada por uma `fixed` instrução é considerada somente leitura. Ocorrerá um erro de tempo de compilação se a instrução inserida tentar modificar essa variável local (por

meio de atribuição ou os `++` `--` operadores e) ou passá-la como um `ref` `out` parâmetro ou.

Um *fixed\_pointer\_initializer* pode ser um dos seguintes:

- O token " `&` " seguido por um *variable\_reference* ([regras precisas para determinar a atribuição definitiva](#)) para uma variável móvel ([variáveis fixas e móveis](#)) de um tipo não gerenciado `T`, desde que o tipo `T*` seja implicitamente conversível para o tipo de ponteiro fornecido na `fixed` instrução. Nesse caso, o inicializador computa o endereço da variável fornecida, e a variável é garantida para permanecer em um endereço fixo para a duração da `fixed` instrução.
- Uma expressão de um *array\_type* com elementos de um tipo não gerenciado `T`, desde que o tipo `T*` seja implicitamente conversível para o tipo de ponteiro fornecido na `fixed` instrução. Nesse caso, o inicializador computa o endereço do primeiro elemento na matriz e toda a matriz é garantida para permanecer em um endereço fixo durante a `fixed` instrução. Se a expressão de matriz for nula ou se a matriz tiver zero elementos, o inicializador computará um endereço igual a zero.
- Uma expressão do tipo `string`, desde que o tipo `char*` seja conversível implicitamente no tipo de ponteiro fornecido na `fixed` instrução. Nesse caso, o inicializador computa o endereço do primeiro caractere na cadeia de caracteres e toda a cadeia de caracteres é garantida para permanecer em um endereço fixo para a duração da `fixed` instrução. O comportamento da `fixed` instrução será definido pela implementação se a expressão de cadeia de caracteres for nula.
- Um *Simple\_name* ou *member\_access* que faz referência a um membro de buffer de tamanho fixo de uma variável móvel, desde que o tipo do membro de buffer de tamanho fixo seja implicitamente conversível para o tipo de ponteiro fornecido na `fixed` instrução. Nesse caso, o inicializador computa um ponteiro para o primeiro elemento do buffer de tamanho fixo ([buffers de tamanho fixo em expressões](#)) e é garantido que o buffer de tamanho fixo permaneça em um endereço fixo para a duração da `fixed` instrução.

Para cada endereço calculado por um *fixed\_pointer\_initializer* a `fixed` instrução garante que a variável referenciada pelo endereço não esteja sujeita a realocação ou descarte pelo coletor de lixo durante a `fixed` instrução. Por exemplo, se o endereço computado por um *fixed\_pointer\_initializer* referenciar um campo de um objeto ou um elemento de uma instância de matriz, a `fixed` instrução garante que a instância de objeto recipiente não seja realocada ou descartada durante o tempo de vida da instrução.

É responsabilidade do programador garantir que os ponteiros criados por `fixed` instruções não sobrevivem além da execução dessas instruções. Por exemplo, quando ponteiros criados por `fixed` instruções são passados para APIs externas, é

responsabilidade do programador garantir que as APIs não mantenham memória desses ponteiros.

Os objetos fixos podem causar a fragmentação do heap (porque eles não podem ser movidos). Por esse motivo, os objetos devem ser corrigidos somente quando absolutamente necessário e, em seguida, apenas pelo menor tempo possível.

O exemplo

C#

```
class Test
{
    static int x;
    int y;

    unsafe static void F(int* p) {
        *p = 1;
    }

    static void Main() {
        Test t = new Test();
        int[] a = new int[10];
        unsafe {
            fixed (int* p = &x) F(p);
            fixed (int* p = &t.y) F(p);
            fixed (int* p = &a[0]) F(p);
            fixed (int* p = a) F(p);
        }
    }
}
```

demonstra vários usos da `fixed` instrução. A primeira instrução corrige e Obtém o endereço de um campo estático, a segunda instrução corrige e Obtém o endereço de um campo de instância, e a terceira instrução corrige e Obtém o endereço de um elemento de matriz. Em cada caso, seria um erro para usar o `&` operador regular, uma vez que as variáveis são todas classificadas como variáveis móveis.

A quarta `fixed` instrução no exemplo acima produz um resultado semelhante ao terceiro.

Este exemplo da `fixed` instrução usa `string` :

C#

```
class Test
{
    static string name = "xx";
```

```

unsafe static void F(char* p) {
    for (int i = 0; p[i] != '\0'; ++i)
        Console.WriteLine(p[i]);
}

static void Main() {
    unsafe {
        fixed (char* p = name) F(p);
        fixed (char* p = "xx") F(p);
    }
}

```

Em um contexto não seguro, os elementos de matrizes unidimensionais são armazenados em ordem de índice crescente, começando com index `0` e terminando com index `Length - 1`. Para matrizes multidimensionais, os elementos de matriz são armazenados de modo que os índices da dimensão mais à direita sejam aumentados primeiro, depois a dimensão à esquerda e assim por diante à esquerda. Dentro de uma `fixed` instrução que obtém um ponteiro `p` para uma instância de matriz `a`, os valores de ponteiro que variam de `p` para `p + a.Length - 1` representar os endereços dos elementos na matriz. Da mesma forma, as variáveis que variam de `p[0]` para `p[a.Length - 1]` representar os elementos reais da matriz. Devido à maneira como as matrizes são armazenadas, podemos tratar uma matriz de qualquer dimensão como se ela fosse linear.

Por exemplo:

```

C#

using System;

class Test
{
    static void Main() {
        int[,] a = new int[2,3,4];
        unsafe {
            fixed (int* p = a) {
                for (int i = 0; i < a.Length; ++i)      // treat as linear
                    p[i] = i;
            }
        }

        for (int i = 0; i < 2; ++i)
            for (int j = 0; j < 3; ++j) {
                for (int k = 0; k < 4; ++k)
                    Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k,
a[i,j,k]);
                Console.WriteLine();
            }
    }
}

```

```
        }
    }
}
```

que produz a saída:

Console

```
[0,0,0] = 0 [0,0,1] = 1 [0,0,2] = 2 [0,0,3] = 3
[0,1,0] = 4 [0,1,1] = 5 [0,1,2] = 6 [0,1,3] = 7
[0,2,0] = 8 [0,2,1] = 9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```

No exemplo

C#

```
class Test
{
    unsafe static void Fill(int* p, int count, int value) {
        for ( ; count != 0; count--) *p++ = value;
    }

    static void Main() {
        int[] a = new int[100];
        unsafe {
            fixed (int* p = a) Fill(p, 100, -1);
        }
    }
}
```

uma `fixed` instrução é usada para corrigir uma matriz para que seu endereço possa ser passado para um método que usa um ponteiro.

No exemplo:

C#

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
```

```

        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    Font f;

    unsafe static void Main()
{
    Test test = new Test();
    test.f.size = 10;
    fixed (char* p = test.f.name) {
        PutString("Times New Roman", p, 32);
    }
}
}

```

uma instrução Fixed é usada para corrigir um buffer de tamanho fixo de uma struct para que seu endereço possa ser usado como um ponteiro.

Um `char*` valor produzido pela correção de uma instância de cadeia de caracteres sempre aponta para uma cadeia de caracteres terminada em nulo. Dentro de uma instrução fixa que obtém um ponteiro `p` para uma instância de cadeia de caracteres `s`, os valores de ponteiro que variam de `p` para `p + s.Length - 1` representar os endereços dos caracteres na cadeia de caracteres e o valor do ponteiro `p + s.Length` sempre aponta para um caractere nulo (o caractere com valor `'\0'`).

Modificar objetos de tipo gerenciado por meio de ponteiros fixos pode resultar em um comportamento indefinido. Por exemplo, como as cadeias de caracteres são imutáveis, é responsabilidade do programador garantir que o caractere referenciado por um ponteiro para uma cadeia de caracteres fixa não seja modificado.

A terminação nula automática de cadeias de caracteres é particularmente conveniente ao chamar APIs externas que esperam cadeias de caracteres "estilo C". Observe, no entanto, que uma instância de cadeia de caracteres tem permissão para conter caracteres nulos. Se esses caracteres nulos estiverem presentes, a cadeia de caracteres aparecerá truncada quando tratada como terminada em nulo `char*`.

## Buffers de tamanho fixo

Buffers de tamanho fixo são usados para declarar matrizes em linha "C style" como membros de structs e são principalmente úteis para a interface com APIs não gerenciadas.

# Declarações de buffer de tamanho fixo

Um *buffer de tamanho fixo* é um membro que representa o armazenamento de um buffer de comprimento fixo de variáveis de um determinado tipo. Uma declaração de buffer de tamanho fixo apresenta um ou mais buffers de tamanho fixo de um determinado tipo de elemento. Buffers de tamanho fixo só são permitidos em declarações struct e só podem ocorrer em contextos não seguros ([contextos não seguros](#)).

```
antlr

struct_member_declaration_unsafe
: fixed_size_buffer_declarator+
;

fixed_size_buffer_declarator
: attributes? fixed_size_buffer_modifier* 'fixed' buffer_element_type
fixed_size_buffer_declarator+ ';'
;

fixed_size_buffer_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'unsafe'
;

buffer_element_type
: type
;

fixed_size_buffer_declarator
: identifier '[' constant_expression ']'
;
```

Uma declaração de buffer de tamanho fixo pode incluir um conjunto de atributos ([atributos](#)), um `new` modificador ([modificadores](#)), uma combinação válida dos quatro modificadores de acesso ([parâmetros de tipo e restrições](#)) e um `unsafe` modificador ([contextos não seguros](#)). Os atributos e os modificadores se aplicam a todos os membros declarados pela declaração de buffer de tamanho fixo. É um erro para que o mesmo modificador apareça várias vezes em uma declaração de buffer de tamanho fixo.

Uma declaração de buffer de tamanho fixo não tem permissão para incluir o `static` modificador.

O tipo de elemento de buffer de uma declaração de buffer de tamanho fixo especifica o tipo de elemento dos buffers introduzidos pela declaração. O tipo de elemento de buffer deve ser um dos tipos predefinidos,,,,`sbyte` `byte` , `short` `ushort` `int` `uint`  
`long` `ulong` `char` `float` `double` ou `bool` .

O tipo de elemento buffer é seguido por uma lista de declaradores de buffer de tamanho fixo, e cada um deles introduz um novo membro. Um Declarador de buffer de tamanho fixo consiste em um identificador que nomeia o membro, seguido por uma expressão constante incluída em [ ] tokens e. A expressão constante denota o número de elementos no membro introduzidos pelo Declarador de buffer de tamanho fixo. O tipo da expressão constante deve ser implicitamente conversível para `int` o tipo, e o valor deve ser um inteiro positivo diferente de zero.

Os elementos de um buffer de tamanho fixo têm garantia de serem dispostos em sequência na memória.

Uma declaração de buffer de tamanho fixo que declara vários buffers de tamanho fixo é equivalente a várias declarações de uma única declaração de buffer de tamanho fixo com os mesmos atributos e tipos de elemento. Por exemplo,

C#

```
unsafe struct A
{
    public fixed int x[5], y[10], z[100];
}
```

é equivalente a

C#

```
unsafe struct A
{
    public fixed int x[5];
    public fixed int y[10];
    public fixed int z[100];
}
```

## Buffers de tamanho fixo em expressões

A pesquisa de Membros ([operadores](#)) de um membro de buffer de tamanho fixo prossegue exatamente como a pesquisa de membros de um campo.

Um buffer de tamanho fixo pode ser referenciado em uma expressão usando uma *Simple\_name* ([inferência de tipos](#)) ou uma *member\_access* ([verificação de tempo de compilação de resolução dinâmica de sobrecarga](#)).

Quando um membro de buffer de tamanho fixo é referenciado como um nome simples, o efeito é o mesmo que um membro de acesso do formulário `this.I`, em que `I` é o membro do buffer de tamanho fixo.

Em um acesso de membro do formulário `E.I`, se `E` for de um tipo struct e uma pesquisa de membro de `I` nesse tipo struct identificar um membro de tamanho fixo, o `E.I` será avaliado como o seguinte:

- Se a expressão `E.I` não ocorrer em um contexto sem segurança, ocorrerá um erro em tempo de compilação.
- Se `E` for classificado como um valor, ocorrerá um erro em tempo de compilação.
- Caso contrário, se `E` for uma variável móvel ([variáveis fixas e móveis](#)) e a expressão `E.I` não for uma *fixed\_pointer\_initializer* ([a instrução Fixed](#)), ocorrerá um erro em tempo de compilação.
- Caso contrário, `E` faz referência a uma variável fixa e o resultado da expressão é um ponteiro para o primeiro elemento do membro de buffer de tamanho fixo `I` em `E`. O resultado é do tipo `S*`, em que `S` é o tipo de elemento de `I` e é classificado como um valor.

Os elementos subsequentes do buffer de tamanho fixo podem ser acessados usando operações de ponteiro do primeiro elemento. Ao contrário do acesso a matrizes, o acesso aos elementos de um buffer de tamanho fixo é uma operação não segura e não é verificado.

O exemplo a seguir declara e usa uma struct com um membro de buffer de tamanho fixo.

C#

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
}
```

```

        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

unsafe static void Main()
{
    Font f;
    f.size = 10;
    PutString("Times New Roman", f.name, 32);
}
}

```

## Verificação de atribuição definitiva

Os buffers de tamanho fixo não estão sujeitos à verificação de atribuição definitiva ([atribuição definitiva](#)) e os membros do buffer de tamanho fixo são ignorados para fins de verificação de atribuição definitiva de variáveis de tipo struct.

Quando o que contém a variável de struct mais externa de um membro de buffer de tamanho fixo é uma variável estática, uma variável de instância de uma instância de classe ou um elemento de matriz, os elementos do buffer de tamanho fixo são inicializados automaticamente para seus valores padrão ([valores padrão](#)). Em todos os outros casos, o conteúdo inicial de um buffer de tamanho fixo é indefinido.

## Alocação da pilha

Em um contexto não seguro, uma declaração de variável local ([declarações de variável local](#)) pode incluir um inicializador de alocação de pilha que aloca memória da pilha de chamadas.

```

antlr

local_variable_initializer_unsafe
: stackalloc_initializer
;

stackalloc_initializer
: 'stackalloc' unmanaged_type '[' expression ']'
;
```

O *unmanaged\_type* indica o tipo dos itens que serão armazenados no local alocado recentemente e a *expressão* indica o número desses itens. Em conjunto, eles especificam o tamanho de alocação necessário. Como o tamanho de uma alocação de pilha não pode ser negativo, é um erro de tempo de compilação para especificar o número de itens como um *constant\_expression* que é avaliado como um valor negativo.

Um inicializador de alocação de pilha do formulário `stackalloc T[E]` requer `T` que seja um tipo não gerenciado ([tipos de ponteiro](#)) e `E` seja uma expressão do tipo `int`. A construção aloca `E * sizeof(T)` bytes da pilha de chamadas e retorna um ponteiro, do tipo `T*`, para o bloco recentemente alocado. Se `E` for um valor negativo, o comportamento será indefinido. Se `E` for zero, nenhuma alocação será feita e o ponteiro retornado será definido como implementação. Se não houver memória suficiente disponível para alocar um bloco de determinado tamanho, um `System.StackOverflowException` será lançado.

O conteúdo da memória recém-alocada é indefinido.

Inicializadores de alocação de pilha não são permitidos em `catch` `finally` blocos ou ([a instrução try](#)).

Não é possível liberar explicitamente a memória alocada usando `stackalloc`. Todos os blocos de memória alocada na pilha criados durante a execução de um membro de função são automaticamente descartados quando esse membro de função retorna. Isso corresponde à `alloca` função, uma extensão normalmente encontrada em implementações de C e C++.

No exemplo

```
C#  
  
using System;  
  
class Test  
{  
    static string IntToString(int value) {  
        int n = value >= 0? value: -value;  
        unsafe {  
            char* buffer = stackalloc char[16];  
            char* p = buffer + 16;  
            do {  
                *--p = (char)(n % 10 + '0');  
                n /= 10;  
            } while (n != 0);  
            if (value < 0) *--p = '-';  
            return new string(p, 0, (int)(buffer + 16 - p));  
        }  
    }  
  
    static void Main() {  
        Console.WriteLine(IntToString(12345));  
        Console.WriteLine(IntToString(-999));  
    }  
}
```

um `stackalloc` inicializador é usado no `IntToString` método para alocar um buffer de 16 caracteres na pilha. O buffer é descartado automaticamente quando o método retorna.

## Alocação de memória dinâmica

Exceto para o `stackalloc` operador, o C# não fornece construções predefinidas para gerenciar memória não coletada pelo lixo. Esses serviços normalmente são fornecidos pelo suporte a bibliotecas de classes ou importados diretamente do sistema operacional subjacente. Por exemplo, a `Memory` classe a seguir ilustra como as funções de heap de um sistema operacional subjacente podem ser acessadas do C#:

C#

```
using System;
using System.Runtime.InteropServices;

public static unsafe class Memory
{
    // Handle for the process heap. This handle is used in all calls to the
    // HeapXXX APIs in the methods below.
    private static readonly IntPtr s_heap = GetProcessHeap();

    // Allocates a memory block of the given size. The allocated memory is
    // automatically initialized to zero.
    public static void* Alloc(int size)
    {
        void* result = HeapAlloc(s_heap, HEAP_ZERO_MEMORY, (UIntPtr)size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Copies count bytes from src to dst. The source and destination
    // blocks are permitted to overlap.
    public static void Copy(void* src, void* dst, int count)
    {
        byte* ps = (byte*)src;
        byte* pd = (byte*)dst;
        if (ps > pd)
        {
            for (; count != 0; count--) *pd++ = *ps++;
        }
        else if (ps < pd)
        {
            for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
        }
    }

    // Frees a memory block.
}
```

```

public static void Free(void* block)
{
    if (!HeapFree(s_heap, 0, block)) throw new
InvalidOperationException();
}

// Re-allocates a memory block. If the reallocation request is for a
// larger size, the additional region of memory is automatically
// initialized to zero.
public static void* ReAlloc(void* block, int size)
{
    void* result = HeapReAlloc(s_heap, HEAP_ZERO_MEMORY, block,
(UIntPtr)size);
    if (result == null) throw new OutOfMemoryException();
    return result;
}

// Returns the size of a memory block.
public static int SizeOf(void* block)
{
    int result = (int)HeapSize(s_heap, 0, block);
    if (result == -1) throw new InvalidOperationException();
    return result;
}

// Heap API flags
private const int HEAP_ZERO_MEMORY = 0x00000008;

// Heap API functions
[DllImport("kernel32")]
private static extern IntPtr GetProcessHeap();

[DllImport("kernel32")]
private static extern void* HeapAlloc(IntPtr hHeap, int flags, UIntPtr
size);

[DllImport("kernel32")]
private static extern bool HeapFree(IntPtr hHeap, int flags, void*
block);

[DllImport("kernel32")]
private static extern void* HeapReAlloc(IntPtr hHeap, int flags, void*
block, UIntPtr size);

}

```

Um exemplo que usa a `Memory` classe é fornecido abaixo:

C#

```
class Test
{
    static unsafe void Main()
    {
        byte* buffer = null;
        try
        {
            const int Size = 256;
            buffer = (byte*)Memory.Alloc(Size);
            for (int i = 0; i < Size; i++) buffer[i] = (byte)i;
            byte[] array = new byte[Size];
            fixed (byte* p = array) Memory.Copy(buffer, p, Size);
            for (int i = 0; i < Size; i++) Console.WriteLine(array[i]);
        }
        finally
        {
            if (buffer != null) Memory.Free(buffer);
        }
    }
}
```

O exemplo aloca 256 bytes de memória por meio do `Memory.Alloc` e inicializa o bloco de memória com valores que aumentam de 0 a 255. Em seguida, ele aloca uma matriz de bytes do elemento 256 e usa `Memory.Copy` para copiar o conteúdo do bloco de memória para a matriz de bytes. Por fim, o bloco de memória é liberado usando `Memory.Free` e o conteúdo da matriz de bytes é apresentado no console.

# Annex A Grammar

Article • 04/07/2023

This clause is informative.

## A.1 General

This annex contains the grammar productions found in the specification, including the optional ones for unsafe code. Productions appear here in the same order in which they appear in the specification.

## A.2 Lexical grammar

ANTLR

```
// Source: §6.3.1 General
DEFAULT  : 'default' ;
NULL     : 'null' ;
TRUE     : 'true' ;
FALSE    : 'false' ;
ASTERISK : '*' ;
SLASH    : '/' ;

// Source: §6.3.1 General
input
: input_section?
;

input_section
: input_section_part+
;

input_section_part
: input_element* New_Line
| PP_Directive
;

input_element
: Whitespace
| Comment
| token
;

// Source: §6.3.2 Line terminators
New_Line
: New_Line_Character
```

```

| '\u000D\u000A'    // carriage return, line feed
;

// Source: §6.3.3 Comments
Comment
: Single_Line_Comment
| Delimited_Comment
;

fragment Single_Line_Comment
: '//' Input_Character*
;

fragment Input_Character
// anything but New_Line_Character
: ~('\u000D' | '\u000A' | '\u0085' | '\u2028' | '\u2029')
;

fragment New_Line_Character
: '\u000D' // carriage return
| '\u000A' // line feed
| '\u0085' // next line
| '\u2028' // line separator
| '\u2029' // paragraph separator
;

fragment Delimited_Comment
: /* Delimited_Comment_Section* ASTERISK+ '/'
;

fragment Delimited_Comment_Section
: SLASH
| ASTERISK* Not_Slash_Or_Asterisk
;

fragment Not_Slash_Or_Asterisk
: ~( '/' | '*' ) // Any except SLASH or ASTERISK
;

// Source: §6.3.4 White space
Whitespace
: [\p{Zs}] // any character with Unicode class Zs
| '\u0009' // horizontal tab
| '\u000B' // vertical tab
| '\u000C' // form feed
;

// Source: §6.4.1 General
token
: identifier
| keyword
| Integer_Literal
| Real_Literal
| Character_Literal
| String_Literal
;
```

```

| operator_or_punctuator
;

// Source: §6.4.2 Unicode character escape sequences
fragment Unicode_Escape_Sequence
: '\u' Hex_Digit Hex_Digit Hex_Digit Hex_Digit
| '\U' Hex_Digit Hex_Digit Hex_Digit Hex_Digit
    Hex_Digit Hex_Digit Hex_Digit Hex_Digit
;
;

// Source: §6.4.3 Identifiers
identifier
: Simple_Identifier
| contextual_keyword
;

Simple_Identifier
: Available_Identifier
| Escaped_Identifier
;

fragment Available_Identifier
// excluding keywords or contextual keywords, see note below
: Basic_Identifier
;

fragment Escaped_Identifier
// Includes keywords and contextual keywords prefixed by '@'.
// See note below.
: '@' Basic_Identifier
;

fragment Basic_Identifier
: Identifier_Start_Character Identifier_Part_Character*
;

fragment Identifier_Start_Character
: Letter_Character
| Underscore_Character
;

fragment Underscore_Character
: '_' // underscore
| '\u005F' [ff] // Unicode_Escape_Sequence for underscore
;

fragment Identifier_Part_Character
: Letter_Character
| Decimal_Digit_Character
| Connecting_Character
| Combining_Character
| Formatting_Character
;

fragment Letter_Character

```

```

// Category Letter, all subcategories; category Number, subcategory
letter.
: [\p{L}\p{Nl}]
// Only escapes for categories L & Nl allowed. See note below.
| Unicode_Escape_Sequence
;

fragment Combining_Character
// Category Mark, subcategories non-spacing and spacing combining.
: [\p{Mn}\p{Mc}]
// Only escapes for categories Mn & Mc allowed. See note below.
| Unicode_Escape_Sequence
;

fragment Decimal_Digit_Character
// Category Number, subcategory decimal digit.
: [\p{Nd}]
// Only escapes for category Nd allowed. See note below.
| Unicode_Escape_Sequence
;

fragment Connecting_Character
// Category Punctuation, subcategory connector.
: [\p{Pc}]
// Only escapes for category Pc allowed. See note below.
| Unicode_Escape_Sequence
;

fragment Formatting_Character
// Category Other, subcategory format.
: [\p{Cf}]
// Only escapes for category Cf allowed, see note below.
| Unicode_Escape_Sequence
;

// Source: §6.4.4 Keywords
keyword
: 'abstract' | 'as' | 'base' | 'bool' | 'break'
| 'byte' | 'case' | 'catch' | 'char' | 'checked'
| 'class' | 'const' | 'continue' | 'decimal' | DEFAULT
| 'delegate' | 'do' | 'double' | 'else' | 'enum'
| 'event' | 'explicit' | 'extern' | FALSE | 'finally'
| 'fixed' | 'float' | 'for' | 'foreach' | 'goto'
| 'if' | 'implicit' | 'in' | 'int' | 'interface'
| 'internal' | 'is' | 'lock' | 'long' | 'namespace'
| 'new' | NULL | 'object' | 'operator' | 'out'
| 'override' | 'params' | 'private' | 'protected' | 'public'
| 'readonly' | 'ref' | 'return' | 'sbyte' | 'sealed'
| 'short' | 'sizeof' | 'stackalloc' | 'static' | 'string'
| 'struct' | 'switch' | 'this' | 'throw' | TRUE
| 'try' | 'typeof' | 'uint' | 'ulong' | 'unchecked'
| 'unsafe' | 'ushort' | 'using' | 'virtual' | 'void'
| 'volatile' | 'while' |
;

```

```

// Source: §6.4.4 Keywords
contextual_keyword
: 'add'      | 'alias'       | 'ascending'   | 'async'        | 'await'
| 'by'        | 'descending'  | 'dynamic'     | 'equals'       | 'from'
| 'get'        | 'global'      | 'group'       | 'into'         | 'join'
| 'let'        | 'nameof'      | 'on'          | 'orderby'      | 'partial'
| 'remove'    | 'select'      | 'set'         | 'unmanaged'   | 'value'
| 'var'        | 'when'        | 'where'       | 'yield'
;

// Source: §6.4.5.1 General
literal
: boolean_literal
| Integer_Literal
| Real_Literal
| Character_Literal
| String_Literal
| null_literal
;

// Source: §6.4.5.2 Boolean literals
boolean_literal
: TRUE
| FALSE
;

// Source: §6.4.5.3 Integer literals
Integer_Literal
: Decimal_Integer_Literal
| Hexadecimal_Integer_Literal
| Binary_Integer_Literal
;

fragment Decimal_Integer_Literal
: Decimal_Digit Decorated.Decimal_Digit* Integer_Type_Suffix?
;

fragment Decorated.Decimal_Digit
: '_'* Decimal_Digit
;

fragment Decimal_Digit
: '0'..'9'
;

fragment Integer_Type_Suffix
: 'U' | 'u' | 'L' | 'l' |
| 'UL' | 'Ul' | 'uL' | 'ul' | 'LU' | 'Lu' | 'lu' | 'lu'
;

fragment Hexadecimal_Integer_Literal
: ('0x' | '0X') Decorated_Hex_Digit+ Integer_Type_Suffix?
;

fragment Decorated_Hex_Digit

```

```

: '_'* Hex_Digit
;

fragment Hex_Digit
: '0'..'9' | 'A'..'F' | 'a'..'f'
;

fragment Binary_Integer_Literal
: ('0b' | '0B') Decorated_Binary_Digit+ Integer_Type_Suffix?
;

fragment Decorated_Binary_Digit
: '_'* Binary_Digit
;

fragment Binary_Digit
: '0' | '1'
;

// Source: §6.4.5.4 Real literals
Real_Literal
: Decimal_Digit Decorated.Decimal_Digit* '.'
  Decimal_Digit Decorated.Decimal_Digit* Exponent_Part?
Real_Type_Suffix?
| '.' Decimal_Digit Decorated.Decimal_Digit* Exponent_Part?
Real_Type_Suffix?
| Decimal_Digit Decorated.Decimal_Digit* Exponent_Part Real_Type_Suffix?
| Decimal_Digit Decorated.Decimal_Digit* Real_Type_Suffix
;

fragment Exponent_Part
: ('e' | 'E') Sign? Decimal_Digit Decorated.Decimal_Digit*
;

fragment Sign
: '+' | '-'
;

fragment Real_Type_Suffix
: 'F' | 'f' | 'D' | 'd' | 'M' | 'm'
;

// Source: §6.4.5.5 Character literals
Character_Literal
: '\'' Character '\''
;

fragment Character
: Single_Character
| Simple_Escape_Sequence
| Hexadecimal_Escape_Sequence
| Unicode_Escape_Sequence
;
;

fragment Single_Character

```

```

// anything but ', \, and New_Line_Character
: ~[ '\u000D\u000A\u0085\u2028\u2029]
;

fragment Simple_Escape_Sequence
: '\\\\' | '\\"' | '\\\\\\' | '\\0' | '\\a' | '\\b' |
  '\\f' | '\\n' | '\\r' | '\\t' | '\\v'
;

fragment Hexadecimal_Escape_Sequence
: '\\x' Hex_Digit Hex_Digit? Hex_Digit? Hex_Digit?
;

// Source: §6.4.5.6 String literals
String_Literal
: Regular_String_Literal
| Verbatim_String_Literal
;

fragment Regular_String_Literal
: '"' Regular_String_Literal_Character* '"'
;

fragment Regular_String_Literal_Character
: Single-Regular_String_Literal_Character
| Simple_Escape_Sequence
| Hexadecimal_Escape_Sequence
| Unicode_Escape_Sequence
;
;

fragment Single-Regular_String_Literal_Character
// anything but ", \, and New_Line_Character
: ~[ '\u000D\u000A\u0085\u2028\u2029]
;

fragment Verbatim_String_Literal
: '@"' Verbatim_String_Literal_Character* '"'
;

fragment Verbatim_String_Literal_Character
: Single_Verbatim_String_Literal_Character
| Quote_Escape_Sequence
;
;

fragment Single_Verbatim_String_Literal_Character
: ~[""] // anything but quotation mark (U+0022)
;
;

fragment Quote_Escape_Sequence
: '""'
;

// Source: §6.4.5.7 The null literal
null_literal
: NULL
;
```

```

;

// Source: §6.4.6 Operators and punctuators
operator_or_punctuator
: '{' | '}' | '[' | ']' | '(' | ')' | '.' | ',' | ';' | ';' |
| '+' | '-' | ASTERISK | SLASH | '%' | '&' | '|' | '^' | '!' |
| '~' |
| '=' | '<' | '>' | '?' | '??' | '::' | '++' | '--' | '&&' | '||' |
| '>' | '==' | '!= | '<=' | '>=' | '+=' | '-=' | '*=' | '/=' | '%=' |
| '&=' | '|=' | '^=' | '<<' | '<<=' | '>=' |
;

right_shift
: '>' | '>' |
;

right_shift_assignment
: '>' | '>=' |
;

// Source: §6.5.1 General
PP_Directive
: PP_Start PP_Kind PP_New_Line
;

fragment PP_Kind
: PP_Declaration
| PP_Conditional
| PP_Line
| PP_Diagnostic
| PP_Region
| PP_Pragma
;

// Only recognised at the beginning of a line
fragment PP_Start
// See note below.
: { getCharPositionInLine() == 0 }? PPWhitespace? '#' PPWhitespace?
;

fragment PPWhitespace
: ( [\p{Zs}] // any character with Unicode class Zs
| '\u0009' // horizontal tab
| '\u000B' // vertical tab
| '\u000C' // form feed
)+
;

fragment PP_New_Line
: PPWhitespace? Single_Line_Comment? New_Line
;

// Source: §6.5.2 Conditional compilation symbols
fragment PP_Conditional_Symbol
// Must not be equal to tokens TRUE or FALSE. See note below.

```

```

: Basic_Identifier
;

// Source: §6.5.3 Pre-processing expressions
fragment PP_Expression
: PP_Whitespace? PP_Or_Expression PP_Whitespace?
;

fragment PP_Or_Expression
: PP_And_Expression (PP_Whitespace? '||' PP_Whitespace?
PP_And_Expression)*
;

fragment PP_And_Expression
: PP_Equality_Expression (PP_Whitespace? '&&' PP_Whitespace?
PP_Equality_Expression)*
;

fragment PP_Equality_Expression
: PP_Unary_Expression (PP_Whitespace? ('==' | '!=') PP_Whitespace?
PP_Unary_Expression)*
;

fragment PP_Unary_Expression
: PP_Primary_Expression
| '!' PP_Whitespace? PP_Unary_Expression
;

fragment PP_Primary_Expression
: TRUE
| FALSE
| PP_Conditional_Symbol
| '(' PP_Whitespace? PP_Expression PP_Whitespace? ')'
;

// Source: §6.5.4 Definition directives
fragment PP_Declaration
: 'define' PP_Whitespace PP_Conditional_Symbol
| 'undef' PP_Whitespace PP_Conditional_Symbol
;

// Source: §6.5.5 Conditional compilation directives
fragment PP_Conditional
: PP_If_Section
| PP_Elif_Section
| PP_Else_Section
| PP_Endif
;

fragment PP_If_Section
: 'if' PP_Whitespace PP_Expression
;

fragment PP_Elif_Section
: 'elif' PP_Whitespace PP_Expression
;

```

```

;

fragment PP_Else_Section
: 'else'
;

fragment PP_Endif
: 'endif'
;

// Source: §6.5.6 Diagnostic directives
fragment PP_Diagnostic
: 'error' PP_Message?
| 'warning' PP_Message?
;

fragment PP_Message
: PP_Whitespace Input_Character*
;

// Source: §6.5.7 Region directives
fragment PP_Region
: PP_Start_Region
| PP_End_Region
;

fragment PP_Start_Region
: 'region' PP_Message?
;

fragment PP_End_Region
: 'endregion' PP_Message?
;

// Source: §6.5.8 Line directives
fragment PP_Line
: 'line' PP_Whitespace PP_Line_Indicator
;

fragment PP_Line_Indicator
: Decimal_Digit+ PP_Whitespace PP_Compilation_Unit_Name
| Decimal_Digit+
| DEFAULT
| 'hidden'
;

fragment PP_Compilation_Unit_Name
: '"' PP_Compilation_Unit_Name_Character+ '"'
;

fragment PP_Compilation_Unit_Name_Character
// Any Input_Character except "
: ~(\u000D | \u000A | \u0085 | \u2028 | \u2029 | '#')
;

```

```
// Source: §6.5.9 Pragma directives
fragment PP_Pragma
    : 'pragma' PP_Pragma_Text?
    ;

fragment PP_Pragma_Text
    : PP_Whitespace Input_Character*
    ;
```

## A.3 Syntactic grammar

ANTLR

```
// Source: §7.8.1 General
namespace_name
    : namespace_or_type_name
    ;

type_name
    : namespace_or_type_name
    ;

namespace_or_type_name
    : identifier type_argument_list?
    | namespace_or_type_name '.' identifier type_argument_list?
    | qualified_alias_member
    ;

// Source: §8.1 General
type
    : reference_type
    | value_type
    | type_parameter
    | pointer_type      // unsafe code support
    ;

// Source: §8.2.1 General
reference_type
    : class_type
    | interface_type
    | array_type
    | delegate_type
    | 'dynamic'
    ;

class_type
    : type_name
    | 'object'
    | 'string'
    ;
```

```
interface_type
: type_name
;

array_type
: non_array_type rank_specifier+
;

non_array_type
: value_type
| class_type
| interface_type
| delegate_type
| 'dynamic'
| type_parameter
| pointer_type      // unsafe code support
;

rank_specifier
: '[' ',' '*' ']'
;

delegate_type
: type_name
;

// Source: §8.3.1 General
value_type
: non_nullable_value_type
| nullable_value_type
;

non_nullable_value_type
: struct_type
| enum_type
;

struct_type
: type_name
| simple_type
| tuple_type
;

simple_type
: numeric_type
| 'bool'
;

numeric_type
: integral_type
| floating_point_type
| 'decimal'
;

integral_type
```

```
: 'sbyte'
| 'byte'
| 'short'
| 'ushort'
| 'int'
| 'uint'
| 'long'
| 'ulong'
| 'char'
;

floating_point_type
: 'float'
| 'double'
;

tuple_type
: '(' tuple_type_element (',' tuple_type_element)* ')'
;

tuple_type_element
: type_identifier?
;

enum_type
: type_name
;

nullable_value_type
: non_nullable_value_type '?'
;

// Source: §8.4.2 Type arguments
type_argument_list
: '<' type_arguments '>'
;

type_arguments
: type_argument (',' type_argument)*
;

type_argument
: type
;

// Source: §8.5 Type parameters
type_parameter
: identifier
;

// Source: §8.8 Unmanaged types
unmanaged_type
: value_type
| pointer_type      // unsafe code support
;
```

```
// Source: §9.5 Variable references
variable_reference
    : expression
    ;

// Source: §11.2.1 General
pattern
    : declaration_pattern
    | constant_pattern
    | var_pattern
    ;

// Source: §11.2.2 Declaration pattern
declaration_pattern
    : type simple_designation
    ;
simple_designation
    : single_variable_designation
    ;
single_variable_designation
    : identifier
    ;

// Source: §11.2.3 Constant pattern
constant_pattern
    : constant_expression
    ;

// Source: §11.2.4 Var pattern
var_pattern
    : 'var' designation
    ;
designation
    : simple_designation
    ;

// Source: §12.6.2.1 General
argument_list
    : argument (',' argument)*
    ;

argument
    : argument_name? argument_value
    ;

argument_name
    : identifier ':'
    ;

argument_value
    : expression
    | 'in' variable_reference
    | 'ref' variable_reference
    | 'out' variable_reference
```

```

;

// Source: §12.8.1 General
primary_expression
: primary_no_array_creation_expression
| array_creation_expression
;

primary_no_array_creation_expression
: literal
| interpolated_string_expression
| simple_name
| parenthesized_expression
| tuple_expression
| member_access
| null_conditional_member_access
| invocation_expression
| element_access
| null_conditional_element_access
| this_access
| base_access
| post_increment_expression
| post_decrement_expression
| object_creation_expression
| delegate_creation_expression
| anonymous_object_creation_expression
| typeof_expression
| sizeof_expression
| checked_expression
| unchecked_expression
| default_value_expression
| nameof_expression
| anonymous_method_expression
| pointer_member_access      // unsafe code support
| pointer_element_access     // unsafe code support
| stackalloc_expression
;

// Source: §12.8.3 Interpolated string expressions
interpolated_string_expression
: interpolated_regular_string_expression
| interpolated_verbatim_string_expression
;

// interpolated regular string expressions

interpolated_regular_string_expression
: Interpolated-Regular-String-Start Interpolated-Regular-String-Mid?
  ('{' regular_interpolation '}' Interpolated-Regular-String-Mid?)*
  Interpolated-Regular-String-End
;

regular_interpolation
: expression (',' interpolation_minimum_width)?
  Regular_Interpolation_Format?
;
```

```

;

interpolation_minimum_width
: constant_expression
;

Interpolated-Regular-String-Start
: '$'
;

// the following three lexical rules are context sensitive, see details
below

Interpolated-Regular-String-Mid
: Interpolated-Regular-String-Element+
;

Regular-Interpolation-Format
: ':' Interpolated-Regular-String-Element+
;

Interpolated-Regular-String-End
: ''
;

fragment Interpolated-Regular-String-Element
: Interpolated-Regular-String-Character
| Simple_Escape_Sequence
| Hexadecimal_Escape_Sequence
| Unicode_Escape_Sequence
| Open_Brace_Escape_Sequence
| Close_Brace_Escape_Sequence
;

fragment Interpolated-Regular-String-Character
// Any character except " (U+0022), \\ (U+005C),
// { (U+007B), } (U+007D), and New_Line_Character.
: ~["\\{}\\u000D\\u000A\\u0085\\u2028\\u2029"]
;

// interpolated verbatim string expressions

interpolated_verbatim_string_expression
: Interpolated_Verbatim_String_Start Interpolated_Verbatim_String_Mid?
('{' verbatim_interpolation '}' Interpolated_Verbatim_String_Mid?)*
Interpolated_Verbatim_String_End
;

verbatim_interpolation
: expression (',' interpolation_minimum_width)?
Verbatim_Interpolation_Format?
;

Interpolated_Verbatim_String_Start
: '$@'
;

```

```

;

// the following three lexical rules are context sensitive, see details
below

Interpolated_Verbatim_String_Mid
: Interpolated_Verbatim_String_Element+
;

Verbatim_Interpolation_Format
: ':' Interpolated_Verbatim_String_Element+
;

Interpolated_Verbatim_String_End
: ''
;

fragment Interpolated_Verbatim_String_Element
: Interpolated_Verbatim_String_Character
| Quote_Escape_Sequence
| Open_Brace_Escape_Sequence
| Close_Brace_Escape_Sequence
;

fragment Interpolated_Verbatim_String_Character
: ~["{}"] // Any character except " (U+0022), { (U+007B) and }
(U+007D)
;

// lexical fragments used by both regular and verbatim interpolated strings

fragment Open_Brace_Escape_Sequence
: '{{'
;

fragment Close_Brace_Escape_Sequence
: '}}'
;

// Source: §12.8.4 Simple names
simple_name
: identifier type_argument_list?
;

// Source: §12.8.5 Parenthesized expressions
parenthesized_expression
: '(' expression ')'
;

// Source: §12.8.6 Tuple expressions
tuple_expression
: '(' tuple_element (',' tuple_element)+ ')'
| deconstruction_expression
;

```

```

tuple_element
: (identifier ':')? expression
;

deconstruction_expression
: 'var' deconstruction_tuple
;

deconstruction_tuple
: '(' deconstruction_element (',' deconstruction_element)+ ')'
;

deconstruction_element
: deconstruction_tuple
| identifier
;

// Source: §12.8.7.1 General
member_access
: primary_expression '.' identifier type_argument_list?
| predefined_type '.' identifier type_argument_list?
| qualified_alias_member '.' identifier type_argument_list?
;

predefined_type
: 'bool' | 'byte' | 'char' | 'decimal' | 'double' | 'float' | 'int'
| 'long' | 'object' | 'sbyte' | 'short' | 'string' | 'uint' | 'ulong'
| 'ushort'
;

// Source: §12.8.8 Null Conditional Member Access
null_conditional_member_access
: primary_expression '?' '.' identifier type_argument_list?
dependent_access*
;

dependent_access
: '.' identifier type_argument_list?           // member access
| '[' argument_list ']'                      // element access
| '(' argument_list? ')'                     // invocation
;

null_conditional_projection_initializer
: primary_expression '?' '.' identifier type_argument_list?
;

// Source: §12.8.9.1 General
invocation_expression
: primary_expression '(' argument_list? ')'
;

// Source: §12.8.10 Null Conditional Invocation Expression
null_conditional_invocation_expression
: null_conditional_member_access '(' argument_list? ')'
| null_conditional_element_access '(' argument_list? ')'
;

```

```
 ;

// Source: §12.8.11.1 General
element_access
: primary_no_array_creation_expression '[' argument_list ']'
;

// Source: §12.8.12 Null Conditional Element Access
null_conditional_element_access
: primary_no_array_creation_expression '?' '[' argument_list ']'
dependent_access*
;
;

// Source: §12.8.13 This access
this_access
: 'this'
;
;

// Source: §12.8.14 Base access
base_access
: 'base' '.' identifier type_argument_list?
| 'base' '[' argument_list ']'
;
;

// Source: §12.8.15 Postfix increment and decrement operators
post_increment_expression
: primary_expression '++'
;
;

post_decrement_expression
: primary_expression '--'
;
;

// Source: §12.8.16.2 Object creation expressions
object_creation_expression
: 'new' type '(' argument_list? ')' object_or_collection_initializer?
| 'new' type object_or_collection_initializer
;
;

object_or_collection_initializer
: object_initializer
| collection_initializer
;
;

// Source: §12.8.16.3 Object initializers
object_initializer
: '{' member_initializer_list? '}'
| '{' member_initializer_list ',' '}'
;
;

member_initializer_list
: member_initializer (',' member_initializer)*
;
;

member_initializer
```

```
: initializer_target '=' initializer_value
;

initializer_target
: identifier
| '[' argument_list ']'
;

initializer_value
: expression
| object_or_collection_initializer
;

// Source: §12.8.16.4 Collection initializers
collection_initializer
: '{' element_initializer_list '}'
| '{' element_initializer_list ',' '}'
;

element_initializer_list
: element_initializer (',' element_initializer)*
;

element_initializer
: non_assignment_expression
| '{' expression_list '}'
;

expression_list
: expression
| expression_list ',' expression
;

// Source: §12.8.16.5 Array creation expressions
array_creation_expression
: 'new' non_array_type '[' expression_list ']' rank_specifier*
  array_initializer?
| 'new' array_type array_initializer
| 'new' rank_specifier array_initializer
;

// Source: §12.8.16.6 Delegate creation expressions
delegate_creation_expression
: 'new' delegate_type '(' expression ')'
;

// Source: §12.8.16.7 Anonymous object creation expressions
anonymous_object_creation_expression
: 'new' anonymous_object_initializer
;

anonymous_object_initializer
: '{' member_declarator_list? '}'
| '{' member_declarator_list ',' '}'
;
```

```
member_declarator_list
: member_declarator (',' member_declarator)*
;

member_declarator
: simple_name
| member_access
| null_conditional_projection_initializer
| base_access
| identifier '=' expression
;

// Source: §12.8.17 The sizeof operator
typeof_expression
: 'typeof' '(' type ')'
| 'typeof' '(' unbound_type_name ')'
| 'typeof' '(' 'void' ')'
;

unbound_type_name
: identifier generic_dimension_specifier?
| identifier '::' identifier generic_dimension_specifier?
| unbound_type_name '.' identifier generic_dimension_specifier?
;

generic_dimension_specifier
: '<' comma* '>'
;

comma
: ','
;

// Source: §12.8.18 The sizeof operator
sizeof_expression
: 'sizeof' '(' unmanaged_type ')'
;

// Source: §12.8.19 The checked and unchecked operators
checked_expression
: 'checked' '(' expression ')'
;

unchecked_expression
: 'unchecked' '(' expression ')'
;

// Source: §12.8.20 Default value expressions
default_value_expression
: explicitly_typed_default
| default_literal
;
```

```
explicitly_typed_default
: 'default' '(' type ')'
;

default_literal
: 'default'
;

// Source: §12.8.21 Stack allocation
stackalloc_expression
: 'stackalloc' unmanaged_type '[' expression ']'
| 'stackalloc' unmanaged_type? '[' constant_expression? ']'
stackalloc_initializer
;
stackalloc_initializer
: '{' stackalloc_initializer_element_list '}'
;

stackalloc_initializer_element_list
: stackalloc_element_initializer (',' stackalloc_element_initializer)*
', '?'
;

stackalloc_element_initializer
: expression
;

// Source: §12.8.22 Nameof expressions
nameof_expression
: 'nameof' '(' named_entity ')'
;

named_entity
: named_entity_target ('.' identifier type_argument_list?)*
;

named_entity_target
: simple_name
| 'this'
| 'base'
| predefined_type
| qualified_alias_member
;

// Source: §12.9.1 General
unary_expression
: primary_expression
| '+' unary_expression
| '-' unary_expression
| '!' unary_expression
| '~' unary_expression
| pre_increment_expression
| pre_decrement_expression
| cast_expression
;
```

```
| await_expression
| pointer_indirection_expression      // unsafe code support
| addressof_expression               // unsafe code support
;

// Source: §12.9.6 Prefix increment and decrement operators
pre_increment_expression
: '++' unary_expression
;

pre_decrement_expression
: '--' unary_expression
;

// Source: §12.9.7 Cast expressions
cast_expression
: '(' type ')' unary_expression
;

// Source: §12.9.8.1 General
await_expression
: 'await' unary_expression
;

// Source: §12.10.1 General
multiplicative_expression
: unary_expression
| multiplicative_expression '*' unary_expression
| multiplicative_expression '/' unary_expression
| multiplicative_expression '%' unary_expression
;

additive_expression
: multiplicative_expression
| additive_expression '+' multiplicative_expression
| additive_expression '-' multiplicative_expression
;

// Source: §12.11 Shift operators
shift_expression
: additive_expression
| shift_expression '<<' additive_expression
| shift_expression right_shift additive_expression
;

// Source: §12.12.1 General
relational_expression
: shift_expression
| relational_expression '<' shift_expression
| relational_expression '>' shift_expression
| relational_expression '<=' shift_expression
| relational_expression '>=' shift_expression
| relational_expression 'is' type
| relational_expression 'is' pattern
| relational_expression 'as' type
```

```
 ;

equality_expression
: relational_expression
| equality_expression '==' relational_expression
| equality_expression '!=' relational_expression
;

// Source: §12.13.1 General
and_expression
: equality_expression
| and_expression '&' equality_expression
;

exclusive_or_expression
: and_expression
| exclusive_or_expression '^' and_expression
;

inclusive_or_expression
: exclusive_or_expression
| inclusive_or_expression '|'| exclusive_or_expression
;

// Source: §12.14.1 General
conditional_and_expression
: inclusive_or_expression
| conditional_and_expression '&&' inclusive_or_expression
;

conditional_or_expression
: conditional_and_expression
| conditional_or_expression '||' conditional_and_expression
;

// Source: §12.15 The null coalescing operator
null_coalescing_expression
: conditional_or_expression
| conditional_or_expression '??' null_coalescing_expression
| throw_expression
;

// Source: §12.16 The throw expression operator
throw_expression
: 'throw' null_coalescing_expression
;

// Source: §12.17 Declaration expressions
declaration_expression
: local_variable_type identifier
;

// Source: §12.18 Conditional operator
conditional_expression
: null_coalescing_expression
```

```
| null_coalescing_expression '?' expression ':' expression
| null_coalescing_expression '?' 'ref' variable_reference ':' 'ref'
variable_reference
;

// Source: §12.19.1 General
lambda_expression
: 'async'? anonymous_function_signature '=>' anonymous_function_body
;

anonymous_method_expression
: 'async'? 'delegate' explicit_anonymous_function_signature? block
;

anonymous_function_signature
: explicit_anonymous_function_signature
| implicit_anonymous_function_signature
;

explicit_anonymous_function_signature
: '(' explicit_anonymous_function_parameter_list? ')'
;

explicit_anonymous_function_parameter_list
: explicit_anonymous_function_parameter
( ',' explicit_anonymous_function_parameter)*
;

explicit_anonymous_function_parameter
: anonymous_function_parameter_modifier? type_identifier
;

anonymous_function_parameter_modifier
: 'ref'
| 'out'
| 'in'
;

implicit_anonymous_function_signature
: '(' implicit_anonymous_function_parameter_list? ')'
| implicit_anonymous_function_parameter
;

implicit_anonymous_function_parameter_list
: implicit_anonymous_function_parameter
( ',' implicit_anonymous_function_parameter)*
;

implicit_anonymous_function_parameter
: identifier
;

anonymous_function_body
: null_conditional_invocation_expression
| expression
;
```

```
| 'ref' variable_reference
| block
;

// Source: §12.20.1 General
query_expression
: from_clause query_body
;

from_clause
: 'from' type? identifier 'in' expression
;

query_body
: query_body_clauses? select_or_group_clause query_continuation?
;

query_body_clauses
: query_body_clause
| query_body_clauses query_body_clause
;

query_body_clause
: from_clause
| let_clause
| where_clause
| join_clause
| join_into_clause
| orderby_clause
;

let_clause
: 'let' identifier '=' expression
;

where_clause
: 'where' boolean_expression
;

join_clause
: 'join' type? identifier 'in' expression 'on' expression
  'equals' expression
;
;

join_into_clause
: 'join' type? identifier 'in' expression 'on' expression
  'equals' expression 'into' identifier
;
;

orderby_clause
: 'orderby' orderings
;
;

orderings
: ordering (',' ordering)*
```

```
 ;

ordering
: expression ordering_direction?
;

ordering_direction
: 'ascending'
| 'descending'
;

select_or_group_clause
: select_clause
| group_clause
;

select_clause
: 'select' expression
;

group_clause
: 'group' expression 'by' expression
;

query_continuation
: 'into' identifier query_body
;

// Source: §12.21.1 General
assignment
: unary_expression assignment_operator expression
;

assignment_operator
: '=' 'ref'? | '+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
'|<<='
| right_shift_assignment
;

// Source: §12.22 Expression
expression
: non_assignment_expression
| assignment
;

non_assignment_expression
: declaration_expression
| conditional_expression
| lambda_expression
| query_expression
;

// Source: §12.23 Constant expressions
constant_expression
: expression
```

```
 ;

// Source: §12.24 Boolean expressions
boolean_expression
    : expression
    ;

// Source: §13.1 General
statement
    : labeled_statement
    | declaration_statement
    | embedded_statement
    ;

embedded_statement
    : block
    | empty_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    | try_statement
    | checked_statement
    | unchecked_statement
    | lock_statement
    | using_statement
    | yield_statement
    | unsafe_statement    // unsafe code support
    | fixed_statement     // unsafe code support
    ;

// Source: §13.3.1 General
block
    : '{' statement_list? '}'
    ;

// Source: §13.3.2 Statement lists
statement_list
    : statement+
    ;

// Source: §13.4 The empty statement
empty_statement
    : ';'
    ;

// Source: §13.5 Labeled statements
labeled_statement
    : identifier ':' statement
    ;

// Source: §13.6.1 General
declaration_statement
    : local_variable_declaration ';'
    | local_constant_declaration ';'
    ;
```

```

| local_function_declarator
;

// Source: §13.6.2 Local variable declarations
local_variable_declarator
: ref_kind? local_variable_type local_variable_declarators
;

local_variable_type
: type
| 'var'
;

local_variable_declarators
: local_variable_declarator
| local_variable_declarators ',' local_variable_declarator
;

local_variable_declarator
: identifier
| identifier '=' local_variable_initializer
;

local_variable_initializer
: expression
| 'ref' variable_reference
| array_initializer
;

// Source: §13.6.3 Local constant declarations
local_constant_declarator
: 'const' type constant_declarators
;

constant_declarators
: constant_declarator (',' constant_declarator)*
;

constant_declarator
: identifier '=' constant_expression
;

// Source: §13.6.4 Local function declarations
local_function_declarator
: local_function_modifier* return_type local_function_header
local_function_body
| ref_kind ref_return_type local_function_header ref_local_function_body
;

local_function_header
: identifier '(' formal_parameter_list? ')'
| identifier type_parameter_list '(' formal_parameter_list? ')'
type_parameter_constraints_clause*
;


```

```
local_function_modifier
: 'async'
| unsafe_modifier // unsafe code support
;

local_function_body
: block
| '>' null_conditional_invocation_expression ';'
| '>' expression ';'
;

ref_local_function_body
: block
| '>' 'ref' variable_reference ';'
;

// Source: §13.7 Expression statements
expression_statement
: statement_expression ';'
;

statement_expression
: null_conditional_invocation_expression
| invocation_expression
| object_creation_expression
| assignment
| post_increment_expression
| post_decrement_expression
| pre_increment_expression
| pre_decrement_expression
| await_expression
;
;

// Source: §13.8.1 General
selection_statement
: if_statement
| switch_statement
;
;

// Source: §13.8.2 The if statement
if_statement
: 'if' '(' boolean_expression ')' embedded_statement
| 'if' '(' boolean_expression ')' embedded_statement
  'else' embedded_statement
;
;

// Source: §13.8.3 The switch statement
switch_statement
: 'switch' '(' expression ')' switch_block
;
;

switch_block
: '{' switch_section* '}'
;
;
```

```
switch_section
: switch_label+ statement_list
;

switch_label
: 'case' pattern case_guard? ':'
| 'default' ':'
;

case_guard
: 'when' expression
;

// Source: §13.9.1 General
iteration_statement
: while_statement
| do_statement
| for_statement
| foreach_statement
;

// Source: §13.9.2 The while statement
while_statement
: 'while' '(' boolean_expression ')' embedded_statement
;

// Source: §13.9.3 The do statement
do_statement
: 'do' embedded_statement 'while' '(' boolean_expression ')' ';'
;

// Source: §13.9.4 The for statement
for_statement
: 'for' '(' for_initializer? ';' for_condition? ';' for_iterator? ')'
embedded_statement
;

for_initializer
: local_variable_declaration
| statement_expression_list
;

for_condition
: boolean_expression
;

for_iterator
: statement_expression_list
;

statement_expression_list
: statement_expression (',' statement_expression)*
;

// Source: §13.9.5 The foreach statement
```

```
foreach_statement
: 'foreach' '(' ref_kind? local_variable_type identifier 'in'
  expression ')' embedded_statement
;

// Source: §13.10.1 General
jump_statement
: break_statement
| continue_statement
| goto_statement
| return_statement
| throw_statement
;

// Source: §13.10.2 The break statement
break_statement
: 'break' ';'
;

// Source: §13.10.3 The continue statement
continue_statement
: 'continue' ';'
;

// Source: §13.10.4 The goto statement
goto_statement
: 'goto' identifier ';'
| 'goto' 'case' constant_expression ';'
| 'goto' 'default' ';'
;

// Source: §13.10.5 The return statement
return_statement
: 'return' ';'
| 'return' expression ';'
| 'return' 'ref' variable_reference ';'
;

// Source: §13.10.6 The throw statement
throw_statement
: 'throw' expression? ';'
;

// Source: §13.11 The try statement
try_statement
: 'try' block catch_clauses
| 'try' block catch_clauses? finally_clause
;

catch_clauses
: specificCatchClause+
| specificCatchClause* generalCatchClause
;

specificCatchClause
```

```
: 'catch' exception_specifier exception_filter? block
| 'catch' exception_filter block
;

exceptionSpecifier
: '(' type_identifier? ')'
;

exceptionFilter
: 'when' '(' boolean_expression ')'
;

generalCatchClause
: 'catch' block
;

finallyClause
: 'finally' block
;

// Source: §13.12 The checked and unchecked statements
checkedStatement
: 'checked' block
;

uncheckedStatement
: 'unchecked' block
;

// Source: §13.13 The lock statement
lockStatement
: 'lock' '(' expression ')' embeddedStatement
;

// Source: §13.14 The using statement
usingStatement
: 'using' '(' resource_acquisition ')' embeddedStatement
;

resourceAcquisition
: local_variable_declaration
| expression
;

// Source: §13.15 The yield statement
yieldStatement
: 'yield' 'return' expression ';'
| 'yield' 'break' ';'
;

// Source: §14.2 Compilation units
compilationUnit
: extern_alias_directive* using_directive* global_attributes?
  namespace_member_declaratoin*
;
```

```
// Source: §14.3 Namespace declarations
namespace_declarator
: 'namespace' qualified_identifier namespace_body ';'?
;

qualified_identifier
: identifier ('.' identifier)*
;

namespace_body
: '{' extern_alias_directive* using_directive*
  namespace_member_declarator* '}'
;

// Source: §14.4 Extern alias directives
extern_alias_directive
: 'extern' 'alias' identifier ';'*
;

// Source: §14.5.1 General
using_directive
: using_alias_directive
| using_namespace_directive
| using_static_directive
;

// Source: §14.5.2 Using alias directives
using_alias_directive
: 'using' identifier '=' namespace_or_type_name ';'*
;

// Source: §14.5.3 Using namespace directives
using_namespace_directive
: 'using' namespace_name ';'*
;

// Source: §14.5.4 Using static directives
using_static_directive
: 'using' 'static' type_name ';'*
;

// Source: §14.6 Namespace member declarations
namespace_member_declarator
: namespace_declarator
| type_declarator
;

// Source: §14.7 Type declarations
type_declarator
: class_declarator
| struct_declarator
| interface_declarator
| enum_declarator
| delegate_declarator
;
```

```
 ;

// Source: §14.8.1 General
qualified_alias_member
: identifier '::' identifier type_argument_list?
;

// Source: §15.2.1 General
class_declarator
: attributes? class_modifier* 'partial'? 'class' identifier
  type_parameter_list? class_base? type_parameter_constraints_clause*
  class_body ';'?
;

// Source: §15.2.2.1 General
class_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'abstract'
| 'sealed'
| 'static'
| unsafe_modifier // unsafe code support
;

// Source: §15.2.3 Type parameters
type_parameter_list
: '<' type_parameters '>'
;

type_parameters
: attributes? type_parameter
| type_parameters ',' attributes? type_parameter
;

// Source: §15.2.4.1 General
class_base
: ':' class_type
| ':' interface_type_list
| ':' class_type ',' interface_type_list
;

interface_type_list
: interface_type (',' interface_type)*
;

// Source: §15.2.5 Type parameter constraints
type_parameter_constraints_clauses
: type_parameter_constraints_clause
| type_parameter_constraints_clauses type_parameter_constraints_clause
;

type_parameter_constraints_clause
```

```
: 'where' type_parameter ':' type_parameter_constraints
;

type_parameter_constraints
: primary_constraint
| secondary_constraints
| constructor_constraint
| primary_constraint ',' secondary_constraints
| primary_constraint ',' constructor_constraint
| secondary_constraints ',' constructor_constraint
| primary_constraint ',' secondary_constraints ',' constructor_constraint
;
constructor_constraint
;

primary_constraint
: class_type
| 'class'
| 'struct'
| 'unmanaged'
;
secondary_constraints
: interface_type
| type_parameter
| secondary_constraints ',' interface_type
| secondary_constraints ',' type_parameter
;
constructor_constraint
: 'new' '(' ')'
;
// Source: §15.2.6 Class body
class_body
: '{' class_member_declaration* '}'
;
// Source: §15.3.1 General
class_member_declaration
: constant_declaration
| field_declaration
| method_declaration
| property_declaration
| event_declaration
| indexer_declaration
| operator_declaration
| constructor_declaration
| finalizer_declaration
| static_constructor_declaration
| type_declaration
;
// Source: §15.4 Constants
constant_declaration
: attributes? constant_modifier* 'const' type constant_declarators ';'
```

```
;

constant_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
;

// Source: §15.5.1 General
field_declaration
: attributes? field_modifier* type variable_declarators ';'
;

field_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'static'
| 'readonly'
| 'volatile'
| unsafe_modifier // unsafe code support
;

variable_declarators
: variable_declarator (',' variable_declarator)*
;

variable_declarator
: identifier ('=' variable_initializer)?
;

// Source: §15.6.1 General
method_declaration
: attributes? method_modifiers return_type method_header method_body
| attributes? ref_method_modifiers ref_kind ref_return_type
method_header ref_method_body
;

method_modifiers
: method_modifier* 'partial'?
;

ref_kind
: 'ref'
| 'ref' 'readonly'
;

ref_method_modifiers
: ref_method_modifier*
;
```

```
method_header
: member_name '(' formal_parameter_list? ')'
| member_name type_parameter_list '(' formal_parameter_list? ')'
type_parameter_constraints_clause*
;

method_modifier
: ref_method_modifier
| 'async'
| unsafe_modifier // unsafe code support
;

ref_method_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'static'
| 'virtual'
| 'sealed'
| 'override'
| 'abstract'
| 'extern'
;

return_type
: ref_return_type
| 'void'
;

ref_return_type
: type
;

member_name
: identifier
| interface_type '.' identifier
;

method_body
: block
| '>' null_conditional_invocation_expression ';'
| '>' expression ';'
| ';'
;

ref_method_body
: block
| '>' 'ref' variable_reference ';'
| ';'
;

// Source: §15.6.2.1 General
formal_parameter_list
```

```
: fixed_parameters
| fixed_parameters ',' parameter_array
| parameter_array
;

fixed_parameters
: fixed_parameter (',' fixed_parameter)*
;

fixed_parameter
: attributes? parameter_modifier? type identifier default_argument?
;

default_argument
: '=' expression
;

parameter_modifier
: parameter_mode_modifier
| 'this'
;

parameter_mode_modifier
: 'ref'
| 'out'
| 'in'
;

parameter_array
: attributes? 'params' array_type identifier
;

// Source: §15.7.1 General
property_declaration
: attributes? property_modifier* type member_name property_body
| attributes? property_modifier* ref_kind type member_name
ref_property_body
;

property_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'static'
| 'virtual'
| 'sealed'
| 'override'
| 'abstract'
| 'extern'
| unsafe_modifier // unsafe code support
;

property_body
```

```
: '{' accessor_declarations '}' property_initializer?
| '>' expression ';'
;

property_initializer
: '=' variable_initializer ';'
;

ref_property_body
: '{' ref_get_accessor_declaration '}'
| '>' 'ref' variable_reference ';'
;

// Source: §15.7.3 Accessors
accessor_declarations
: get_accessor_declaration set_accessor_declaration?
| set_accessor_declaration get_accessor_declaration?
;

get_accessor_declaration
: attributes? accessor_modifier? 'get' accessor_body
;

set_accessor_declaration
: attributes? accessor_modifier? 'set' accessor_body
;

accessor_modifier
: 'protected'
| 'internal'
| 'private'
| 'protected' 'internal'
| 'internal' 'protected'
| 'protected' 'private'
| 'private' 'protected'
;

accessor_body
: block
| '>' expression ';'
| ';'
;

ref_get_accessor_declaration
: attributes? accessor_modifier? 'get' ref_accessor_body
;

ref_accessor_body
: block
| '>' 'ref' variable_reference ';'
| ';'
;

// Source: §15.8.1 General
event_declaration
```

```
: attributes? event_modifier* 'event' type variable_declarators ';'
| attributes? event_modifier* 'event' type member_name
  '{' event_accessor_declarations '}'
;

event_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'static'
| 'virtual'
| 'sealed'
| 'override'
| 'abstract'
| 'extern'
| unsafe_modifier // unsafe code support
;

event_accessor_declarations
: add_accessor_declaration remove_accessor_declaration
| remove_accessor_declaration add_accessor_declaration
;

add_accessor_declaration
: attributes? 'add' block
;

remove_accessor_declaration
: attributes? 'remove' block
;

// Source: §15.9.1 General
indexer_declaration
: attributes? indexer_modifier* indexer_declarator indexer_body
| attributes? indexer_modifier* ref_kind indexer_declarator
ref_indexer_body
;

indexer_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| 'virtual'
| 'sealed'
| 'override'
| 'abstract'
| 'extern'
| unsafe_modifier // unsafe code support
;

indexer_declarator
```

```

: type 'this' '[' formal_parameter_list ']'
| type interface_type '.' 'this' '[' formal_parameter_list ']'
;

indexer_body
: '{' accessor_declarations '}'
| '>' expression ';'
;

ref_indexer_body
: '{' ref_get_accessor_declaration '}'
| '>' 'ref' variable_reference ';'
;

// Source: §15.10.1 General
operator_declaration
: attributes? operator_modifier+ operator_declarator operator_body
;

operator_modifier
: 'public'
| 'static'
| 'extern'
| unsafe_modifier // unsafe code support
;

operator_declarator
: unary_operator_declarator
| binary_operator_declarator
| conversion_operator_declarator
;

unary_operator_declarator
: type 'operator' overloadable_unary_operator '(' fixed_parameter ')'
;

overloadable_unary_operator
: '+' | '-' | '!' | '~' | '++' | '--' | 'true' | 'false'
;

binary_operator_declarator
: type 'operator' overloadable_binary_operator
  '(' fixed_parameter ',' fixed_parameter ')'
;

overloadable_binary_operator
: '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '<<'
| right_shift | '==' | '!='
| '>' | '<' | '>=' | '<='
;

conversion_operator_declarator
: 'implicit' 'operator' type '(' fixed_parameter ')'
| 'explicit' 'operator' type '(' fixed_parameter ')'
;

```

```

operator_body
: block
| '>' expression ';'
| ';'
;

// Source: §15.11.1 General

constructor_declaration
: attributes? constructor_modifier* constructor_declarator
constructor_body
;

constructor_modifier
: 'public'
| 'protected'
| 'internal'
| 'private'
| 'extern'
| unsafe_modifier // unsafe code support
;

constructor_declarator
: identifier '(' formal_parameter_list? ')' constructor_initializer?
;

constructor_initializer
: ':' 'base' '(' argument_list? ')'
| ':' 'this' '(' argument_list? ')'
;

constructor_body
: block
| '>' expression ';'
| ';'
;

// Source: §15.12 Static constructors

static_constructor_declaration
: attributes? static_constructor_modifiers identifier '(' ')'
    static_constructor_body
;

static_constructor_modifiers
: 'static'
| 'static' 'extern' unsafe_modifier?
| 'static' unsafe_modifier 'extern'?
| 'extern' 'static' unsafe_modifier?
| 'extern' unsafe_modifier 'static'
| unsafe_modifier 'static' 'extern'?
| unsafe_modifier 'extern' 'static'
;

static_constructor_body
: block
| '>' expression ';'
;

```

```
| ';'  
;  
  
// Source: §15.13 Finalizers  
finalizer_declaration  
: attributes? '~' identifier '(' ')' finalizer_body  
| attributes? 'extern' unsafe_modifier? '~' identifier '(' ')' finalizer_body  
| attributes? unsafe_modifier 'extern'? '~' identifier '(' ')' finalizer_body  
;  
  
finalizer_body  
: block  
| '=>' expression ';'  
| ';'  
;  
  
// Source: §16.2.1 General  
struct_declaration  
: attributes? struct_modifier* 'ref'? 'partial'? 'struct'  
    identifier type_parameter_list? struct_interfaces?  
    type_parameter_constraints_clause* struct_body ';'?  
;  
  
// Source: §16.2.2 Struct modifiers  
struct_modifier  
: 'new'  
| 'public'  
| 'protected'  
| 'internal'  
| 'private'  
| 'readonly'  
| unsafe_modifier // unsafe code support  
;  
  
// Source: §16.2.5 Struct interfaces  
struct_interfaces  
: ':' interface_type_list  
;  
  
// Source: §16.2.6 Struct body  
struct_body  
: '{' struct_member_declarator* '}'  
;  
  
// Source: §16.3 Struct members  
struct_member_declarator  
: constant_declarator  
| field_declarator  
| method_declarator  
| property_declarator  
| event_declarator  
| indexer_declarator  
| operator_declarator
```

```

| constructor_declarator
| static_constructor_declarator
| type_declarator
| fixed_size_buffer_declarator // unsafe code support
;

// Source: §17.7 Array initializers
array_initializer
: '{' variable_initializer_list? '}'
| '{' variable_initializer_list ',' '}'
;

variable_initializer_list
: variable_initializer (',' variable_initializer)*
;

variable_initializer
: expression
| array_initializer
;

// Source: §18.2.1 General
interface_declarator
: attributes? interface_modifier* 'partial'? 'interface'
  identifier variant_type_parameter_list? interface_base?
  type_parameter_constraints_clause* interface_body ';'?
;

// Source: §18.2.2 Interface modifiers
interface_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
| unsafe_modifier // unsafe code support
;

// Source: §18.2.3.1 General
variant_type_parameter_list
: '<' variant_type_parameters '>'
;

// Source: §18.2.3.1 General
variant_type_parameters
: attributes? variance_annotation? type_parameter
| variant_type_parameters ',' attributes? variance_annotation?
  type_parameter
;

// Source: §18.2.3.1 General
variance_annotation
: 'in'
| 'out'
;

```

```
// Source: §18.2.4 Base interfaces
interface_base
  : ':' interface_type_list
;

// Source: §18.3 Interface body
interface_body
  : '{' interface_member_declarator* '}'
;

// Source: §18.4.1 General
interface_member_declarator
  : interface_method_declarator
| interface_property_declarator
| interface_event_declarator
| interface_indexer_declarator
;

// Source: §18.4.2 Interface methods
interface_method_declarator
  : attributes? 'new'? return_type interface_method_header
| attributes? 'new'? ref_kind ref_return_type interface_method_header
;

interface_method_header
  : identifier '(' formal_parameter_list? ')' ';'
| identifier type_parameter_list '(' formal_parameter_list? ')'
type_parameter_constraints_clause* ';'
;

// Source: §18.4.3 Interface properties
interface_property_declarator
  : attributes? 'new'? type identifier '{' interface_accessors '}'
| attributes? 'new'? ref_kind type identifier '{' ref_interface_accessor '}'
;

interface_accessors
  : attributes? 'get' ';'
| attributes? 'set' ';'
| attributes? 'get' ';' attributes? 'set' ';'
| attributes? 'set' ';' attributes? 'get' ';'
;

ref_interface_accessor
  : attributes? 'get' ';'
;

// Source: §18.4.4 Interface events
interface_event_declarator
  : attributes? 'new'? 'event' type identifier ';'
;

// Source: §18.4.5 Interface indexers
```

```

interface_indexer_declaration
: attributes? 'new'? type 'this' '[' formal_parameter_list ']'
  '{' interface_accessors '}'
| attributes? 'new'? ref_kind type 'this' '[' formal_parameter_list ']'
  '{' ref_interface_accessor '}'
;

// Source: §19.2 Enum declarations
enum_declaration
: attributes? enum_modifier* 'enum' identifier enum_base? enum_body ';'?
;

enum_base
: ':' integral_type
| ':' integral_type_name
;

integral_type_name
: type_name // Shall resolve to an integral type other than char
;

enum_body
: '{' enum_member_declarations? '}'
| '{' enum_member_declarations ',' '}'
;

// Source: §19.3 Enum modifiers
enum_modifier
: 'new'
| 'public'
| 'protected'
| 'internal'
| 'private'
;

// Source: §19.4 Enum members
enum_member_declarations
: enum_member_declaration (',' enum_member_declaration)*
;

// Source: §19.4 Enum members
enum_member_declaration
: attributes? identifier ('=' constant_expression)?
;

// Source: §20.2 Delegate declarations
delegate_declaration
: attributes? delegate_modifier* 'delegate' return_type delegate_header
| attributes? ref_delegate_modifier* 'delegate' ref_kind ref_return_type
delegate_header
;

delegate_header
: identifier '(' formal_parameter_list? ')' ';'
| identifier variant_type_parameter_list '(' formal_parameter_list? ')'
;

```

```
type_parameter_constraints_clause* ';'  
;  
  
delegate_modifier  
: ref_delegate_modifier  
| unsafe_modifier // unsafe code support  
;  
  
ref_delegate_modifier  
: 'new'  
| 'public'  
| 'protected'  
| 'internal'  
| 'private'  
;  
  
// Source: §22.3 Attribute specification  
global_attributes  
: global_attribute_section+  
;  
  
global_attribute_section  
: '[' global_attribute_targetSpecifier attribute_list ']'  
| '[' global_attribute_targetSpecifier attribute_list ',' ']'  
;  
  
global_attribute_targetSpecifier  
: global_attribute_target ':'  
;  
  
global_attribute_target  
: identifier  
;  
  
attributes  
: attribute_section+  
;  
  
attribute_section  
: '[' attribute_targetSpecifier? attribute_list ']'  
| '[' attribute_targetSpecifier? attribute_list ',' ']'  
;  
  
attribute_targetSpecifier  
: attribute_target ':'  
;  
  
attribute_target  
: identifier  
| keyword  
;  
  
attribute_list  
: attribute (',' attribute)*  
;
```

```

attribute
: attribute_name attribute_arguments?
;

attribute_name
: type_name
;

attribute_arguments
: '(' positional_argument_list? ')'
| '(' positional_argument_list ',' named_argument_list ')'
| '(' named_argument_list ')'
;

positional_argument_list
: positional_argument (',' positional_argument)*
;

positional_argument
: argument_name? attribute_argument_expression
;

named_argument_list
: named_argument (',' named_argument)*
;

named_argument
: identifier '=' attribute_argument_expression
;

attribute_argument_expression
: expression
;

```

## A.4 Grammar extensions for unsafe code

ANTLR

```

// Source: §23.2 Unsafe contexts
unsafe_modifier
: 'unsafe'
;

unsafe_statement
: 'unsafe' block
;

// Source: §23.3 Pointer types
pointer_type
: value_type ('*')+
```

```
| 'void' ('*')+  
;  
  
// Source: §23.6.2 Pointer indirection  
pointer_indirection_expression  
: '*' unary_expression  
;  
  
// Source: §23.6.3 Pointer member access  
pointer_member_access  
: primary_expression '->' identifier type_argument_list?  
;  
  
// Source: §23.6.4 Pointer element access  
pointer_element_access  
: primary_no_array_creation_expression '[' expression ']'  
;  
  
// Source: §23.6.5 The address-of operator  
addressof_expression  
: '&' unary_expression  
;  
  
// Source: §23.7 The fixed statement  
fixed_statement  
: 'fixed' '(' pointer_type fixed_pointer_declarators ')' embedded_statement  
;  
  
fixed_pointer_declarators  
: fixed_pointer_declarator (',' fixed_pointer_declarator)*  
;  
  
fixed_pointer_declarator  
: identifier '=' fixed_pointer_initializer  
;  
  
fixed_pointer_initializer  
: '&' variable_reference  
| expression  
;  
  
// Source: §23.8.2 Fixed-size buffer declarations  
fixed_size_buffer_declaration  
: attributes? fixed_size_buffer_modifier* 'fixed' buffer_element_type fixed_size_buffer_declarators ';'   
;  
  
fixed_size_buffer_modifier  
: 'new'  
| 'public'  
| 'internal'  
| 'private'  
| 'unsafe'  
;
```

```
buffer_element_type
: type
;

fixed_size_buffer_declarators
: fixed_size_buffer_declarator (',' fixed_size_buffer_declarator)*
;

fixed_size_buffer_declarator
: identifier '[' constant_expression ']'
;
```

End of informative text.

# Annex B Portability issues

Article • 07/14/2023

This clause is informative.

## B.1 General

This annex collects some information about portability that appears in this specification.

## B.2 Undefined behavior

The behavior is undefined in the following circumstances:

1. The behavior of the enclosing `async` function when an awainer's implementation of the interface methods `INotifyCompletion.OnCompleted` and `ICriticalNotifyCompletion.UnsafeOnCompleted` does not cause the resumption delegate to be invoked at most once ([§12.9.8.4](#)).
2. Passing pointers as `ref` or `out` parameters ([§23.3](#)).
3. When dereferencing the result of converting one pointer type to another and the resulting pointer is not correctly aligned for the pointed-to type. ([§23.5.1](#)).
4. When the unary `*` operator is applied to a pointer containing an invalid value ([§23.6.2](#)).
5. When a pointer is subscripted to access an out-of-bounds element ([§23.6.4](#)).
6. Modifying objects of managed type through fixed pointers ([§23.7](#)).
7. The content of memory newly allocated by `stackalloc` ([§12.8.21](#)).
8. Attempting to allocate a negative number of items using `stackalloc` ([§12.8.21](#)).
9. Implicit dynamic conversions ([§10.2.10](#)) of `in` parameters with value arguments ([§12.6.4.2](#)).

## B.3 Implementation-defined behavior

A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:

1. The behavior when an identifier not in Normalization Form C is encountered ([§6.4.3](#)).
2. The maximum value allowed for `Decimal_Digit+` in `PP_Line_Indicator` ([§6.5.8](#)).
3. The interpretation of the `input_characters` in the `pp_pragma-text` of a `#pragma` directive ([§6.5.9](#)).

4. The values of any application parameters passed to `Main` by the host environment prior to application startup ([§7.1](#)).
5. The precise structure of the expression tree, as well as the exact process for creating it, when an anonymous function is converted to an expression-tree ([§10.7.3](#)).
6. Whether a `System.ArithemeticException` (or a subclass thereof) is thrown or the overflow goes unreported with the resulting value being that of the left operand, when in an `unchecked` context and the left operand of an integer division is the maximum negative `int` or `long` value and the right operand is `-1` ([§12.10.3](#)).
7. When a `System.ArithemeticException` (or a subclass thereof) is thrown when performing a decimal remainder operation ([§12.10.4](#)).
8. The impact of thread termination when a thread has no handler for an exception, and the thread is itself terminated ([§13.10.6](#)).
9. The impact of thread termination when no matching `catch` clause is found for an exception and the code that initially started that thread is reached. ([§21.4](#)).
10. The mappings between pointers and integers ([§23.5.1](#)).
11. The effect of applying the unary `*` operator to a `null` pointer ([§23.6.2](#)).
12. The behavior when pointer arithmetic overflows the domain of the pointer type ([§23.6.6](#), [§23.6.7](#)).
13. The result of the `sizeof` operator for non-pre-defined value types ([§23.6.9](#)).
14. The behavior of the `fixed` statement if the array expression is `null` or if the array has zero elements ([§23.7](#)).
15. The behavior of the `fixed` statement if the string expression is `null` ([§23.7](#)).
16. The value returned when a stack allocation of size zero is made ([§12.8.21](#)).

## B.4 Unspecified behavior

1. The time at which the finalizer (if any) for an object is run, once that object has become eligible for finalization ([§7.9](#)).
2. The representation of `true` ([§8.3.9](#)).
3. The value of the result when converting out-of-range values from `float` or `double` values to an integral type in an `unchecked` context ([§10.3.2](#)).
4. The exact target object and target method of the delegate produced from an *anonymous\_method\_expression* contains ([§10.7.2](#)).
5. The layout of arrays, except in an unsafe context ([§12.8.16.5](#)).
6. Whether there is any way to execute the *block* of an anonymous function other than through evaluation and invocation of the *lambda\_expression* or *anonymous\_method-expression* ([§12.19.3](#)).
7. The exact timing of static field initialization ([§15.5.6.2](#)).

8. The result of invoking `MoveNext` when an enumerator object is running ([§15.14.5.2](#)).
9. The result of accessing `Current` when an enumerator object is in the before, running, or after states ([§15.14.5.3](#)).
10. The result of invoking `Dispose` when an enumerator object is in the running state ([§15.14.5.4](#)).
11. The attributes of a type declared in multiple parts are determined by combining, in an unspecified order, the attributes of each of its parts ([§22.3](#)).
12. The order in which members are packed into a struct ([§23.6.9](#)).
13. An exception occurs during finalizer execution, and that exception is not caught ([§21.4](#)).
14. If more than one member matches, which member is the implementation of `I.M` ([§18.6.5](#)).

## B.5 Other issues

1. The exact results of floating-point expression evaluation can vary from one implementation to another, because an implementation is permitted to evaluate such expressions using a greater range and/or precision than is required ([§8.3.7](#)).
2. The CLI reserves certain signatures for compatibility with other programming languages ([§15.3.10](#)).

End of informative text.

# Annex C Standard library

Article • 04/07/2023

## C.1 General

A conforming C# implementation shall provide a minimum set of types having specific semantics. These types and their members are listed here, in alphabetical order by namespace and type. For a formal definition of these types and their members, refer to ISO/IEC 23271:2012 *Common Language Infrastructure (CLI), Partition IV; Base Class Library (BCL), Extended Numerics Library, and Extended Array Library*, which are included by reference in this specification.

**This text is informative.**

The standard library is intended to be the minimum set of types and members required by a conforming C# implementation. As such, it contains only those members that are explicitly required by the C# language specification.

It is expected that a conforming C# implementation will supply a significantly more extensive library that enables useful programs to be written. For example, a conforming implementation might extend this library by

- Adding namespaces.
- Adding types.
- Adding members to non-interface types.
- Adding intervening base classes or interfaces.
- Having struct and class types implement additional interfaces.
- Adding attributes (other than the `ConditionalAttribute`) to existing types and members.

**End of informative text.**

## C.2 Standard Library Types defined in ISO/IEC 23271

*Note:* Some `struct` types below have the `readonly` modifier. This modifier was not available when ISO/IEC 23271 was released, but is required for conforming implementations of this specification. *end note*

C#

```
namespace System
{
    public delegate void Action();

    public class ArgumentException : SystemException
    {
        public ArgumentException();
        public ArgumentException(string message);
        public ArgumentException(string message, Exception innerException);
    }

    public class ArithmeticException : Exception
    {
        public ArithmeticException();
        public ArithmeticException(string message);
        public ArithmeticException(string message, Exception
innerException);
    }

    public abstract class Array : IList, ICollection, IEnumerable
    {
        public int Length { get; }
        public int Rank { get; }
        public int GetLength(int dimension);
    }

    public class ArrayTypeMismatchException : Exception
    {
        public ArrayTypeMismatchException();
        public ArrayTypeMismatchException(string message);
        public ArrayTypeMismatchException(string message,
            Exception innerException);
    }

    [AttributeUsageAttribute(AttributeTargets.All, Inherited = true,
        AllowMultiple = false)]
    public abstract class Attribute
    {
        protected Attribute();
    }

    public enum AttributeTargets
    {
        Assembly = 0x1,
        Module = 0x2,
        Class = 0x4,
        Struct = 0x8,
        Enum = 0x10,
        Constructor = 0x20,
        Method = 0x40,
        Property = 0x80,
        Field = 0x100,
```

```
    Event = 0x200,
    Interface = 0x400,
    Parameter = 0x800,
    Delegate = 0x1000,
    ReturnValue = 0x2000,
    GenericParameter = 0x4000,
    All = 0x7FFF
}

[AttributeUsageAttribute(AttributeTargets.Class, Inherited = true)]
public sealed class AttributeUsageAttribute : Attribute
{
    public AttributeUsageAttribute(AttributeTargets validOn);
    public bool AllowMultiple { get; set; }
    public bool Inherited { get; set; }
    public AttributeTargets ValidOn { get; }
}

public readonly struct Boolean { }
public readonly struct Byte { }
public readonly struct Char { }
public readonly struct Decimal { }
public abstract class Delegate { }

public class DivideByZeroException : ArithmeticException
{
    public DivideByZeroException();
    public DivideByZeroException(string message);
    public DivideByZeroException(string message, Exception
innerException);
}

public readonly struct Double { }

public abstract class Enum : ValueType
{
    protected Enum();
}

public class Exception
{
    public Exception();
    public Exception(string message);
    public Exception(string message, Exception innerException);
    public sealed Exception InnerException { get; }
    public virtual string Message { get; }
}

public class GC { }

public interface IDisposable
{
    void Dispose();
}
```

```
public interface IFormattable { }

public sealed class IndexOutOfRangeException : Exception
{
    public IndexOutOfRangeException();
    public IndexOutOfRangeException(string message);
    public IndexOutOfRangeException(string message,
        Exception innerException);
}

public readonly struct Int16 { }
public readonly struct Int32 { }
public readonly struct Int64 { }
public readonly struct IntPtr { }

public class InvalidCastException : Exception
{
    public InvalidCastException();
    public InvalidCastException(string message);
    public InvalidCastException(string message, Exception
innerException);
}

public class InvalidOperationException : Exception
{
    public InvalidOperationException();
    public InvalidOperationException(string message);
    public InvalidOperationException(string message,
        Exception innerException);
}

public class NotSupportedException : Exception
{
    public NotSupportedException();
    public NotSupportedException(string message);
    public NotSupportedException(string message, Exception
innerException);
}

public struct Nullable<T>
{
    public bool HasValue { get; }
    public T Value { get; }
}

public class NullReferenceException : Exception
{
    public NullReferenceException();
    public NullReferenceException(string message);
    public NullReferenceException(string message, Exception
innerException);
}

public class Object
{
```

```
    public Object();
    ~Object();
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
}

[AttributeUsageAttribute(AttributeTargets.Class | AttributeTargets.Struct | AttributeTargets.Enum | AttributeTargets.Interface | AttributeTargets.Constructor | AttributeTargets.Method | AttributeTargets.Property | AttributeTargets.Field | AttributeTargets.Event | AttributeTargets.Delegate, Inherited = false)]
public sealed class ObsoleteAttribute : Attribute
{
    public ObsoleteAttribute();
    public ObsoleteAttribute(string message);
    public ObsoleteAttribute(string message, bool error);
    public bool IsError { get; }
    public string Message { get; }
}

public class OutOfMemoryException : Exception
{
    public OutOfMemoryException();
    public OutOfMemoryException(string message);
    public OutOfMemoryException(string message, Exception innerException);
}

public class OverflowException : ArithmeticException
{
    public OverflowException();
    public OverflowException(string message);
    public OverflowException(string message, Exception innerException);
}

public readonly struct SByte { }
public readonly struct Single { }

public sealed class StackOverflowException : Exception
{
    public StackOverflowException();
    public StackOverflowException(string message);
    public StackOverflowException(string message, Exception innerException);
}

public sealed class String : IEnumerable<Char>, IEnumerable
{
    public int Length { get; }
    public char this [int index] { get; }
    public static string Format(string format, params object[] args);
```

```
}

public abstract class Type : MemberInfo { }

public sealed class TypeInitializationException : Exception
{
    public TypeInitializationException(string fullTypeName,
        Exception innerException);
}

public readonly struct UInt16 { }
public readonly struct UInt32 { }
public readonly struct UInt64 { }
public readonly struct UIntPtr { }

public struct ValueTuple<T1>
{
    public T1 Item1;
    public ValueTuple(T1 item1);
}
public struct ValueTuple<T1, T2>
{
    public T1 Item1;
    public T2 Item2;
    public ValueTuple(T1 item1, T2 item2);
}
public struct ValueTuple<T1, T2, T3>
{
    public T1 Item1;
    public T2 Item2;
    public T3 Item3;
    public ValueTuple(T1 item1, T2 item2, T3 item3);
}
public struct ValueTuple<T1, T2, T3, T4>
{
    public T1 Item1;
    public T2 Item2;
    public T3 Item3;
    public T4 Item4;
    public ValueTuple(T1 item1, T2 item2, T3 item3, T4 item4);
}
public struct ValueTuple<T1, T2, T3, T4, T5>
{
    public T1 Item1;
    public T2 Item2;
    public T3 Item3;
    public T4 Item4;
    public T5 Item5;
    public ValueTuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5);
}
public struct ValueTuple<T1, T2, T3, T4, T5, T6>
{
    public T1 Item1;
    public T2 Item2;
    public T3 Item3;
```

```
        public T4 Item4;
        public T5 Item5;
        public T6 Item6;
        public ValueTuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5,
T6 item6);
    }
    public struct ValueTuple<T1, T2, T3, T4, T5, T6, T7>
{
    public T1 Item1;
    public T2 Item2;
    public T3 Item3;
    public T4 Item4;
    public T5 Item5;
    public T6 Item6;
    public T7 Item7;
    public ValueTuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5,
T6 item6, T7 item7);
}
    public struct ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>
{
    public T1 Item1;
    public T2 Item2;
    public T3 Item3;
    public T4 Item4;
    public T5 Item5;
    public T6 Item6;
    public T7 Item7;
    public TRest Rest;
    public ValueTuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5,
T6 item6, T7 item7, TRest rest);
}
}

public abstract class ValueType
{
    protected ValueType();
}
}

namespace System.Collections
{
    public interface ICollection : IEnumerable
    {
        int Count { get; }
        bool IsSynchronized { get; }
        object SyncRoot { get; }
        void CopyTo(Array array, int index);
    }

    public interface IEnumerable
    {
        IEnumerator GetEnumerator();
    }

    public interface IEnumerator
    {
```

```
        object Current { get; }
        bool MoveNext();
        void Reset();
    }

    public interface IList : ICollection, IEnumerable
    {
        bool IsFixedSize { get; }
        bool IsReadOnly { get; }
        object this [int index] { get; set; }
        int Add(object value);
        void Clear();
        bool Contains(object value);
        int IndexOf(object value);
        void Insert(int index, object value);
        void Remove(object value);
        void RemoveAt(int index);
    }
}

namespace System.Collections.Generic
{
    public interface ICollection<T> : IEnumerable<T>
    {
        int Count { get; }
        bool IsReadOnly { get; }
        void Add(T item);
        void Clear();
        bool Contains(T item);
        void CopyTo(T[] array, int arrayIndex);
        bool Remove(T item);
    }

    public interface IEnumerable<T> : IEnumerable
    {
        Ienumerator<T> GetEnumerator();
    }

    public interface Ienumerator<T> : IDisposable, Ienumerator
    {
        T Current { get; }
    }

    public interface IList<T> : ICollection<T>
    {
        T this [int index] { get; set; }
        int IndexOf(T item);
        void Insert(int index, T item);
        void RemoveAt(int index);
    }

    public interface IReadOnlyCollection<out T> : IEnumerable<T>
    {
        int Count { get; }
    }
}
```

```

public interface IReadOnlyList<out T> : IReadOnlyCollection<T>
{
    T this [int index] { get; }
}

namespace System.Diagnostics
{
    [AttributeUsageAttribute(AttributeTargets.Method | AttributeTargets.Class,
                           AllowMultiple = true)]
    public sealed class ConditionalAttribute : Attribute
    {
        public ConditionalAttribute(string conditionString);
        public string ConditionString { get; }
    }
}

namespace System.Reflection
{
    public abstract class MethodInfo
    {
        protected MethodInfo();
    }
}

namespace System.Runtime.CompilerServices
{
    public sealed class IndexerNameAttribute : Attribute
    {
        public IndexerNameAttribute(String indexerName);
    }

    public static class Unsafe
    {
        public static ref T NullRef<T>();
    }
}

namespace System.Threading
{
    public static class Monitor
    {
        public static void Enter(object obj);
        public static void Exit(object obj);
    }
}

```

## C.3 Standard Library Types not defined in ISO/IEC 23271

The following types, including the members listed, must be defined in a conforming standard library. (These types might be defined in a future edition of ISO/IEC 23271.) It is expected that many of these types will have more members available than are listed.

A conforming implementation may provide `Task.GetAwaiter()` and `Task<TResult>.GetAwaiter()` as extension methods.

C#

```
namespace System
{
    public class FormattableString : IFormattable { }

namespace System.Linq.Expressions
{
    public sealed class Expression<TDelegate>
    {
        public TDelegate Compile();
    }
}

namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct | 
                    AttributeTargets.Interface,
                    Inherited = false, AllowMultiple = false)]
    public sealed class AsyncMethodBuilderAttribute : Attribute
    {
        public AsyncMethodBuilderAttribute(Type builderType) {}

        public Type BuilderType { get; }
    }

    [AttributeUsage(AttributeTargets.Parameter, Inherited = false)]
    public sealed class CallerFilePathAttribute : Attribute
    {
        public CallerFilePathAttribute() { }
    }

    [AttributeUsage(AttributeTargets.Parameter, Inherited = false)]
    public sealed class CallerLineNumberAttribute : Attribute
    {
        public CallerLineNumberAttribute() { }
    }

    [AttributeUsage(AttributeTargets.Parameter, Inherited = false)]
    public sealed class CallerMemberNameAttribute : Attribute
    {
        public CallerMemberNameAttribute() { }
    }

    public static class FormattableStringFactory
    {

```

```
        public static FormattableString Create(string format,
            params object[] arguments);
    }

    public interface ICriticalNotifyCompletion : INotifyCompletion
    {
        void UnsafeOnCompleted(Action continuation);
    }

    public interface INotifyCompletion
    {
        void OnCompleted(Action continuation);
    }

    public readonly struct TaskAwaiter : ICriticalNotifyCompletion,
INotifyCompletion
    {
        public bool IsCompleted { get; }
        public void GetResult();
    }

    public readonly struct TaskAwaiter<TResult> : ICriticalNotifyCompletion,
INotifyCompletion
    {
        public bool IsCompleted { get; }
        public TResult GetResult();
    }

    public readonly struct ValueTaskAwaiter : ICriticalNotifyCompletion,
INotifyCompletion
    {
        public bool IsCompleted { get; }
        public void GetResult();
    }

    public readonly struct ValueTaskAwaiter<TResult> :
ICriticalNotifyCompletion, INotifyCompletion
    {
        public bool IsCompleted { get; }
        public TResult GetResult();
    }
}

namespace System.Threading.Tasks
{
    public class Task
    {
        public System.Runtime.CompilerServices.TaskAwaiter GetAwaiter();
    }
    public class Task<TResult> : Task
    {
        public new System.Runtime.CompilerServices.TaskAwaiter<T>
GetAwaiter();
    }
}
```

```
public readonly struct ValueTask : System.IEquatable<ValueTask>
{
    public System.Runtime.CompilerServices.ValueTaskAwaiter
GetAwaiter();
}
public readonly struct ValueTask<TResult> :
System.IEquatable<ValueTask<TResult>>
{
    public new System.Runtime.CompilerServices.ValueTaskAwaiter<TResult>
GetAwaiter();
}
}
```

C#

```
namespace System
{
    public readonly ref struct ReadOnlySpan<T>
    {
        public int Length { get; }
        public ref readonly T this[int index] { get; }
    }
    public readonly ref struct Span<T>
    {
        public int Length { get; }
        public ref T this[int index] { get; }
        public static implicit operator ReadOnlySpan<T>(Span<T> span);
    }
}
```

## C.4 Format Specifications

The meaning of the formats, as used in interpolated string expressions ([§12.8.3](#)), are defined in ISO/IEC 23271:2012. For convenience the following text is copied from the description of `System.IFormattable`.

This text is informative.

A *format* is a string that describes the appearance of an object when it is converted to a string. Either standard or custom formats can be used. A standard format takes the form *Axx*, where *A* is a single alphabetic character called the *format specifier*, and *xx* is an integer between zero and 99 inclusive, called the *precision specifier*. The *format specifier* controls the type of formatting applied to the value being represented as a string. The *precision specifier* controls the number of significant digits or decimal places in the string, if applicable.

*Note:* For the list of standard format specifiers, see the table below. Note that a given data type, such as `System.Int32`, might not support one or more of the standard format specifiers. *end note*

*Note:* When a format includes symbols that vary by culture, such as the `currencysymbol` included by the 'C' and 'c' formats, a formatting object supplies the actual characters used in the string representation. A method might include a parameter to pass a `System.IFormatProvider` object that supplies a formatting object, or the method might use the default formatting object, which contains the symbol definitions for the current culture. The current culture typically uses the same set of symbols used system-wide by default. In the Base Class Library, the formatting object for system-supplied numeric types is a `System.Globalization.NumberFormatInfo` instance. For `System.DateTime` instances, a `System.Globalization.DateTimeFormatInfo` is used. *end note*

The following table describes the standard format specifiers and associated formatting object members that are used with numeric data types in the Base Class Library.

Format Specifier	Description
C	<p><b>Currency Format:</b> Used for strings containing a monetary value. The <code>System.Globalization.NumberFormatInfo.CurrencySymbol</code>, <code>System.Globalization.NumberFormatInfo.CurrencyGroupSizes</code>, <code>System.Globalization.NumberFormatInfo.CurrencyGroupSeparator</code>, and <code>System.Globalization.NumberFormatInfo.CurrencyDecimalSeparator</code> members of a <code>System.Globalization.NumberFormatInfo</code> supply the currency symbol, size and separator for digit groupings, and decimal separator, respectively.</p> <p><code>System.Globalization.NumberFormatInfo.CurrencyNegativePattern</code> and <code>System.Globalization.NumberFormatInfo.CurrencyPositivePattern</code> determine the symbols used to represent negative and positive values. For example, a negative value can be prefixed with a minus sign, or enclosed in parentheses.</p> <p>If the precision specifier is omitted, <code>System.Globalization.NumberFormatInfo.CurrencyDecimalDigits</code> determines the number of decimal places in the string. Results are rounded to the nearest representable value when necessary.</p>
D	<p><b>Decimal Format:</b> (This format is valid only when specified with integral data types.) Used for strings containing integer values. Negative numbers are prefixed with the negative number symbol specified by the <code>System.Globalization.NumberFormatInfo.NegativeSign</code> property.</p>

The precision specifier determines the minimum number of digits that appear in the string. If the specified precision requires more digits than the value contains, the string is left-padded with zeros. If the precision specifier specifies fewer digits than are in the value, the precision specifier is ignored.

**E** **Scientific (Engineering) Format:** Used for strings in one of the following forms:

**e**  $[-]m.aaaaaaaaE+xxx$

$[-]m.aaaaaaaaE-xxx$

$[-]m.aaaaaaaae+xxx$

$[-]m.aaaaaaaae-xxx$

The negative number symbol ('-') appears only if the value is negative, and is supplied by the `System.Globalization.NumberFormatInfo.NegativeSign` property.

Exactly one non-zero decimal digit (*m*) precedes the decimal separator ('.'), which is supplied by the `System.Globalization.NumberFormatInfo.NumberDecimalSeparator` property.

The precision specifier determines the number of decimal places (*aaaaaaaa*) in the string. If the precision specifier is omitted, six decimal places are included in the string.

The exponent (+/-*xxx*) consists of either a positive or negative number symbol followed by a minimum of three digits (*xxx*). The exponent is left-padded with zeros, if necessary. The case of the format specifier ('E' or 'e') determines the case used for the exponent prefix (E or e) in the string. Results are rounded to the nearest representable value when necessary. The positive number symbol is supplied by the `System.Globalization.NumberFormatInfo.PositiveSign` property.

**F** **Fixed-Point Format:** Used for strings in the following form:

**f**  $[-]m.dd...d$

At least one non-zero decimal digit (*m*) precedes the decimal separator ('.'), which is supplied by the `System.Globalization.NumberFormatInfo.NumberDecimalSeparator` property.

A negative number symbol sign ('-') precedes *m* only if the value is negative. This symbol is supplied by the `System.Globalization.NumberFormatInfo.NegativeSign` property.

The precision specifier determines the number of decimal places (*dd...d*) in the string. If the precision specifier is omitted,

`System.Globalization.NumberFormatInfo.NumberDecimalDigits` determines the number of decimal places in the string. Results are rounded to the nearest representable value when necessary.

**G** **General Format:** The string is formatted in either fixed-point format ('F' or 'f') or scientific format ('E' or 'e').

g

For integral types:

Values are formatted using fixed-point format if *exponent* < precision specifier, where *exponent* is the exponent of the value in scientific format. For all other values, scientific format is used.

If the precision specifier is omitted, a default precision equal to the field width required to display the maximum value for the data type is used, which results in the value being formatted in fixed-point format. The default precisions for integral types are as follows:

`System.Int16, System.UInt16 : 5`

`System.Int32, System.UInt32 : 10`

`System.Int64, System.UInt64 : 19`

For Single, Decimal and Double types:

Values are formatted using fixed-point format if *exponent*  $\geq -4$  and *exponent* < precision specifier, where *exponent* is the exponent of the value in scientific format. For all other values, scientific format is used. Results are rounded to the nearest representable value when necessary.

If the precision specifier is omitted, the following default precisions are used:

`System.Single : 7`

`System.Double : 15`

`System.Decimal : 29`

For all types:

- The number of digits that appear in the result (not including the exponent) will not exceed the value of the precision specifier; values are rounded as necessary.
- The decimal point and any trailing zeros after the decimal point are removed whenever possible.
- The case of the format specifier ('G' or 'g') determines whether 'E' or 'e' prefixes the scientific format exponent.

N

**Number Format:** Used for strings in the following form:

n

`[-]d,ddd,ddd.dd...d`

The representation of negative values is determined by the `System.Globalization.NumberFormatInfo.NumberNegativePattern` property. If the pattern includes a negative number symbol ('-'), this symbol is supplied by the `System.Globalization.NumberFormatInfo.NegativeSign` property.

At least one non-zero decimal digit (*d*) precedes the decimal separator ('.'), which is supplied by the `System.Globalization.NumberFormatInfo.NumberDecimalSeparator`

property. Digits between the decimal point and the most significant digit in the value are grouped using the group size specified by the

`System.Globalization.NumberFormatInfo.NumberGroupSizes` property. The group separator (',') is inserted between each digit group, and is supplied by the `System.Globalization.NumberFormatInfo.NumberGroupSeparator` property.

The precision specifier determines the number of decimal places (*dd...d*). If the precision specifier is omitted,

`System.Globalization.NumberFormatInfo.NumberDecimalDigits` determines the number of decimal places in the string. Results are rounded to the nearest representable value when necessary.

P

**Percent Format:** Used for strings containing a percentage. The `System.Globalization.NumberFormatInfo.PercentSymbol`, `System.Globalization.NumberFormatInfo.PercentGroupSizes`, `System.Globalization.NumberFormatInfo.PercentGroupSeparator`, and `System.Globalization.NumberFormatInfo.PercentDecimalSeparator` members of a `System.Globalization.NumberFormatInfo` supply the percent symbol, size and separator for digit groupings, and decimal separator, respectively.

`System.Globalization.NumberFormatInfo.PercentNegativePattern` and `System.Globalization.NumberFormatInfo.PercentPositivePattern` determine the symbols used to represent negative and positive values. For example, a negative value can be prefixed with a minus sign, or enclosed in parentheses.

If no precision is specified, the number of decimal places in the result is determined by `System.Globalization.NumberFormatInfo.PercentDecimalDigits`. Results are rounded to the nearest representable value when necessary.

The result is scaled by 100 (.99 becomes 99%).

R

**Round trip Format:** (This format is valid only when specified with `System.Double` or `System.Single`.) Used to ensure that the precision of the string representation of a floating-point value is such that parsing the string does not result in a loss of precision when compared to the original value. If the maximum precision of the data type (7 for `System.Single`, and 15 for `System.Double`) would result in a loss of precision, the precision is increased by two decimal places. If a precision specifier is supplied with this format specifier, it is ignored. This format is otherwise identical to the fixed-point format.

X

**Hexadecimal Format:** (This format is valid only when specified with integral data types.) Used for string representations of numbers in Base 16. The precision determines the minimum number of digits in the string. If the precision specifies more digits than the number contains, the number is left-padded with zeros. The case of the format specifier ('X' or 'x') determines whether upper case or lower case letters are used in the hexadecimal representation.

If the numerical value is a `System.Single` or `System.Double` with a value of `NaN`, `PositiveInfinity`, or `NegativeInfinity`, the format specifier is ignored, and one of the

following is returned: `System.Globalization.NumberFormatInfo.NaNSymbol`,

`System.Globalization.NumberFormatInfo.PositiveInfinitySymbol`, or

`System.Globalization.NumberFormatInfo.NegativeInfinitySymbol`.

A custom format is any string specified as a format that is not in the form of a standard format string (Axx) described above. The following table describes the characters that are used in constructing custom formats.

Format Specifier	Description
<code>0</code> (zero)	<b>Zero placeholder:</b> If the value being formatted has a digit in the position where a '0' appears in the custom format, then that digit is copied to the output string; otherwise a zero is stored in that position in the output string. The position of the leftmost '0' before the decimal separator and the rightmost '0' after the decimal separator determine the range of digits that are always present in the output string.  The number of Zero and/or Digit placeholders after the decimal separator determines the number of digits that appear after the decimal separator. Values are rounded as necessary.
<code>#</code>	<b>Digit placeholder:</b> If the value being formatted has a digit in the position where a '#' appears in the custom format, then that digit is copied to the output string; otherwise, nothing is stored in that position in the output string. Note that this specifier never stores the '0' character if it is not a significant digit, even if '0' is the only digit in the string. (It does display the '0' character in the output string if it is a significant digit.)  The number of Zero and/or Digit placeholders after the decimal separator determines the number of digits that appear after the decimal separator. Values are rounded as necessary.
<code>.</code> (period)	<b>Decimal separator:</b> The left most '.' character in the format string determines the location of the decimal separator in the formatted value; any additional '.' characters are ignored. The <code>System.Globalization.NumberFormatInfo.NumberDecimalSeparator</code> property determines the symbol used as the decimal separator.
<code>,</code> (comma)	<b>Group separator and number scaling:</b> The ',' character serves two purposes. First, if the custom format contains this character between two Zero or Digit placeholders (0 or #) and to the left of the decimal separator if one is present, then the output will have group separators inserted between each group of digits to the left of the decimal separator. The <code>System.Globalization.NumberFormatInfo.NumberGroupSeparator</code> and <code>System.Globalization.NumberFormatInfo.NumberGroupSizes</code> properties determine the symbol used as the group separator and the number of digits in each group, respectively.

If the format string contains one or more ‘,’ characters immediately to the left of the decimal separator, then the number will be scaled. The scale factor is determined by the number of group separator characters immediately to the left of the decimal separator. If there are  $x$  characters, then the value is divided by  $1000^x$  before it is formatted. For example, the format string ‘0,’ will divide a value by one million. Note that the presence of the ‘,’ character to indicate scaling does not insert group separators in the output string. Thus, to scale a number by 1 million and insert group separators, use a custom format similar to ‘#,##0,’.

% (percent)	<b>Percentage placeholder:</b> The presence of a ‘%’ character in a custom format causes a number to be multiplied by 100 before it is formatted. The percent symbol is inserted in the output string at the location where the ‘%’ appears in the format string. The <code>System.Globalization.NumberFormatInfo.PercentSymbol</code> property determines the percent symbol.
E0	<b>Engineering format:</b> If any of the strings ‘E’, ‘E+’, ‘E-’, ‘e’, ‘e+’, or ‘e-’ are present in a custom format and is followed immediately by at least one ‘0’ character, then the value is formatted using scientific notation. The number of ‘0’ characters following the exponent prefix (E or e) determines the minimum number of digits in the exponent. The ‘E+’ and ‘e+’ formats indicate that a positive or negative number symbol always precedes the exponent. The ‘E’, ‘E-’, ‘e’, or ‘e-’ formats indicate that a negative number symbol precedes negative exponents; no symbol precedes positive exponents. The positive number symbol is supplied by the <code>System.Globalization.NumberFormatInfo.PositiveSign</code> property. The negative number symbol is supplied by the <code>System.Globalization.NumberFormatInfo.NegativeSign</code> property.
E+0	
E-0	
e0	
e+0	
e-0	
\ (backslash)	<b>Escape character:</b> In some languages, such as C#, the backslash character causes the next character in the custom format to be interpreted as an escape sequence. It is used with C language formatting sequences, such as ‘\n’ (newline). In some languages, the escape character itself is required to be preceded by an escape character when used as a literal. Otherwise, the compiler interprets the character as an escape sequence. This escape character is not required to be supported in all programming languages.
'ABC'	<b>Literal string:</b> Characters enclosed in single or double quotes are copied to the output string literally, and do not affect formatting.
"ABC"	
;	<b>Section separator:</b> The ‘;’ character is used to separate sections for positive, negative, and zero numbers in the format string. (This feature is described in detail below.)
(semicolon)	
Other	<b>All other characters:</b> All other characters are stored in the output string as literals in the position in which they appear.

Note that for fixed-point format strings (strings not containing an ‘E0’, ‘E+0’, ‘E-0’, ‘e0’, ‘e+0’, or ‘e-0’), numbers are rounded to as many decimal places as there are Zero or Digit placeholders to the right of the decimal separator. If the custom format does not contain a decimal separator, the number is rounded to the nearest integer. If the

number has more digits than there are Zero or Digit placeholders to the left of the decimal separator, the extra digits are copied to the output string immediately before the first Zero or Digit placeholder.

A custom format can contain up to three sections separated by section separator characters, to specify different formatting for positive, negative, and zero values. The sections are interpreted as follows:

- **One section:** The custom format applies to all values (positive, negative and zero). Negative values include a negative sign.
- **Two sections:** The first section applies to positive values and zeros, and the second section applies to negative values. If the value to be formatted is negative, but becomes zero after rounding according to the format in the second section, then the resulting zero is formatted according to the first section. Negative values do not include a negative sign to allow full control over representations of negative values. For example, a negative can be represented in parenthesis using a custom format similar to '#####.#####;(#####.#####)'.
- **Three sections:** The first section applies to positive values, the second section applies to negative values, and the third section applies to zeros. The second section can be empty (nothing appears between the semicolons), in which case the first section applies to all nonzero values, and negative values include a negative sign. If the number to be formatted is nonzero, but becomes zero after rounding according to the format in the first or second section, then the resulting zero is formatted according to the third section.

The `System.Enum` and `System.DateTime` types also support using format specifiers to format string representations of values. The meaning of a specific format specifier varies according to the kind of data (numeric, date/time, enumeration) being formatted. See `System.Enum` and `System.Globalization.DateTimeFormatInfo` for a comprehensive list of the format specifiers supported by each type.

## C.5 Library Type Abbreviations

The following library types are referenced in this specification. The full names of those types, including the global namespace qualifier are listed below. Throughout this specification, these types appear as either the fully qualified name; with the global namespace qualifier omitted; or as a simple unqualified type name, with the namespace omitted as well. For example, the type `ICollection<T>`, when used in this specification, always means the type `global::System.Collections.Generic.ICollection<T>`.

- `global::System.Action`
- `global::System.ArgumentException`
- `global::System.ArithmeticException`
- `global::System.Array`
- `global::System.ArrayTypeMismatchException`
- `global::System.Attribute`
- `global::System.AttributeTargets`
- `global::System.AttributeUsageAttribute`
- `global::System.Boolean`
- `global::System.Byte`
- `global::System.Char`
- `global::System.Collections.Generic.ICollection<T>`
- `global::System.Collections.Generic.IEnumerable<T>`
- `global::System.Collections.Generic.IEnumerator<T>`
- `global::System.Collections.Generic.IList<T>`
- `global::System.Collections.Generic.IReadOnlyCollection<out T>`
- `global::System.Collections.Generic.IReadOnlyList<out T>`
- `global::System.Collections.ICollection`
- `global::System.Collections.IEnumerable`
- `global::System.Collections.IList`
- `global::System.Collections.IEnumerator`
- `global::System.Decimal`
- `global::System.Delegate`
- `global::System.Diagnostics.ConditionalAttribute`
- `global::System.DivideByZeroException`
- `global::System.Double`
- `global::System.Enum`
- `global::System.Exception`
- `global::System.GC`
- `global::System.ICollection`
- `global::System.IDisposable`
- `global::System.IEnumerable`
- `global::System.IEnumerable<out T>`
- `global::System.IList`
- `global::System.IndexOutOfRangeException`
- `global::System.Int16`
- `global::System.Int32`
- `global::System.Int64`

- `global::System.IntPtr`
- `global::System.InvalidCastException`
- `global::System.InvalidOperationException`
- `global::System.Linq.Expressions.Expression<TDelegate>`
- `global::System.MemberInfo`
- `global::System.NotSupportedException`
- `global::System.Nullable<T>`
- `global::System.NullReferenceException`
- `global::System.Object`
- `global::System.ObsoleteAttribute`
- `global::System.OutOfMemoryException`
- `global::System.OverflowException`
- `global::System.Runtime.CompilerServices.CallerFileAttribute`
- `global::System.Runtime.CompilerServices.CallerLineNumberAttribute`
- `global::System.Runtime.CompilerServices.CallerMemberNameAttribute`
- `global::System.Runtime.CompilerServices.ICriticalNotifyCompletion`
- `global::System.Runtime.CompilerServices.IndexerNameAttribute`
- `global::System.Runtime.CompilerServices.INotifyCompletion`
- `global::System.Runtime.CompilerServices.TaskAwaiter`
- `global::System.Runtime.CompilerServices.TaskAwaiter<T>`
- `global::System.SByte`
- `global::System.Single`
- `global::System.StackOverflowException`
- `global::System.String`
- `global::System.SystemException`
- `global::System.Threading.Monitor`
- `global::System.Threading.Tasks.Task`
- `global::System.Threading.Tasks.Task<TResult>`
- `global::System.Type`
- `global::System.TypeInitializationException`
- `global::System.UInt16`
- `global::System.UInt32`
- `global::System.UInt64`
- `global::System.UIntPtr`
- `global::System.ValueType`

**End of informative text.**

# Comentários de documentação

Artigo • 16/09/2021

O C# fornece um mecanismo para os programadores documentarem seu código usando uma sintaxe de comentário especial que contém texto XML. Em arquivos de código-fonte, os comentários com um determinado formulário podem ser usados para direcionar uma ferramenta para produzir XML a partir desses comentários e dos elementos de código-fonte, que eles precedem. Os comentários que usam essa sintaxe são chamados **\*comentários da documentação\***. Eles devem preceder imediatamente um tipo definido pelo usuário (como uma classe, um delegado ou uma interface) ou um membro (como um campo, evento, propriedade ou método). A ferramenta de geração de XML é chamada de *gerador de documentação*. (Esse gerador poderia ser, mas não precisa ser, o próprio compilador do C#.) A saída produzida pelo gerador de documentação é chamada de *arquivo de documentação*. Um arquivo de documentação é usado como entrada para o *Visualizador de documentação*; uma ferramenta destinada a produzir algum tipo de exibição visual de informações de tipo e sua documentação associada.

Essa especificação sugere um conjunto de marcas a serem usadas em comentários de documentação, mas o uso dessas marcas não é necessário e outras marcas podem ser usadas, se desejado, desde que as regras de XML bem formadas sejam seguidas.

## Introdução

Os comentários com um formulário especial podem ser usados para direcionar uma ferramenta para produzir XML a partir desses comentários e dos elementos de código-fonte, que eles precedem. Esses comentários são comentários de linha única que começam com três barras (`///`) ou comentários delimitados que começam com uma barra e duas estrelas (`/**`). Eles devem preceder imediatamente um tipo definido pelo usuário (como uma classe, um delegado ou uma interface) ou um membro (como um campo, evento, propriedade ou método) que eles anotam. As seções de atributo ([especificação de atributo](#)) são consideradas parte das declarações; portanto, os comentários de documentação devem preceder os atributos aplicados a um tipo ou membro.

### Sintaxe:

antlr

```
single_line_doc_comment
    : '///' input_character*
```

```
;  
  
delimited_doc_comment  
: '/*' delimited_comment_section* asterisk+ '/'  
;
```

Em uma *single\_line\_doc\_comment*, se houver um caractere de *espaço em branco* após os `///` caracteres em cada um dos *single\_line\_doc\_comment* adjacentes ao *single\_line\_doc\_comment* atual, esse caractere de espaço em *branco* não será incluído na saída XML.

Em um documento delimitado-doc-Comment, se o primeiro caractere que não seja espaço em branco na segunda linha for um asterisco e o mesmo padrão de caracteres de espaço em branco opcionais e um caractere asterisco for repetido no início de cada linha dentro do comentário delimitado-doc-Comment, os caracteres do padrão repetido não serão incluídos na saída XML. O padrão pode incluir caracteres de espaço em branco após, bem como antes, o caractere de asterisco.

**Exemplo:**

C#

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional  
/// plane.</summary>  
///  
public class Point  
{  
    /// <summary>method <c>draw</c> renders the point.</summary>  
    void draw() {...}  
}
```

O texto nos comentários da documentação deve estar bem formado de acordo com as regras de XML (<https://www.w3.org/TR/REC-xml>). Se o XML estiver mal formado, um aviso será gerado e o arquivo de documentação conterá um comentário dizendo que um erro foi encontrado.

Embora os desenvolvedores estejam livres para criar seu próprio conjunto de marcas, um conjunto recomendado é definido em [marcas recomendadas](#). Algumas das marcas recomendadas têm significado especial:

- A `<param>` marca é usada para descrever os parâmetros. Se tal marca for usada, o gerador de documentação deverá verificar se o parâmetro especificado existe e se todos os parâmetros estão descritos nos comentários da documentação. Se essa verificação falhar, o gerador de documentação emitirá um aviso.

- O atributo `cref` pode ser anexado a qualquer marca para fornecer uma referência a um elemento de código. O gerador de documentação deve verificar se esse elemento de código existe. Se a verificação falhar, o gerador de documentação emitirá um aviso. Ao procurar um nome descrito em um `cref` atributo, o gerador de documentação deve respeitar a visibilidade do namespace de acordo com `using` as instruções que aparecem no código-fonte. Para elementos de código genéricos, a sintaxe genérica normal (ou seja, "`List<T>`") não pode ser usada porque produz XML inválido. Chaves podem ser usadas em vez de colchetes (ou seja, "`List{T}`") ou a sintaxe de escape XML pode ser usada (ou seja, "`List<T>`").
- A `<summary>` marca destina-se a ser usada por um visualizador de documentação para exibir informações adicionais sobre um tipo ou membro.
- A `<include>` marca inclui informações de um arquivo XML externo.

Observe atentamente que o arquivo de documentação não fornece informações completas sobre o tipo e os membros (por exemplo, ele não contém nenhuma informação de tipo). Para obter informações sobre um tipo ou membro, o arquivo de documentação deve ser usado em conjunto com reflexão no tipo ou membro real.

## Marcações recomendadas

O gerador de documentação deve aceitar e processar qualquer marca que seja válida de acordo com as regras de XML. As marcas a seguir fornecem funcionalidade comumente usada na documentação do usuário. (É claro que outras marcas são possíveis.)

Tag	Seção	Finalidade
<code>&lt;c&gt;</code>	<code>&lt;c&gt;</code>	Definir texto em uma fonte do tipo código
<code>&lt;code&gt;</code>	<code>&lt;code&gt;</code>	Definir uma ou mais linhas do código-fonte ou saída do programa
<code>&lt;example&gt;</code>	<code>&lt;example&gt;</code>	Indicar um exemplo
<code>&lt;exception&gt;</code>	<code>&lt;exception&gt;</code>	Identifica as exceções que um método pode gerar
<code>&lt;include&gt;</code>	<code>&lt;include&gt;</code>	Inclui XML de um arquivo externo
<code>&lt;list&gt;</code>	<code>&lt;list&gt;</code>	Criar uma lista ou tabela
<code>&lt;para&gt;</code>	<code>&lt;para&gt;</code>	Permitir que a estrutura seja adicionada ao texto
<code>&lt;param&gt;</code>	<code>&lt;param&gt;</code>	Descrever um parâmetro para um método ou Construtor

Tag	Seção	Finalidade
<paramref>	<paramref>	Identificar que uma palavra é um nome de parâmetro
<permission>	<permission>	Documentar a acessibilidade de segurança de um membro
<remarks>	<remarks>	Descrever informações adicionais sobre um tipo
<returns>	<returns>	Descrever o valor de retorno de um método
<see>	<see>	Especificar um link
<seealso>	<seealso>	Gerar uma entrada Consulte também
<summary>	<summary>	Descrever um tipo ou um membro de um tipo
<value>	<value>	Descrever uma propriedade
<typeparam>		Descrever um parâmetro de tipo genérico
<typeparamref>		Identificar que uma palavra é um nome de parâmetro de tipo

## <C>

Essa marca fornece um mecanismo para indicar que um fragmento de texto dentro de uma descrição deve ser definido em uma fonte especial, como a usada para um bloco de código. Para linhas de código real, use <code> ( <code> ).

### Sintaxe:

#### XML

```
<c>text</c>
```

### Exemplo:

#### C#

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>

public class Point
{
    // ...
}
```

## <code>

Essa marca é usada para definir uma ou mais linhas de código-fonte ou saída do programa em alguma fonte especial. Para fragmentos de código pequenos em narrativas, use <c> ( <c> ).

Sintaxe:

XML

```
<code>source code or program output</code>
```

Exemplo:

C#

```
/// <summary>This method changes the point's location by  
///      the given x- and y-offsets.  
/// <example>For example:  
/// <code>  
///     Point p = new Point(3,5);  
///     p.Translate(-1,3);  
/// </code>  
/// results in <c>p</c>'s having the value (2,8).  
/// </example>  
/// </summary>  
  
public void Translate(int xor, int yor) {  
    X += xor;  
    Y += yor;  
}
```

## <example>

Essa marca permite código de exemplo dentro de um comentário, para especificar como um método ou outro membro da biblioteca pode ser usado. Normalmente, isso também envolveria o uso da marca <code> ( <code> ).

Sintaxe:

XML

```
<example>description</example>
```

Exemplo:

Consulte `<code>` ( `<code>` ) para obter um exemplo.

## `<exception>`

Essa marca fornece uma maneira de documentar as exceções que um método pode gerar.

**Sintaxe:**

XML

```
<exception cref="member">description</exception>
```

where

- `member` é o nome de um membro. O gerador de documentação verifica se o membro fornecido existe e se traduz no `member` nome do elemento canônico no arquivo de documentação.
- `description` é uma descrição das circunstâncias em que a exceção é lançada.

**Exemplo:**

C#

```
public class DataBaseOperations
{
    /// <exception cref="MasterFileFormatCorruptException"></exception>
    /// <exception cref="MasterFileLockedOpenException"></exception>
    public static void ReadRecord(int flag) {
        if (flag == 1)
            throw new MasterFileFormatCorruptException();
        else if (flag == 2)
            throw new MasterFileLockedOpenException();
        // ...
    }
}
```

## `<include>`

Essa marca permite incluir informações de um documento XML externo ao arquivo de código-fonte. O arquivo externo deve ser um documento XML bem formado e uma expressão XPath é aplicada a esse documento para especificar o XML desse documento a ser incluído. `<include>` Em seguida, a marca é substituída pelo XML selecionado do documento externo.

## Sintaxe:

XML

```
<include file="filename" path="xpath" />
```

where

- `filename` é o nome de arquivo de um arquivo XML externo. O nome do arquivo é interpretado em relação ao arquivo que contém a marca include.
- `xpath` é uma expressão XPath que seleciona parte do XML no arquivo XML externo.

## Exemplo:

Se o código-fonte contiver uma declaração como:

C#

```
/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { ... }
```

e o arquivo externo "docs.xml" tinha o seguinte conteúdo:

XML

```
<?xml version="1.0"?>
<extradoc>
    <class name="IntList">
        <summary>
            Contains a list of integers.
        </summary>
    </class>
    <class name="StringList">
        <summary>
            Contains a list of integers.
        </summary>
    </class>
</extradoc>
```

em seguida, a mesma documentação é saída como se o código-fonte contivesse:

C#

```
/// <summary>
///     Contains a list of integers.
```

```
/// </summary>
public class IntList { ... }
```

## <list>

Essa marca é usada para criar uma lista ou tabela de itens. Ele pode conter um <listheader> bloco para definir a linha de cabeçalho de uma tabela ou lista de definições. (Ao definir uma tabela, apenas uma entrada para `term` no cabeçalho precisa ser fornecida.)

Cada item na lista é especificado com um <item> bloco. Ao criar uma lista de definições, `term` e `description` deve ser especificado. No entanto, para uma tabela, lista com marcadores ou lista numerada, só `description` é necessário especificar.

### Sintaxe:

XML

```
<list type="bullet" | "number" | "table">
  <listheader>
    <term>term</term>
    <description>*description*</description>
  </listheader>
  <item>
    <term>term</term>
    <description>*description*</description>
  </item>
  ...
  <item>
    <term>term</term>
    <description>description</description>
  </item>
</list>
```

where

- `term` é o termo a ser definido, cuja definição está `description`.
- `description` é um item em uma lista com marcadores ou numerada ou a definição de um `term`.

### Exemplo:

C#

```
public class MyClass
{
```

```
/// <summary>Here is an example of a bulleted list:</summary>
/// <list type="bullet">
/// <item>
/// <description>Item 1.</description>
/// </item>
/// <item>
/// <description>Item 2.</description>
/// </item>
/// </list>
/// </summary>
public static void Main () {
    // ...
}
}
```

## <para>

Essa marca é para uso dentro de outras marcas, como `<summary>` ( `<remarks>` ) ou `<returns>` ( `<returns>` ), e permite que a estrutura seja adicionada ao texto.

**Sintaxe:**

XML

```
<para>content</para>
```

em que `content` é o texto do parágrafo.

**Exemplo:**

C#

```
/// <summary>This is the entry point of the Point class testing program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any non-trivial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // ...
}
```

## <param>

Essa marca é usada para descrever um parâmetro para um método, Construtor ou indexador.

**Sintaxe:**

## XML

```
<param name="name">description</param>
```

where

- `name` é o nome do parâmetro.
- `description` é uma descrição do parâmetro.

**Exemplo:**

## C#

```
/// <summary>This method changes the point's location to  
///      the given coordinates.</summary>  
/// <param name="xor">the new x-coordinate.</param>  
/// <param name="yor">the new y-coordinate.</param>  
public void Move(int xor, int yor) {  
    X = xor;  
    Y = yor;  
}
```

## <paramref>

Essa marca é usada para indicar que uma palavra é um parâmetro. O arquivo de documentação pode ser processado para formatar esse parâmetro de uma maneira distinta.

**Sintaxe:**

## XML

```
<paramref name="name"/>
```

em que `name` é o nome do parâmetro.

**Exemplo:**

## C#

```
/// <summary>This constructor initializes the new Point to  
///      (<paramref name="xor"/>,<paramref name="yor"/>).</summary>  
/// <param name="xor">the new Point's x-coordinate.</param>  
/// <param name="yor">the new Point's y-coordinate.</param>  
  
public Point(int xor, int yor) {
```

```
X = xor;  
Y = yor;  
}
```

## <permission>

Essa marca permite que a acessibilidade de segurança de um membro seja documentada.

**Sintaxe:**

XML

```
<permission cref="member">description</permission>
```

where

- `member` é o nome de um membro. O gerador de documentação verifica se o elemento de código fornecido existe e traduz o *membro* para o nome do elemento canônico no arquivo de documentação.
- `description` é uma descrição do acesso ao membro.

**Exemplo:**

C#

```
/// <permission cref="System.Security.PermissionSet">Everyone can  
/// access this method.</permission>  
  
public static void Test() {  
    // ...  
}
```

## <remarks>

Essa marca é usada para especificar informações adicionais sobre um tipo. (Use `<summary>` ( `<summary>` ) para descrever o próprio tipo e os membros de um tipo.)

**Sintaxe:**

XML

```
<remarks>description</remarks>
```

em que `description` é o texto do comentário.

### Exemplo:

```
C#  
  
/// <summary>Class <c>Point</c> models a point in a  
/// two-dimensional plane.</summary>  
/// <remarks>Uses polar coordinates</remarks>  
public class Point  
{  
    // ...  
}
```

## <returns>

Essa marca é usada para descrever o valor de retorno de um método.

### Sintaxe:

```
XML  
  
<returns>description</returns>
```

em que `description` é uma descrição do valor de retorno.

### Exemplo:

```
C#  
  
/// <summary>Report a point's location as a string.</summary>  
/// <returns>A string representing a point's location, in the form (x,y),  
/// without any leading, trailing, or embedded whitespace.</returns>  
public override string ToString() {  
    return "(" + X + "," + Y + ")";  
}
```

## <see>

Essa marca permite que um link seja especificado dentro do texto. Use `<seealso>` (`<seealso>`) para indicar o texto que deve aparecer em uma seção ver também.

### Sintaxe:

```
XML
```

```
<see cref="member"/>
```

em que `member` é o nome de um membro. O gerador de documentação verifica se o elemento de código fornecido existe e altera o *membro* para o nome do elemento no arquivo de documentação gerado.

**Exemplo:**

C#

```
/// <summary>This method changes the point's location to  
///     the given coordinates.</summary>  
/// <see cref="Translate"/>  
public void Move(int xor, int yor) {  
    X = xor;  
    Y = yor;  
}  
  
/// <summary>This method changes the point's location by  
///     the given x- and y-offsets.  
/// </summary>  
/// <see cref="Move"/>  
public void Translate(int xor, int yor) {  
    X += xor;  
    Y += yor;  
}
```

## <seealso>

Essa marca permite que uma entrada seja gerada para a seção Consulte também. Use `<see>` ( `<see>` ) para especificar um link de dentro do texto.

**Sintaxe:**

XML

```
<seealso cref="member"/>
```

em que `member` é o nome de um membro. O gerador de documentação verifica se o elemento de código fornecido existe e altera o *membro* para o nome do elemento no arquivo de documentação gerado.

**Exemplo:**

C#

```
/// <summary>This method determines whether two Points have the same  
///     location.</summary>  
/// <seealso cref="operator=="/>  
/// <seealso cref="operator!="/>  
public override bool Equals(object o) {  
    // ...  
}
```

## <summary>

Essa marca pode ser usada para descrever um tipo ou um membro de um tipo. Use `<remarks>` ( `<remarks>` ) para descrever o próprio tipo.

**Sintaxe:**

XML

```
<summary>description</summary>
```

em que `description` é um resumo do tipo ou membro.

**Exemplo:**

C#

```
/// <summary>This constructor initializes the new Point to (0,0).</summary>  
public Point() : this(0,0) {  
}
```

## <value>

Essa marca permite que uma propriedade seja descrita.

**Sintaxe:**

XML

```
<value>property description</value>
```

em que `property description` é uma descrição para a propriedade.

**Exemplo:**

C#

```
/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
{
    get { return x; }
    set { x = value; }
}
```

## <typeparam>

Essa marca é usada para descrever um parâmetro de tipo genérico para uma classe, struct, interface, delegado ou método.

Sintaxe:

XML

```
<typeparam name="name">description</typeparam>
```

em que `name` é o nome do parâmetro de tipo e `description` é sua descrição.

Exemplo:

C#

```
/// <summary>A generic list class.</summary>
/// <typeparam name="T">The type stored by the list.</typeparam>
public class myList<T> {
    ...
}
```

## <typeparamref>

Essa marca é usada para indicar que uma palavra é um parâmetro de tipo. O arquivo de documentação pode ser processado para formatar esse parâmetro de tipo de alguma maneira distinta.

Sintaxe:

XML

```
<typeparamref name="name"/>
```

em que `name` é o nome do parâmetro de tipo.

## Exemplo:

C#

```
/// <summary>This method fetches data and returns a list of <typeparamref name="T"/>.</summary>
/// <param name="query">query to execute</param>
public List<T> FetchData<T>(string query) {
    ...
}
```

## Processando o arquivo de documentação

O gerador de documentação gera uma cadeia de caracteres de ID para cada elemento no código-fonte que é marcado com um comentário de documentação. Essa cadeia de caracteres de ID identifica exclusivamente um elemento de origem. Um visualizador de documentação pode usar uma cadeia de caracteres de ID para identificar o item de metadados/reflexão correspondente ao qual a documentação se aplica.

O arquivo de documentação não é uma representação hierárquica do código-fonte; em vez disso, é uma lista simples com uma cadeia de caracteres de ID gerada para cada elemento.

## Formato da cadeia de caracteres de ID

O gerador de documentação observa as seguintes regras ao gerar as cadeias de caracteres de ID:

- Nenhum espaço em branco é colocado na cadeia de caracteres.
- A primeira parte da cadeia de caracteres identifica o tipo de membro que está sendo documentado, por meio de um único caractere seguido por dois-pontos. Os seguintes tipos de membros são definidos:

Caractere	Descrição
E	Evento
F	Campo
M	Método (incluindo construtores, destruidores e operadores)
N	Namespace
P	Propriedade (incluindo indexadores)

Caractere	Descrição
T	Tipo (como classe, delegado, enumeração, interface e struct)
!	Cadeia de caracteres de erro; o restante da cadeia de caracteres fornece informações sobre o erro. Por exemplo, o gerador de documentação gera informações de erro para links que não podem ser resolvidos.

- A segunda parte da cadeia de caracteres é o nome totalmente qualificado do elemento, começando na raiz do namespace. O nome do elemento, seus tipos de delimitador e o namespace são separados por pontos. Se o nome do próprio item tiver pontos, eles serão substituídos por `#(U+0023)` caracteres. (Supõe-se que nenhum elemento tenha esse caractere em seu nome.)
- Para métodos e propriedades com argumentos, a lista de argumentos segue, entre parênteses. Para aqueles sem argumentos, os parênteses são omitidos. Os argumentos são separados por vírgulas. A codificação de cada argumento é a mesma que uma assinatura da CLI, da seguinte maneira:
  - Os argumentos são representados pelo nome da documentação, que se baseia em seu nome totalmente qualificado, modificado da seguinte maneira:
  - Os argumentos que representam tipos genéricos têm um ` caractere anexado (de marca de seleção) seguido pelo número de parâmetros de tipo
  - Os argumentos que `out` `ref` têm o modificador ou têm um `@` nome de tipo a seguir. Os argumentos passados por valor ou via `params` não têm notação especial.
  - Os argumentos que são matrizes são representados como `[lowerbound:size, ... , lowerbound:size]` onde o número de vírgulas é a classificação menos uma, e os limites inferiores e o tamanho de cada dimensão, se conhecido, são representados em decimal. Se um limite inferior ou tamanho não for especificado, ele será omitido. Se o limite inferior e o tamanho de uma determinada dimensão forem omitidos, o `:` também será omitido. As matrizes denteadas são representadas por um `[]` por nível.
  - Os argumentos que têm tipos de ponteiro diferentes de void são representados usando `*` o nome do tipo a seguir. Um ponteiro void é representado usando um nome de tipo de `System.Void`.
  - Os argumentos que se referem a parâmetros de tipo genérico definidos em tipos são codificados usando o ` caractere (backtick) seguido pelo índice de base zero do parâmetro de tipo.
  - Os argumentos que usam parâmetros de tipo genérico definidos em métodos usam uma marca de seleção dupla ` ` em vez de ` usados para tipos.

- Os argumentos que se referem a tipos genéricos construídos são codificados usando o tipo genérico, seguido por `{`, seguido por uma lista separada por vírgulas de argumentos de tipo, seguida por `}`.

## Exemplos de cadeia de caracteres de ID

Os exemplos a seguir mostram um fragmento do código C#, juntamente com a cadeia de caracteres de ID produzida de cada elemento de origem capaz de ter um comentário de documentação:

- Os tipos são representados usando seu nome totalmente qualificado, aumentados com informações genéricas:

```
C#  
  
enum Color { Red, Blue, Green }  
  
namespace Acme  
{  
    interface IProcess {...}  
  
    struct ValueType {...}  
  
    class Widget: IProcess  
    {  
        public class NestedClass {...}  
        public interface IMenuItem {...}  
        public delegate void Del(int i);  
        public enum Direction { North, South, East, West }  
    }  
  
    class MyList<T>  
    {  
        class Helper<U,V> {...}  
    }  
}  
  
"T:Color"  
"T:Acme.IProcess"  
"T:Acme.ValueType"  
"T:Acme.Widget"  
"T:Acme.Widget.NestedClass"  
"T:Acme.Widget.IMenuItem"  
"T:Acme.Widget.Del"  
"T:Acme.Widget.Direction"  
"T:Acme.MyList`1"  
"T:Acme.MyList`1.Helper`2"
```

- Os campos são representados pelo nome totalmente qualificado:

C#

```
namespace Acme
{
    struct ValueType
    {
        private int total;
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            private int value;
        }

        private string message;
        private static Color defaultColor;
        private const double PI = 3.14159;
        protected readonly double monthlyAverage;
        private long[] array1;
        private Widget[,] array2;
        private unsafe int *pCount;
        private unsafe float **ppValues;
    }
}

"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

- Construtores.

C#

```
namespace Acme
{
    class Widget: IProcess
    {
        static Widget() {...}
        public Widget() {...}
        public Widget(string s) {...}
    }
}

"M:Acme.Widget.#cctor"
```

```
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- Destruidores.

C#

```
namespace Acme
{
    class Widget: IProcess
    {
        ~Widget() {...}
    }
}

"M:Acme.Widget.Finalize"
```

- Maneiras.

C#

```
namespace Acme
{
    struct ValueType
    {
        public void M(int i) {...}
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            public void M(int i) {...}
        }

        public static void M0() {...}
        public void M1(char c, out float f, ref ValueType v) {...}
        public void M2(short[] x1, int[,] x2, long[][] x3) {...}
        public void M3(long[][] x3, Widget[, ,] x4) {...}
        public unsafe void M4(char *pc, Color **pf) {...}
        public unsafe void M5(void *pv, double *[,] pd) {...}
        public void M6(int i, params object[] args) {...}
    }

    class MyList<T>
    {
        public void Test(T t) { }
    }

    class UseList
    {
        public void Process(MyList<int> list) { }
    }
}
```

```

        public MyList<T> GetValues<T>(T inputValue) { return null; }
    }

    "M:Acme.ValueType.M(System.Int32)"
    "M:Acme.Widget.NestedClass.M(System.Int32)"
    "M:Acme.Widget.M0"
    "M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
    "M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][][])"
    "M:Acme.Widget.M3(System.Int64[],Acme.Widget[0:,0:,0:][])"
    "M:Acme.Widget.M4(System.Char*,Color**)"
    "M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
    "M:Acme.Widget.M6(System.Int32,System.Object[])"
    "M:Acme.MyList`1.Test(`0)"
    "M:Acme.UseList.Process(Acme.MyList{System.Int32})"
    "M:Acme.UseList.GetValues``(``0)"

```

- Propriedades e indexadores.

C#

```

namespace Acme
{
    class Widget: IProcess
    {
        public int Width { get {...} set {...} }
        public int this[int i] { get {...} set {...} }
        public int this[string s, int i] { get {...} set {...} }
    }
}

"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String, System.Int32)"

```

- Eventos.

C#

```

namespace Acme
{
    class Widget: IProcess
    {
        public event Del AnEvent;
    }
}

"E:Acme.Widget.AnEvent"

```

- Operadores unários.

C#

```
namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x) {...}
    }
}

"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
```

O conjunto completo de nomes de funções de operador unários usado é o seguinte:,,, `op_UnaryPlus` `op_UnaryNegation` `op_LogicalNot` `op_OnesComplement` `op_Increment` `op_Decrement` , `op_True` e `op_False` .

- Operadores binários.

C#

```
namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x1, Widget x2) {...}
    }
}

"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"
```

O conjunto completo de nomes de funções de operador binários usado é o seguinte:,, `op>Addition` `op_Subtraction` , `op_Multiply` `op_Division` `op_Modulus` `op_BitwiseAnd` , `op_BitwiseOr` , `op_ExclusiveOr` , `op_LeftShift` , `op_RightShift` , `op_Equality` , `op_Inequality` , `op_LessThan` , `op_LessThanOrEqual` , `op_GreaterThan` e `op_GreaterThanOrEqual` .

- Os operadores de conversão têm um " ~ " à direita seguido pelo tipo de retorno.

C#

```
namespace Acme
{
    class Widget: IProcess
    {
        public static explicit operator int(Widget x) {...}
        public static implicit operator long(Widget x) {...}
    }
}
```

```
"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"
```

## Um exemplo

### Código-fonte C#

O exemplo a seguir mostra o código-fonte de uma `Point` classe:

C#

```
namespace Graphics
{
    /// <summary>Class <c>Point</c> models a point in a two-dimensional plane.
    /// </summary>
    public class Point
    {
        /// <summary>Instance variable <c>x</c> represents the point's
        ///     x-coordinate.</summary>
        private int x;

        /// <summary>Instance variable <c>y</c> represents the point's
        ///     y-coordinate.</summary>
        private int y;

        /// <value>Property <c>X</c> represents the point's x-coordinate.
        </value>
        public int X
        {
            get { return x; }
            set { x = value; }
        }

        /// <value>Property <c>Y</c> represents the point's y-coordinate.
        </value>
        public int Y
        {
            get { return y; }
            set { y = value; }
        }

        /// <summary>This constructor initializes the new Point to
        ///     (0,0).</summary>
        public Point() : this(0,0) {}

        /// <summary>This constructor initializes the new Point to
        ///     (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
```

```

/// <param><c>xor</c> is the new Point's x-coordinate.</param>
/// <param><c>yor</c> is the new Point's y-coordinate.</param>
public Point(int xor, int yor) {
    X = xor;
    Y = yor;
}

/// <summary>This method changes the point's location to
/// the given coordinates.</summary>
/// <param><c>xor</c> is the new x-coordinate.</param>
/// <param><c>yor</c> is the new y-coordinate.</param>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}

/// <summary>This method changes the point's location by
/// the given x- and y-offsets.
/// <example>For example:
/// <code>
///     Point p = new Point(3,5);
///     p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>
/// <param><c>xor</c> is the relative x-offset.</param>
/// <param><c>yor</c> is the relative y-offset.</param>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}

/// <summary>This method determines whether two Points have the same
/// location.</summary>
/// <param><c>o</c> is the object to be compared to the current object.
/// </param>
/// <returns>True if the Points have the same location and they have
/// the exact same type; otherwise, false.</returns>
/// <seealso cref="operator==">
/// <seealso cref="operator!=">
public override bool Equals(object o) {
    if (o == null) {
        return false;
    }

    if (this == o) {
        return true;
    }

    if (GetType() == o.GetType()) {
        Point p = (Point)o;
        return (X == p.X) && (Y == p.Y);
    }
}

```

```
        }
        return false;
    }

    /// <summary>Report a point's location as a string.</summary>
    /// <returns>A string representing a point's location, in the form
    (x,y),
    /// without any leading, trailing, or embedded whitespace.</returns>
    public override string ToString() {
        return "(" + X + "," + Y + ")";
    }

    /// <summary>This operator determines whether two Points have the same
    /// location.</summary>
    /// <param><c>p1</c> is the first Point to be compared.</param>
    /// <param><c>p2</c> is the second Point to be compared.</param>
    /// <returns>True if the Points have the same location and they have
    /// the exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator!="/>
    public static bool operator==(Point p1, Point p2) {
        if ((object)p1 == null || (object)p2 == null) {
            return false;
        }

        if (p1.GetType() == p2.GetType()) {
            return (p1.X == p2.X) && (p1.Y == p2.Y);
        }

        return false;
    }

    /// <summary>This operator determines whether two Points have the same
    /// location.</summary>
    /// <param><c>p1</c> is the first Point to be compared.</param>
    /// <param><c>p2</c> is the second Point to be compared.</param>
    /// <returns>True if the Points do not have the same location and the
    /// exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator=="/>
    public static bool operator!=(Point p1, Point p2) {
        return !(p1 == p2);
    }

    /// <summary>This is the entry point of the Point class testing
    /// program.
    /// <para>This program tests each method and operator, and
    /// is intended to be run after any non-trivial maintenance has
    /// been performed on the Point class.</para></summary>
    public static void Main() {
        // class test code goes here
    }
}
```

## XML resultante

Aqui está a saída produzida por um gerador de documentação quando é fornecido o código-fonte para a classe `Point`, mostrado acima:

XML

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Point</name>
    </assembly>
    <members>
        <member name="T:Graphics.Point">
            <summary>Class <c>Point</c> models a point in a two-dimensional plane.
            </summary>
        </member>

        <member name="F:Graphics.Point.x">
            <summary>Instance variable <c>x</c> represents the point's x-coordinate.</summary>
        </member>

        <member name="F:Graphics.Point.y">
            <summary>Instance variable <c>y</c> represents the point's y-coordinate.</summary>
        </member>

        <member name="M:Graphics.Point.#ctor">
            <summary>This constructor initializes the new Point to (0,0).</summary>
        </member>

        <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
            <summary>This constructor initializes the new Point to (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
            <param><c>xor</c> is the new Point's x-coordinate.</param>
            <param><c>yor</c> is the new Point's y-coordinate.</param>
        </member>

        <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
            <summary>This method changes the point's location to the given coordinates.</summary>
            <param><c>xor</c> is the new x-coordinate.</param>
            <param><c>yor</c> is the new y-coordinate.</param>
            <see
cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
        </member>

        <member
            name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
            <summary>This method changes the point's location by

```

```
the given x- and y-offsets.
<example>For example:
<code>
Point p = new Point(3,5);
p.Translate(-1,3);
</code>
results in <c>p</c>'s having the value (2,8).
</example>
</summary>
<param><c>xor</c> is the relative x-offset.</param>
<param><c>yor</c> is the relative y-offset.</param>
<see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
</member>

<member name="M:Graphics.Point.Equals(System.Object)">
<summary>This method determines whether two Points have the same
location.</summary>
<param><c>o</c> is the object to be compared to the current
object.
</param>
<returns>True if the Points have the same location and they have
the exact same type; otherwise, false.</returns>
<seealso
href="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
<seealso
href="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
</member>

<member name="M:Graphics.Point.ToString">
<summary>Report a point's location as a string.</summary>
<returns>A string representing a point's location, in the form
(x,y),
without any leading, trailing, or embedded whitespace.</returns>
</member>

<member
name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
<summary>This operator determines whether two Points have the
same
location.</summary>
<param><c>p1</c> is the first Point to be compared.</param>
<param><c>p2</c> is the second Point to be compared.</param>
<returns>True if the Points have the same location and they have
the exact same type; otherwise, false.</returns>
<seealso cref="M:Graphics.Point.Equals(System.Object)"/>
<seealso
href="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
</member>

<member
name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
<summary>This operator determines whether two Points have the
same
location.</summary>
<param><c>p1</c> is the first Point to be compared.</param>
```

```
<param><c>p2</c> is the second Point to be compared.</param>
<returns>True if the Points do not have the same location and
the
exact same type; otherwise, false.</returns>
<seealso href="M:Graphics.Point.Equals(System.Object)"/>
<seealso
href="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
</member>

<member name="M:Graphics.Point.Main">
<summary>This is the entry point of the Point class testing
program.
<para>This program tests each method and operator, and
is intended to be run after any non-trivial maintenance has
been performed on the Point class.</para></summary>
</member>

<member name="P:Graphics.Point.X">
<value>Property <c>X</c> represents the point's
x-coordinate.</value>
</member>

<member name="P:Graphics.Point.Y">
<value>Property <c>Y</c> represents the point's
y-coordinate.</value>
</member>
</members>
</doc>
```

# Annex E Bibliography

Article • 01/13/2022

**This annex is informative.**

ANSI X3.274-1996, *Programming Language REXX*. (This document is useful in understanding floating-point decimal arithmetic rules.)

ISO/IEC 9075-1, *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*

ISO/IEC 9899, *Programming languages — C*.

ISO/IEC 14882 *Programming languages — C++*

ISO 80000-1, Quantities and units — Part 1: General. (This document defines “banker’s rounding.”)

**End of informative text.**

# Tipos de referência anuláveis em C #

Artigo • 16/09/2021

O objetivo desse recurso é:

- Permitir que os desenvolvedores expressem se uma variável, um parâmetro ou um resultado de um tipo de referência deve ser nulo ou não.
- Forneça avisos quando tais variáveis, parâmetros e resultados não forem usados de acordo com essa intenção.

## Expressão de intenção

O idioma já contém a `T?` sintaxe para tipos de valor. É simples estender essa sintaxe para tipos de referência.

Supõe-se que a intenção de um tipo de referência não adornado `T` seja para que ele seja não nulo.

## Verificação de referências anuláveis

Uma análise de fluxo acompanha variáveis de referência anuláveis. Quando a análise considera que ela não seria nula (por exemplo, após uma verificação ou uma atribuição), seu valor será considerado uma referência não nula.

Uma referência anulável também pode ser tratada explicitamente como não-nula com o operador de sufixo `x!` (o operador "damnit"), para quando a análise de fluxo não puder estabelecer uma situação não nula que o desenvolvedor saiba.

Caso contrário, um aviso será fornecido se uma referência anulável for desreferenciada ou for convertida em um tipo não nulo.

Um aviso é fornecido ao converter de `S[]` para `T?[]` e de `S?[]` para `T[]`.

Um aviso é fornecido ao converter de `C<S>` para `C<T?>`, exceto quando o parâmetro de tipo é covariant (`out`) e ao converter de `C<S?>` para `C<T>`, exceto quando o parâmetro de tipo é contravariant (`in`).

Um aviso será fornecido em `C<T?>` se o parâmetro de tipo tiver restrições não nulas.

## Verificando referências não nulas

Um aviso será fornecido se um literal nulo for atribuído a uma variável não nula ou passado como um parâmetro não nulo.

Um aviso também será fornecido se um construtor não inicializar explicitamente os campos de referência não nulos.

Não podemos rastrear adequadamente que todos os elementos de uma matriz de referências não nulas são inicializados. No entanto, poderíamos emitir um aviso se nenhum elemento de uma matriz recém-criada for atribuído antes de a matriz ser lida ou passada. Isso pode lidar com o caso comum sem muito ruído.

Precisamos decidir se `default(T)` o gera um aviso ou é simplesmente tratado como sendo do tipo `T?`.

## Representação de metadados

Adornos de nulidade devem ser representados em metadados como atributos. Isso significa que os compiladores de nível inferior irão ignorá-los.

Precisamos decidir se apenas anotações anuláveis estão incluídas, ou também há alguma indicação de se não nulo foi "ligado" no assembly.

## Genéricos

Se um parâmetro de tipo `T` tiver restrições não anuláveis, ele será tratado como não anulável dentro de seu escopo.

Se um parâmetro de tipo for irrestrito ou tiver apenas restrições anuláveis, a situação será um pouco mais complexa: isso significa que o argumento de tipo correspondente pode ser *anulável* ou *não anulável*. A coisa segura a fazer nessa situação é tratar o *parâmetro de tipo como anulável* e não anulável, fornecendo avisos quando um deles for violado.

Vale a pena considerar se as restrições de referência anuláveis explícitas devem ser permitidas. No entanto, observe que não podemos evitar ter tipos de referência anuláveis *implicitamente* restrições em determinados casos (restrições herdadas).

A `class` restrição não é nula. Podemos considerar se `class?` deve ser uma restrição anulável válida, indicando "tipo de referência anulável".

## Inferência de tipos

Na inferência de tipos, se um tipo de contribuição for um tipo de referência anulável, o tipo resultante deverá ser anulável. Em outras palavras, a nulidade é propagada.

Devemos considerar se o `null` literal como uma expressão participante deve contribuir com a nulidade. Não hoje: para tipos de valor, ele leva a um erro, ao passo que, para tipos de referência, o `NULL` converte com êxito o tipo Plain.

```
C#
```

```
string? n = "world";
var x = b ? "Hello" : n; // string?
var y = b ? "Hello" : null; // string? or error
var z = b ? 7 : null; // Error today, could be int?
```

## Diretrizes de proteção nula

Como um recurso, os tipos de referência anuláveis permitem aos desenvolvedores expressar suas intenções e fornecem avisos por meio da análise de fluxo se essa intenção for contraditória. Há uma pergunta comum sobre se as proteções nulas devem ou não ser necessárias.

## Exemplo de proteção nula

```
C#
```

```
public void DoWork(Worker worker)
{
    // Guard against worker being null
    if (worker is null)
    {
        throw new ArgumentNullException(nameof(worker));
    }

    // Otherwise use worker argument
}
```

No exemplo anterior, a `DoWork` função aceita um `Worker` e protege contra ele potencialmente `null`. Se o `worker` argumento for `null`, a `DoWork` função irá `throw`. Com tipos de referência anuláveis, o código no exemplo anterior faz a intenção de que o `Worker` parâmetro *não* seria `null`. Se a `DoWork` função era uma API pública, como um pacote NuGet ou uma biblioteca compartilhada, como orientação, você deve deixar proteções nulas em vigor. Como uma API pública, a única garantia de que um chamador não `null` está passando é a proteção contra ele.

## Intenção expressa

Um uso mais atraente do exemplo anterior é expressar que o `Worker` parâmetro poderia ser `null`, tornando a proteção nula mais apropriada. Se você remover a proteção nula no exemplo a seguir, o compilador avisará que você pode estar desreferenciando NULL. Independentemente de, ambas as proteções nulas ainda são válidas.

C#

```
public void DoWork(Worker? worker)
{
    // Guard against worker being null
    if (worker is null)
    {
        throw new ArgumentNullException(nameof(worker));
    }

    // Otherwise use worker argument
}
```

Para APIs não públicas, como o código-fonte totalmente no controle de uma equipe de desenvolvedor ou de desenvolvimento, os tipos de referência anuláveis podem permitir a remoção segura de proteções nulas, nas quais os desenvolvedores podem garantir que isso não seja necessário. O recurso pode ajudar com avisos, mas não pode garantir que na execução de código de tempo de execução possa resultar em um `NullReferenceException`.

## Alterações de quebra

Avisos não nulos são uma alteração significativa de quebra no código existente e devem ser acompanhados por um mecanismo de aceitação.

Menos óbvio, os avisos de tipos anuláveis (conforme descrito acima) são uma alteração significativa no código existente em determinados cenários em que a nulidade é implícita:

- Os parâmetros de tipo irrestrito serão tratados como anuláveis implicitamente, portanto, atribuí-los `object` ou acessar, por exemplo, `ToString` resultarão em avisos.
- se a inferência de tipos inferir a nulidade de `null` expressões, o código existente, às vezes, resultará em tipos anuláveis em vez de não anuláveis, o que pode levar a novos avisos.

Portanto, os avisos anuláveis também precisam ser opcionais

Por fim, adicionar anotações a uma API existente será uma alteração significativa para os usuários que optaram por avisos, quando eles atualizarem a biblioteca. Isso também merece a capacidade de aceitar ou sair. "Desejo as correções de bugs, mas não estou pronto para lidar com suas novas anotações"

Em resumo, você precisa ser capaz de aceitar/sair de:

- Avisos anuláveis
- Avisos não nulos
- Avisos de anotações em outros arquivos

A granularidade da aceitação sugere um modelo como o analisador, em que faixas de código pode aceitar e cancelar com pragmas e níveis de severidade podem ser escolhidos pelo usuário. Além disso, as opções por biblioteca ("ignorar as anotações de JSON.NET até que eu esteja pronto para lidar com a saída") podem ser expressas em código como atributos.

O design da experiência de aceitação/transição é crucial para o sucesso e a utilidade desse recurso. Precisamos garantir que:

- Os usuários podem adotar a verificação de nulidade gradualmente, como desejam
- Os autores de biblioteca podem adicionar anotações de nulidade sem medo de quebrar clientes
- Apesar desses, não há uma noção de "pesadelo de configuração"

## Ajustes

Poderíamos considerar não usar as `?`  anotações em locais, mas apenas observando se elas são usadas de acordo com o que é atribuído a elas. Eu não prefiro isso; Acho que devemos permitir que as pessoas expressem suas intenções.

Poderíamos considerar uma abreviação `T! x` de parâmetros, que gera automaticamente uma verificação nula em tempo de execução.

Determinados padrões em tipos genéricos, como `FirstOrDefault` ou `TryGet`, têm um comportamento um pouco estranho com argumentos de tipo não anuláveis, pois eles implicitamente geram valores padrão em determinadas situações. Poderíamos tentar nuancer o sistema de tipos para acomodá-lo melhor. Por exemplo, poderíamos permitir `?`  em parâmetros de tipo irrestrito, embora o argumento de tipo já possa ser anulável. Acho que vale a pena, e isso leva à estranhidade relacionada à interação com tipos de valor anulável.

# Tipos de valor anuláveis

Poderíamos considerar a adoção de algumas das semânticas acima para tipos de valor anuláveis também.

Já mencionamos inferência de tipos, onde poderíamos inferir `int? (7, null)`, em vez de apenas dar um erro.

Outra oportunidade é aplicar a análise de fluxo a tipos de valor anuláveis. Quando eles são considerados não nulos, podemos realmente permitir o uso como o tipo não anulável de determinadas maneiras (por exemplo, acesso de membro). Precisamos apenas ter cuidado para que as coisas que você já possa fazer em um tipo de valor anulável sejam preferenciais, para fins de compatibilidade de volta.

# Correspondência de padrão recursivo

Artigo • 16/09/2021

## Resumo

As extensões de correspondência de padrões para C# habilitam muitos dos benefícios dos tipos de dados algébricas e a correspondência de padrões de linguagens funcionais, mas de uma forma que se integre perfeitamente com a sensação da linguagem subjacente. Os elementos dessa abordagem são inspirados por recursos relacionados nas linguagens de programação [F#](#) e [escala](#).

## Design detalhado

### Expressão is

O `is` operador é estendido para testar uma expressão em relação a um *padrão*.

```
antlr

relational_expression
    : is_pattern_expression
    ;
is_pattern_expression
    : relational_expression 'is' pattern
    ;
```

Essa forma de *relational\_expression* é além dos formulários existentes na especificação do C#. Será um erro de tempo de compilação se o *relational\_expression* à esquerda do token não `is` designar um valor ou não tiver um tipo.

Cada *identificador* do padrão apresenta uma nova variável local que é *definitivamente atribuída* após o `is` operador `true` (ou seja, *definitivamente atribuído quando true*).

Observação: há tecnicamente uma ambiguidade entre o *tipo* em um `is-expression` e `constant_pattern`, que pode ser uma análise válida de um identificador qualificado. Tentamos associá-lo como um tipo de compatibilidade com as versões anteriores do idioma; somente se isso falhar, resolveremos como fazemos uma expressão em outros contextos, para a primeira coisa encontrada (que deve ser uma constante ou um tipo). Essa ambiguidade só está presente no lado direito de uma `is` expressão.

# Padrões

Os padrões são usados no operador de `is_pattern`, em uma `switch_statement` e em uma `switch_expression` para expressar a forma de dados em que os dados de entrada (que chamamos de valor de entrada) devem ser comparados. Os padrões podem ser recursivos para que as partes dos dados possam ser correspondidas em subpadrões.

antlr

```
pattern
: declaration_pattern
| constant_pattern
| var_pattern
| positional_pattern
| property_pattern
| discard_pattern
;
declaration_pattern
: type simple_designation
;
constant_pattern
: constant_expression
;
var_pattern
: 'var' designation
;
positional_pattern
: type? '(' subpatterns? ')' property_subpattern? simple_designation?
;
subpatterns
: subpattern
| subpattern ',' subpatterns
;
subpattern
: pattern
| identifier ':' pattern
;
property_subpattern
: '{}''
| '{}' subpatterns ','? '}'
;
property_pattern
: type? property_subpattern simple_designation?
;
simple_designation
: single_variable_designation
| discard_designation
;
discard_pattern
: '_'
;
```

## Padrão de declaração

```
antlr
```

```
declaration_pattern
  : type simple_designation
  ;
```

O *declaration\_pattern* testa se uma expressão é de um determinado tipo e a converte para esse tipo se o teste for bem sucedido. Isso pode introduzir uma variável local do tipo fornecido nomeado pelo identificador fornecido, se a designação for uma *single\_variable\_designation*. Essa variável local é *definitivamente atribuída* quando o resultado da operação de correspondência de padrões é `true`.

A semântica de tempo de execução dessa expressão é que ela testa o tipo de tempo de execução do operando de *relational\_expression* à esquerda em relação ao *tipo* no padrão. Se for desse tipo de tempo de execução (ou algum subtipo) e não `null`, o resultado de `is operator` é `true`.

Determinadas combinações de tipo estático do lado esquerdo e do tipo fornecido são consideradas incompatíveis e resultam em erro de tempo de compilação. Um valor de tipo estático `E` é considerado *compatível* com um tipo, `T` se houver uma conversão de identidade, uma conversão de referência implícita, uma conversão de boxing, uma conversão de referência explícita ou uma conversão de unboxing de `E` para `T`, ou se um desses tipos for um tipo aberto. É um erro de tempo de compilação se uma entrada do tipo `E` não for *compatível* com o padrão com o *tipo* em um padrão de tipo com o qual ele é correspondido.

O padrão de tipo é útil para executar testes de tipo de tempo de execução de tipos de referência e substitui o idioma

```
C#
```

```
var v = expr as Type;
if (v != null) { // code using v
```

Com um pouco mais conciso

```
C#
```

```
if (expr is Type v) { // code using v
```

Erro se o *tipo* for um tipo de valor anulável.

O padrão de tipo pode ser usado para testar valores de tipos anuláveis: um valor do tipo `Nullable<T>` (ou um Boxed `T`) corresponde a um padrão de tipo `T2 id` se o valor for não nulo e o tipo de `T2` for ou `T` algum tipo ou interface base de `T`. Por exemplo, no fragmento de código

```
C#  
  
int? x = 3;  
if (x is int v) { // code using v }
```

A condição da `if` instrução está `true` em tempo de execução e a variável `v` contém o valor `3` do tipo `int` dentro do bloco. Depois de bloquear, a variável `v` está no escopo, mas não definitivamente atribuída.

## Padrão de constante

```
antlr  
  
constant_pattern  
: constant_expression  
;
```

Um padrão constante testa o valor de uma expressão em relação a um valor constante. A constante pode ser qualquer expressão constante, como um literal, o nome de uma variável declarada `const` ou uma constante de enumeração. Quando o valor de entrada não é um tipo aberto, a expressão constante é convertida implicitamente no tipo da expressão correspondente; Se o tipo do valor de entrada não for *compatível* com o padrão com o tipo da expressão constante, a operação de correspondência de padrões será um erro.

O padrão `c` é considerado compatível com o valor de entrada convertido *e se* `object.Equals(c, e)` retornar `true`.

Esperamos ver `e is null` como a maneira mais comum de testar `null` no código recém escrito, pois ele não pode invocar um definido pelo usuário `operator==`.

## Padrão de var

```
antlr  
  
var_pattern  
: 'var' designation  
;
```

```

designation
: simple_designation
| tuple_designation
;
simple_designation
: single_variable_designation
| discard_designation
;
single_variable_designation
: identifier
;
discard_designation
: '_'
;
tuple_designation
: '(' designations? ')'
;
designations
: designation
| designations ',' designation
;

```

Se a *designação* for uma *simple\_designation*, uma expressão e corresponder ao padrão. Em outras palavras, uma correspondência a um padrão *var* sempre é realizada com um *simple\_designation*. Se o *simple\_designation* for um *single\_variable\_designation*, o valor de *e* será associado a uma variável local introduzida recentemente. O tipo da variável local é o tipo estático de *e*.

Se a *designação* for uma *tuple\_designation*, o padrão será equivalente a uma *positional\_pattern* da `(var designação de formulário,...)` onde os *s* de *design* são aqueles encontrados no *tuple\_designation*. Por exemplo, o padrão `var (x, (y, z))` é equivalente a `(var x, (var y, var z))`.

Erro se o nome for `var` associado a um tipo.

## Descartar padrão

```

antlr

discard_pattern
: '_'
;
```

Uma expressão *e* corresponde ao padrão `_` sempre. Em outras palavras, cada expressão corresponde ao padrão de descarte.

Um padrão de descarte não pode ser usado como o padrão de um *is\_pattern\_expression*.

## Padrão posicional

Um padrão posicional verifica se o valor de entrada não é `null`, invoca um `Deconstruct` método apropriado e executa uma correspondência de padrão adicional nos valores resultantes. Ele também dá suporte a uma sintaxe de padrão semelhante a tupla (sem o tipo fornecido) quando o tipo do valor de entrada é o mesmo que o tipo que contém `Deconstruct`, ou se o tipo do valor de entrada é um tipo de tupla, ou se o tipo do valor de entrada é `object` ou `ITuple` e o tipo de tempo de execução da expressão implementa `ITuple`.

```
antlr

positional_pattern
    : type? '(' subpatterns? ')' property_subpattern? simple_designation?
    ;
subpatterns
    : subpattern
    | subpattern ',' subpatterns
    ;
subpattern
    : pattern
    | identifier ':' pattern
    ;
```

Se o *tipo* for omitido, levaremos para ser o tipo estático do valor de entrada.

Dada uma correspondência de um valor de entrada para o *tipo* de padrão (`subpattern_list`), um método é selecionado pesquisando em *tipo* para declarações acessíveis `Deconstruct` e selecionando um entre eles usando as mesmas regras que para a declaração de desconstrução.

Erro se um *positional\_pattern* omitir o tipo, tiver um único *subpadrão* sem um *identificador*, não tiver nenhum *property\_subpattern* e não tiver *simple\_designation*. Essa ambiguidade é desambiguada entre um *constant\_pattern* que está entre parênteses e um *positional\_pattern*.

Para extrair os valores para corresponder aos padrões na lista,

- Se o *tipo* foi omitido e o tipo do valor de entrada é um tipo de tupla, o número de subpadrões é necessário para ser o mesmo que a cardinalidade da tupla. Cada elemento de tupla é correspondido em relação ao *subpadrão* correspondente e a

correspondência é realizada com sucesso se todas elas forem bem sucedidos. Se qualquer *subpadrão* tiver um *identificador*, isso deverá nomear um elemento de tupla na posição correspondente no tipo de tupla.

- Caso contrário, se houver uma adequada `Deconstruct` como um membro *do tipo*, será um erro de tempo de compilação se o tipo do valor de entrada não for compatível com o *padrão* com o *tipo*. Em tempo de execução, o valor de entrada é testado em relação ao *tipo*. Se isso falhar, a correspondência de padrão posicional falhará. Se tiver sucesso, o valor de entrada será convertido para esse tipo e `Deconstruct` será invocado com novas variáveis geradas pelo compilador para receber os `out` parâmetros. Cada valor recebido é correspondido em relação ao *subpadrão* correspondente e a correspondência é realizada com êxito se todas elas forem bem sucedidos. Se qualquer *subpadrão* tiver um *identificador*, isso deverá nomear um parâmetro na posição correspondente de `Deconstruct`.
- Caso contrário, se o *tipo* foi omitido, e o valor de entrada for do tipo `object` ou `ITuple` ou algum tipo que possa ser convertido `ITuple` por uma conversão de referência implícita e nenhum *identificador* aparecer entre os subpadrões, Corresponderei usando `ITuple`.
- Caso contrário, o padrão é um erro de tempo de compilação.

A ordem na qual os subpadrões são correspondidos em tempo de execução não é especificado e uma correspondência com falha pode não tentar corresponder a todos os subpadrões.

## Exemplo

Este exemplo usa muitos dos recursos descritos nesta especificação

```
c#  
  
var newState = (GetState(), action, hasKey) switch {  
    (DoorState.Closed, Action.Open, _) => DoorState.Opened,  
    (DoorState.Opened, Action.Close, _) => DoorState.Closed,  
    (DoorState.Closed, Action.Lock, true) => DoorState.Locked,  
    (DoorState.Locked, Action.Unlock, true) => DoorState.Closed,  
    (var state, _, _) => state };
```

## Padrão de propriedade

Um padrão de propriedade verifica se o valor de entrada não é `null` e corresponde recursivamente valores extraídos pelo uso de propriedades ou campos acessíveis.

```

property_pattern
  : type? property_subpattern simple_designation?
  ;
property_subpattern
  : '{' '}'
  | '{' subpatterns ','? '}''
  ;

```

Erro se qualquer *subpadrão* de um *property\_pattern* não contiver um *identificador* (ele deve ser do segundo formulário, que tem um *identificador*). Uma vírgula à direita após o último subpadrão é opcional.

Observe que um padrão de verificação nula sai de um padrão de propriedade trivial. Para verificar se a cadeia de caracteres `s` é não nula, você pode escrever qualquers dos seguintes formulários

C#

```

if (s is object o) ... // o is of type object
if (s is string x) ... // x is of type string
if (s is {} x) ... // x is of type string
if (s is {}) ...

```

Considerando uma correspondência de uma expressão *e* para o *tipo* de padrão `{ property_pattern_list }`, será um erro de tempo de compilação se a expressão *e* não for compatível com o *padrão* com o tipo *T* designado por *tipo*. Se o tipo estiver ausente, levaremos para ser o tipo estático de *e*. Se o *identificador* estiver presente, ele declara uma variável de padrão *do tipo Type*. Cada um dos identificadores que aparecem no lado esquerdo de seu *property\_pattern\_list* deve designar uma propriedade legível ou um campo de *T* acessível. Se o *simple\_designation* do *property\_pattern* estiver presente, ele definirá uma variável de padrão do tipo *T*.

Em tempo de execução, a expressão é testada em relação a *T*. Se isso falhar, a correspondência de padrão de propriedade falhará e o resultado será `false`. Se tiver sucesso, cada campo ou propriedade de *property\_subpattern* será lido e seu valor corresponderá ao seu padrão correspondente. O resultado da correspondência inteira é `false` apenas se o resultado de qualquer um deles for `false`. A ordem na qual os subpadrões são correspondidos não é especificada e uma correspondência com falha pode não corresponder a todos os subpadrões em tempo de execução. Se a correspondência for realizada com sucesso e a *simple\_designation* da *property\_pattern* for uma *single\_variable\_designation*, ela definirá uma variável do tipo *T* que recebe o valor correspondente.

Observação: o padrão de propriedade pode ser usado para correspondência de padrões com tipos anônimos.

## Exemplo

```
C#  
  
if (o is string { Length: 5 } s)
```

## Expressão switch

Um *switch\_expression* é adicionado à `switch` semântica do tipo suporte para um contexto de expressão.

A sintaxe da linguagem C# é aumentada com as seguintes produções sintáticas:

```
antlr  
  
multiplicative_expression  
: switch_expression  
| multiplicative_expression '*' switch_expression  
| multiplicative_expression '/' switch_expression  
| multiplicative_expression '%' switch_expression  
;  
switch_expression  
: range_expression 'switch' '{' '}'  
| range_expression 'switch' '{' switch_expression_arms ','? '}'  
;  
switch_expression_arms  
: switch_expression_arm  
| switch_expression_arms ',' switch_expression_arm  
;  
switch_expression_arm  
: pattern case_guard? '=>' expression  
;  
case_guard  
: 'when' null_coalescing_expression  
;
```

O *switch\_expression* não é permitido como um *expression\_statement*.

Estamos pensando em relaxar isso em uma revisão futura.

O tipo de *switch\_expression* é o [melhor tipo comum](#) de expressões que aparecem à direita dos `=>` tokens da *switch\_expression\_arm*s se existir um tipo e a expressão em todos os ARM da expressão switch puder ser convertida implicitamente nesse tipo. Além

disso, adicionamos uma nova conversão de expressão de switch, que é uma conversão implícita predefinida de uma expressão de switch para cada tipo `T` para o qual existe uma conversão implícita da expressão de cada ARM para `T`.

Ocorrerá um erro se algum padrão de `switch_expression_arm` não puder afetar o resultado, pois algum padrão e proteção anteriores sempre corresponderá.

Uma expressão de switch é considerada *exaustiva* se algum ARM da expressão switch tratar cada valor de sua entrada. O compilador deverá produzir um aviso se uma expressão de comutador não for *exaustiva*.

No tempo de execução, o resultado da `switch_expression` é o valor da expressão da primeira `switch_expression_arm` para a qual a expressão no lado esquerdo da `switch_expression` corresponde ao padrão do `switch_expression_arm` e para o qual a `case_guard` do `switch_expression_arm`, se presente, é avaliada como `true`. Se não houver tal `switch_expression_arm`, o `switch_expression` lançará uma instância da exceção `System.Runtime.CompilerServices.SwitchExpressionException`.

## Parênteses opcional ao alternar para um literal de tupla

Para alternar para um literal de tupla usando o `switch_statement`, você precisa escrever o que parece ser parênteses redundante

```
C#  
  
switch ((a, b))  
{
```

Para permitir

```
C#  
  
switch (a, b)  
{
```

os parênteses da instrução switch são opcionais quando a expressão que está sendo ativada é um literal de tupla.

## Ordem de avaliação na correspondência de padrões

Dar a flexibilidade do compilador ao reordenar as operações executadas durante a correspondência de padrões pode permitir a flexibilidade que pode ser usada para

melhorar a eficiência da correspondência de padrões. O requisito (não imposto) seria que as propriedades acessadas em um padrão, e os métodos desconstruir, precisam ser "puras" (com efeito colateral gratuito, idempotente, etc.). Isso não significa que adicionaremos pureza como um conceito de linguagem, apenas que permitiremos a flexibilidade do compilador em operações de reordenação.

**Resolução 2018-04-04 LDM:** confirmada: o compilador tem permissão para reordenar chamadas para `Deconstruct`, acessos de propriedade e invocações de métodos no `ITuple`, e pode assumir que os valores retornados são os mesmos de várias chamadas. O compilador não deve invocar funções que não afetem o resultado, e teremos muito cuidado antes de fazer qualquer alteração na ordem de avaliação gerada pelo compilador no futuro.

## Algumas otimizações possíveis

A compilação da correspondência de padrões pode tirar proveito de partes comuns de padrões. Por exemplo, se o teste de tipo de nível superior de dois padrões sucessivos em um *switch\_statement* for o mesmo tipo, o código gerado poderá ignorar o teste de tipo para o segundo padrão.

Quando alguns dos padrões são inteiros ou cadeias de caracteres, o compilador pode gerar o mesmo tipo de código que ele gera para uma instrução *switch* em versões anteriores do idioma.

Para obter mais informações sobre esses tipos de otimizações, consulte [\[Scott e Ramsey \(2000\)\]](#).

# métodos de interface padrão

Artigo • 16/09/2021

- [x] proposta
- [] Protótipo: [em andamento](#)
- [] Implementação: nenhuma
- [] Especificação: em andamento, abaixo

## Resumo

Adicionar suporte para *métodos de extensão virtual* – métodos em interfaces com implementações concretas. Uma classe ou estrutura que implementa tal interface é necessária para ter uma única implementação *mais específica* para o método de interface, implementada pela classe ou struct, ou herdada de suas classes ou interfaces base. Os métodos de extensão virtual permitem que um autor de API adicione métodos a uma interface em versões futuras sem perder a origem ou a compatibilidade binária com as implementações existentes dessa interface.

Eles são semelhantes aos "[métodos padrão](#)" do Java.

(Com base na provável técnica de implementação), esse recurso requer suporte correspondente no CLI/CLR. Os programas que tiram proveito desse recurso não podem ser executados em versões anteriores da plataforma.

## Motivação

As principais motivações para esse recurso são

- Os métodos de interface padrão permitem que um autor de API adicione métodos a uma interface em versões futuras sem perder a origem ou a compatibilidade binária com as implementações existentes dessa interface.
- O recurso permite que o C# interopere com APIs voltadas para [Android \(Java\)](#) e [iOS \(Swift\)](#), que dão suporte a recursos semelhantes.
- Como acontece, adicionar implementações de interface padrão fornece os elementos do recurso de linguagem "características" ([https://en.wikipedia.org/wiki/Trait\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Trait_(computer_programming))). As características comprovaram ser uma técnica de programação avançada (<http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf>).

# Design detalhado

A sintaxe de uma interface é estendida para permitir

- declarações de membro que declaram constantes, operadores, construtores estáticos e tipos aninhados;
- um *corpo* para um método, um indexador, uma propriedade ou um acessador de evento (ou seja, uma implementação "padrão");
- declarações de membro que declaram campos estáticos, métodos, propriedades, indexadores e eventos;
- declarações de membro usando a sintaxe de implementação de interface explícita; e
- Modificadores de acesso explícitos (o acesso padrão é `public`).

Membros com corpos permitem que a interface forneça uma implementação "padrão" para o método em classes e estruturas que não fornecem uma implementação de substituição.

As interfaces não podem conter o estado da instância. Embora os campos estáticos agora sejam permitidos, os campos de instância não são permitidos em interfaces. Não há suporte para propriedades automáticas de instância em interfaces, pois elas declarariam implicitamente um campo oculto.

Os métodos estáticos e privados permitem a refatoração útil e a organização do código usado para implementar a API pública da interface.

Uma substituição de método em uma interface deve usar a sintaxe de implementação de interface explícita.

É um erro declarar um tipo de classe, tipo de struct ou tipo de enumeração dentro do escopo de um parâmetro de tipo que foi declarado com um *variance\_annotation*. Por exemplo, a declaração de `C` abaixo é um erro.

C#

```
interface IOuter<out T>
{
    class C { } // error: class declaration within the scope of variant type
parameter 'T'
}
```

## Métodos concretos em interfaces

A forma mais simples desse recurso é a capacidade de declarar um *método concreto* em uma interface, que é um método com um corpo.

```
C#
```

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
```

Uma classe que implementa essa interface não precisa implementar seu método concreto.

```
C#
```

```
class C : IA { } // OK

IA i = new C();
i.M(); // prints "IA.M"
```

A substituição final de `IA.M` na classe `C` é o método concreto `M` declarado em `IA`.

Observe que uma classe não herda membros de suas interfaces; Isso não é alterado por esse recurso:

```
C#
```

```
new C().M(); // error: class 'C' does not contain a member 'M'
```

Dentro de um membro de instância de uma interface, `this` tem o tipo da interface delimitadora.

## Modificadores em interfaces

A sintaxe de uma interface é relaxada para permitir modificadores em seus membros. Os itens a seguir são permitidos: `private` `protected` `internal` `public` `virtual` `abstract` `sealed`, `static`, `extern` e `partial`.

***To do:*** Verifique quais outros modificadores existem.

Um membro de interface cuja declaração inclui um corpo é um `virtual` membro, a menos que o `sealed` `private` modificador ou seja usado. O `virtual` modificador pode ser usado em um membro de função que, de outra forma, seria implicitamente `virtual`.

. Da mesma forma, embora `abstract` o seja o padrão em membros de interface sem corpos, esse modificador pode ser fornecido explicitamente. Um membro não virtual pode ser declarado usando a `sealed` palavra-chave.

É um erro para um `private` membro de `sealed` função ou de uma interface não ter corpo. Um `private` membro de função não pode ter o modificador `sealed`.

Os modificadores de acesso podem ser usados em membros de interface de todos os tipos de membros que são permitidos. O nível de acesso `public` é o padrão, mas pode ser fornecido explicitamente.

**Abrir problema:** Precisamos especificar o significado preciso dos modificadores de acesso, como `protected` e `internal`, e quais declarações fazem e não os substituem (em uma interface derivada) ou os implementam (em uma classe que implementa a interface).

As interfaces podem declarar `static` Membros, incluindo tipos aninhados, métodos, indexadores, propriedades, eventos e construtores estáticos. O nível de acesso padrão para todos os membros de interface é `public`.

As interfaces não podem declarar construtores, destruidores ou campos de instância.

**\*Problema fechado:** \_ devem ser permitidas declarações de operador em uma interface? Provavelmente não há operadores de conversão, mas e quanto a outros?  
*Decisão:* operadores são permitidos *except \* para operadores de conversão, igualdade e desigualdade*.  
*Decision\_:* Operators are permitted \_except\* for conversion, equality, and inequality operators.

**\*Problema fechado:** \_ deve `new` ser permitido em declarações de membro de interface que ocultam membros de interfaces base? \_ *Decisão \*:* Sim.

**\*Problema fechado:** \_ não permitimos atualmente `partial` em uma interface ou seus membros. Isso exigiria uma proposta separada. \_ *Decisão \*:* Sim.  
[https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface ↴](https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface)

## Substituições em interfaces

As declarações de substituição (ou seja, aquelas que contêm o `override` Modificador) permitem que o programador forneça uma implementação mais específica de um

membro virtual em uma interface na qual o compilador ou tempo de execução não encontraria um. Ele também permite transformar um membro abstrato de uma superinterface em um membro padrão em uma interface derivada. Uma declaração de substituição tem permissão para substituir *explicitamente* um método de interface base específico qualificando a declaração com o nome da interface (nenhum modificador de acesso é permitido nesse caso). Substituições implícitas não são permitidas.

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); } // explicitly named
}
interface IC : IA
{
    override void M() { WriteLine("IC.M"); } // implicitly named
}
```

As declarações de substituição em interfaces não podem ser declaradas `sealed`.

`virtual` Membros de funções públicas em uma interface podem ser substituídos em uma interface derivada explicitamente (qualificando o nome na declaração de substituição com o tipo de interface que declarou originalmente o método e omitindo um modificador de acesso).

`virtual` os membros de função em uma interface só podem ser substituídos explicitamente (não implicitamente) em interfaces derivadas e os membros que não são `public` apenas implantados em uma classe ou struct explicitamente (não implicitamente). Em ambos os casos, o membro substituído ou implementado deve estar *acessível* onde é substituído.

## Reabstração

Um método virtual (concreto) declarado em uma interface pode ser substituído para ser abstrato em uma interface derivada

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
```

```
interface IB : IA
{
    abstract void IA.M();
}
class C : IB { } // error: class 'C' does not implement 'IA.M'.
```

O `abstract` modificador não é necessário na declaração de `IB.M` (que é o padrão nas interfaces), mas é provável que seja uma prática recomendada ser explícita em uma declaração de substituição.

Isso é útil em interfaces derivadas em que a implementação padrão de um método é inadequada e uma implementação mais apropriada deve ser fornecida pela implementação de classes.

**Abrir problema:** A reabstração deve ser permitida?

## A regra de substituição mais específica

Exigimos que cada interface e classe tenham uma *substituição mais específica* para cada membro virtual entre as substituições que aparecem no tipo ou suas interfaces diretas e indiretas. A *substituição mais específica* é uma substituição exclusiva que é mais específica do que todas as outras substituições. Se não houver nenhuma substituição, o membro em si será considerado a substituição mais específica.

Uma substituição `M1` é considerada *mais específica* do que outra substituição `M2` se `M1` é declarada no tipo `T1`, `M2` é declarada no tipo `T2` e qualquer uma

1. `T1` contém `T2` entre suas interfaces diretas ou indiretas, ou
2. `T2` é um tipo de interface, mas `T1` não é um tipo de interface.

Por exemplo:

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    void IA.M() { WriteLine("IC.M"); }
}
```

```
interface ID : IB, IC { } // error: no most specific override for 'IA.M'
abstract class C : IB, IC { } // error: no most specific override for 'IA.M'
abstract class D : IA, IB, IC // ok
{
    public abstract void M();
}
```

A regra de substituição mais específica garante que um conflito (ou seja, uma ambiguidade resultante da herança de losango) seja resolvido explicitamente pelo programador no ponto em que surge o conflito.

Como damos suporte a substituições abstratas explícitas em interfaces, poderíamos fazer isso em classes também

C#

```
abstract class E : IA, IB, IC // ok
{
    abstract void IA.M();
}
```

**Problema aberto:** devemos dar suporte a substituições abstratas de interface explícita em classes?

Além disso, é um erro se, em uma declaração de classe, a substituição mais específica de algum método de interface for uma substituição abstrata que foi declarada em uma interface. Esta é uma regra existente renovada usando a nova terminologia.

C#

```
interface IF
{
    void M();
}
abstract class F : IF { } // error: 'F' does not implement 'IF.M'
```

É possível que uma propriedade virtual declarada em uma interface tenha uma substituição mais específica para seu `get` acessador em uma interface e uma substituição mais específica para seu `set` acessador em uma interface diferente. Isso é considerado uma violação da regra de *substituição mais específica*.

## Métodos `static` e `private`

Como as interfaces agora podem conter código executável, é útil abstrair o código comum em métodos privados e estáticos. Agora, permitimos isso em interfaces.

\***Problema fechado**: devemos dar suporte a métodos privados? *Devemos dar suporte a métodos estáticos?* *Decisão: Sim*\*

**Problema de abertura**: devemos permitir que os métodos de interface sejam `protected` ou `internal` ou outro acesso? Em caso afirmativo, quais são as semânticas? Eles são `virtual` por padrão? Nesse caso, há uma maneira de torná-los não virtuais?

**Problema aberto**: se damos suporte a métodos estáticos, devemos dar suporte a operadores (estáticos)?

## Invocações de interface base

O código em um tipo derivado de uma interface com um método padrão pode invocar explicitamente a implementação de "base" dessa interface.

C#

```
interface I0
{
    void M() { Console.WriteLine("I0"); }
}
interface I1 : I0
{
    override void M() { Console.WriteLine("I1"); }
}
interface I2 : I0
{
    override void M() { Console.WriteLine("I2"); }
}
interface I3 : I1, I2
{
    // an explicit override that invoke's a base interface's default method
    void I0.M() { I2.base.M(); }
}
```

Um método de instância (não estático) é permitido para invocar a implementação de um método de instância acessível em uma interface base direta não virtualmente ao nomeá-lo usando a sintaxe `base(Type).M`. Isso é útil quando uma substituição necessária para ser fornecida devido à herança de losango é resolvida pela delegação a uma implementação de base específica.

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    override void IA.M() { WriteLine("IC.M"); }
}

class D : IA, IB, IC
{
    void IA.M() { base(IB).M(); }
}
```

Quando um `virtual abstract` membro ou é acessado usando a sintaxe `base(Type).M`, é necessário que `Type` contenha uma *substituição exclusiva mais específica* para o `M`.

## Vinculando cláusulas base

Agora, as interfaces contêm tipos. Esses tipos podem ser usados na cláusula base como interfaces base. Ao ligar uma cláusula base, talvez seja necessário saber o conjunto de interfaces base para associar esses tipos (por exemplo, para pesquisar neles e resolver o acesso protegido). O significado da cláusula base de uma interface é, portanto, definido circularmente. Para interromper o ciclo, adicionamos novas regras de idioma correspondentes a uma regra semelhante já em vigor para classes.

Ao determinar o significado da *interface\_base* de uma interface, as interfaces base são temporariamente consideradas vazias. Intuitivamente, isso garante que o significado de uma cláusula base não possa ser recursivamente dependente de si mesma.

Nós usamos as seguintes regras:

"Quando uma classe B deriva de uma classe A, é um erro de tempo de compilação para um para depender de B. Uma classe **depende diretamente** de sua classe base direta (se houver) e **depende diretamente da classe** na qual ela é imediatamente aninhada (se houver). Dada essa definição, o conjunto completo de **classes** sobre as quais uma classe depende é o fechamento reflexivo e transitivo da relação **depende diretamente de**.

É um erro de tempo de compilação para uma interface herdar direta ou indiretamente de si mesma. As **interfaces base** de uma interface são as interfaces base explícitas e suas

interfaces base. Em outras palavras, o conjunto de interfaces base é o fechamento transitivo completo das interfaces base explícitas, suas interfaces base explícitas e assim por diante.

Estamos ajustando-os da seguinte maneira:

Quando uma classe B deriva de uma classe A, é um erro de tempo de compilação para um para depender de B. Uma classe **depende diretamente** de sua classe base direta (se houver) e **depende diretamente do tipo** no qual ele é imediatamente aninhado (se houver).

Quando uma interface IB estende uma interface IA, é um erro de tempo de compilação para IA depender da IB. Uma interface **depende diretamente** de suas interfaces base diretas (se houver) e **depende diretamente do tipo** no qual ele é imediatamente aninhado (se houver).

Dadas essas definições, o conjunto completo de **tipos** sobre os quais um tipo depende é o fechamento reflexivo e transitivo da relação **depende diretamente** de.

## Efeito em programas existentes

As regras apresentadas aqui destinam-se a não afetar o significado dos programas existentes.

Exemplo 1:

```
C#  
  
interface IA  
{  
    void M();  
}  
class C: IA // Error: IA.M has no concrete most specific override in C  
{  
    public static void M() { } // method unrelated to 'IA.M' because static  
}
```

Exemplo 2:

```
C#  
  
interface IA  
{  
    void M();  
}  
class Base: IA
```

```
{  
    void IA.M() { }  
}  
class Derived: Base, IA // OK, all interface members have a concrete most  
specific override  
{  
    private void M() { } // method unrelated to 'IA.M' because private  
}
```

As mesmas regras fornecem resultados semelhantes à situação análoga que envolve métodos de interface padrão:

C#

```
interface IA  
{  
    void M() { }  
}  
class Derived: IA // OK, all interface members have a concrete most specific  
override  
{  
    private void M() { } // method unrelated to 'IA.M' because private  
}
```

\***Problema fechado:** Confirme se essa é uma consequência pretendida da especificação. *Decisão: Sim*\*

## Resolução do método de tempo de execução

**Problema fechado:** A especificação deve descrever o algoritmo de resolução do método de tempo de execução na face dos métodos padrão de interface. Precisamos garantir que a semântica seja consistente com a semântica da linguagem, por exemplo, quais métodos declarados fazem e não substituem ou implementam um `internal` método.

## API de suporte CLR

Para que os compiladores detectem quando estão compilando para um tempo de execução que dá suporte a esse recurso, as bibliotecas para esses tempos de execução são modificadas para anunciar esse fato por meio da API discutida em <https://github.com/dotnet/corefx/issues/17116>. Adicionamos

C#

```
namespace System.Runtime.CompilerServices
{
    public static class RuntimeFeature
    {
        // Presence of the field indicates runtime support
        public const string DefaultInterfaceImplementation =
nameof(DefaultInterfaceImplementation);
    }
}
```

\*Abrir problema \_: é o melhor nome para o recurso \_CLR \*? O recurso CLR faz muito mais do que apenas isso (por exemplo, libera as restrições de proteção, dá suporte a substituições em interfaces, etc). Talvez ele deva ser chamado de algo como "métodos concretos em interfaces" ou "características"?

## Outras áreas a serem especificadas

- [] Seria útil catalogar os tipos de efeitos de compatibilidade binária e de origem causados pela adição de métodos de interface padrão e substituições a interfaces existentes.

## Desvantagens

Essa proposta requer uma atualização coordenada para a especificação do CLR (para dar suporte a métodos concretos em interfaces e resolução de métodos). Portanto, é razoavelmente "caro" e pode valer a pena fazer isso em combinação com outros recursos que também antecipamos a necessidade de alterações no CLR.

## Alternativas

Nenhum.

## Perguntas não resolvidas

- As perguntas abertas são chamadas em toda a proposta, acima.
- Consulte também [https://github.com/dotnet/csharplang/issues/406 ↗](https://github.com/dotnet/csharplang/issues/406) para obter uma lista de perguntas abertas.
- A especificação detalhada deve descrever o mecanismo de resolução usado em tempo de execução para selecionar o método preciso a ser invocado.

- A interação dos metadados produzidos por novos compiladores e consumidos por compiladores mais antigos precisa ser realizada em detalhes. Por exemplo, precisamos garantir que a representação de metadados que usamos não cause a adição de uma implementação padrão em uma interface para interromper uma classe existente que implementa essa interface quando compilada por um compilador mais antigo. Isso pode afetar a representação de metadados que podemos usar.
- O design deve considerar a interoperabilidade com outras linguagens e compiladores existentes para outras linguagens.

## Perguntas resolvidas

### Substituição abstrata

A especificação de rascunho anterior continha a capacidade de "reabstrair" um método herdado:

```
C#  
  
interface IA  
{  
    void M();  
}  
interface IB : IA  
{  
    override void M() { }  
}  
interface IC : IB  
{  
    override void M(); // make it abstract again  
}
```

Minhas notas para 2017-03-20 mostraram que decidimos não permitir isso. No entanto, há pelo menos dois casos de uso para ele:

1. As APIs do Java, com as quais alguns usuários desse recurso esperamos interoperar dependem dessa instalação.
2. A programação com *características* se beneficia disso. A reabstração é um dos elementos do recurso de linguagem "características" ([https://en.wikipedia.org/wiki/Trait\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Trait_(computer_programming))) . O seguinte é permitido com classes:

```
C#
```

```
public abstract class Base
{
    public abstract void M();
}
public abstract class A : Base
{
    public override void M() { }
}
public abstract class B : A
{
    public override abstract void M(); // reabstract Base.M
}
```

Infelizmente, esse código não pode ser refatorado como um conjunto de interfaces (características), a menos que isso seja permitido. Pelo *princípio de Jared de ganância*, ele deve ser permitido.

**Problema fechado:** A reabstração deve ser permitida? Ok Minhas anotações estavam erradas. As [observações do LDM ↗](#) dizem que a reabstração é permitida em uma interface. Não está em uma classe.

## Modificador e modificador virtual vs lacrado

De [Aleksey Tsingauz ↗](#):

Decidimos permitir que modificadores explicitamente declarados em membros de interface, a menos que haja um motivo para não permitir alguns deles. Isso traz uma pergunta interessante sobre o modificador virtual. Eles devem ser necessários em membros com implementação padrão?

Poderíamos dizer que:

- Se não houver nenhuma implementação e nenhum virtual, nem lacrado for especificado, supomos que o membro é abstrato.
- Se houver uma implementação e nenhum resumo, nem lacrado for especificado, supomos que o membro é virtual.
- o modificador lacrado é necessário para tornar um método não virtual nem abstrato.

Como alternativa, poderíamos dizer que o modificador virtual é necessário para um membro virtual. Ou seja, se houver um membro com implementação não explicitamente marcado com modificador virtual, ele não será virtual nem abstrato. Essa abordagem pode proporcionar uma melhor experiência quando um método é movido de uma classe para uma interface:

- um método abstrato permanece abstrato.
- um método virtual permanece virtual.
- um método sem nenhum modificador não permanece virtual nem abstract.
- o modificador lacrado não pode ser aplicado a um método que não seja uma substituição.

O que você acha?

**Problema fechado:** Um método concreto (com implementação) deve ser implicitamente `virtual`? Ok

**Decisões:** Feitas no LDM 2017-04-05:

1. Não `virtual` deve ser expresso explicitamente por meio `sealed` de ou `private`.
2. `sealed` é a palavra-chave para tornar membros da instância da interface com corpos não virtuais
3. Queremos permitir todos os modificadores em interfaces
4. A acessibilidade padrão para membros de interface é pública, incluindo tipos aninhados
5. Membros de função privada em interfaces são lacrados implicitamente e `sealed` não são permitidos neles.
6. As classes privadas (em interfaces) são permitidas e podem ser seladas, e isso significa lacrado na classe sensação de `sealed`.
7. Uma boa proposta está ausente, parcial ainda não é permitida em interfaces ou seus membros.

## Compatibilidade binária 1

Quando uma biblioteca fornece uma implementação padrão

C#

```
interface I1
{
    void M() { Impl1 }
}
interface I2 : I1
{
}
class C : I2
{}
```

Entendemos que a implementação de `I1.M` no `C` é `I1.M`. E se o assembly contendo `I2` for alterado da seguinte maneira e recompilada

```
C#  
  
interface I2 : I1  
{  
    override void M() { Impl2 }  
}
```

Mas `C` não é recompilado. O que acontece quando o programa é executado? Uma invocação de `(C as I1).M()`

1. Torna `I1.M`
2. Torna `I2.M`
3. Gera algum tipo de erro de tempo de execução

**Decisão:** Tornou-se 2017-04-11: execuções `I2.M`, que é a substituição mais específica, sem ambigüidade, no tempo de execução.

## Acessadores de evento (fechados)

**Problema fechado:** Um evento pode ser substituído "piecewise"?

Considere este caso:

```
C#  
  
public interface I1  
{  
    event T e1;  
}  
public interface I2 : I1  
{  
    override event T  
    {  
        add { }  
        // error: "remove" accessor missing  
    }  
}
```

Essa implementação "parcial" do evento não é permitida porque, como em uma classe, a sintaxe de uma declaração de evento não permite apenas um acessador; ambos (ou nenhum) devem ser fornecidos. Você poderia realizar a mesma coisa permitindo que o

acessador remove abstract na sintaxe seja implicitamente abstrato pela ausência de um corpo:

```
C#  
  
public interface I1  
{  
    event T e1;  
}  
public interface I2 : I1  
{  
    override event T  
    {  
        add {}  
        remove; // implicitly abstract  
    }  
}
```

Observe que *essa é uma sintaxe nova (proposta)*. Na gramática atual, os acessadores de evento têm um corpo obrigatório.

**Problema fechado:** Um acessador de evento pode ser (implicitamente) abstrato pela omissão de um corpo, da mesma forma que os métodos em interfaces e acessadores de propriedade são (implicitamente) abstratos pela omissão de um corpo?

**Decisão:** (2017-04-18) não, as declarações de evento exigem tanto acessadores concretos (ou nenhum).

## Reabstração em uma classe (fechada)

**Problema fechado:** Devemos confirmar que isso é permitido (caso contrário, adicionar uma implementação padrão seria uma alteração significativa):

```
C#  
  
interface I1  
{  
    void M() {}  
}  
abstract class C : I1  
{  
    public abstract void M(); // implement I1.M with an abstract method in C  
}
```

**Decisão:** (2017-04-18) Sim, a adição de um corpo a uma declaração de membro de interface não deve quebrar C.

## Substituição lacrada (fechada)

A pergunta anterior pressupõe implicitamente que o `sealed` modificador pode ser aplicado a um `override` em uma interface. Isso contradiz a especificação de rascunho. Queremos permitir o lacre de uma substituição? Devem ser considerados efeitos de compatibilidade de origem e binário de lacre.

**Problema fechado:** Devemos permitir lacrar uma substituição?

**Decisão:** (2017-04-18) não é permitido `sealed` em substituições em interfaces. O único uso de `sealed` membros de interface é torná-los não virtuais em sua declaração inicial.

## Herança de diamante e classes (fechadas)

O rascunho da proposta prefere substituições de classe a substituições de interface em cenários de herança em losango:

Exigimos que cada interface e classe tenham uma *substituição mais específica* para cada método de interface entre as substituições que aparecem no tipo ou suas interfaces diretas e indiretas. A *substituição mais específica* é uma substituição exclusiva que é mais específica do que todas as outras substituições. Se não houver nenhuma substituição, o próprio método será considerado a substituição mais específica.

Uma substituição `M1` é considerada *mais específica* do que outra substituição `M2` se `M1` é declarada no tipo `T1`, `M2` é declarada no tipo `T2` e qualquer uma

1. `T1` contém `T2` entre suas interfaces diretas ou indiretas, ou
2. `T2` é um tipo de interface, mas `T1` não é um tipo de interface.

O cenário é este

C#

```
interface IA
{
    void M();
}

interface IB : IA
```

```

{
    override void M() { WriteLine("IB"); }
}
class Base : IA
{
    void IA.M() { WriteLine("Base"); }
}
class Derived : Base, IB // allowed?
{
    static void Main()
    {
        Ia a = new Derived();
        a.M();           // what does it do?
    }
}

```

Devemos confirmar esse comportamento (ou decidir de outra forma)

\***Problema fechado:** \_ confirme a especificação de rascunho, acima, para \_most substituição específica \*, pois ela se aplica a classes e interfaces mistas (uma classe tem prioridade sobre uma interface). Consulte

[https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#diamonds-with-classes ↴](https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#diamonds-with-classes).

## Métodos de interface vs structs (fechados)

Há algumas interações de infeliz entre as estruturas e os métodos de interface padrão.

C#

```

interface IA
{
    public void M() { }
}
struct S : IA
{
}

```

Observe que os membros da interface não são herdados:

C#

```

var s = default(S);
s.M(); // error: 'S' does not contain a member 'M'

```

Consequentemente, o cliente deve caixar a estrutura para invocar métodos de interface

```
C#
```

```
IA s = default(S); // an S, boxed
s.M(); // ok
```

A Boxing dessa forma derrota os principais benefícios de um `struct` tipo. Além disso, qualquer método de mutação não terá efeito aparente, pois eles estão operando em uma *cópia em caixa* da estrutura:

```
C#
```

```
interface IB
{
    public void Increment() { P += 1; }
    public int P { get; set; }
}

struct T : IB
{
    public int P { get; set; } // auto-property
}

T t = default(T);
Console.WriteLine(t.P); // prints 0
(t as IB).Increment();
Console.WriteLine(t.P); // prints 0
```

**Problema fechado:** O que podemos fazer a respeito:

1. Proíba um `struct` de herdar uma implementação padrão. Todos os métodos de interface seriam tratados como abstratos em um `struct`. Em seguida, poderemos levar tempo mais tarde para decidir como fazê-lo funcionar melhor.
2. Surgirão com algum tipo de estratégia de geração de código que evita boxing. Dentro de um método como `IB.Increment`, o tipo de `this` talvez seria semelhante a um parâmetro de tipo restrito a `IB`. Em conjunto com isso, para evitar boxing no chamador, os métodos não abstratos seriam herdados de interfaces. Isso pode aumentar o trabalho de implementação do compilador e do CLR.
3. Não se preocupe e simplesmente deixe-o como um imperfeição.
4. Outras ideias?

**Decisão:** Não se preocupe e simplesmente deixe-o como um imperfeição. Consulte <https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#structs-and-default-implementations>.

## Invocações de interface base (fechadas)

A especificação de rascunho sugere uma sintaxe para invocações de interface base inspiradas por Java: `Interface.base.M()` . Precisamos selecionar uma sintaxe, pelo menos para o protótipo inicial. Meu favorito é `base<Interface>.M()` .

**Problema fechado:** Qual é a sintaxe para uma invocação de membro base?

**Decisão:** A sintaxe é `base(Interface).M()` . Consulte

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation> . A interface, portanto, denominada deve ser uma interface base, mas não precisa ser uma interface base direta.

**Abrir problema:** As invocações de interface base devem ser permitidas em membros de classe?

**Decisão:** Sim. <https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation>

## Substituindo membros de interface não-pública (fechado)

Em uma interface, membros não públicos de interfaces base são substituídos usando o `override` modificador. Se for uma substituição "explícita" que nomeia a interface que contém o membro, o modificador de acesso será omitido.

**Problema fechado:** Se for uma substituição "implícita" que não nomeie a interface, o modificador de acesso precisará corresponder?

**Decisão:** Somente membros públicos podem ser substituídos implicitamente e o acesso deve corresponder. Consulte

<https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing-a-non-public-interface-member-not-in-list>.

**Abrir problema:** O modificador de acesso é necessário, opcional ou omitido em uma substituição explícita, como `override void IB.M() {}` ?

**Abrir problema:** É `override` obrigatório, opcional ou omitido em uma substituição explícita, como `void IB.M() {}` ?

Como uma implementação de um membro de interface não-pública em uma classe?  
Talvez ele deva ser feito explicitamente?

```
C#  
  
interface IA  
{  
    internal void MI();  
    protected void MP();  
}  
class C : IA  
{  
    // are these implementations?  
    internal void MI() {}  
    protected void MP() {}  
}
```

**Problema fechado:** Como uma implementação de um membro de interface não-pública em uma classe?

**Decisão:** Você só pode implementar membros de interface não pública explicitamente.  
Consulte <https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing-a-non-public-interface-member-not-in-list>.

**Decisão:** nenhuma `override` palavra-chave permitida em membros de interface.  
<https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member>

## Compatibilidade binária 2 (fechado)

Considere o seguinte código no qual cada tipo está em um assembly separado

```
C#  
  
interface I1  
{  
    void M() { Impl1 }  
}  
interface I2 : I1  
{  
    override void M() { Impl2 }  
}  
interface I3 : I1  
{  
}  
class C : I2, I3
```

```
{  
}
```

Entendemos que a implementação de `I1.M` no `C` é `I2.M`. E se o assembly contendo `I3` for alterado da seguinte maneira e recompilada

```
C#
```

```
interface I3 : I1  
{  
    override void M() { Impl3 }  
}
```

Mas `C` não é recompilado. O que acontece quando o programa é executado? Uma invocação de `(C as I1).M()`

1. Tour `I1.M`
2. Tour `I2.M`
3. Tour `I3.M`
4. 2 ou 3, de forma determinista
5. Gera algum tipo de exceção de tempo de execução

*Decisão:* lançar uma exceção (5). Consulte

<https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#issues-in-default-interface-methods>.

## Permitir `partial` na interface? Legenda

Considerando que as interfaces podem ser usadas de maneiras análogas à forma como as classes abstratas são usadas, pode ser útil declará-las `partial`. Isso seria particularmente útil na face de geradores.

*Proposta:* Remova a restrição de idioma que as interfaces e os membros das interfaces não podem ser declarados `partial`.

*Decisão:* Sim. Consulte

<https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface>.

## Main em uma interface? Legenda

**Abrir problema:** Um `static Main` método em uma interface é candidato ao ponto de entrada do programa?

**Decisão:** Sim. Consulte

[https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#main-in-an-interface ↴](https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#main-in-an-interface).

## Confirmar a intenção de dar suporte a métodos não virtuais públicos (fechados)

Podemos confirmar (ou reverter) nossa decisão de permitir métodos públicos não virtuais em uma interface?

C#

```
interface IA
{
    public sealed void M() { }
```

**Problema semifechado:** (2017-04-18) achamos que ele será útil, mas voltará a ele. Esse é um bloco mental de blocos de modelo.

**Decisão:** Sim. [https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#confirm-that-we-support-public-non-virtual-methods ↴](https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#confirm-that-we-support-public-non-virtual-methods).

## Um `override` em uma interface introduz um novo membro? Legenda

Há algumas maneiras de observar se uma declaração de substituição introduz um novo membro ou não.

C#

```
interface IA
{
    void M(int x) { }
```

```
interface IB : IA
{
    override void M(int y) { }
```

```
interface IC : IB
```

```

{
    static void M2()
    {
        M(y: 3); // permitted?
    }
    override void IB.M(int z) { } // permitted? What does it override?
}

```

**Abrir problema:** Uma declaração de substituição em uma interface introduz um novo membro? Legenda

Em uma classe, um método de substituição é "visível" em alguns sentidos. Por exemplo, os nomes de seus parâmetros têm precedência sobre os nomes dos parâmetros no método substituído. Pode ser possível duplicar esse comportamento em interfaces, pois sempre há uma substituição mais específica. Mas queremos duplicar esse comportamento?

Além disso, é possível "substituir" um método de substituição? Sentido

**Decisão:** nenhuma `override` palavra-chave permitida em membros de interface.  
[https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member).

## Propriedades com um acessador privado (fechado)

Dizemos que os membros privados não são virtuais e a combinação de virtual e privada não é permitida. Mas e quanto a uma propriedade com um acessador particular?

C#

```

interface IA
{
    public virtual int P
    {
        get => 3;
        private set => { }
    }
}

```

Isso é permitido? O `set` acessador é aqui `virtual` ou não? Ele pode ser substituído onde estiver acessível? O seguinte implementa implicitamente o `get` acessador?

C#

```

class C : IA
{
    public int P
    {
        get => 4;
        set { }
    }
}

```

O seguinte é supostamente um erro porque IA.P.set não é virtual e também porque não está acessível?

```

C#

class C : IA
{
    int IA.P
    {
        get => 4;
        set { }
    }
}

```

**Decisão:** o primeiro exemplo é válido, enquanto o último não. Isso é resolvido de forma análoga em como ele já funciona em C#.

[https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#properties-with-a-private-accessor ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#properties-with-a-private-accessor)

## Invocações de interface base, redondo 2 (fechado)

Nossa "resolução" anterior de como lidar com invocações de base não oferece, na verdade, uma expressividade suficiente. Acontece que, em C# e no CLR, ao contrário do Java, você precisa especificar a interface que contém a declaração do método e o local da implementação que você deseja invocar.

Eu proponho a sintaxe a seguir para chamadas base em interfaces. Não estou de amor, mas ilustra o que qualquer sintaxe deve ser capaz de expressar:

```

C#

interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I4 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3, I4
{

```

```

void I1.M()
{
    base<I3>(I1).M(); // calls I3's implementation of I1.M
    base<I4>(I1).M(); // calls I4's implementation of I1.M
}
void I2.M()
{
    base<I3>(I2).M(); // calls I3's implementation of I2.M
    base<I4>(I2).M(); // calls I4's implementation of I2.M
}
}

```

Se não houver nenhuma ambiguidade, você poderá escrevê-la mais simplesmente

C#

```

interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I4 : I1 { void I1.M() { } }
interface I5 : I3, I4
{
    void I1.M()
    {
        base<I3>.M(); // calls I3's implementation of I1.M
        base<I4>.M(); // calls I4's implementation of I1.M
    }
}

```

Ou

C#

```

interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base(I1).M(); // calls I3's implementation of I1.M
    }
    void I2.M()
    {
        base(I2).M(); // calls I3's implementation of I2.M
    }
}

```

Ou

C#

```

interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base.M(); // calls I3's implementation of I1.M
    }
}

```

**Decisão:** decidida `base(N.I1<T>).M(s)`, em que, se tivermos uma associação de invocação, pode haver um problema aqui mais tarde.

[https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-11-14.md#default-interface-implementations ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-11-14.md#default-interface-implementations)

## Aviso para struct não implementar o método padrão? Legenda

@vancem declara que devemos considerar seriamente a geração de um aviso se uma declaração de tipo de valor não substituir algum método de interface, mesmo que herde uma implementação desse método de uma interface. Porque faz com que a boxing e a subminam chamadas restritas.

**Decisão:** isso parece algo mais adequado para um analisador. Também parece que esse aviso pode ser barulhento, pois ele será disparado mesmo se o método de interface padrão nunca for chamado e nenhuma Boxing ocorrerá.

[https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#warning-for-struct-not-implementing-default-method ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#warning-for-struct-not-implementing-default-method)

## Construtores estáticos de interface (fechados)

Quando os construtores estáticos da interface são executados? O rascunho da CLI atual propõe que ele ocorre quando o primeiro método ou campo estático é acessado. Se não houver nenhuma delas, ela pode nunca ser executada?

[2018-10-09 a equipe do CLR propõe "vai espelhar o que fazemos para valuetypes (verificação de cctor no acesso a cada método de instância)"]

**Decisão:** construtores estáticos também são executados na entrada para métodos de instância, se o construtor estático não tiver sido `beforefieldinit`, caso em que construtores estáticos são executados antes do acesso ao primeiro campo estático.

[https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#when-are-interface-static-constructors-run ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#when-are-interface-static-constructors-run)

# Criar reuniões

[2017-03-08 notas ↗](#) de reunião do LDM [2017-03-21 notas ↗](#) de reunião do LDM [2017-03-23 reunião "comportamento do CLR para métodos de interface padrão" ↗](#) [2017-04-05 notas ↗](#) de reunião do LDM [2017-04-11 notas ↗](#) de reunião do LDM [2017-04-18 notas ↗](#) de reunião do LDM [2017-04-19 notas ↗](#) de reunião do LDM [2017-05-17 notas ↗](#) de reunião do LDM [2017-05-31 notas ↗](#) de reunião do LDM [2017-06-14 notas ↗](#) de reunião do LDM [2018-10-17 notas ↗](#) de reunião do LDM [2018-11-14 notas de reunião do LDM ↗](#)

# Fluxos assíncronos

Artigo • 16/09/2021

- [x] proposta
- [x] protótipo
- [] Implementação
- [] Especificação

## Resumo

O C# tem suporte para métodos iteradores e métodos assíncronos, mas sem suporte para um método que seja um iterador e um método assíncrono. Devemos corrigir isso permitindo que o seja `await` usado em uma nova forma de `async` iterador, um que retorne um `IAsyncEnumerable<T>` ou `IAsyncEnumerator<T>` em vez de um `IEnumerable<T>` ou `IEnumerator<T>`, com `IAsyncEnumerable<T>` o consumo de um novo `await foreach`. Uma `IAsyncDisposable` interface também é usada para habilitar a limpeza assíncrona.

## Discussão relacionada

- [https://github.com/dotnet/roslyn/issues/261 ↗](https://github.com/dotnet/roslyn/issues/261)
- [https://github.com/dotnet/roslyn/issues/114 ↗](https://github.com/dotnet/roslyn/issues/114)

## Design detalhado

## Interfaces

### IAsyncDisposable

Houve muita discussão sobre `IAsyncDisposable` (por exemplo, [https://github.com/dotnet/roslyn/issues/114 ↗](https://github.com/dotnet/roslyn/issues/114)) e se é uma boa ideia. No entanto, é um conceito necessário para adicionar suporte a iteradores assíncronos. Como os `finally` blocos podem conter `await`s e, como `finally` os blocos precisam ser executados como parte do descarte de iteradores, precisamos de uma alienação assíncrona. Ele também é geralmente útil sempre que a limpeza de recursos pode levar algum tempo, por exemplo, fechar arquivos (exigindo liberações), cancelar o registro de retornos de chamada e fornecer uma maneira de saber quando o cancelamento do registro foi concluído, etc.

A interface a seguir é adicionada às bibliotecas principais do .NET (por exemplo, System. privado. CoreLib/System. Runtime):

```
C#  
  
namespace System  
{  
    public interface IAsyncDisposable  
    {  
        ValueTask DisposeAsync();  
    }  
}
```

Assim como acontece com `Dispose`, invocar `DisposeAsync` várias vezes é aceitável e as invocações subsequentes após o primeiro devem ser tratadas como NOPs, retornando uma tarefa bem-sucedida sincronizada de forma síncrona (`DisposeAsync` não é necessário ser thread-safe, porém e não precisa dar suporte à invocação simultânea). Além disso, os tipos podem implementar `IDisposable` e `IAsyncDisposable`, e se fizerem, é aceitável invocar `Dispose` e, em seguida, `DisposeAsync` ou vice-versa, mas somente o primeiro deve ser significativo e as invocações subsequentes de devem ser um NOP. Dessa forma, se um tipo implementar ambos, os consumidores serão incentivados a chamar uma única vez e apenas uma vez o método mais relevante com base no contexto, `Dispose` em contextos síncronos e `DisposeAsync` em assíncronos.

(Estou deixando a discussão de como `IAsyncDisposable` interage com `using` uma discussão separada. E a cobertura de como ela interage `foreach` é tratada posteriormente nesta proposta.)

Alternativas consideradas:

- aceitando: embora, em teoria, faz sentido que qualquer coisa assíncrona possa ser cancelada, a alienação é sobre a limpeza, o fechamento de coisas, recursos de free'ing, etc., que geralmente não é algo que deve ser cancelado; a limpeza ainda é importante para o trabalho que foi cancelado. `DisposeAsync CancellationToken` O mesmo `CancellationToken` que fazia com que o trabalho real a ser cancelado normalmente seria o mesmo token passado para `DisposeAsync`, fazendo `DisposeAsync` inúteis porque o cancelamento do trabalho faria com que `DisposeAsync` fosse um NOP. Se alguém quiser evitar ser bloqueado aguardando a alienação, ele poderá evitar esperar o resultado `ValueTask`, ou esperar por um período de tempo.
- retornando: agora que um não genérico existe e pode ser construído a partir de um, o retorno `DisposeAsync` de permite que um objeto existente seja reutilizado

como a promessa que representa a eventual conclusão assíncrona de, salvando uma alocação no caso em que é concluído de forma assíncrona.

`Task` `ValueTask`  
`IValueTaskSource` `ValueTask` `DisposeAsync` `DisposeAsync` `Task` `DisposeAsync`

- Configurando `DisposeAsync` com um `bool continueOnCapturedContext` (`ConfigureAwait`): embora possa haver problemas relacionados à forma como esse conceito é exposto a `using`, `foreach` e a outras construções de linguagem que consomem isso, de uma perspectiva de interface, na verdade, não está fazendo nenhum `await`ing`` e não há nada para configurar... os consumidores do `ValueTask` podem consumi-lo, no entanto, eles querem.
- herança: como apenas uma ou outra deve ser usada, não faz sentido forçar os tipos a implementar ambos. `IAsyncDisposable` `IDisposable`
- `IDisposableAsync` em vez `IAsyncDisposable` de: estamos seguindo a nomenclatura de que as coisas/tipos são um "assíncrono de algo", enquanto operações são "feitas assincronamente", de modo que os tipos têm "Async" como um prefixo e os métodos têm "Async" como um sufixo.

## IAsyncEnumerable/IAsyncEnumerator

Duas interfaces são adicionadas às principais bibliotecas do .NET:

C#

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

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

O consumo típico (sem recursos de linguagem adicional) seria semelhante a:

C#

```
IAsyncEnumerable<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
```

```

    {
        Use(enumerator.Current);
    }
}
finally { await enumerator.DisposeAsync(); }

```

Opções descartadas consideradas:

- `Task<bool> MoveNextAsync(); T current { get; }`: Usar `Task<bool>` o ofereceria suporte ao uso de um objeto de tarefa em cache para representar chamadas síncronas e bem-sucedidas `MoveNextAsync`, mas uma alocação ainda seria necessária para a conclusão assíncrona. Ao retornar `ValueTask<bool>`, habilitamos o objeto Enumerator para que ele próprio implemente `IValueTaskSource<bool>` e seja usado como o backup para o `ValueTask<bool>` retornado de `MoveNextAsync`, que, por sua vez, permite sobrecargas significativamente reduzidas.
- `ValueTask<(bool, T)> MoveNextAsync();`: Não é apenas mais difícil de consumir, mas isso significa que `T` não pode mais ser covariante.
- `ValueTask<T?> TryMoveNextAsync();`: Não covariant.
- `Task<T?> TryMoveNextAsync();`: Não covariant, alocações em todas as chamadas, etc.
- `ITask<T?> TryMoveNextAsync();`: Não covariant, alocações em todas as chamadas, etc.
- `ITask<(bool, T)> TryMoveNextAsync();`: Não covariant, alocações em todas as chamadas, etc.
- `Task<bool> TryMoveNextAsync(out T result);`: O `out` resultado precisaria ser definido quando a operação retorna de forma síncrona, não quando ele conclui a tarefa de maneira assíncrona, em algum momento, no futuro, nesse ponto não haveria nenhuma maneira de comunicar o resultado.
- `IAsyncEnumerator<T> não implementando IAsyncDisposable`: poderíamos optar por separá-las. No entanto, isso complica algumas outras áreas da proposta, pois o código deve ser capaz de lidar com a possibilidade de um enumerador não fornecer descarte, o que dificulta a gravação de auxiliares baseados em padrões. Além disso, será comum que os enumeradores tenham uma necessidade de descarte (por exemplo, qualquer iterador assíncrono do C# que tenha um bloco `finally`, a maioria das coisas que enumeram dados de uma conexão de rede etc.) e, se não houver, será simples implementar o método puramente como `public ValueTask DisposeAsync() => default(ValueTask);` com sobrecarga adicional mínima.
- `_ IAsyncEnumerator<T> GetAsyncEnumerator()` : Nenhum parâmetro de token de cancelamento.

## Alternativa viável:

C#

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

    public interface IAsyncEnumerator<out T> : IAsyncDisposable
    {
        ValueTask<bool> WaitForNextAsync();
        T TryGetNext(out bool success);
    }
}
```

`TryGetNext` é usado em um loop interno para consumir itens com uma única chamada de interface, desde que estejam disponíveis de forma síncrona. Quando o próximo item não pode ser recuperado de forma síncrona, ele retorna false e, sempre que retorna false, um chamador deve ser invocado posteriormente `WaitForNextAsync` para esperar que o próximo item esteja disponível ou para determinar que nunca haverá outro item. O consumo típico (sem recursos de linguagem adicional) seria semelhante a:

C#

```
IAsyncEnumerator<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.WaitForNextAsync())
    {
        while (true)
        {
            int item = enumerator.TryGetNext(out bool success);
            if (!success) break;
            Use(item);
        }
    }
}
finally { await enumerator.DisposeAsync(); }
```

A vantagem disso é duas dobras, uma secundária e uma maior:

- *Minor: permite que um enumerador dê suporte a vários consumidores.* Pode haver cenários em que é valioso para um enumerador dar suporte a vários consumidores simultâneos. Isso não pode ser obtido quando `MoveNextAsync` e `Current` são separados de modo que uma implementação não possa fazer seu uso atômico. Por

outro lado, essa abordagem fornece um método único `TryGetNext` que dá suporte ao envio por push do enumerador e à obtenção do próximo item, de modo que o enumerador possa habilitar a atomicidade, se desejado. No entanto, é provável que esses cenários também possam ser habilitados fornecendo a cada consumidor seu próprio enumerador de um `Enumerable` compartilhado. Além disso, não queremos impor que todos os enumeradores suportam uso simultâneo, pois isso adicionaria sobrecargas não triviais ao caso principal que não exige isso, o que significa que um consumidor da interface geralmente não podia depender dessa forma.

- *Principal: desempenho.* A `MoveNextAsync` / `Current` abordagem requer duas chamadas de interface por operação, enquanto o melhor caso `WaitForNextAsync` / `TryGetNext` é que a maioria das iterações é concluída de forma síncrona, permitindo um loop interno apertado com `TryGetNext`, de modo que tenhamos apenas uma chamada de interface por operação. Isso pode ter um impacto mensurável em situações em que a interface chama o dominando o cálculo.

No entanto, há desvantagens não triviais, incluindo uma complexidade significativamente maior ao consumi-las manualmente e uma maior chance de introduzir bugs ao usá-las. E, embora os benefícios de desempenho sejam mostrados em netbenchmarks, não acreditamos que eles serão impactados na grande maioria do uso real. Se isso acontece, podemos introduzir um segundo conjunto de interfaces de um modo claro.

Opções descartadas consideradas:

- `ValueTask<bool> WaitForNextAsync(); bool TryGetNext(out T result);`: os `out` parâmetros não podem ser covariantes. Há também um pequeno impacto aqui (um problema com o padrão try em geral) que isso provavelmente incorre em uma barreira de gravação em tempo de execução para resultados do tipo de referência.

## Cancelamento

Há várias abordagens possíveis para dar suporte ao cancelamento:

1. `IAsyncEnumerable<T>/IAsyncEnumerator<T>` Os cancelamentos são independentes: `CancellationToken` não aparecem em nenhum lugar. O cancelamento é obtido logicamente trazendo o `CancellationToken` em enumerável e/ou enumerador de qualquer maneira que seja apropriada, por exemplo, ao chamar um iterador, passar o `CancellationToken` como um argumento para o método iterador e usá-lo no corpo do iterador, como é feito com qualquer outro parâmetro.
2. `IAsyncEnumerator<T>.GetAsyncEnumerator(CancellationToken)`: Você passa um `CancellationToken` para `GetAsyncEnumerator`, e `MoveNextAsync` as operaçōes

subsequentes respeitam isso, no entanto, ele pode.

3. `IAsyncEnumerator<T>.MoveNextAsync(CancellationToken)`: Você passa um `CancellationToken` para cada `MoveNextAsync` chamada individual.
4. 1 && 2: você incorpora os `CancellationToken`s em seu enumerador/enumerável e passa os `CancellationToken`s para `GetAsyncEnumerator`.
5. 1 && 3: você incorpora os `CancellationToken`s em seu enumerador/enumerável e passa os `CancellationToken`s para `MoveNextAsync`.

De uma perspectiva puramente teórica, (5) é a mais robusta, `MoveNextAsync` pois a aceitação de um `CancellationToken` habilita o controle mais refinado sobre o que foi cancelado e (b) `CancellationToken` é apenas qualquer outro tipo que possa passar como um argumento em iteradores, inseridos em tipos arbitrários, etc.

No entanto, há vários problemas com essa abordagem:

- Como um `CancellationToken` passado para `GetAsyncEnumerator` transformá-lo no corpo do iterador? Poderíamos expor uma nova `iterator` palavra-chave que você poderia retirar para obter acesso ao `CancellationToken` passado para `GetEnumerator`, mas a) que é uma grande quantidade de máquinas adicionais, b) estamos fazendo dele um cidadão de primeira classe e c) o caso de 99% pareceria ser o mesmo código que chamamos um iterador e `GetAsyncEnumerator`, nesse caso, ele pode apenas passar o `CancellationToken` como um argumento para o método.
- Como um `CancellationToken` passado para `MoveNextAsync` entrar no corpo do método? Isso é ainda pior, como se fosse exposto a partir de um `iterator` objeto local, seu valor pode ser alterado em Awaits, o que significa que qualquer código registrado com o token precisaria cancelar o registro dele antes de aguardar e, em seguida, registrar novamente depois; também é potencialmente muito caro precisar fazer tal registro e cancelamento de registro em cada `MoveNextAsync` chamada, independentemente de ser implementado pelo compilador em um iterador ou por um desenvolvedor manualmente.
- Como um desenvolvedor cancela um `foreach` loop? Se isso for feito fornecendo um `CancellationToken` para um enumerador/enumeraer, em seguida, um) precisamos dar suporte aos `foreach` enumeradores 'ing over', o que os gera para cidadãos de primeira classe, e agora você precisa começar a pensar em um ecossistema criado em torno de enumeradores (por exemplo, métodos LINQ) ou b), precisamos inserir o `CancellationToken` no enumerável de qualquer forma, tendo algum `WithCancellation` método de extensão de `IAsyncEnumerable<T>` que armazenaria o token fornecido e, em seguida, passá-lo para o Enumerable recordado `GetAsyncEnumerator` quando o `GetAsyncEnumerator` no struct retornado

for invocado (ignorando esse token). Ou você pode usar apenas o `CancellationToken` que tem no corpo do foreach.

- Se/quando houver suporte para compreensão de consulta, como seria `CancellationToken` fornecido `GetEnumerator` ou `MoveNextAsync` passado para cada cláusula? A maneira mais fácil seria simplesmente para a cláusula capturá-la, e nesse ponto qualquer token passado para `GetAsyncEnumerator` / `MoveNextAsync` é ignorado.

Uma versão anterior deste documento é recomendada (1), mas, como mudamos para (4).

Os dois principais problemas com (1):

- os produtores de enumeráveis canceláveis precisam implementar algum texto clichê e só podem aproveitar o suporte do compilador para os iteradores assíncronos para implementar um `IAsyncEnumerator<T>` `GetAsyncEnumerator(CancellationToken)` método.
- é provável que muitos produtores sejam tentados a simplesmente adicionar um `CancellationToken` parâmetro à sua assinatura `Async-Enumerable`, o que impedirá que os consumidores passem o token de cancelamento que desejam quando recebem um `IEnumerable` tipo.

Há dois cenários de consumo principais:

1. `await foreach (var i in GetData(token)) ...` onde o consumidor chama o método `Async-Iterator`,
2. `await foreach (var i in givenIAsyncEnumerable.WithCancellation(token)) ...` onde o consumidor lida com uma determinada `IAsyncEnumerable` instância.

Descobrimos que um comprometimento razoável para dar suporte a ambos os cenários de uma maneira que seja conveniente para produtores e consumidores de transmissões assíncronas é usar um parâmetro especialmente anotado no método `Async-Iterator`. O `[EnumeratorCancellation]` atributo é usado para essa finalidade. Colocar esse atributo em um parâmetro informa ao compilador que, se um token for passado para o `GetAsyncEnumerator` método, esse token deverá ser usado em vez do valor passado originalmente para o parâmetro.

Considere o `IAsyncEnumerable<int> GetData([EnumeratorCancellation] CancellationToken token = default)`. O implementador desse método pode simplesmente usar o parâmetro no corpo do método. O consumidor pode usar os padrões de consumo acima:

1. Se você usar `GetData(token)` , o token será salvo no Async-Enumerable e será usado na iteração,
2. Se você usar `givenIAsyncEnumerable.WithCancellation(token)` , o token passado para `GetAsyncEnumerator` substituirá qualquer token salvo no Async-Enumerable.

## foreach

`foreach` será aumentado para dar suporte além de `IAsyncEnumerable<T>` seu suporte existente para o `IEnumerable<T>` . Ele dará suporte ao equivalente de `IAsyncEnumerable<T>` como um padrão se os membros relevantes forem expostos publicamente, voltando ao uso da interface diretamente, se não, para habilitar extensões baseadas em struct que evitem alocar, bem como usar awaitables alternativas como o tipo de retorno de `MoveNextAsync` e `DisposeAsync` .

## Syntax

Usando a sintaxe:

```
C#  
  
foreach (var i in enumerable)
```

O C# continuará a tratar `enumerable` como um Enumerable síncrono, de modo que, mesmo que ele exponha APIs relevantes para enumeráveis assíncronos (expondo o padrão ou implementando a interface), ele considerará apenas as APIs síncronas.

Para forçar `foreach` , considere apenas as APIs assíncronas, `await` é inserido da seguinte maneira:

```
C#  
  
await foreach (var i in enumerable)
```

Nenhuma sintaxe seria fornecida para dar suporte ao uso das APIs Async ou Sync; o desenvolvedor deve escolher com base na sintaxe usada.

Opções descartadas consideradas:

- `foreach (var i in await enumerable)`: Essa sintaxe já é válida e alterar seu significado seria uma alteração significativa. Isso significa para `await` o `enumerable` ,

obter algo de forma síncrona iterável a partir dele e, em seguida, iterar de forma síncrona.

- `foreach (var i await in enumerable)`, `foreach (var await i in enumerable)`, `foreach (await var i in enumerable)` Todos sugerem que estamos aguardando o próximo item, mas há outros Awaits envolvidos em foreach, em particular se o Enumerable for um `IAsyncDisposable`, será `await`'ing seu descarte assíncrono. O Await é como o escopo do foreach em vez de para cada elemento individual e, portanto, a `await` palavra-chave merece estar no `foreach` nível. Além disso, tê-lo associado ao `foreach` nos oferece uma maneira de descrever o `foreach` com um termo diferente, por exemplo, um "Await foreach". Mas o mais importante é que há um valor `foreach` para considerar a sintaxe ao mesmo tempo que a `using` sintaxe, para que eles permaneçam consistentes entre si e `using (await ...)` já seja uma sintaxe válida.
- `foreach await (var i in enumerable)`

Você ainda deve considerar:

- `foreach` Atualmente, o não oferece suporte à iteração por meio de um enumerador. Esperamos que seja mais comum ter os `IAsyncEnumerator<T>`s em mãos e, portanto, é tentador dar suporte `await foreach` com `IAsyncEnumerable<T>` e `IAsyncEnumerator<T>`. Mas depois de adicionar esse suporte, ele apresenta a pergunta se `IAsyncEnumerator<T>` é um cidadão de primeira classe e se precisamos ter sobrecargas de combinadores que operam em enumeradores, além de enumeráveis? Queremos incentivar métodos para retornar enumeradores em vez de enumeráveis? Devemos continuar a discutir isso. Se decidirmos que não queremos dar suporte a ela, talvez queiramos introduzir um método de extensão `public static IAsyncEnumerable<T> AsEnumerable<T>(this IAsyncEnumerator<T> enumerator);` que permitisse que um enumerador ainda fosse `foreach`. Se decidirmos que desejamos dar suporte a ele, também precisaremos decidir se o `await foreach` seria responsável por chamar `DisposeAsync` o enumerador e, provavelmente, a resposta é "não, o controle sobre a alienação deve ser tratado por quem chamou `GetEnumerator`".

## Compilação baseada em padrões

O compilador se associará às APIs baseadas em padrão, se existirem, preferirem aquelas usando a interface (o padrão pode ser satisfeito com métodos de instância ou métodos de extensão). Os requisitos para o padrão são:

- O Enumerable deve expor um `GetAsyncEnumerator` método que pode ser chamado sem argumentos e que retorne um enumerador que atenda ao padrão relevante.
- O enumerador deve expor um `MoveNextAsync` método que pode ser chamado sem argumentos e que retorna algo que pode ser `await` Ed e cujo `GetResult()` retorna um `bool`.
- O enumerador também deve expor `current` a propriedade cujo getter retorna um `T` representando o tipo de dados que está sendo enumerado.
- O enumerador pode, opcionalmente, expor um `DisposeAsync` método que pode ser invocado sem argumentos e que retorna algo que pode ser `await` Ed e cujo `GetResult()` retorna `void`.

Esse código:

```
C#
var enumerable = ...;
await foreach (T item in enumerable)
{
    ...
}
```

é convertido para o equivalente de:

```
C#
var enumerable = ...;
var enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        T item = enumerator.Current;
        ...
    }
}
finally
{
    await enumerator.DisposeAsync(); // omitted, along with the try/finally,
if the enumerator doesn't expose DisposeAsync
}
```

Se o tipo iterado não expor o padrão correto, as interfaces serão usadas.

## ConfigureAwait

Essa compilação baseada em padrão permitirá que `ConfigureAwait` seja usada em todos os Awaits, por meio de um `ConfigureAwait` método de extensão:

C#

```
await foreach (T item in enumerable.ConfigureAwait(false))
{
    ...
}
```

Isso se baseará em tipos que também adicionaremos ao .NET, provavelmente `System.Threading.Tasks.Extensions.dll`:

C#

```
// Approximate implementation, omitting arg validation and the like
namespace System.Threading.Tasks
{
    public static class AsyncEnumerableExtensions
    {
        public static ConfiguredAsyncEnumerable<T> ConfigureAwait<T>(this
IAsyncEnumerable<T> enumerable, bool continueOnCapturedContext) =>
            new ConfiguredAsyncEnumerable<T>(enumerable,
continueOnCapturedContext);

        public struct ConfiguredAsyncEnumerable<T>
        {
            private readonly IAsyncEnumerable<T> _enumerable;
            private readonly bool _continueOnCapturedContext;

            internal ConfiguredAsyncEnumerable(IAsyncEnumerable<T>
enumerable, bool continueOnCapturedContext)
            {
                _enumerable = enumerable;
                _continueOnCapturedContext = continueOnCapturedContext;
            }

            public ConfiguredAsyncEnumerator<T> GetAsyncEnumerator() =>
                new ConfiguredAsyncEnumerator<T>
(_enumerable.GetAsyncEnumerator(), _continueOnCapturedContext);

            public struct Enumerator
            {
                private readonly IAsyncEnumerator<T> _enumerator;
                private readonly bool _continueOnCapturedContext;

                internal Enumerator(IAsyncEnumerator<T> enumerator, bool
continueOnCapturedContext)
                {
                    _enumerator = enumerator;
                    _continueOnCapturedContext = continueOnCapturedContext;
                }
            }
        }
    }
}
```

```

    public ConfiguredValueTaskAwaitable<bool> MoveNextAsync() =>

    _enumerator.MoveNextAsync().ConfigureAwait(_continueOnCapturedContext);

        public T Current => _enumerator.Current;

        public ConfiguredValueTaskAwaitable DisposeAsync() =>

    _enumerator.DisposeAsync().ConfigureAwait(_continueOnCapturedContext);
    }

}

}

```

Observe que essa abordagem não permitirá que `ConfigureAwait` seja usada com enumeráveis baseados em padrões, mas, novamente, ele já é o caso de `ConfigureAwait` ser exposto apenas como uma extensão em `Task` / `Task<T>` / `ValueTask` / `ValueTask<T>` e não pode ser aplicado a coisas notentes arbitrárias, pois faz sentido quando aplicado às tarefas (ele controla um comportamento implementado no suporte à continuação da tarefa) e, portanto, não faz sentido ao usar um padrão em que as coisas que podem ser esperadas não sejam tarefas. Qualquer pessoa que retornar coisas awaitable pode fornecer seu próprio comportamento personalizado nesses cenários avançados.

(Se pudermos surgir alguma maneira de dar suporte a uma solução em nível de escopo ou de assembly `ConfigureAwait`, isso não será necessário.)

## Iteradores assíncronos

O idioma/compilador dará suporte à produção `IAsyncEnumerable<T>` de `IAsyncEnumerator<T>`s, além de consumi-los. Atualmente, a linguagem dá suporte à gravação de um iterador como:

C#

```

static I Enumerable<int> MyIterator()
{
    try
    {
        for (int i = 0; i < 100; i++)
        {
            Thread.Sleep(1000);
            yield return i;
        }
    }
    finally
    {
    }
}

```

```

        Thread.Sleep(200);
        Console.WriteLine("finally");
    }
}

```

Mas `await` não podem ser usados no corpo desses iteradores. Adicionaremos esse suporte.

## Syntax

O suporte de idioma existente para iteradores infere a natureza do iterador do método com base no fato de ele conter qualquer `yield`s. O mesmo será verdadeiro para iteradores assíncronos. Esses iteradores assíncronos serão demarcadas e diferenciados dos iteradores síncronos por meio `async` da adição à assinatura e, em seguida, também devem ter um `IAsyncEnumerable<T>` ou `IAsyncEnumerator<T>` como seu tipo de retorno. Por exemplo, o exemplo acima poderia ser escrito como um iterador assíncrono da seguinte maneira:

```
C#
static async IAsyncEnumerable<int> MyIterator()
{
    try
    {
        for (int i = 0; i < 100; i++)
        {
            await Task.Delay(1000);
            yield return i;
        }
    }
    finally
    {
        await Task.Delay(200);
        Console.WriteLine("finally");
    }
}
```

Alternativas consideradas:

- *Não usar `async` na assinatura:* o uso do `async` provavelmente é tecnicamente exigido pelo compilador, pois ele o utiliza para determinar se `await` é válido nesse contexto. Mas mesmo que não seja necessário, estabelecemos que `await` pode ser usado apenas em métodos marcados como `async`, e parece importante manter a consistência.

- *Habilitando construtores personalizados para `IAsyncEnumerable<T>`*: isso é algo que poderíamos examinar para o futuro, mas a maquinade é complicada e não damos suporte a isso para as contrapartes síncronas.
- *Ter uma `iterator` palavra-chave na assinatura*: os iteradores assíncronos usariam `async iterator` na assinatura e `yield` só poderiam ser usados em `async` métodos que incluíssem `iterator`; `iterator` seria opcional em iteradores síncronos.

Dependendo da sua perspectiva, isso tem a vantagem de torná-lo muito claro pela assinatura do método, independentemente de ser `yield` permitido e se o método é, na verdade, destinado a retornar instâncias do tipo `IAsyncEnumerable<T>` em vez do compilador fabricado um com base em se o código usa `yield` ou não. Mas é diferente dos iteradores síncronos, que não são e não podem ser feitos para exigir um. Além disso, alguns desenvolvedores não gostam da sintaxe extra. Se estivessemos projetando a partir do zero, provavelmente teríamos que fazer isso, mas neste ponto há muito mais valor em manter iteradores assíncronos próximos aos iteradores de sincronização.

## LINQ

Há mais de ~ 200 sobrecargas de métodos na `System.Linq.Enumerable` classe, todos eles funcionam em termos de `IEnumerable<T>`; algumas dessas aceitas `IEnumerable<T>`, algumas delas produzem `IEnumerable<T>`, e muitas são ambas. Adicionar suporte a LINQ for `IAsyncEnumerable<T>` provavelmente envolveria a duplicação de todas essas sobrecargas para ela, para outro ~ 200. E, como `IAsyncEnumerator<T>` é provável que seja mais comum como uma entidade autônoma no mundo assíncrono do que `IEnumerator<T>` está no mundo síncrono, podemos potencialmente precisar de outras sobrecargas de ~ 200 que funcionam com o `IAsyncEnumerator<T>`. Além disso, um grande número de sobrecargas lida com predicados (por exemplo `Where`, que leva um `Func<T, bool>`), e pode ser desejável ter `IAsyncEnumerable<T>` sobrecargas com base no que lidam com predicados síncronos e assíncronos (por exemplo, além `Func<T,` `ValueTask<bool>>` de `Func<T, bool>`). Embora isso não seja aplicável a todas as novas sobrecargas de agora ~ 400, um cálculo aproximado é que seria aplicável à metade, o que significa que outras ~ 200 sobrecargas, para um total de ~ 600 novos métodos.

Esse é um número cada vez maior de APIs, com o potencial para ainda mais quando bibliotecas de extensão como extensões interativas (IX) são consideradas. Mas o IX já tem uma implementação de muitos desses, e parece que não há um grande motivo para duplicar esse trabalho; em vez disso, devemos ajudar a Comunidade a melhorar o IX e recomendar isso para quando os desenvolvedores desejarem usar o LINQ com o `IAsyncEnumerable<T>`.

Também há o problema de sintaxe de compreensão da consulta. A natureza baseada em padrões das compreensões das consultas permitiria "simplesmente funcionar" com alguns operadores, por exemplo, se o IX fornecer os seguintes métodos:

C#

```
public static IAsyncEnumerable<TResult> Select<TSource, TResult>(this  
IAsyncEnumerable<TSource> source, Func<TSource, TResult> func);  
public static IAsyncEnumerable<T> Where(this IAsyncEnumerable<T> source,  
Func<T, bool> func);
```

Então, esse código C# vai "apenas funcionar":

C#

```
IAsyncEnumerable<int> enumerable = ...;  
IAsyncEnumerable<int> result = from item in enumerable  
                                where item % 2 == 0  
                                select item * 2;
```

No entanto, não há nenhuma sintaxe de compreensão de consulta que ofereça suporte ao uso `await` nas cláusulas, portanto, se o IX for adicionado, por exemplo:

C#

```
public static IAsyncEnumerable<TResult> Select<TSource, TResult>(this  
IAsyncEnumerable<TSource> source, Func<TSource, ValueTask<TResult>> func);
```

Então, isso "simplesmente funciona":

C#

```
IAsyncEnumerable<string> result = from url in urls  
                                    where item % 2 == 0  
                                    select SomeAsyncMethod(item);  
  
async ValueTask<int> SomeAsyncMethod(int item)  
{  
    await Task.Yield();  
    return item * 2;  
}
```

Mas não haveria nenhuma maneira de escrevê-lo com o `await` embutido na `select` cláusula. Como um esforço separado, poderíamos examinar a adição de `async { ... }` expressões à linguagem, ponto em que poderíamos permitir que elas sejam usadas em compreensão de consulta e, em vez disso, o seguinte poderia ser escrito como:

C#

```
IAsyncEnumerable<int> result = from item in enumerable
                                where item % 2 == 0
                                select async
                                {
                                    await Task.Yield();
                                    return item * 2;
                                };

```

ou para habilitar o `await` para ser usado diretamente em expressões, como ao dar suporte a `async from`. No entanto, é improvável que um design aqui afete o restante do conjunto de recursos de uma maneira ou o outro, e essa não é uma coisa particularmente de alto valor para investir no momento, portanto, a proposta é não fazer nada mais aqui no momento.

## Integração com outras estruturas assíncronas

A integração com o `IObservable<T>` e outras estruturas assíncronas (por exemplo, fluxos reativos) seria feita no nível da biblioteca, e não no nível de linguagem. Por exemplo, todos os dados de um `IAsyncEnumerator<T>` podem ser publicados em um `IObserver<T>` simples por `await foreach` 'ing sobre o enumerador e `OnNext` 'ing os dados para o observador, portanto, um método de `AsObservable<T>` extensão é possível. O consumo de um `IObservable<T>` em `await foreach` requer o armazenamento em buffer dos dados (caso outro item seja enviado por push enquanto o item anterior ainda estiver sendo processado), mas esse adaptador pode ser facilmente implementado para permitir que um seja `IObservable<T>` extraído de um `IAsyncEnumerator<T>`. Diante. O RX/IX já fornece protótipos dessas implementações e bibliotecas como <https://github.com/dotnet/corefx/tree/master/src/System.Threading.Channels> ↗ fornecer vários tipos de estruturas de dados em buffer. O idioma não precisa estar envolvido neste estágio.

# Intervalos

Artigo • 16/09/2021

## Resumo

Esse recurso está prestes a fornecer dois novos operadores que permitem `System.Index` a construção e `System.Range` os objetos, e usá-los para indexar/dividir as coleções em tempo de execução.

## Visão geral

### Membros e tipos bem conhecidos

Para usar as novas formas sintáticas para o `System.Index` e o `System.Range`, novos tipos e membros bem conhecidos podem ser necessários, dependendo de quais formas sintáticas são usadas.

Para usar o operador "Hat" (`^`), é necessário o seguinte

```
C#  
  
namespace System  
{  
    public readonly struct Index  
    {  
        public Index(int value, bool fromEnd);  
    }  
}
```

Para usar o `System.Index` tipo como um argumento em um acesso de elemento de matriz, o membro a seguir é necessário:

```
C#  
  
int System.Index.GetOffset(int length);
```

A `..` sintaxe do `System.Range` exigirá o `System.Range` tipo, bem como um ou mais dos seguintes membros:

```
C#
```

```

namespace System
{
    public readonly struct Range
    {
        public Range(System.Index start, System.Index end);
        public static Range StartAt(System.Index start);
        public static Range EndAt(System.Index end);
        public static Range All { get; }
    }
}

```

A `..` sintaxe permite que ambos ou nenhum dos seus argumentos estejam ausentes. Independentemente do número de argumentos, o `Range` Construtor é sempre suficiente para usar a `Range` sintaxe. No entanto, se qualquer um dos outros membros estiver presente e um ou mais `..` argumentos estiverem ausentes, o membro apropriado poderá ser substituído.

Por fim, para um valor do tipo `System.Range` a ser usado em uma expressão de acesso de elemento de matriz, o seguinte membro deve estar presente:

```

C#

namespace System.Runtime.CompilerServices
{
    public static class RuntimeHelpers
    {
        public static T[] GetSubArray<T>(T[] array, System.Range range);
    }
}

```

## System. index

O C# não tem como indexar uma coleção a partir do final, mas sim a maioria dos indexadores usam a noção "de início" ou uma expressão de "comprimento i". Apresentamos uma nova expressão de índice que significa "do final". O recurso apresentará um novo operador unário "Hat". Seu único operando deve ser conversível para `System.Int32`. Ele será reduzido na chamada de método de `System.Index` fábrica apropriada.

Aumentamos a gramática para *unary\_expression* com o seguinte formulário de sintaxe adicional:

```
unary_expression
: '^' unary_expression
;
```

Chamamos isso de *índice do operador end*. O índice predefinido dos operadores end é o seguinte:

C#

```
System.Index operator ^(int fromEnd);
```

O comportamento desse operador é definido apenas para valores de entrada maiores ou iguais a zero.

Exemplos:

C#

```
var array = new int[] { 1, 2, 3, 4, 5 };
var thirdItem = array[2];      // array[2]
var lastItem = array[^1];      // array[new Index(1, fromEnd: true)]
```

## System. Range

O C# não tem uma forma sintática de acessar "intervalos" ou "fatias" de coleções. Normalmente, os usuários são forçados a implementar estruturas complexas para filtrar/operar em fatias de memória ou recorrer a métodos LINQ como `list.Skip(5).Take(2)`. Com a adição do `System.Span<T>` e de outros tipos semelhantes, torna-se mais importante ter esse tipo de operação com suporte em um nível mais profundo no idioma/tempo de execução e ter a interface unificada.

O idioma introduzirá um novo operador Range `x..y`. É um operador binário infixo que aceita duas expressões. Ambos os operandos podem ser omitidos (exemplos abaixo) e devem ser conversíveis `System.Index`. Ele será reduzido para a `System.Range` chamada de método de fábrica apropriada.

Substituimos as regras de gramática do C# por *multiplicative\_expression* pelo seguinte (para introduzir um novo nível de precedência):

antlr

```
range_expression
: unary_expression
```

```

| range_expression? '...' range_expression?
;

multiplicative_expression
: range_expression
| multiplicative_expression '*' range_expression
| multiplicative_expression '/' range_expression
| multiplicative_expression '%' range_expression
;

```

Todas as formas do *operador Range* têm a mesma precedência. Esse novo grupo de precedência é menor do que os *operadores unários* e superiores aos *operadores aritméticos multiplicativa*.

Chamamos o `..` operador do *operador Range*. O operador de intervalo interno pode aproximadamente ser compreendido para corresponder à invocação de um operador interno deste formulário:

C#

```
System.Range operator ..(Index start = 0, Index end = ^0);
```

Exemplos:

C#

```

var array = new int[] { 1, 2, 3, 4, 5 };
var slice1 = array[2..^3];      // array[new Range(2, new Index(3, fromEnd:
true))]
var slice2 = array[..^3];       // array[Range.EndAt(new Index(3, fromEnd:
true))]
var slice3 = array[2..];        // array[Range.StartAt(2)]
var slice4 = array[..];         // array[Range.All]

```

Além disso, `System.Index` deve ter uma conversão implícita do `System.Int32`, a fim de evitar a necessidade de sobrecarga de combinação de inteiros e índices em assinaturas multidimensionais.

## Adicionando suporte a índice e intervalo a tipos de biblioteca existentes

### Suporte a índice implícito

O idioma fornecerá um membro do indexador de instância com um único parâmetro do tipo `Index` para tipos que atendam aos seguintes critérios:

- O tipo é contável.
- O tipo tem um indexador de instância acessível que usa um único `int` como o argumento.
- O tipo não tem um indexador de instância acessível que usa um `Index` como o primeiro parâmetro. O `Index` deve ser o único parâmetro ou os parâmetros restantes devem ser opcionais.

Um tipo pode ser *contabilizado* se tiver uma propriedade chamada `Length` ou `Count` com um getter acessível e um tipo de retorno de `int`. O idioma pode fazer uso dessa propriedade para converter uma expressão do tipo `Index` em um `int` no ponto da expressão sem a necessidade de usar o tipo `Index`. No caso de `Length` e `Count` estão presentes, `Length` será preferível. Para simplificar o futuro, a proposta usará o nome `Length` para representar `Count` ou `Length`.

Para esses tipos, a linguagem agirá como se houver um membro do indexador do formulário `T this[Index index]` em que `T` é o tipo de retorno do `int` indexador baseado, incluindo quaisquer `ref` anotações de estilo. O novo membro terá o mesmo `get` e `set` os membros com a acessibilidade correspondente como o `int` indexador.

O novo indexador será implementado convertendo o argumento do tipo `Index` em um `int` e emitindo uma chamada para o `int` indexador baseado. Para fins de discussão, vamos usar o exemplo de `receiver[expr]`. A conversão de `expr` para `int` ocorrerá da seguinte maneira:

- Quando o argumento estiver no formato `^expr2` e o tipo de `expr2` for `int`, ele será convertido em `receiver.Length - expr2`.
- Caso contrário, ele será traduzido como `expr.GetOffset(receiver.Length)`.

Isso permite que os desenvolvedores usem o `Index` recurso em tipos existentes sem a necessidade de modificação. Por exemplo:

C#

```
List<char> list = ...;
var value = list[^1];

// Gets translated to
var value = list[list.Count - 1];
```

As `receiver` `Length` expressões e serão despejadas conforme apropriado para garantir que os efeitos colaterais sejam executados apenas uma vez. Por exemplo:

```
C#  
  
class Collection {  
    private int[] _array = new[] { 1, 2, 3 };  
  
    public int Length {  
        get {  
            Console.Write("Length ");  
            return _array.Length;  
        }  
    }  
  
    public int this[int index] => _array[index];  
}  
  
class SideEffect {  
    Collection Get() {  
        Console.Write("Get ");  
        return new Collection();  
    }  
  
    void Use() {  
        int i = Get()[^1];  
        Console.WriteLine(i);  
    }  
}
```

Esse código imprimirá "obter tamanho 3".

## Supporte a intervalo implícito

O idioma fornecerá um membro do indexador de instância com um único parâmetro do tipo `Range` para tipos que atendam aos seguintes critérios:

- O tipo é contável.
- O tipo tem um membro acessível chamado `slice` que tem dois parâmetros do tipo `int`.
- O tipo não tem um indexador de instância que usa um único `Range` como o primeiro parâmetro. O `Range` deve ser o único parâmetro ou os parâmetros restantes devem ser opcionais.

Para esses tipos, a linguagem será vinculada como se houver um membro do indexador no formato `T this[Range range]` em que `T` é o tipo de retorno do `Slice` método,

incluindo quaisquer anotações de `ref` estilo. O novo membro também terá a acessibilidade correspondente com o `Slice`.

Quando o `Range` indexador baseado estiver associado a uma expressão denominada `receiver`, ele será reduzido convertendo a `Range` expressão em dois valores que são passados para o `Slice` método. Para fins de discussão, vamos usar o exemplo de `receiver[expr]`.

O primeiro argumento de `Slice` será obtido convertendo a expressão com tipo de intervalo da seguinte maneira:

- Quando `expr` está no formato `expr1..expr2` (onde `expr2` pode ser omitido) e `expr1` tem tipo `int`, ele será emitido como `expr1`.
- Quando `expr` está no formato `^expr1..expr2` (onde `expr2` pode ser omitido), ele será emitido como `receiver.Length - expr1`.
- Quando `expr` está no formato `..expr2` (onde `expr2` pode ser omitido), ele será emitido como `0`.
- Caso contrário, ele será emitido como `expr.Start.GetOffset(receiver.Length)`.

Esse valor será reutilizado no cálculo do segundo `slice` argumento. Ao fazer isso, ele será referido como `start`. O segundo argumento de `Slice` será obtido convertendo a expressão com tipo de intervalo da seguinte maneira:

- Quando `expr` está no formato `expr1..expr2` (onde `expr1` pode ser omitido) e `expr2` tem tipo `int`, ele será emitido como `expr2 - start`.
- Quando `expr` está no formato `expr1..^expr2` (onde `expr1` pode ser omitido), ele será emitido como `(receiver.Length - expr2) - start`.
- Quando `expr` está no formato `expr1..` (onde `expr1` pode ser omitido), ele será emitido como `receiver.Length - start`.
- Caso contrário, ele será emitido como `expr.End.GetOffset(receiver.Length) - start`.

As `receiver Length` expressões,, e `expr` serão despejadas conforme apropriado para garantir que os efeitos colaterais sejam executados apenas uma vez. Por exemplo:

C#

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.Write("Length ");
        }
    }
}
```

```

        return _array.Length;
    }
}

public int[] Slice(int start, int length) {
    var slice = new int[length];
    Array.Copy(_array, start, slice, 0, length);
    return slice;
}
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        var array = Get()[0..2];
        Console.WriteLine(array.Length);
    }
}

```

Esse código imprimirá "obter comprimento 2".

O idioma fará o caso especial dos seguintes tipos conhecidos:

- `string`: o método `Substring` será usado em vez de `Slice`.
- `array`: o método `System.Runtime.CompilerServices.RuntimeHelpers.GetSubArray` será usado em vez de `Slice`.

## Alternativas

Os novos operadores (`^` e `..`) são uma simplificação sintática. A funcionalidade pode ser implementada por chamadas explícitas para `System.Index` `System.Range` métodos de fábrica, mas isso resultará em muito mais código clichê, e a experiência será não intuitiva.

## Representação de IL

Esses dois operadores serão reduzidos para chamadas regulares de indexador/método, sem alteração nas camadas subsequentes do compilador.

## Comportamento do tempo de execução

- O compilador pode otimizar indexadores para tipos internos, como matrizes e cadeias de caracteres, e reduzir a indexação para os métodos existentes apropriados.
- `System.Index` será gerado se construído com um valor negativo.
- `^0` Não lança, mas se traduz no comprimento da coleção/enumerável para o qual ela é fornecida.
- `Range.All` é semanticamente equivalente a `0..^0` e pode ser desconstruído para esses índices.

## Considerações

### Detectar indexável com base em `ICollection`

A inspiração para esse comportamento foram os inicializadores de coleção. Usando a estrutura de um tipo para transmitir que ele optou por um recurso. No caso de tipos de inicializadores de coleção podem optar pelo recurso implementando a interface `IEnumerable` (não genérica).

Inicialmente, essa proposta exigia que os tipos `ICollection` implementasse para se qualificarem como Indexáveis. No entanto, isso exigia vários casos especiais:

- `ref struct`: eles não podem implementar interfaces, mas tipos como `Span<T>` são ideais para suporte a índice/intervalo.
- `string`: não implementa e `ICollection` adiciona que tem um custo `interface` grande.

Isso significa que para dar suporte a tipos de chave, o uso de uso especial de uso de caixa já é necessário. O uso de uso especial de é menos interessante, pois a linguagem faz isso em outras `string foreach` áreas (redução, constantes etc.). O uso de uso especial de é mais importante, pois é um uso especial de `ref struct` uso de dados em uma classe inteira de tipos. Eles serão rotulados como Indexáveis se simplesmente têm uma propriedade chamada `Count` com um tipo de retorno de `int`.

Após a consideração, o design foi normalizado para dizer que qualquer tipo que tenha uma propriedade com um tipo de retorno `Count` / `Length` de `int` é Indexável. Isso remove todas as coberturas especiais, mesmo para `string` matrizes e .

### Detectar apenas Contagem

Detectar os nomes de propriedade `Count` ou complica um pouco o `Length` design.

Escolher apenas um para padronizar, porém, não é suficiente, pois acaba excluindo um grande número de tipos:

- Use `Length` : exclui praticamente todas as coleções em `System.Collections` e sub-namespaces. Elas tendem a derivar de `ICollection` e, portanto, preferem `Count` em vez de comprimento.
- Usar `Count` : exclui , `string` matrizes e a maioria dos tipos `Span<T>` `ref struct` baseados

A complicação extra na detecção inicial de tipos indexáveis é superada por sua simplificação em outros aspectos.

## Escolha de Fatia como um nome

O nome foi escolhido como o nome padrão de facto para `Slice` operações de estilo de fatia no .NET. A partir do netcoreapp2.1, todos os tipos de estilo de intervalo usam o nome `Slice` para operações de fatiamento. Antes do netcoreapp2.1, na verdade, não há nenhum exemplo de fatia para procurar por um exemplo. Tipos como , seriam ideais para a fatia, mas o conceito não `List<T>` existia quando os tipos foram `ArraySegment<T>` `SortedList<T>` adicionados.

Portanto, `Slice` sendo o único exemplo, ele foi escolhido como o nome.

## Conversão de tipo de destino de índice

Outra maneira de exibir a transformação `Index` em uma expressão de indexador é como uma conversão de tipo de destino. Em vez de vincular como se houvesse um membro do formulário , o idioma em vez disso atribui uma `return_type this[Index]` conversão com tipo de destino para `int` .

Esse conceito pode ser generalizado para todo o acesso de membro em tipos `countable`. Sempre que uma expressão com tipo for usada como um argumento para uma invocação de membro de instância e o receptor for `Countable`, a expressão terá uma conversão de tipo de `Index` destino para `int` . As invocações de membro aplicáveis a essa conversão incluem métodos, indexadores, propriedades, métodos de extensão etc... Somente construtores são excluídos, pois não têm nenhum receptor.

A conversão de tipo de destino será implementada da seguinte forma para qualquer expressão que tenha um tipo de `Index` . Para fins de discussão, vamos usar o exemplo de `receiver[expr]` :

- Quando `expr` for do formulário e o tipo de `for`, ele será convertido em `^expr2`  
`expr2 int receiver.Length - expr2`.
- Caso contrário, ele será convertido como `expr.GetOffset(receiver.Length)`.

As `receiver Length` expressões e serão apropriadas para garantir que os efeitos colaterais sejam executados apenas uma vez. Por exemplo:

C#

```

class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }

    public int GetAt(int index) => _array[index];
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        int i = Get().GetAt(^1);
        Console.WriteLine(i);
    }
}

```

Esse código imprimirá "Obter Comprimento 3".

Esse recurso seria benéfico para qualquer membro que tivesse um parâmetro que representasse um índice. Por exemplo, `List<T>.InsertAt`. Isso também tem o potencial de confusão, pois o idioma não pode dar nenhuma orientação sobre se uma expressão se trata ou não de indexação. Tudo o que ele pode fazer é converter qualquer expressão em ao `Index` invocar um membro em um tipo `int Countable`.

Restrições:

- Essa conversão só é aplicável quando a expressão com tipo `Index` é diretamente um argumento para o membro. Ele não se aplicaria a nenhuma expressão aninhada.

# Decisões tomadas durante a implementação

- Todos os membros no padrão devem ser membros da instância
- Se um método Length for encontrado, mas tiver o tipo de retorno errado, continue procurando Contagem
- O indexador usado para o padrão Index deve ter exatamente um parâmetro int
- O método Slice usado para o padrão Range deve ter exatamente dois parâmetros int
- Ao procurar os membros padrão, buscamos definições originais, não membros construídos

## Reuniões de design

- [10 de janeiro de 2018 ↗](#)
- [18 de janeiro de 2018 ↗](#)
- [22 de janeiro de 2018 ↗](#)
- [3 de dezembro de 2018 ↗](#)
- [25 de março de 2019 ↗](#)
- [1º de abril de 2019 ↗](#)
- [15 de abril de 2019 ↗](#)

# "uso baseado em padrão" e "usando declarações"

Artigo • 16/09/2021

## Resumo

O idioma adicionará dois novos recursos em torno da instrução para simplificar o gerenciamento de recursos: deve reconhecer um padrão descartável além de e adicionar uma `using` declaração ao `IDisposable` `using` idioma.

## Motivação

A `using` instrução é uma ferramenta efetiva para o gerenciamento de recursos atualmente, mas requer bastante soltura. Os métodos que têm vários recursos a gerenciar podem ser sintetizados sintaticamente com uma série de `using` instruções. Essa carga de sintaxe é suficiente para que a maioria das diretrizes de estilo de codificação tenha explicitamente uma exceção em torno de chaves para esse cenário.

A declaração remove grande parte da independência aqui e obtém C# em `using` par com outras linguagens que incluem blocos de gerenciamento de recursos. Além disso, o baseado em `using` padrão permite que os desenvolvedores expandam o conjunto de tipos que podem participar aqui. Em muitos casos, a remoção da necessidade de criar tipos de wrapper que existem apenas para permitir o uso de valores em uma `using` instrução .

Juntos, esses recursos permitem que os desenvolvedores simplifiquem e expandam os cenários em `using` que podem ser aplicados.

## Design detalhado

### usando declaração

O idioma permitirá que `using` seja adicionado a uma declaração de variável local. Essa declaração terá o mesmo efeito que declarar a variável em uma `using` instrução no mesmo local.

```
if (...)  
{  
    using FileStream f = new FileStream(@"C:\users\jaredpar\using.md");  
    // statements  
}  
  
// Equivalent to  
if (...)  
{  
    using (FileStream f = new FileStream(@"C:\users\jaredpar\using.md"))  
    {  
        // statements  
    }  
}
```

O tempo de vida `using` de um local se estenderá até o final do escopo no qual ele é declarado. Em `using` seguida, os locais serão descartados na ordem inversa em que são declarados.

C#

```
{  
    using var f1 = new FileStream("...");  
    using var f2 = new FileStream("..."), f3 = new FileStream("...");  
    ...  
    // Dispose f3  
    // Dispose f2  
    // Dispose f1  
}
```

Não há restrições em torno de ou qualquer outro constructo de fluxo `goto` de controle no rosto de uma `using` declaração. Em vez disso, o código atua exatamente como faria para a `using` instrução equivalente:

C#

```
{  
    using var f1 = new FileStream("...");  
    target:  
        using var f2 = new FileStream("...");  
        if (someCondition)  
        {  
            // Causes f2 to be disposed but has no effect on f1  
            goto target;  
        }  
}
```

Um local declarado em uma declaração `using` local será implicitamente somente leitura. Isso corresponde ao comportamento de locais declarados em uma `using` instrução .

A gramática da linguagem `using` para declarações será a seguinte:

```
antlr

local-using-declaration:
    using type using-declarators

using-declarators:
    using-declarator
    using-declarators , using-declarator

using-declarator:
    identifier = expression
```

Restrições em torno `using` da declaração:

- Pode não aparecer diretamente dentro de um `case` rótulo, mas em vez disso deve estar dentro de um bloco dentro do `case` rótulo.
- Pode não aparecer como parte de uma `out` declaração de variável.
- Deve ter um inicializador para cada Declarador.
- O tipo local deve ser implicitamente conversível para `IDisposable` ou atender ao `using` padrão.

## baseado em padrões usando

O idioma adicionará a noção de um padrão descartável: que é um tipo que tem um `Dispose` método de instância acessível. Tipos que se ajustam ao padrão descartável podem participar de uma declaração `using` ou declaração sem necessidade de implementação `IDisposable` .

```
C#

class Resource
{
    public void Dispose() { ... }

using (var r = new Resource())
{
    // statements
}
```

Isso permitirá que os desenvolvedores utilizem `using` vários cenários novos:

- `ref struct`: Esses tipos não podem implementar interfaces hoje e, portanto, não podem participar de `using` instruções.
- Os métodos de extensão permitirão que os desenvolvedores ampliem os tipos em outros assemblies para participar de `using` instruções.

Na situação em que um tipo pode ser convertido implicitamente `IDisposable` e também se adapta ao padrão descartável, `IDisposable` será preferível. Embora isso aceite a abordagem oposta de `foreach` (padrão preferencial na interface), é necessário para compatibilidade com versões anteriores.

As mesmas restrições de uma `using` instrução tradicional se aplicam aqui também: variáveis locais declaradas no `using` são somente leitura, um `null` valor não fará com que uma exceção seja gerada, etc... A geração de código será diferente apenas no que não haverá uma conversão `IDisposable` antes de chamar `Dispose`:

```
C#  
  
{  
    Resource r = new Resource();  
    try {  
        // statements  
    }  
    finally {  
        if (r != null) r.Dispose();  
    }  
}
```

Para ajustar o padrão descartável, o `Dispose` método deve ser acessível, sem parâmetros e ter um `void` tipo de retorno. Não há nenhuma outra restrição. Isso significa explicitamente que os métodos de extensão podem ser usados aqui.

## Considerações

### rótulos de caso sem blocos

Um `using declaration` é ilegal diretamente dentro de um `case` rótulo devido a complicações em relação ao seu tempo de vida real. Uma solução potencial é simplesmente dar o mesmo tempo de vida como um `out var` no mesmo local. Foi considerada a complexidade extra para a implementação do recurso e a facilidade de

resolver o problema (basta adicionar um bloco ao rótulo) não justificava a realização `case` dessa rota.

## Expansões futuras

### locais fixos

Uma `fixed` instrução tem todas as propriedades de `using` instruções que motivaram a capacidade de ter `using` locais. Deve-se considerar a extensão desse recurso `fixed` para locais também. O tempo de vida e as regras de ordenação devem se aplicar igualmente bem `using` para e `fixed` aqui.

# Funções locais estáticas

Artigo • 16/09/2021

## Resumo

Suporte a funções locais que não permitem capturar o estado do escopo delimitador.

## Motivação

Evite capturar de forma não intencional o estado do contexto delimitador. Permita que as funções locais sejam usadas em cenários em que um `static` método é necessário.

## Design detalhado

Uma função local declarada `static` não pode capturar o estado do escopo delimitador. Como resultado, locais, parâmetros e `this` do escopo de delimitação não estão disponíveis em uma `static` função local.

Uma `static` função local não pode referenciar membros de instância de um implícito ou explícito `this` ou de `base` referência.

Uma `static` função local pode referenciar `static` membros do escopo delimitador.

Uma `static` função local pode referenciar `constant` definições do escopo delimitador.

`nameof()` em uma `static` função local pode referenciar locais, parâmetros ou `this` ou `base` do escopo delimitador.

As regras de acessibilidade para `private` Membros no escopo delimitador são as mesmas para `static static` funções não locais.

Uma `static` definição de função local é emitida como um `static` método nos metadados, mesmo se usada apenas em um delegado.

Uma função não `static` local ou lambda pode capturar o estado de uma função local delimitadora `static`, mas não pode capturar o estado fora da função local de circunscrição `static`.

Uma `static` função local não pode ser invocada em uma árvore de expressão.

Uma chamada para uma função local é emitida como `call` em vez de `callvirt` , independentemente se a função local é `static` .

A resolução de sobrecarga de uma chamada em uma função local não é afetada pelo fato de a função local ser `static` .

A remoção do `static` modificador de uma função local em um programa válido não altera o significado do programa.

## Criar reuniões

[https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-09-10.md#static-local-functions ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-09-10.md#static-local-functions)

# Atribuição de coalescência nula

Artigo • 16/09/2021

- [x] proposta
- [x] protótipo: concluído
- [x] implementação: concluída
- [x] especificação: abaixo

## Resumo

Simplifica um padrão de codificação comum em que uma variável é atribuída a um valor se ela for nula.

Como parte dessa proposta, também desintegrarei os requisitos de tipo `??` para permitir uma expressão cujo tipo é um parâmetro de tipo irrestrito a ser usado no lado esquerdo.

## Motivação

É comum ver o código do formulário

```
C#  
  
if (variable == null)  
{  
    variable = expression;  
}
```

Essa proposta adiciona um operador binário não sobrecarregável à linguagem que executa essa função.

Houve pelo menos oito solicitações de comunidade separadas para esse recurso.

## Design detalhado

Adicionamos um novo formulário de operador de atribuição

```
antlr  
  
assignment_operator  
    : '??= '  
    ;
```

Que segue as [regras semânticas existentes para operadores de atribuição compostos](#), exceto que elide a atribuição se o lado esquerdo for não nulo. As regras para esse recurso são as seguintes.

Dado `a ??= b`, em que `A` é o tipo de `a`, `B` é o tipo de `b`, e `A0` é o tipo subjacente de `A`. If `A` é um tipo de valor anulável:

1. Se não `A` existir ou for um tipo de valor não anulável, ocorrerá um erro em tempo de compilação.
2. Se `B` não for implicitamente conversível para `A` ou `A0` (se `A0` existir), ocorrerá um erro em tempo de compilação.
3. Se `A0` existir e `B` for implicitamente conversível para `A0`, e `B` não for dinâmico, o tipo de `a ??= b` será `A0`. `a ??= b` é avaliado em tempo de execução como:

C#

```
var tmp = a.GetValueOrDefault();
if (!a.HasValue) { tmp = b; a = tmp; }
tmp
```

Exceto que `a` é avaliado apenas uma vez.

4. Caso contrário, o tipo de `a ??= b` é `A`. `a ??= b` é avaliado em tempo de execução como `a ?? (a = b)`, exceto que `a` é avaliado apenas uma vez.

Para o relaxamento dos requisitos de tipo do `??`, atualizamos a especificação em que ele declara atualmente, dado `a ?? b`, em que `A` é o tipo de `a`:

1. Se existir e não for um tipo anulável ou um tipo de referência, ocorrerá um erro em tempo de compilação.

Nós liberamos esse requisito para:

1. Se existir e for um tipo de valor não anulável, ocorrerá um erro em tempo de compilação.

Isso permite que o operador de União nulo funcione em parâmetros de tipo irrestrito, pois o parâmetro de tipo não restrito `T` existe, não é um tipo anulável e não é um tipo de referência.

## Desvantagens

Assim como ocorre com qualquer recurso de linguagem, devemos questionar se a complexidade adicional para o idioma é repagada na clareza adicional oferecida ao corpo de programas em C# que se beneficiaria do recurso.

## Alternativas

O programador pode escrever `(x = x ?? y)`, `if (x == null) x = y;` ou `x ?? (x = y)` manualmente.

## Perguntas não resolvidas

- [] Requer revisão LDM
- [] Também devemos dar suporte `&&=` a `||=` operadores e?

## Criar reuniões

Nenhum.

# Membros da instância ReadOnly

Artigo • 16/09/2021

Problema defensor: [https://github.com/dotnet/csharplang/issues/1710 ↗](https://github.com/dotnet/csharplang/issues/1710)

## Resumo

Forneça uma maneira de especificar membros de instância individuais em um struct não modificar o estado, da mesma maneira que `readonly struct` especifica que nenhum membro de instância modifica o estado.

Vale a pena observar que `readonly instance member` != `pure instance member`. Um `pure` membro de instância garante que nenhum Estado seja modificado. Um `readonly` membro de instância só garante que o estado da instância não será modificado.

Todos os membros de instância em um `readonly struct` podem ser considerados implicitamente `readonly instance members`. Explicit `readonly instance members` declarado em structs não ReadOnly se comportaria da mesma maneira. Por exemplo, eles ainda criaria cópias ocultas se você chamou um membro de instância (na instância atual ou em um campo da instância) que, por si só, não é ReadOnly.

## Motivação

Hoje, os usuários têm a capacidade de criar `readonly struct` tipos que o compilador impõe que todos os campos sejam ReadOnly (e por extensão, que nenhum membro de instância modifique o estado). No entanto, há alguns cenários em que você tem uma API existente que expõe campos acessíveis ou que tem uma combinação de membros que fazem mutação e não mutação. Sob essas circunstâncias, você não pode marcar o tipo como `readonly` (seria uma alteração significativa).

Isso normalmente não tem muito impacto, exceto no caso de `in` parâmetros. Com `in` parâmetros para structs não ReadOnly, o compilador fará uma cópia do parâmetro para cada invocação de membro de instância, pois não pode garantir que a invocação não modifique o estado interno. Isso pode levar a uma infinidade de cópias e pior desempenho geral do que se você acabou de passar o struct diretamente por valor. Para obter um exemplo, consulte este código em [sharplab ↗](#)

Alguns outros cenários em que as cópias ocultas podem ocorrer incluem `static readonly fields` e `literals`. Se eles tiverem suporte no futuro, `blittable constants`

acabarão no mesmo barco; ou seja, todos eles precisarão de uma cópia completa (em invocação de membro de instância) se a estrutura não estiver marcada `readonly`.

## Design

Permitir que um usuário especifique que um membro de instância é, ele mesmo, `readonly` e não modifica o estado da instância (com toda a verificação apropriada feita pelo compilador, é claro). Por exemplo:

C#

```
public struct Vector2
{
    public float x;
    public float y;

    public readonly float GetLengthReadonly()
    {
        return MathF.Sqrt(LengthSquared);
    }

    public float GetLength()
    {
        return MathF.Sqrt(LengthSquared);
    }

    public readonly float GetLengthIllegal()
    {
        var tmp = MathF.Sqrt(LengthSquared);

        x = tmp;      // Compiler error, cannot write x
        y = tmp;      // Compiler error, cannot write y

        return tmp;
    }

    public readonly float LengthSquared
    {
        get
        {
            return (x * x) +
                   (y * y);
        }
    }
}

public static class MyClass
{
    public static float ExistingBehavior(in Vector2 vector)
    {
        // This code causes a hidden copy, the compiler effectively emits:
```

```

        //     var tmpVector = vector;
        //     return tmpVector.GetLength();
        //
        // This is done because the compiler doesn't know that `GetLength()`
        // won't mutate `vector`.

        return vector.GetLength();
    }

    public static float ReadonlyBehavior(in Vector2 vector)
{
    // This code is emitted exactly as listed. There are no hidden
    // copies as the `readonly` modifier indicates that the method
    // won't mutate `vector`.

    return vector.GetLengthReadonly();
}
}

```

ReadOnly pode ser aplicado a acessadores de propriedade para indicar que `this` não serão modificados no acessador. Os exemplos a seguir têm setters ReadOnly porque esses acessadores modificam o estado do campo de membro, mas não modificam o valor desse campo de membro.

C#

```

public readonly int Prop1
{
    get
    {
        return this._store["Prop1"];
    }
    set
    {
        this._store["Prop1"] = value;
    }
}

```

Quando `readonly` é aplicado à sintaxe de propriedade, isso significa que todos os acessadores são `readonly`.

C#

```

public readonly int Prop2
{
    get
    {
        return this._store["Prop2"];
    }
    set

```

```
{  
    this._store["Prop2"] = value;  
}  
}
```

ReadOnly só pode ser aplicado a acessadores que não permutam o tipo recipiente.

C#

```
public int Prop3  
{  
    readonly get  
    {  
        return this._prop3;  
    }  
    set  
    {  
        this._prop3 = value;  
    }  
}
```

ReadOnly pode ser aplicado a algumas propriedades implementadas automaticamente, mas não terá um efeito significativo. O compilador tratará todos os getters autoimplementados como ReadOnly se a `readonly` palavra-chave estiver presente ou não.

C#

```
// Allowed  
public readonly int Prop4 { get; }  
public int Prop5 { readonly get; }  
public int Prop6 { readonly get; set; }  
  
// Not allowed  
public readonly int Prop7 { get; set; }  
public int Prop8 { get; readonly set; }
```

ReadOnly pode ser aplicado a eventos implementados manualmente, mas não a eventos do tipo campo. ReadOnly não pode ser aplicado a acessadores de evento individuais (Adicionar/remover).

C#

```
// Allowed  
public readonly event Action<EventArgs> Event1  
{  
    add { }  
    remove { }  
}
```

```
}
```

```
// Not allowed
```

```
public readonly event Action<EventArgs> Event2;
```

```
public event Action<EventArgs> Event3
```

```
{
```

```
    readonly add { }
```

```
    readonly remove { }
```

```
}
```

```
public static readonly event Event4
```

```
{
```

```
    add { }
```

```
    remove { }
```

```
}
```

Alguns exemplos de sintaxe:

- Membros de apto para de expressão: `public readonly float ExpressionBodiedMember => (x * x) + (y * y);`
- Restrições genéricas: `public readonly void GenericMethod<T>(T value) where T : struct { }`

O compilador emitiria o membro da instância, como de costume, e também emitiria um atributo reconhecido do compilador indicando que o membro da instância não modifica o estado. Isso efetivamente faz com que o `this` parâmetro Hidden se torne `in T` em vez de `ref T`.

Isso permitiria que o usuário chamassem o método de instância dito com segurança sem que o compilador precise fazer uma cópia.

As restrições incluem:

- O `readonly` modificador não pode ser aplicado a métodos estáticos, construtores ou destruidores.
- O `readonly` modificador não pode ser aplicado a delegados.
- O `readonly` modificador não pode ser aplicado a membros da classe ou interface.

## Desvantagens

As mesmas desvantagens que existem com os `readonly struct` métodos hoje. Determinado código ainda pode causar cópias ocultas.

## Observações

Usar um atributo ou outra palavra-chave também pode ser possível.

Essa proposta está um pouco relacionada a (mas é mais um subconjunto de) `functional purity` e/ou `constant expressions`, ambas com propostas existentes.

# Permitir `stackalloc` em contextos aninhados

Artigo • 16/09/2021

## Alocação da pilha

Modificamos a [alocação da pilha](#) de seções da especificação da linguagem C# para relaxar os locais quando uma `stackalloc` expressão pode aparecer. Nós excluímos

```
antlr

local_variable_initializer_unsafe
: stackalloc_initializer
;

stackalloc_initializer
: 'stackalloc' unmanaged_type '[' expression ']'
;
```

e substitua-os por

```
antlr

primary_no_array_creation_expression
: stackalloc_initializer
;

stackalloc_initializer
: 'stackalloc' unmanaged_type '[' expression? ']' array_initializer?
| 'stackalloc' '[' expression? ']' array_initializer
;
```

Observe que a adição de um `array_initializer` para `stackalloc_initializer` (e a criação da expressão de índice opcional) era uma [extensão no C# 7,3](#) e não é descrita aqui.

O *tipo de elemento* da `stackalloc` expressão é o *unmanaged\_type* nomeado na expressão `stackalloc`, se houver, ou o tipo comum entre os elementos da `array_initializer` caso contrário.

O tipo de `stackalloc_initializer` com o *tipo de elemento* `K` depende de seu contexto sintático:

- Se a *stackalloc\_initializer* aparecer diretamente como a *local\_variable\_initializer* de uma instrução *local\_variable\_declaration* ou uma *for\_initializer*, seu tipo será `K*` .
- Caso contrário, seu tipo é `System.Span<K>` .

## Conversão de stackalloc

A *conversão stackalloc* é uma nova conversão interna implícita da expressão. Quando o tipo de um *stackalloc\_initializer* é `K*` , há uma *conversão* implícita de *stackalloc* do *stackalloc\_initializer* para o tipo `System.Span<K>` .

# Registros

Artigo • 16/09/2021

Essa proposta acompanha a especificação do recurso de registros do C# 9, como acordado pela equipe de design da linguagem C#.

A sintaxe de um registro é a seguinte:

```
antlr

record_declaration
    : attributes? class_modifier* 'partial'? 'record' identifier
    type_parameter_list?
        parameter_list? record_base? type_parameter_constraints_clause*
    record_body
    ;

record_base
    : ':' class_type argument_list?
    | ':' interface_type_list
    | ':' class_type argument_list? ',' interface_type_list
    ;

record_body
    : '{' class_member_declaration* '}' ';'?
    | ';'
    ;
```

Tipos de registro são tipos de referência, semelhantes a uma declaração de classe. É um erro para um registro fornecer um `record_base argument_list` se o não `record_declaration` contiver um `parameter_list`. No máximo uma declaração de tipo parcial de um registro parcial pode fornecer um `parameter_list`.

Os parâmetros de registro não podem usar `ref` `out` ou `this` modificadores (mas `in` e `params` são permitidos).

## Herança

Os registros não podem herdar de classes, a menos que a classe seja `object`, e as classes não podem herdar de registros. Os registros podem ser herdados de outros registros.

## Membros de um tipo de registro

Além dos membros declarados no corpo do registro, um tipo de registro tem membros sintetizados adicionais. Os membros são sintetizados, a menos que um membro com uma assinatura "correspondente" seja declarado no corpo do registro ou um membro não virtual concreto acessível com uma assinatura "correspondente" seja herdado. Dois membros são considerados correspondentes se tiverem a mesma assinatura ou serão considerados "ocultos" em um cenário de herança. É um erro para um membro de um registro ser nomeado "clone". É um erro para um campo de instância de um registro ter um tipo não seguro.

Os membros sintetizados são os seguintes:

## Membros de igualdade

Se o registro for derivado de `object`, o tipo de registro incluirá uma propriedade `ReadOnly` sintetizada equivalente a uma propriedade declarada da seguinte maneira:

C#

```
Type EqualityContract { get; };
```

A propriedade será `private` se o tipo de registro for `sealed`. Caso contrário, a propriedade será `virtual` e `protected`. A propriedade pode ser declarada explicitamente. Erro se a declaração explícita não corresponder à assinatura ou acessibilidade esperada ou se a declaração explícita não permitir a substituição em um tipo derivado e o tipo de registro não for `sealed`.

Se o tipo de registro for derivado de um tipo de registro base `Base`, o tipo de registro incluirá uma propriedade `ReadOnly` sintetizada equivalente a uma propriedade declarada da seguinte maneira:

C#

```
protected override Type EqualityContract { get; };
```

A propriedade pode ser declarada explicitamente. Erro se a declaração explícita não corresponder à assinatura ou acessibilidade esperada ou se a declaração explícita não permitir a substituição em um tipo derivado e o tipo de registro não for `sealed`. Erro se a propriedade sintetizada ou declarada explicitamente não substituir uma propriedade com essa assinatura no tipo de registro `Base` (por exemplo, se a propriedade estiver ausente no `Base`, ou lacrado, ou não virtual, etc.). A propriedade sintetizada retorna `typeof(R)` onde `R` é o tipo de registro.

O tipo de registro implementa `System.IEquatable<R>` e inclui uma sobrecarga com rigidez de tipos sintetizada de `Equals(R? other)` onde `R` é o tipo de registro. O método é `public`, e o método é, a `virtual` menos que o tipo de registro seja `sealed`. O método pode ser declarado explicitamente. Erro se a declaração explícita não corresponder à assinatura ou acessibilidade esperada ou se a declaração explícita não permitir substituí-la em um tipo derivado e o tipo de registro não for `sealed`.

Se `Equals(R? other)` for definido pelo usuário (não sintetizado), mas `GetHashCode` não for, um aviso será produzido.

C#

```
public virtual bool Equals(R? other);
```

O sintetizado `Equals(R?)` retorna `true` se e somente se cada um dos seguintes for `true`:

- `other` Não é `null`, e
- Para cada campo de instância `fieldN` no tipo de registro que não é herdado, o valor de `System.Collections.Generic.EqualityComparer<TN>.Default.Equals(fieldN, other.fieldN)` onde `TN` é o tipo de campo e
- Se houver um tipo de registro base, o valor de `base.Equals(other)` (uma chamada não virtual para `public virtual bool Equals(Base? other)`); caso contrário, o valor de `EqualityContract == other.EqualityContract`.

O tipo de registro inclui sintetizado `==` e `!=` operadores equivalentes aos operadores declarados da seguinte maneira:

C#

```
public static bool operator==(R? left, R? right)
    => (object)left == right || (left?.Equals(right) ?? false);
public static bool operator!=(R? left, R? right)
    => !(left == right);
```

O `Equals` método chamado pelo `==` operador é o `Equals(R? other)` método especificado acima. O `!=` operador delega para o `==` operador. Erro se os operadores forem declarados explicitamente.

Se o tipo de registro for derivado de um tipo de registro base `Base`, o tipo de registro incluirá uma substituição sintetizada equivalente a um método declarado da seguinte

maneira:

C#

```
public sealed override bool Equals(Base? other);
```

Erro se a substituição for declarada explicitamente. Ocorrerá um erro se o método não substituir um método com a mesma assinatura no tipo de registro `Base` (por exemplo, se o método estiver ausente no `Base`, ou lacrado, ou não virtual, etc.). A substituição sintetizada retorna `Equals((object?)other)`.

O tipo de registro inclui uma substituição sintetizada equivalente a um método declarado da seguinte maneira:

C#

```
public override bool Equals(object? obj);
```

Erro se a substituição for declarada explicitamente. Erro se o método não substituir `object.Equals(object? obj)` (por exemplo, devido ao sombreamento em tipos base intermediários, etc.). A substituição sintetizada retorna `Equals(other as R)` onde `R` é o tipo de registro.

O tipo de registro inclui uma substituição sintetizada equivalente a um método declarado da seguinte maneira:

C#

```
public override int GetHashCode();
```

O método pode ser declarado explicitamente. Erro se a declaração explícita não permitir substituí-la em um tipo derivado e o tipo de registro não for `sealed`. Erro se o método sintetizado ou declarado explicitamente não substituir `object.GetHashCode()` (por exemplo, devido ao sombreamento em tipos base intermediários, etc.).

Um aviso será relatado se um dos `Equals(R?)` e `GetHashCode()` for declarado explicitamente, mas o outro método não for explícito.

A substituição sintetizada de `GetHashCode()` retorna um `int` resultado da combinação dos seguintes valores:

- Para cada campo de instância `fieldN` no tipo de registro que não é herdado, o valor de

```
System.Collections.Generic.EqualityComparer<TN>.Default.GetHashCode(fieldN)
```

onde `TN` é o tipo de campo e

- Se houver um tipo de registro base, o valor de `base.GetHashCode()` ; caso contrário, o valor de

```
System.Collections.Generic.EqualityComparer<System.Type>.Default.GetHashCode(E  
qualityContract) .
```

Por exemplo, considere os seguintes tipos de registro:

C#

```
record R1(T1 P1);  
record R2(T1 P1, T2 P2) : R1(P1);  
record R3(T1 P1, T2 P2, T3 P3) : R2(P1, P2);
```

Para esses tipos de registro, os membros de igualdade sintetizados seriam algo como:

C#

```
class R1 : IEquatable<R1>  
{  
    public T1 P1 { get; init; }  
    protected virtual Type EqualityContract => typeof(R1);  
    public override bool Equals(object? obj) => Equals(obj as R1);  
    public virtual bool Equals(R1? other)  
    {  
        return !(other is null) &&  
            EqualityContract == other.EqualityContract &&  
            EqualityComparer<T1>.Default.Equals(P1, other.P1);  
    }  
    public static bool operator==(R1? left, R1? right)  
        => (object)left == right || (left?.Equals(right) ?? false);  
    public static bool operator!=(R1? left, R1? right)  
        => !(left == right);  
    public override int GetHashCode()  
    {  
        return  
Combine(EqualityComparer<Type>.Default.GetHashCode(EqualityContract),  
                EqualityComparer<T1>.Default.GetHashCode(P1));  
    }  
}  
  
class R2 : R1, IEquatable<R2>  
{  
    public T2 P2 { get; init; }  
    protected override Type EqualityContract => typeof(R2);  
    public override bool Equals(object? obj) => Equals(obj as R2);  
    public sealed override bool Equals(R1? other) => Equals((object?)other);  
    public virtual bool Equals(R2? other)  
    {
```

```

        return base.Equals((R1?)other) &&
            EqualityComparer<T2>.Default.Equals(P2, other.P2);
    }
    public static bool operator==(R2? left, R2? right)
        => (object)left == right || (left?.Equals(right) ?? false);
    public static bool operator!=(R2? left, R2? right)
        => !(left == right);
    public override int GetHashCode()
    {
        return Combine(base.GetHashCode(),
            EqualityComparer<T2>.Default.GetHashCode(P2));
    }
}

class R3 : R2, IEquatable<R3>
{
    public T3 P3 { get; init; }
    protected override Type EqualityContract => typeof(R3);
    public override bool Equals(object? obj) => Equals(obj as R3);
    public sealed override bool Equals(R2? other) => Equals((object?)other);
    public virtual bool Equals(R3? other)
    {
        return base.Equals((R2?)other) &&
            EqualityComparer<T3>.Default.Equals(P3, other.P3);
    }
    public static bool operator==(R3? left, R3? right)
        => (object)left == right || (left?.Equals(right) ?? false);
    public static bool operator!=(R3? left, R3? right)
        => !(left == right);
    public override int GetHashCode()
    {
        return Combine(base.GetHashCode(),
            EqualityComparer<T3>.Default.GetHashCode(P3));
    }
}

```

## Copiar e clonar Membros

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

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 lacrado, o Construtor será privado, caso contrário, ele será protegido. Um construtor de cópia explicitamente declarado deve ser público ou protegido, a menos que o registro seja lacrado. A primeira coisa que o construtor deve fazer é chamar um construtor de cópia da base ou um construtor de objeto sem parâmetro se o registro herdar de Object. Um erro será relatado se um construtor de cópia definido pelo usuário usar um inicializador de Construtor implícito ou explícito que não atenda a esse requisito. Depois que um construtor de cópia base é invocado, um construtor de cópia sintetizado copia valores para todos os campos de instância implicitamente ou explicitamente declarados dentro do tipo de registro. A única presença de um construtor de cópia, seja explícita ou implícita, não impede uma adição automática de um construtor de instância padrão.

Se um método de "clonagem" virtual estiver presente no registro base, o método "clone" sintetizado o substituirá e o tipo de retorno do método será o tipo atual contido se o recurso "Covariance Return" tiver suporte e o tipo de retorno de substituição for diferente. Um erro será produzido se o método clone de registro base estiver lacrado. Se um método "clone" virtual não estiver presente no registro base, o tipo de retorno do método clone será o tipo recipiente e o método será virtual, a menos que o registro seja lacrado ou abstrato. Se o registro recipiente for abstrato, o método clone sintetizado também será abstrato. Se o método "clone" não for abstrato, ele retornará o resultado de uma chamada para um construtor de cópia.

## Imprimindo Membros: os métodos `Dimembers` e `ToString`

Se o registro for derivado de `object`, o registro incluirá um método sintetizado equivalente a um método declarado da seguinte maneira:

C#

```
bool PrintMembers(System.Text.StringBuilder builder);
```

O método é `private` se o tipo de registro for `sealed`. Caso contrário, o método é `virtual` e `protected`.

O método:

1. para cada um dos membros imprimíveis do registro (campo público não estático e membros de propriedade legível), o acrescenta o nome desse membro seguido por "=" seguido pelo valor do membro separado por ",",
2. Retorna true se o registro tiver membros imprimíveis.

Para um membro que tem um tipo de valor, converteremos seu valor em uma representação de cadeia de caracteres usando o método mais eficiente disponível para a plataforma de destino. No momento, isso significa chamar `ToString` antes de passar para `StringBuilder.Append`.

Se o tipo de registro for derivado de um registro base `Base`, o registro incluirá uma substituição sintetizada equivalente a um método declarado da seguinte maneira:

C#

```
protected override bool PrintMembers(StringBuilder builder);
```

Se o registro não tiver nenhum membro imprimível, o método chamará o `PrintMembers` método base com um argumento (seu `builder` parâmetro) e retornará o resultado.

Caso contrário, o método:

1. chama o `PrintMembers` método base com um argumento (seu `builder` parâmetro),
2. Se o `PrintMembers` método retornou true, acrescente "," ao construtor,
3. para cada um dos membros imprimíveis do registro, o acrescenta o nome desse membro seguido por "=" seguido pelo valor do membro: `this.member` (ou `this.member.ToString()` para tipos de valor), separados por ",",
4. retornar true.

O `PrintMembers` método pode ser declarado explicitamente. Erro se a declaração explícita não corresponder à assinatura ou acessibilidade esperada ou se a declaração explícita não permitir a substituição em um tipo derivado e o tipo de registro não for `sealed`.

O registro inclui um método sintetizado equivalente a um método declarado da seguinte maneira:

C#

```
public override string ToString();
```

O método pode ser declarado explicitamente. Erro se a declaração explícita não corresponder à assinatura ou acessibilidade esperada ou se a declaração explícita não permitir a substituição em um tipo derivado e o tipo de registro não for `sealed`. Erro se o método sintetizado ou declarado explicitamente não substituir `object.ToString()` (por exemplo, devido ao sombreamento em tipos base intermediários, etc.).

O método sintetizado:

1. Cria uma `StringBuilder` instância do,
2. anexa o nome do registro ao construtor, seguido por "{",
3. invoca o método do registro `PrintMembers` , fornecendo a ele o construtor, seguido por "" se ele retornou verdadeiro,
4. acrescenta "}",
5. Retorna o conteúdo do construtor com `builder.ToString()` .

Por exemplo, considere os seguintes tipos de registro:

C#

```
record R1(T1 P1);
record R2(T1 P1, T2 P2, T3 P3) : R1(P1);
```

Para esses tipos de registro, os membros de impressão sintetizados seriam algo como:

C#

```
class R1 : IEquatable<R1>
{
    public T1 P1 { get; init; }

    protected virtual bool PrintMembers(StringBuilder builder)
    {
        builder.Append(nameof(P1));
        builder.Append(" = ");
        builder.Append(this.P1); // or builder.Append(this.P1.ToString());
    }

    if P1 has a value type

        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R1));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}

class R2 : R1, IEquatable<R2>
{
    public T2 P2 { get; init; }
    public T3 P3 { get; init; }
```

```

protected override bool PrintMembers(StringBuilder builder)
{
    if (base.PrintMembers(builder))
        builder.Append(", ");

    builder.Append(nameof(P2));
    builder.Append(" = ");
    builder.Append(this.P2); // or builder.Append(this.P2); if P2 has a
value type

    builder.Append(nameof(P3));
    builder.Append(" = ");
    builder.Append(this.P3); // or builder.Append(this.P3); if P3 has a
value type

    return true;
}

public override string ToString()
{
    var builder = new StringBuilder();
    builder.Append(nameof(R2));
    builder.Append(" { ");

    if (PrintMembers(builder))
        builder.Append(" ");

    builder.Append("}");
    return builder.ToString();
}
}

```

## Membros do registro posicional

Além dos membros acima, os registros com uma lista de parâmetros ("registros posicionais") sintetizam membros adicionais com as mesmas condições que os membros acima.

## Construtor principal

Um tipo de registro tem um construtor público cuja assinatura corresponde aos parâmetros de valor da declaração de tipo. Isso é chamado de Construtor principal para o tipo e faz com que o construtor de classe padrão declarado implicitamente, se presente, seja suprimido. É um erro ter um construtor primário e um construtor com a mesma assinatura já presentes na classe.

Em tempo de execução, o construtor primário

1. executa os inicializadores de instância que aparecem no corpo da classe
2. invoca o construtor da classe base com os argumentos fornecidos na `record_base` cláusula, se presente

Se um registro tiver um construtor primário, qualquer Construtor definido pelo usuário, exceto "Construtor de cópia", deverá ter um `this` inicializador de construtor explícito.

Os parâmetros do construtor primário, bem como os membros do registro, estão no escopo dentro do `argument_list` da `record_base` cláusula e em inicializadores de campos de instância ou propriedades. Os membros da instância seriam um erro nesses locais (semelhante a como os membros da instância estão no escopo em inicializadores de Construtor regulares hoje, mas um erro a ser usado), mas os parâmetros do construtor primário estaria no escopo e utilizáveis e sombreariam os membros. Os membros estáticos também seriam utilizáveis, semelhante a como as chamadas base e inicializadores funcionam em construtores comuns hoje.

Um aviso será produzido se um parâmetro do construtor primário não for lido.

As variáveis de expressão declaradas no `argument_list` estão no escopo dentro do `argument_list`. As mesmas regras de sombreamento em uma lista de argumentos de um inicializador de Construtor regular se aplicam.

## Propriedades

Para cada parâmetro de registro de uma declaração de tipo de registro, há um membro de propriedade pública correspondente cujo nome e tipo são tirados da declaração de parâmetro de valor.

Para um registro:

- Uma `get` propriedade pública e `init` automática é criada (consulte Especificação de `init` acessador separado). Uma propriedade herdada `abstract` com tipo correspondente é substituída. Erro se a propriedade herdada não tiver `public` substituíveis `get` e `init` acessadores. Erro se a propriedade herdada estiver oculta. A propriedade automática é inicializada para o valor do parâmetro de construtor primário correspondente. Os atributos podem ser aplicados à propriedade automática sintetizada e a seu campo de apoio usando `property:` ou `field:` destinos para atributos aplicados sintaticamente ao parâmetro de registro correspondente.

# Desconstruir

Um registro posicional com pelo menos um parâmetro sintetiza um método de instância de retorno de void público chamado desconstruir com uma declaração de parâmetro out para cada parâmetro da declaração de Construtor principal. Cada parâmetro do método desconstruir tem o mesmo tipo que o parâmetro correspondente da declaração de Construtor principal. O corpo do método atribui cada parâmetro do método desconstruir ao valor de um membro de instância acesso a um membro do mesmo nome. O método pode ser declarado explicitamente. Erro se a declaração explícita não corresponder à assinatura ou acessibilidade esperada ou for estática.

## Expressão with

Uma `with` expressão é uma nova expressão usando a sintaxe a seguir.

antlr

```
with_expression
  : switch_expression
  | switch_expression 'with' '{' member_initializer_list? '}'
  ;

member_initializer_list
  : member_initializer (',' member_initializer)*
  ;

member_initializer
  : identifier '=' expression
  ;
```

Uma `with` expressão não é permitida como uma instrução.

Uma `with` expressão permite uma "mutação não destrutiva", projetada para produzir uma cópia da expressão do destinatário com modificações em atribuições no `member_initializer_list`.

Uma `with` expressão válida tem um receptor com um tipo não void. O tipo de receptor deve ser um registro.

No lado direito da `with` expressão, há um `member_initializer_list` com uma sequência de atribuições para o *identificador*, que deve ser um campo de instância acessível ou Propriedade do tipo do destinatário.

Primeiro, o método "clone" do destinatário (especificado acima) é invocado e seu resultado é convertido no tipo do destinatário. Em seguida, cada `member_initializer` uma 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.

# Instruções de nível superior

Artigo • 16/09/2021

- [x] proposta
- [x] protótipo: iniciado
- [x] implementação: iniciada
- [] Especificação: não iniciada

## Resumo

Permita que uma sequência de *instruções* ocorra logo antes da *namespace\_member\_declaration*s de um *compilation\_unit* (ou seja, o arquivo de origem).

A semântica é que se tal sequência de *instruções* estiver presente, a seguinte declaração de tipo, módulo, o nome do tipo real e o nome do método, seria emitido:

```
c#  
  
static class Program  
{  
    static async Task Main(string[] args)  
    {  
        // statements  
    }  
}
```

Consulte também <https://github.com/dotnet/csharplang/issues/3117>.

## Motivação

Há uma certa quantidade de clichê em torno mesmo dos programas mais simples, devido à necessidade de um `Main` método explícito. Isso parece chegar à forma de aprendizado de idioma e clareza do programa. O objetivo principal do recurso é, portanto, permitir programas em C# sem um texto clichê desnecessário em relação a eles, para fins de aprendizes e a clareza do código.

## Design detalhado

### Syntax

A única sintaxe adicional é permitir uma sequência de *instruções* s em uma unidade de compilação, logo antes do *namespace\_member\_declaration* s:

```
antlr

compilation_unit
    : extern_alias_directive* using_directive* global_attributes? statement*
namespace_member_declaration*
;
```

Somente um *compilation\_unit* pode ter a *instrução* s.

Exemplo:

```
c#

if (args.Length == 0
    || !int.TryParse(args[0], out int n)
    || n < 0) return;
Console.WriteLine(Fib(n).curr);

(int curr, int prev) Fib(int i)
{
    if (i == 0) return (1, 0);
    var (curr, prev) = Fib(i - 1);
    return (curr + prev, curr);
}
```

## Semântica

Se quaisquer instruções de nível superior estiverem presentes em qualquer unidade de compilação do programa, o significado será como se elas fossem combinadas no corpo do bloco de um `Main` método de uma `Program` classe no namespace global, da seguinte maneira:

```
c#

static class Program
{
    static async Task Main(string[] args)
    {
        // statements
    }
}
```

Observe que os nomes "Program" e "Main" são usados apenas para fins ilustrativos, os nomes reais usados pelo compilador são dependentes de implementação e nem o tipo, nem o método pode ser referenciado pelo nome do código-fonte.

O método é designado como o ponto de entrada do programa. Métodos explicitamente declarados que por convenção podem ser considerados como candidatos de ponto de entrada são ignorados. Um aviso é relatado quando isso acontece. É um erro especificar o `-main:<type>` comutador do compilador quando há instruções de nível superior.

O método de ponto de entrada sempre tem um parâmetro formal, `string[] args`. O ambiente de execução cria e passa um `string[]` argumento contendo os argumentos de linha de comando que foram especificados quando o aplicativo foi iniciado. O `string[]` argumento nunca é nulo, mas pode ter um comprimento zero se nenhum argumento de linha de comando foi especificado. O parâmetro 'args' está no escopo nas instruções de nível superior e não está no escopo fora deles. Regras de conflito/sombreamento de nome regular se aplicam.

Operações assíncronas são permitidas em instruções de nível superior para o grau em que são permitidas em instruções dentro de um método de ponto de entrada assíncrono regular. No entanto, eles não são necessários, se `await` as expressões e outras operações assíncronas forem omitidas, nenhum aviso será produzido.

A assinatura do método de ponto de entrada gerado é determinada com base nas operações usadas pelas instruções de nível superior da seguinte maneira:

Async-operations\Return-with-expression	Presente	Ausente
Presente	<code>static Task&lt;int&gt; Main(string[] args)</code>	<code>static Task Main(string[] args)</code>
Ausente	<code>static int Main(string[] args)</code>	<code>static void Main(string[] args)</code>

O exemplo acima produziria a seguinte `$Main` declaração de método:

```
c#  
  
static class $Program  
{  
    static void $Main(string[] args)  
    {  
        if (args.Length == 0  
            || !int.TryParse(args[0], out int n)  
            || n < 0) return;  
        Console.WriteLine(Fib(n).curr);  
    }  
}
```

```
(int curr, int prev) Fib(int i)
{
    if (i == 0) return (1, 0);
    var (curr, prev) = Fib(i - 1);
    return (curr + prev, curr);
}
```

Ao mesmo tempo, um exemplo como este:

```
c#
await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");
```

produziria:

```
c#
static class $Program
{
    static async Task $Main(string[] args)
    {
        await System.Threading.Tasks.Task.Delay(1000);
        System.Console.WriteLine("Hi!");
    }
}
```

Um exemplo como este:

```
c#
await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");
return 0;
```

produziria:

```
c#
static class $Program
{
    static async Task<int> $Main(string[] args)
    {
        await System.Threading.Tasks.Task.Delay(1000);
        System.Console.WriteLine("Hi!");
        return 0;
    }
}
```

```
    }  
}
```

E um exemplo como este:

```
c#  
  
System.Console.WriteLine("Hi!");  
return 2;
```

produziria:

```
c#  
  
static class $Program  
{  
    static int $Main(string[] args)  
    {  
        System.Console.WriteLine("Hi!");  
        return 2;  
    }  
}
```

## Escopo de variáveis locais de nível superior e funções locais

Embora as variáveis e funções locais de nível superior sejam "encapsuladas" no método de ponto de entrada gerado, elas ainda devem estar no escopo em todo o programa em cada unidade de compilação. Para fins de avaliação de nome simples, depois que o namespace global for atingido:

- Primeiro, é feita uma tentativa de avaliar o nome dentro do método de ponto de entrada gerado e somente se essa tentativa falhar
- A avaliação "regular" dentro da declaração de namespace global é executada.

Isso pode levar ao nome sombreamento de namespaces e tipos declarados dentro do namespace global, bem como para sombreamento de nomes importados.

Se a avaliação de nome simples ocorrer fora das instruções de nível superior e a avaliação gerar uma variável ou função local de nível superior, isso deverá levar a um erro.

Dessa forma, protegemos nossa capacidade futura de abordar melhor as "funções de nível superior" (cenário 2 no <https://github.com/dotnet/csharplang/issues/3117>) e são

capazes de fornecer diagnósticos úteis aos usuários que acreditam erroneamente que têm suporte.

# Especificação de tipos de referência anulável

Artigo • 16/09/2021

*Este é um trabalho em andamento-várias partes estão ausentes ou incompletas.*

Esse recurso adiciona dois novos tipos de tipos anuláveis (tipos de referência anuláveis e tipos genéricos anuláveis) aos tipos de valor anulável existentes e apresenta uma análise de fluxo estático para fins de segurança nula.

## Syntax

### Tipos de referência anulável e parâmetros de tipo anuláveis

Tipos de referência anuláveis e parâmetros de tipo anuláveis têm a mesma sintaxe `T?` que a forma abreviada de tipos de valor anulável, mas não têm um formato longo correspondente.

Para os fins da especificação, a produção atual `nullable_type` é renomeada para `nullable_value_type` e as `nullable_reference_type` `nullable_type_parameter` produções são adicionadas:

```
antlr

type
: value_type
| reference_type
| nullable_type_parameter
| type_parameter
| type_unsafe
;

reference_type
: ...
| nullable_reference_type
;

nullable_reference_type
: non_nullable_reference_type '?'
;

non_nullable_reference_type
: reference_type
```

```

;

nullable_type_parameter
: non_nullable_non_value_type_parameter '?'
;

non_nullable_non_value_type_parameter
: type_parameter
;

```

O `non_nullable_reference_type` em um `nullable_reference_type` deve ser um tipo de referência não nulo (classe, interface, delegado ou matriz).

O `non_nullable_non_value_type_parameter` in `nullable_type_parameter` deve ser um parâmetro de tipo que não seja restrito a ser um tipo de valor.

Tipos de referência anuláveis e parâmetros de tipo anulável não podem ocorrer nas seguintes posições:

- como uma classe base ou interface
- como receptor de um `member_access`
- como o `type` em um `object_creation_expression`
- como o `delegate_type` em um `delegate_creation_expression`
- como `type` no, a `is_expression` `catch_clause` ou a `type_pattern`
- como o `interface` em um nome de membro de interface totalmente qualificado

Um aviso é fornecido em um `nullable_reference_type` e `nullable_type_parameter` em um contexto de anotação Anulável *desabilitado*.

## `class``class?` restrição and

A `class` restrição tem uma contraparte Anulável `class?`:

```

antlr

primary_constraint
: ...
| 'class' '?'
;
```

Um parâmetro de tipo restrito com `class` (em um contexto de anotação *habilitado*) deve ser instanciado com um tipo de referência não nulo.

Um parâmetro de tipo restrito com `class?` (ou `class` em um contexto de anotação *desabilitado*) pode ser instanciado com um tipo de referência anulável ou não nulo.

Um aviso é fornecido em uma `class?` restrição em um contexto de anotação *desabilitado*.

## notnull Constraint

Um parâmetro de tipo restrito com `notnull` pode não ser um tipo anulável (tipo de valor anulável, tipo de referência anulável ou parâmetro de tipo anulável).

```
antlr
```

```
primary_constraint
: ...
| 'notnull'
;
```

## default Constraint

A `default` restrição pode ser usada em uma substituição de método ou implementação explícita para ambiguidade, `T?` significando "parâmetro de tipo anulável" de "tipo de valor anulável" (`Nullable<T>`). Sem a `default` restrição, uma `T?` sintaxe em uma substituição ou implementação explícita será interpretada como `Nullable<T>`

Veja <https://github.com/dotnet/csharplang/blob/master/proposals/csharp-9.0/unconstrained-type-parameter-annotations.md#default-constraint>

## O operador NULL-tolerante

O operador post-Fix `!` é chamado de operador NULL-tolerante. Ele pode ser aplicado em um *primary\_expression* ou em um *null\_conditional\_expression*:

```
antlr
```

```
primary_expression
: ...
| null_forgiving_expression
;

null_forgiving_expression
: primary_expression '!'
;

null_conditional_expression
: primary_expression null_conditional_operations_no_suppression
suppression?
;
```

```

null_conditional_operations_no_suppression
: null_conditional_operations? '?' '.' identifier type_argument_list?
| null_conditional_operations? '?' '[' argument_list ']'
| null_conditional_operations '.' identifier type_argument_list?
| null_conditional_operations '[' argument_list ']'
| null_conditional_operations '(' argument_list? ')'
;

null_conditional_operations
: null_conditional_operations_no_suppression suppression?
;

suppression
: '!'
;

```

Por exemplo:

C#

```

var v = expr!;
expr!.M();
_ = a?.b!.c;

```

O `primary_expression` e `null_conditional_operations_no_suppression` deve ser de um tipo anulável.

O operador de sufixo `!` não tem efeito de tempo de execução-ele é avaliado como o resultado da expressão subjacente. Sua única função é alterar o estado nulo da expressão para "NOT NULL" e limitar os avisos fornecidos em seu uso.

## Diretivas de compilador anuláveis

`#nullable` as diretivas controlam os contextos de anotação e de aviso que permitem valor nulo.

antlr

```

pp_directive
: ...
| pp_nullable
;

pp_nullable
: whitespace? '#' whitespace? 'nullable' whitespace nullable_action
(nullable_target)? pp_new_line
;

```

```
nullable_action
  : 'disable'
  | 'enable'
  | 'restore'
;

nullable_target
  : 'warnings'
  | 'annotations'
;
```

#pragma warning as diretivas são expandidas para permitir a alteração do contexto de aviso anulável:

```
antlr
```

```
pragma_warning_body
  : ...
  | 'warning' whitespace warning_action whitespace 'nullable'
;
```

Por exemplo:

```
C#
```

```
#pragma warning disable nullable
```

## Contextos que permitem valor nulo

Cada linha de código-fonte tem um *contexto de anotação anulável* e um *contexto de aviso anulável*. Eles controlam se as anotações anuláveis têm efeito e se são fornecidos avisos de nulidade. O contexto de anotação de uma determinada linha está *desabilitado* ou *habilitado*. O contexto de aviso de uma determinada linha está *desabilitado* ou *habilitado*.

Ambos os contextos podem ser especificados no nível do projeto (fora do código-fonte do C#) ou em qualquer lugar dentro de um arquivo de origem por meio de #nullable diretivas de pré-processador. Se nenhuma configuração de nível de projeto for fornecida, o padrão será para os dois contextos a serem *desabilitados*.

A #nullable diretiva controla os contextos de anotação e de aviso dentro do texto de origem e tem precedência sobre as configurações de nível de projeto.

Uma diretiva define os contextos que ele controla para linhas de código subsequentes, até que outra diretiva a substitua ou até o final do arquivo de origem.

O efeito das diretivas é o seguinte:

- `#nullable disable`: Define a anotação anulável e contextos de aviso como *desabilitado*
- `#nullable enable`: Define a anotação anulável e contextos de aviso como *habilitados*
- `#nullable restore`: Restaura os contextos de anotação e de aviso anuláveis para as configurações do projeto
- `#nullable disable annotations`: Define o contexto de anotação anulável como *desabilitado*
- `#nullable enable annotations`: Define o contexto de anotação anulável como *habilitado*
- `#nullable restore annotations`: Restaura o contexto de anotação anulável para as configurações do projeto
- `#nullable disable warnings`: Define o contexto de aviso anulável como *desabilitado*
- `#nullable enable warnings`: Define o contexto de aviso anulável como *habilitado*
- `#nullable restore warnings`: Restaura o contexto de aviso anulável para as configurações do projeto

## Possibilidade de nulidade de tipos

Um determinado tipo pode ter um dos três nullabilities: *alheios*, *nonnull* e *Nullable*.

Tipos não *nulos* poderão causar avisos se um `null` valor potencial for atribuído a eles.

Os tipos *alheios* e *Nullable*, no entanto, são "*nulos atribuíveis*" e podem ter `null` valores atribuídos a eles sem avisos.

Os valores dos tipos *alheios* e *nonnull* podem ser desreferenciados ou atribuídos sem avisos. Os valores de tipos *anuláveis*, no entanto, são "*nulos*" e podem causar avisos quando desreferenciados ou atribuídos sem verificação nula adequada.

O *estado nulo padrão* de um tipo de rendimento nulo é "Talvez nulo" ou "Talvez padrão". O estado nulo padrão de um tipo não nulo de rendimento é "NOT NULL".

O tipo e o contexto de anotação anulável que ele ocorre em determinar sua nulidade:

- Um tipo de valor não nulo `s` é sempre não *nulo*
- Um tipo de valor Anulável `s?` é sempre *anulável*

- Um tipo de referência não anotado `c` em um contexto de anotação *desabilitado* é *alheios*
- Um tipo de referência não anotado `c` em um contexto de anotação *habilitado* é *não nulo*
- Um tipo de referência Anulável `c?` é *anulável* (mas um aviso pode ser produzido em um contexto de anotação *desabilitado*)

Além disso, os parâmetros de tipo levam suas restrições em conta:

- Um parâmetro de tipo `T` em que todas as restrições (se houver) são tipos anuláveis ou a `class?` restrição é *anulável*
- Um parâmetro de tipo `T` em que pelo menos uma restrição é *alheios* ou *não nulo*, ou uma das `struct` restrições or ou `class notnull is`
  - *alheios* em um contexto de anotação *desabilitado*
  - *Não nulo* em um contexto de anotação *habilitado*
- Um parâmetro de tipo anulável `T?` é *anulável*, mas um aviso é produzido em um contexto de anotação *desabilitado* se `T` não for um tipo de valor

## Alheios vs não nulo

`R type` é considerado para ocorrer em um determinado contexto de anotação quando o último token do tipo está dentro desse contexto.

Se um determinado tipo `c` de referência no código-fonte é interpretado como alheios ou não nulo depende do contexto de anotação desse código-fonte. Mas, uma vez estabelecida, ele é considerado parte desse tipo e "viaja com ele", por exemplo, durante a substituição de argumentos de tipo genérico. É como se houver uma anotação como `? type` no tipo, mas invisível.

## Restrições

Tipos de referência anuláveis podem ser usados como restrições genéricas.

`class?` é uma nova restrição que indica "tipo de referência possivelmente anulável", enquanto `class` em um contexto de anotação *habilitado* denota "tipo de referência não nula".

`default` é uma nova restrição que indica um parâmetro de tipo que não é conhecido como um tipo de referência ou de valor. Ele só pode ser usado em métodos substituídos e explicitamente implementados. Com essa restrição, `T?` significa um parâmetro de tipo anulável, em oposição a ser uma abreviação para `Nullable<T>`.

`notnull` é uma nova restrição que indica um parâmetro de tipo que não é nulo.

A nulidade de um argumento de tipo ou de uma restrição não afeta se o tipo satisfaz a restrição, exceto onde esse já é o caso hoje (os tipos de valores anuláveis não atendem à `struct` restrição). No entanto, se o argumento de tipo não atender aos requisitos de nulidade da restrição, um aviso poderá ser dado.

## Estado nulo e acompanhamento nulo

Cada expressão em um local de origem específico tem um *estado nulo*, que indica se acredita potencialmente ser possível avaliar como NULL. O estado NULL é "NOT NULL", "Talvez NULL" ou "Talvez default". O estado NULL é usado para determinar se um aviso deve ser fornecido sobre conversões e desreferências sem segurança nula.

A distinção entre "Talvez NULL" e "Talvez padrão" é sutil e se aplica aos parâmetros de tipo. A distinção é que um parâmetro `T` de tipo que tem o estado "Talvez seja nulo" significa que o valor está no domínio de valores válidos, `T` no entanto, o valor legal pode incluir `null`. Onde "Talvez o padrão" significa que o valor pode estar fora do domínio legal de valores para `T`.

Exemplo:

```
c#  
  
// The value `t` here has the state "maybe null". It's possible for `T` to  
be instantiated  
// with `string?` in which case `null` would be within the domain of legal  
values here. The  
// assumption though is the value provided here is within the legal values  
of `T`. Hence  
// if `T` is `string` then `null` will not be a value, just as we assume  
that `null` is not  
// provided for a normal `string` parameter  
void M<T>(T t)  
{  
    // There is no guarantee that default(T) is within the legal values for  
T hence the  
    // state *must* be "maybe-default" and hence `local` must be `T?`  
    T? local = default(T);  
}
```

## Acompanhamento nulo para variáveis

Para determinadas expressões que denotam variáveis, campos ou propriedades, o estado nulo é acompanhado entre ocorrências, com base nas atribuições a elas, nos

testes executados neles e no fluxo de controle entre elas. Isso é semelhante a como a atribuição definitiva é controlada para variáveis. As expressões rastreadas são aquelas do seguinte formato:

```
antlr

tracked_expression
: simple_name
| this
| base
| tracked_expression '.' identifier
;
```

Onde os identificadores denotam campos ou propriedades.

O estado nulo de variáveis rastreadas é "NOT NULL" em código inacessível. Isso segue outras decisões relacionadas ao código inacessível, como considerar que todos os locais serão definitivamente atribuídos.

*Descrever as transições de estado nulos semelhantes à atribuição definitiva*

## Estado nulo para expressões

O estado nulo de uma expressão é derivado de seu formulário e tipo e do estado nulo das variáveis envolvidas.

## Literais

O estado nulo de um `null` literal depende do tipo de destino da expressão. Se o tipo de destino for um parâmetro de tipo restrito a um tipo de referência, será "Talvez padrão". Caso contrário, será "Talvez NULL".

O estado nulo de um `default` literal depende do tipo de destino do `default` literal. Um `default` literal com o tipo `T` de destino tem o mesmo estado nulo que a `default(T)` expressão.

O estado nulo de qualquer outro literal é "NOT NULL".

## Nomes simples

Se um `simple_name` não for classificado como um valor, seu estado nulo será "NOT NULL". Caso contrário, é uma expressão rastreada e seu estado nulo é seu estado nulo rastreado neste local de origem.

## Acesso de membros

Se um `member_access` não for classificado como um valor, seu estado nulo será "NOT NULL". Caso contrário, se for uma expressão rastreada, seu estado nulo será seu estado NULL rastreado nesse local de origem. Caso contrário, seu estado nulo será o estado nulo padrão de seu tipo.

```
c#  
  
var person = new Person();  
  
// The receiver is a tracked expression hence the member_access of the  
// property  
// is tracked as well  
if (person.FirstName is not null)  
{  
    Use(person.FirstName);  
}  
  
// The return of an invocation is not a tracked expression hence the  
// member_access  
// of the return is also not tracked  
if (GetAnonymous().FirstName is not null)  
{  
    // Warning: Cannot convert null literal to non-nullable reference type.  
    Use(GetAnonymous().FirstName);  
}  
  
void Use(string s)  
{  
    // ...  
}  
  
public class Person  
{  
    public string? FirstName { get; set; }  
    public string? LastName { get; set; }  
  
    private static Person s_anonymous = new Person();  
    public static Person GetAnonymous() => s_anonymous;  
}
```

## Expressões de invocação

Se um `invocation_expression` invocar um membro declarado com um ou mais atributos para um comportamento nulo especial, o estado NULL será determinado por esses atributos. Caso contrário, o estado nulo da expressão será o estado nulo padrão de seu tipo.

O estado nulo de um `invocation_expression` não é acompanhado pelo compilador.

```
c#  
  
// The result of an invocation_expression is not tracked  
if (GetText() is not null)  
{  
    // Warning: Converting null literal or possible null value to non-  
    nullable type.  
    string s = GetText();  
    // Warning: Dereference of a possibly null reference.  
    Use(s);  
}  
  
// Nullable friendly pattern  
if (GetText() is string s)  
{  
    Use(s);  
}  
  
string? GetText() => ...  
Use(string s) { }
```

## Acesso a elemento

Se um `element_access` chamar um indexador declarado com um ou mais atributos para um comportamento nulo especial, o estado NULL será determinado por esses atributos. Caso contrário, o estado nulo da expressão será o estado nulo padrão de seu tipo.

```
c#  
  
object?[] array = ...;  
if (array[0] != null)  
{  
    // Warning: Converting null literal or possible null value to non-  
    nullable type.  
    object o = array[0];  
    // Warning: Dereference of a possibly null reference.  
    Console.WriteLine(o.ToString());  
}  
  
// Nullable friendly pattern  
if (array[0] is {} o)  
{  
    Console.WriteLine(o.ToString());  
}
```

## Acesso de base

Se `B` denota o tipo base do tipo delimitador, `base.I` tem o mesmo estado nulo que `((B)this).I` e `base[E]` tem o mesmo estado nulo que `((B)this)[E]`.

## Expressões padrão

`default(T)` tem o estado nulo com base nas propriedades do tipo `T`:

- Se o tipo for um tipo não *nulo*, ele terá o estado nulo "NOT NULL"
- Caso contrário, se o tipo for um parâmetro de tipo, ele terá o estado nulo "Talvez padrão"
- Caso contrário, ele tem o estado nulo "Talvez nulo"

## Expressões condicionais nulas?

Um `null_conditional_expression` tem o estado nulo com base no tipo de expressão.

Observe que isso se refere ao tipo de `null_conditional_expression`, e não ao tipo original do membro que está sendo invocado:

- Se o tipo for um tipo de valor *anulável*, ele terá o estado nulo "Talvez NULL"
- Caso contrário, se o tipo for um parâmetro de tipo *anulável*, ele terá o estado nulo "Talvez padrão"
- Caso contrário, ele tem o estado nulo "Talvez nulo"

## Expressões cast

Se uma expressão de conversão `(T)E` invocar uma conversão definida pelo usuário, o estado nulo da expressão será o estado nulo padrão para o tipo da conversão definida pelo usuário. Caso contrário:

- Se `T` for um tipo de valor não *nulo*, `T` o estado nulo será "NOT NULL"
- Senão `T`, se for um tipo de valor *anulável*, `T` terá o estado nulo "Talvez nulo"
- Senão `T`, se for um tipo *anulável* no formulário `U?` em que `U` é um parâmetro de tipo, `T` o estado nulo será "Talvez padrão"
- Caso contrário `T`, se for um tipo *anulável* e `E` tiver um estado nulo "Talvez nulo" ou "Talvez padrão", `T` o estado nulo será "Talvez nulo"
- Caso contrário `T`, se for um parâmetro de tipo e `E` tiver um estado nulo "Talvez nulo" ou "Talvez padrão", `T` terá o estado nulo "Talvez padrão"
- Mais `T` tem o mesmo estado nulo que `E`

# Operadores unários e binários

Se um operador unário ou binário invocar um operador definido pelo usuário, o estado nulo da expressão será o estado nulo padrão para o tipo do operador definido pelo usuário. Caso contrário, é o estado nulo da expressão.

*Algo especial a fazer para binário + sobre cadeias de caracteres e delegados?*

## Expressões Await

O estado nulo de `await E` é o estado nulo padrão de seu tipo.

## O operador `as`

O estado nulo de uma `E as T` expressão depende primeiro das propriedades do tipo `T`. Se o tipo de `T` for *nonnullable*, o estado NULL será "NOT NULL". Caso contrário, o estado nulo dependerá da conversão do tipo de `E` para tipo `T`:

- Se a conversão for uma identidade, boxing, referência implícita ou conversão implícita de anulável, o estado NULL será o estado nulo de `E`
- Caso contrário `T`, se for um parâmetro de tipo, ele terá o estado nulo "Talvez padrão"
- Caso contrário, ele tem o estado nulo "Talvez nulo"

## O operador de União nula

O estado nulo de `E1 ?? E2` é o estado nulo de `E2`

## O operador condicional

O estado nulo de `E1 ? E2 : E3` é baseado no estado nulo de `E2` e `E3`:

- Se ambos forem "NOT NULL", o estado NULL será "NOT NULL"
- Caso contrário, se for "Talvez padrão", o estado nulo será "Talvez padrão"
- Caso contrário, o estado NULL será "NOT NULL"

## Expressões de consulta

O estado nulo de uma expressão de consulta é o estado nulo padrão de seu tipo.

*Trabalho adicional necessário aqui*

# Operadores de atribuição

`E1 = E2` e `E1 op= E2` têm o mesmo estado nulo que `E2` após qualquer conversões implícitas terem sido aplicadas.

## Expressões que propagam o estado nulo

`(E) ``checked(E)` e `unchecked(E)` todos têm o mesmo estado nulo que `E`.

## Expressões que nunca são nulas

O estado nulo dos formulários de expressão a seguir é sempre "NOT NULL":

- `this` acesso
- cadeias de caracteres interpoladas
- `new` expressões (objeto, delegado, objeto anônimo e expressões de criação de matriz)
- Expressões `typeof`
- Expressões `nameof`
- funções anônimas (métodos anônimos e expressões lambda)
- expressões tolerante nulas
- Expressões `is`

## Funções aninhadas

Funções aninhadas (lambda e funções locais) são tratadas como métodos, exceto em relação às variáveis capturadas. O estado inicial de uma variável capturada dentro de uma função lambda ou local é a interseção do estado anulável da variável em todos os "usos" dessa função ou lambda aninhado. Um uso de uma função local é uma chamada para essa função ou onde ela é convertida em um delegado. Um uso de um lambda é o ponto no qual ele é definido na origem.

## Inferência de tipos

### variáveis de local digitadas implicitamente de tipo nulo

`var` infere um tipo anotado para tipos de referência e parâmetros de tipo que não são restritos a serem um tipo de valor. Por exemplo:

- no `var s = "";` `var` é inferido como `string?`.

- em `var t = new T();` com um irrestrito `T` `var`, o é inferido como `T?`.

## Inferência de tipo genérico

A inferência de tipo genérico é aprimorada para ajudar a decidir se os tipos de referência inferidos devem ser anuláveis ou não. Isso é um melhor esforço. Ele pode gerar avisos sobre restrições de nulidade e pode levar a avisos anuláveis quando os tipos inferidos da sobrecarga selecionada são aplicados aos argumentos.

## A primeira fase

Os tipos de referência anuláveis fluem para os limites das expressões iniciais, conforme descrito abaixo. Além disso, dois novos tipos de limites, `null` ou seja, `default` são apresentados. Sua finalidade é realizar ocorrências de `null` ou `default` nas expressões de entrada, o que pode fazer com que um tipo inferido seja anulável, mesmo quando não fosse. Isso funciona mesmo para tipos de *valores* anuláveis, que são aprimorados para pegar a "nulidade" no processo de inferência.

A determinação dos limites a serem adicionados na primeira fase é aprimorada da seguinte maneira:

Se um argumento `Ei` tiver um tipo de referência, o tipo `U` usado para a inferência dependerá do estado nulo do, `Ei` bem como do seu tipo declarado:

- Se o tipo declarado for um tipo de referência não nulo `U0` ou um tipo de referência Anulável `U0?`,
  - Se o estado nulo de `Ei` for "NOT NULL", `U` será `U0`
  - Se o estado nulo de `Ei` for "Talvez NULL", `U` será `U0?`
- Caso contrário `Ei`, se tiver um tipo declarado, `U` será esse tipo
- Caso contrário `Ei`, se for `null`, `U` o limite especial será `null`
- Caso contrário `Ei`, se for `default`, `U` o limite especial será `default`
- Caso contrário, nenhuma inferência será feita.

## Inferências exatas, de limite superior e de limite inferior

Em inferências *do tipo* `U` *para o tipo* `V`, se `V` for um tipo de referência Anulável `V0?`, `V0` será usado em vez de `V` nas cláusulas a seguir.

- Se `V` for uma das variáveis de tipo não fixas, `U` será adicionado como um limite exato, superior ou inferior como antes

- Caso contrário, se `U` for `null` ou `default`, nenhuma inferência será feita
- Caso contrário, se `U` for um tipo de referência Anulável `U0?`, `U0` será usado em vez de `U` nas cláusulas subsequentes.

A essência é que a nulidade que se refere diretamente a uma das variáveis de tipo não fixos é preservada em seus limites. Por outro lado, para as inferências que recursivamente os tipos de origem e destino, a nulidade é ignorada. Pode ou não corresponder, mas se não for, um aviso será emitido mais tarde se a sobrecarga for escolhida e aplicada.

## Resolvendo

Atualmente, a especificação não faz um bom trabalho descrevendo o que acontece quando vários limites são conversíveis de identidade entre si, mas são diferentes. Isso pode acontecer entre `object` e `dynamic`, entre os tipos de tupla que diferem somente em nomes de elementos, entre os tipos construídos e agora também entre `C` e `C?` para tipos de referência.

Além disso, precisamos propagar "nulidade" das expressões de entrada para o tipo de resultado.

Para lidar com isso, adicionamos mais fases para corrigir, que agora é:

1. Reunir todos os tipos em todos os limites como candidatos, removendo `?` de todos os que são tipos de referência anuláveis
2. Elimine os candidatos com base nos requisitos de limites exatos, inferiores e superiores (mantendo `null` e `default` limitado)
3. Elimine os candidatos que não têm uma conversão implícita para todos os outros candidatos
4. Se todos os candidatos restantes não tiverem conversões de identidade entre si, a inferência de tipos falhará
5. *Mescle* os candidatos restantes conforme descrito abaixo
6. Se o candidato resultante for um tipo de referência ou um tipo de valor não nulo e *todos* os limites exatos ou *qualquer* um dos limites inferiores forem tipos de valor anulável, tipos de referência anuláveis ou `null` `default`, em seguida, `?` será adicionado ao candidato resultante, tornando-o um tipo de valor anulável ou tipo de referência.

A *mesclagem* é descrita entre dois tipos candidatos. Ele é transitivo e comutador, portanto, os candidatos podem ser mesclados em qualquer ordem com o mesmo resultado final. Ele será indefinido se os dois tipos candidatos não forem conversíveis de identidade entre si.

A função *Merge* usa dois tipos candidatos e uma direção (+ ou -):

- $\text{Merge}(\text{T}, \text{T}, d) = \text{T}$
- $\text{Merge}(\text{S}, \text{T?}, +) = \text{mesclar}(\text{S?}, \text{T}, +) = \text{Merge}(\text{S}, \text{T}, +)?$
- $\text{Merge}(\text{S}, \text{T?}, -) = \text{mesclar}(\text{S?}, \text{T}, -) = \text{Merge}(\text{S}, \text{T}, -)$
- $\text{Mesclar}(\text{C}\langle\text{S}_1, \dots, \text{S}_n\rangle, \text{C}\langle\text{T}_1, \dots, \text{T}_n\rangle, +) = \text{mesclar} \text{C}\langle (\text{S}_1, \text{T}_1, \text{D}_1), \dots, \text{mesclagem}(\text{S}_n, \text{T}_n, \text{D}_n) \rangle, \text{em que}$ 
  - $\text{di} = +$  Se o  $i$  parâmetro 'th Type de  $\text{C}\langle \dots \rangle$  for Covariance
  - $\text{di} = -$  Se o  $i$  parâmetro 'th Type de  $\text{C}\langle \dots \rangle$  for contrato ou constante
- $\text{Mesclar}(\text{C}\langle\text{S}_1, \dots, \text{S}_n\rangle, \text{C}\langle\text{T}_1, \dots, \text{T}_n\rangle, -) = \text{mesclar} \text{C}\langle (\text{S}_1, \text{T}_1, \text{D}_1), \dots, \text{mesclagem}(\text{S}_n, \text{T}_n, \text{D}_n) \rangle, \text{em que}$ 
  - $\text{di} = -$  Se o  $i$  parâmetro 'th Type de  $\text{C}\langle \dots \rangle$  for Covariance
  - $\text{di} = +$  Se o  $i$  parâmetro 'th Type de  $\text{C}\langle \dots \rangle$  for contrato ou constante
- $\text{Mesclar}((\text{S}_1 \text{s}_1, \dots, \text{S}_n \text{s}_n), (\text{T}_1 \text{t}_1, \dots, \text{T}_n \text{t}_n), d) = \text{mesclar} ((\text{S}_1, \text{T}_1, d) \text{n}_1, \dots, \text{mesclar}(\text{S}_n, \text{T}_n, d) \text{n}_n), \text{em que}$ 
  - $\text{ni}$  está ausente se  $\text{si}$  e  $\text{ti}$  diferir ou se ambos estiverem ausentes
  - $\text{ni}$  é  $\text{si}$  If  $\text{si}$  e é  $\text{ti}$  o mesmo
- $\text{Mesclar}(\text{object}, \text{dynamic}) = \text{mesclar}(\text{dynamic}, \text{object}) = \text{dynamic}$

## Warnings

### Possível atribuição nula

### Desreferência de nulo potencial

### Incompatibilidade de nulidade de restrição

### Tipos anuláveis no contexto de anotação desabilitado

### Incompatibilidade de anulação de substituição e implementação

### Atributos para comportamento nulo especial

# Alterações de correspondência de padrões para C# 9,0

Artigo • 16/09/2021

Estamos considerando uma pequena quantidade de aprimoramentos na correspondência de padrões para C# 9,0 que têm sinergia natural e funcionam bem para resolver vários problemas comuns de programação:

- <https://github.com/dotnet/csharplang/issues/2925> Padrões de tipo
- <https://github.com/dotnet/csharplang/issues/1350> Padrões entre parênteses para impor ou enfatizar a precedência dos novos combinadores
- <https://github.com/dotnet/csharplang/issues/1350> Padrões conjuntiva que exigem dois de dois padrões diferentes para corresponder;
- <https://github.com/dotnet/csharplang/issues/1350> Padrões disjunctive que exigem um dos dois padrões diferentes para corresponder;
- <https://github.com/dotnet/csharplang/issues/1350> Padrões negados `not` que exigem um determinado padrão para *não* corresponder; e
- <https://github.com/dotnet/csharplang/issues/812> Padrões relacionais que exigem que o valor de entrada seja menor que, menor ou igual a, etc. uma determinada constante.

## Padrões entre parênteses

Padrões entre parênteses permitem que o programador Coloque parênteses em qualquer padrão. Isso não é tão útil com os padrões existentes no C# 8,0, no entanto, os novos combinadores de padrões apresentam uma precedência que o programador pode querer substituir.

```
antlr

primary_pattern
    : parenthesized_pattern
    | // all of the existing forms
    ;
parenthesized_pattern
    : '(' pattern ')'
    ;
```

## Padrões de tipo

Permitimos um tipo como um padrão:

```
antlr

primary_pattern
    : type-pattern
    | // all of the existing forms
    ;
type_pattern
    : type
    ;
```

Isso retcons que o *tipo de expressão is-Type* existente seja uma *expressão is-Pattern-*, na qual o padrão é um *padrão de tipo*, embora não mudemos a árvore de sintaxe produzida pelo compilador.

Um problema de implementação sutil é que essa gramática é ambígua. Uma cadeia de caracteres como `a.b` pode ser analisada como um nome qualificado (em um contexto de tipo) ou uma expressão pontilhada (em um contexto de expressão). O compilador já é capaz de tratar um nome qualificado da mesma forma que uma expressão pontilhada para lidar com algo como `e is Color.Red`. A análise semântica do compilador seria mais estendida para ser capaz de ligar um padrão constante (sintático) (por exemplo, uma expressão pontilhada) como um tipo para tratá-lo como um padrão de tipo vinculado para dar suporte a esse constructo.

Após essa alteração, você poderá escrever

```
C#

void M(object o1, object o2)
{
    var t = (o1, o2);
    if (t is (int, string)) {} // test if o1 is an int and o2 is a string
    switch (o1) {
        case int: break; // test if o1 is an int
        case System.String: break; // test if o1 is a string
    }
}
```

## Padrões relacionais

Os padrões relacionais permitem que o programador expresse que um valor de entrada deve satisfazer uma restrição relacional em comparação a um valor constante:

```
C#
```

```

public static LifeStage LifeStageAtAge(int age) => age switch
{
    < 0 => LifeStage.Prenatal,
    < 2 => LifeStage.Infant,
    < 4 => LifeStage.Toddler,
    < 6 => LifeStage.EarlyChild,
    < 12 => LifeStage.MiddleChild,
    < 20 => LifeStage.Adolescent,
    < 40 => LifeStage.EarlyAdult,
    < 65 => LifeStage.MiddleAdult,
    _ => LifeStage.LateAdult,
};

```

Os padrões relacionais dão suporte aos operadores relacionais `<`, `<=`, `>` e `>=` em todos os tipos internos que dão suporte a esses operadores relacionais binários com dois operandos do mesmo tipo em uma expressão. Especificamente, damos suporte a todos esses padrões relacionais para,,, `sbyte` `byte` , `short` `ushort` `int` `uint` `long` `ulong` `char` , `float` , `double` , `decimal` , `nint` e `nuint` .

antlr

```

primary_pattern
: relational_pattern
;
relational_pattern
: '<' relational_expression
| '<=' relational_expression
| '>' relational_expression
| '>=' relational_expression
;

```

A expressão é necessária para ser avaliada como um valor constante. Erro se esse valor constante for `double.NaN` ou `float.NaN` . Erro se a expressão for uma constante nula.

Quando a entrada é um tipo para o qual um operador relacional binário interno adequado é definido, que é aplicável com a entrada como seu operando esquerdo e a constante determinada como operando à direita, a avaliação desse operador é usada como o significado do padrão relacional. Caso contrário, convertemos a entrada para o tipo da expressão usando uma conversão explícita anulável ou unboxing. É um erro de tempo de compilação se essa conversão não existir. O padrão é considerado para não corresponder se a conversão falhar. Se a conversão for realizada com sucesso, o resultado da operação de correspondência de padrões será o resultado da avaliação da expressão `e OP v` em que `e` é a entrada convertida, `OP` é o operador relacional e `v` é a expressão constante.

# Combinadores de padrões

Os *combinadores* de padrões permitem a correspondência de ambos os dois padrões diferentes usando `and` (isso pode ser estendido para qualquer número de padrões pelo uso repetido de `and`), de dois padrões diferentes usando `or` (Idem) ou a *negação* de um padrão usando `not`.

Um uso comum de um combinador será o idioma

```
c#  
  
if (e is not null) ...
```

Mais legível do que o idioma atual `e is object`, esse padrão claramente expressa que um deles está verificando um valor não nulo.

Os `and` `or` combinadores e serão úteis para testar intervalos de valores

```
c#  
  
bool IsLetter(char c) => c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

Este exemplo ilustra que terá `and` uma prioridade de análise maior (ou seja, ligará mais de forma mais detalhada) do que `or`. O programador pode usar o *padrão entre parênteses* para tornar a precedência explícita:

```
c#  
  
bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');
```

Como todos os padrões, esses combinadores podem ser usados em qualquer contexto no qual um padrão é esperado, incluindo padrões aninhados, a *expressão is-Pattern-*, a *expressão switch* e o padrão de um rótulo case da instrução switch.

```
antlr  
  
pattern  
    : disjunctive_pattern  
    ;  
disjunctive_pattern  
    : disjunctive_pattern 'or' conjunctive_pattern  
    | conjunctive_pattern  
    ;  
conjunctive_pattern  
    : conjunctive_pattern 'and' negated_pattern
```

```

| negated_pattern
;
negated_pattern
: 'not' negated_pattern
| primary_pattern
;
primary_pattern
: // all of the patterns forms previously defined
;

```

## Alterar para ambiguidades de gramática 7.5.4.2

Devido à introdução do padrão de *tipo*, é possível que um tipo genérico apareça antes do token `=>`. Portanto, adicionamos `=>` ao conjunto de tokens listados em *ambiguidades de gramática 7.5.4.2* para permitir a desambiguidade do `<` que começa a lista de argumentos de tipo. Consulte também <https://github.com/dotnet/roslyn/issues/47614>.

## Problemas em aberto com alterações propostas

### Sintaxe para operadores relacionais

São `and`, `or` e `not` algum tipo de palavra-chave contextual? Nesse caso, há uma alteração significativa (por exemplo, em comparação com seu uso como um designador em um *padrão de declaração*).

### Semântica (por exemplo, tipo) para operadores relacionais

Esperamos dar suporte a todos os tipos primitivos que podem ser comparados em uma expressão usando um operador relacional. O significado em casos simples é claro

```
c#
bool IsValidPercentage(int x) => x is >= 0 and <= 100;
```

Mas quando a entrada não é um tipo primitivo, para qual tipo tentar convertê-la?

```
c#
bool IsValidPercentage(object x) => x is >= 0 and <= 100;
```

Sugerimos que, quando o tipo de entrada já for um primitivo comparável, seja o tipo de comparação. No entanto, quando a entrada não é um primitivo comparável, tratamos o relacional como incluindo um teste de tipo implícito para o tipo da constante no lado direito da relacional. Se o programador pretende dar suporte a mais de um tipo de entrada, isso deve ser feito explicitamente:

```
C#  
  
bool IsValidPercentage(object x) => x is  
    >= 0 and <= 100 or      // integer tests  
    >= 0F and <= 100F or    // float tests  
    >= 0D and <= 100D;      // double tests
```

## Informações de tipo de fluxo da esquerda para a direita de `and`

Foi sugerido que, ao gravar um `and` combinador, as informações de tipo aprendidas à esquerda sobre o tipo de nível superior poderiam fluir para a direita. Por exemplo,

```
C#  
  
bool isSmallByte(object o) => o is byte and < 100;
```

Aqui, o *tipo de entrada* para o segundo padrão é limitado pelos requisitos de *restrição de tipo* da esquerda do `and`. Definimos a semântica de restrição de tipo para todos os padrões da seguinte maneira. O *tipo limitado* de um padrão `P` é definido da seguinte maneira:

1. Se `P` for um padrão de tipo, o *tipo limitado* será o tipo do tipo do padrão de tipo.
2. Se `P` for um padrão de declaração, o *tipo limitado* será o tipo do tipo de padrão de declaração.
3. Se `P` é um padrão recursivo que fornece um tipo explícito, o *tipo limitado* é aquele tipo.
4. Se `P` for correspondido por meio das regras para `ITuple`, o *tipo limitado* será o tipo `System.Runtime.CompilerServices.ITuple`.
5. Se `P` é um padrão constante em que a constante não é a constante nula e onde a expressão não tem nenhuma *conversão de expressão constante* para o *tipo de entrada*, o *tipo limitado* é o tipo da constante.
6. Se `P` é um padrão relacional em que a expressão constante não tem *conversão de expressão constante* para o *tipo de entrada*, o *tipo limitado* é o tipo da constante.

7. Se `P` for um `or` padrão, o *tipo limitado* será o tipo comum do *tipo restrito* dos subpadrões se existir um tipo comum. Para essa finalidade, o algoritmo de tipo comum considera apenas a identidade, a conversão boxing e as conversões de referência implícitas e considera todos os subpadrões de uma seqüência de `or` padrões (ignorando padrões entre parênteses).
8. Se `P` for um `and` padrão, o *tipo limitado* será o *tipo limitado* do padrão correto. Além disso, o *tipo limitado* do padrão esquerdo é o *tipo de entrada* do padrão correto.
9. Caso contrário, o *tipo limitado* de `P` é o *tipo de P* entrada.

## Definições de variáveis e atribuição definitiva

A adição de `or` `not` padrões cria alguns novos problemas interessantes em relação às variáveis de padrão e à atribuição definitiva. Como as variáveis normalmente podem ser declaradas no máximo uma vez, parece que qualquer variável de padrão declarada em um lado de um `or` padrão não seria definitivamente atribuída quando o padrão corresponde. Da mesma forma, uma variável declarada dentro de um `not` padrão não deve ser atribuída definitivamente quando o padrão for correspondente. A maneira mais simples de resolver isso é proibir a declaração de variáveis de padrão nesses contextos. No entanto, isso pode ser muito restritivo. Há outras abordagens a serem consideradas.

Um cenário que vale a pena considerar é isso

```
C#  
  
if (e is not int i) return;  
M(i); // is i definitely assigned here?
```

Isso não funciona hoje porque, para uma *expressão is-Pattern*, as variáveis de padrão são consideradas *definitivamente atribuídas* somente onde a *expressão is-Pattern-padrão* é true ("definitivamente atribuída quando true").

Dar suporte a isso seria mais simples (da perspectiva do programador) do que também adicionar suporte para uma instrução de condição negada `if`. Mesmo que tenhamos adicionado esse suporte, os programadores se perguntariam por que o trecho acima não funciona. Por outro lado, o mesmo cenário de um `switch` faz menos sentido, pois não há nenhum ponto correspondente no programa onde *definitivamente atribuído quando falso* seria significativo. Permitimos isso em uma *expressão is-Pattern*, mas não em outros contextos onde os padrões são permitidos? Parece irregular.

Relacionado a esse problema é a atribuição definitiva em um *disjunctive-Pattern*.

C#

```
if (e is 0 or int i)
{
    M(i); // is i definitely assigned here?
}
```

Só esperamos `i` ser atribuído definitivamente quando a entrada não for zero. Mas como não sabemos se a entrada é zero ou não dentro do bloco, `i` não é definitivamente atribuída. No entanto, e se permitirmos `i` ser declarados em diferentes padrões mutuamente exclusivos?

C#

```
if ((e1, e2) is (0, int i) or (int i, 0))
{
    M(i);
}
```

Aqui, a variável `i` é definitivamente atribuída dentro do bloco e usa o valor do outro elemento da tupla quando um elemento zero é encontrado.

Também foi recomendável permitir que as variáveis sejam (multiplique) definidas em cada caso de um bloco de caso:

C#

```
case (0, int x):
case (int x, 0):
    Console.WriteLine(x);
```

Para fazer qualquer um desses trabalhos, teríamos que definir cuidadosamente onde essas várias definições são permitidas e sob quais condições essa variável é considerada definitivamente atribuída.

Devemos optar por adiar tal trabalho até mais tarde (o que eu recomendo), poderíamos dizer em C# 9

- abaixo de uma `not` ou `or`, as variáveis de padrão não podem ser declaradas.

Em seguida, teríamos tempo para desenvolver alguma experiência que fornecesse informações sobre o possível valor de relaxar mais tarde.

## Diagnóstico, subtomada e exaustiva

Esses novos formulários de padrão apresentam muitas novas oportunidades de erro de programador diagnosticado. Precisaremos decidir quais tipos de erros serão diagnosticados e como fazer isso. Estes são alguns exemplos:

```
C#
```

```
case >= 0 and <= 100D:
```

Esse caso nunca pode corresponder (porque a entrada não pode ser um `int` e um `double`). Já temos um erro quando detectamos um caso que nunca pode corresponder, mas suas palavras ("o caso de alternância já foi manipulado por um caso anterior" e "o padrão já foi manipulado por um braço anterior da expressão do comutador") pode ser enganoso em novos cenários. Talvez seja necessário modificar as palavras para apenas dizer que o padrão nunca corresponderá à entrada.

```
C#
```

```
case 1 and 2:
```

Da mesma forma, isso seria um erro porque um valor não pode ser `1` e `2`.

```
C#
```

```
case 1 or 2 or 3 or 1:
```

Esse caso é possível fazer a correspondência, mas o `or 1` no final não adiciona nenhum significado ao padrão. Sugiro que devemos criar um erro sempre que algum conjunção ou disjunct de um padrão composto não definir uma variável de padrão ou afetar o conjunto de valores correspondentes.

```
C#
```

```
case < 2: break;
case 0 or 1 or 2 or 3 or 4 or 5: break;
```

Aqui, não `0 or 1 or` adiciona nada ao segundo caso, pois esses valores teriam sido tratados pelo primeiro caso. Isso merece um erro.

```
C#
```

```
byte b = ...;
int x = b switch { <100 => 0, 100 => 1, 101 => 2, >101 => 3 };
```

Uma expressão de comutador como essa deve ser considerada *exaustiva* (ela lida com todos os valores de entrada possíveis).

No C# 8.0, uma expressão de switch com uma entrada de tipo `byte` é considerada exaustiva apenas se contiver um braço final cujo padrão corresponde a tudo (um padrão de *descarte* ou de *var*). Até mesmo uma expressão de switch que tem um ARM para cada `byte` valor distinto não é considerada exaustiva no C# 8. Para lidar corretamente com a exaustividade dos padrões relacionais, também teremos que lidar com esse caso. Tecnicamente, isso será uma alteração significativa, mas nenhum usuário é provavelmente notado.

# Somente setters init

Artigo • 16/09/2021

## Resumo

Essa proposta adiciona o conceito de propriedades e indexadores somente de init ao C#. Essas propriedades e indexadores podem ser definidos no ponto da criação do objeto, mas são efetivamente `get` apenas quando a criação do objeto é concluída. Isso permite um modelo imutável muito mais flexível em C#.

## Motivação

Os mecanismos subjacentes para a criação de dados imutáveis em C# não foram alterados desde 1.0. Eles permanecem:

1. Declarando campos como `readonly`.
2. Declarando propriedades que contêm apenas um `get` acessador.

Esses mecanismos são eficazes em permitir a construção de dados imutáveis, mas eles fazem isso adicionando custo ao código clichê de tipos e optando por tais tipos de recursos, como inicializadores de objeto e de coleção. Isso significa que os desenvolvedores devem escolher entre facilidade de uso e imutabilidade.

Um objeto imutável simples, como o `Point` exige duas vezes o código de chapa para dar suporte à construção, como faz para declarar o tipo. Quanto maior o tipo, maior o custo desta placa:

C#

```
struct Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }
}
```

O `init` acessador torna os objetos imutáveis mais flexíveis, permitindo que o chamador permutasse os membros durante o ato de construção. Isso significa que as propriedades imutáveis do objeto podem participar de inicializadores de objeto e, portanto, remove a necessidade de todos os textos clichês do Construtor no tipo. O `Point` tipo agora é simplesmente:

```
C#  
  
struct Point  
{  
    public int X { get; init; }  
    public int Y { get; init; }  
}
```

O consumidor pode usar inicializadores de objeto para criar o objeto

```
C#  
  
var p = new Point() { X = 42, Y = 13 };
```

## Design detalhado

### acessadores de inicialização

Uma propriedade somente `init` (ou indexador) é declarada usando o `init` acessador no lugar do `set` acessador:

```
C#  
  
class Student  
{  
    public string FirstName { get; init; }  
    public string LastName { get; init; }  
}
```

Uma propriedade de instância que contém um `init` acessador é considerada configurável nas seguintes circunstâncias, exceto quando em uma função local ou lambda:

- Durante um inicializador de objeto
- Durante um `with` inicializador de expressão

- Dentro de um construtor de instância do tipo contido ou derivado, em `this` ou `base`
- Dentro do `init` acessador de qualquer propriedade, em `this` ou `base`
- Dentro de usos de atributo com parâmetros nomeados

Os tempos acima nos quais os `init` acessadores são settable são mencionados coletivamente neste documento como a fase de construção do objeto.

Isso significa que a `Student` classe pode ser usada das seguintes maneiras:

C#

```
var s = new Student()
{
    FirstName = "Jared",
    LastName = "Parosns",
};
s.LastName = "Parsons"; // Error: LastName is not settable
```

As regras em relação ao quando os `init` acessadores são transtendem a extensão entre hierarquias de tipo. Se o membro estiver acessível e o objeto estiver na fase de construção, o membro será configurável. Isso permite especificamente o seguinte:

C#

```
class Base
{
    public bool Value { get; init; }
}

class Derived : Base
{
    Derived()
    {
        // Not allowed with get only properties but allowed with init
        Value = true;
    }
}

class Consumption
{
    void Example()
    {
        var d = new Derived() { Value = true };
    }
}
```

No ponto em que um `init` acessador é invocado, a instância é conhecida por estar na fase de construção aberta. Portanto, um `init` acessador tem permissão para executar as seguintes ações além do que um `set` acessador normal pode fazer:

1. Chamar outros `init` acessadores disponíveis por meio `this` de ou `base`
2. Atribuir `readonly` campos declarados no mesmo tipo por meio de `this`

C#

```
class Complex
{
    readonly int Field1;
    int Field2;
    int Prop1 { get; init; }
    int Prop2
    {
        get => 42;
        init
        {
            Field1 = 13; // okay
            Field2 = 13; // okay
            Prop1 = 13; // okay
        }
    }
}
```

A capacidade de atribuir `readonly` campos de um `init` acessador é limitada a esses campos declarados no mesmo tipo que o acessador. Ele não pode ser usado para atribuir `readonly` campos em um tipo base. Essa regra garante que os autores de tipo permaneçam no controle sobre o comportamento de imutabilidade de seu tipo. Os desenvolvedores que não querem utilizar `init` não podem ser afetados de outros tipos, optando por fazer isso:

C#

```
class Base
{
    internal readonly int Field;
    internal int Property
    {
        get => Field;
        init => Field = value; // Okay
    }

    internal int OtherProperty { get; init; }
}

class Derived : Base
{
```

```

internal readonly int DerivedField;
internal int DerivedProperty
{
    get => DerivedField;
    init
    {
        DerivedField = 42; // Okay
        Property = 0; // Okay
        Field = 13; // Error Field is readonly
    }
}

public Derived()
{
    Property = 42; // Okay
    Field = 13; // Error Field is readonly
}
}

```

Quando `init` é usado em uma propriedade virtual, todas as substituições também devem ser marcadas como `init`. Da mesma forma, não é possível substituir um simples `set` por `init`.

C#

```

class Base
{
    public virtual int Property { get; init; }
}

class C1 : Base
{
    public override int Property { get; init; }
}

class C2 : Base
{
    // Error: Property must have init to override Base.Property
    public override int Property { get; set; }
}

```

Uma `interface` declaração também pode participar da `init` inicialização de estilo por meio do seguinte padrão:

C#

```

interface IPerson
{
    string Name { get; init; }
}

```

```

class Init
{
    void M<T>() where T : IPerson, new()
    {
        var local = new T()
        {
            Name = "Jared"
        };
        local.Name = "Jraed"; // Error
    }
}

```

Restrições deste recurso:

- O `init` acessador só pode ser usado em Propriedades de instância
- Uma propriedade não pode conter um `init` `set` acessador e
- Todas as substituições de uma propriedade devem ter `init` se a base tivesse `init`. Essa regra também se aplica à implementação de interface.

## Structs ReadOnly

`init` os acessadores (acessadores implementados automaticamente e acessadores implementados manualmente) são permitidos nas propriedades de `readonly struct`s, bem como `readonly` Propriedades. `init` os acessadores não têm permissão para serem marcados `readonly` em si, em `readonly` e em não `readonly` `struct` tipos.

C#

```

readonly struct ReadonlyStruct1
{
    public int Prop1 { get; init; } // Allowed
}

struct ReadonlyStruct2
{
    public readonly int Prop2 { get; init; } // Allowed

    public int Prop3 { get; readonly init; } // Error
}

```

## Codificação de metadados

Acessadores `init` de propriedade serão emitidos como um `set` acessador padrão com o tipo de retorno marcado com um modreq de `IsExternalInit`. Esse é um novo tipo

que terá a seguinte definição:

```
C#  
  
namespace System.Runtime.CompilerServices  
{  
    public sealed class IsExternalInit  
    {  
    }  
}
```

O compilador corresponderá ao tipo por nome completo. Não há nenhum requisito de que ele apareça na biblioteca principal. Se houver vários tipos por esse nome, o compilador vinculará a quebra na seguinte ordem:

1. Aquele definido no projeto que está sendo compilado
2. Aquele definido em corelib

Se nenhuma delas existir, um erro de ambiguidade de tipo será emitido.

O design para o `IsExternalInit` é mais abordado neste [problema ↗](#)

## Perguntas

### Alterações de quebra

Um dos principais pontos dinâmicos de como esse recurso é codificado será a seguinte pergunta:

É uma alteração de quebra binária para substituir `init` por `set` ?

Substituir `init` por `set` e, portanto, tornar uma propriedade totalmente gravável nunca é uma alteração significativa de origem em uma propriedade não virtual. Ele simplesmente expande o conjunto de cenários em que a propriedade pode ser gravada. O único comportamento em questão é se isso permanece ou não uma alteração de quebra binária.

Se quisermos fazer a alteração de `init` para `set` uma alteração compatível com origem e binário, ela forçará a nossa mão na decisão de modreq vs. Attributes abaixo porque ela impedirá a modreqs como uma solução. Se, por outro lado, isso for visto como não interessante, isso fará com que a decisão de modreq versus atributo seja menos afetada.

**Resolução** Esse cenário não é visto como convincente pelo LDM.

## Modreqs vs. atributos

A estratégia de emissão para `init` acessadores de propriedade deve escolher entre usar atributos ou modreqs ao emitir durante os metadados. Eles têm diferentes compensações que precisam ser consideradas.

Anotar um acessador de conjunto de propriedades com uma declaração modreq significa que os compiladores compatíveis com a CLI ignorarão o acessador, a menos que entenda o modreq. Isso significa que somente os compiladores cientes do `init` vão ler o membro. Os compiladores `init` que não sabem do irão ignorar o `set` acessador e, portanto, não tratarão accidentalmente a propriedade como leitura/gravação.

A desvantagem de modreq se `init` torna uma parte da assinatura binária do `set` acessador. Adicionar ou remover `init` interromperá o compatibility binário do aplicativo.

Usar atributos para anotar o `set` acessador significa que somente os compiladores que entendem o atributo saberão limitar o acesso a ele. Um compilador não está ciente de que o `init` verá como uma propriedade de leitura/gravação simples e permite o acesso.

Isso aparentemente significaria que essa decisão é uma opção entre a segurança extra às custas da compatibilidade binária. Se aprofundando em um pouco, a segurança extra não é exatamente o que parece. Ele não se protegerá das seguintes circunstâncias:

1. Reflexão sobre `public` Membros
2. O uso de `dynamic`
3. Compiladores que não reconhecem modreqs

Também deve ser considerado que, quando concluirmos as regras de verificação de IL para o .NET 5, `init` será uma dessas regras. Isso significa que a imposição extra será obtida da simples verificação de compiladores que emitirão IL verificável.

Os principais idiomas do .NET (C#, F# e VB) serão atualizados para reconhecer esses `init` acessadores. Portanto, o único cenário realista é quando um compilador do C# 9 emite `init` Propriedades e elas são vistas por um conjunto de ferramentas mais antigo, como C# 8, VB 15, etc... C# 8. Esse é o compensador a considerar e avaliar a compatibilidade binária.

**Observação** Essa discussão se aplica principalmente somente a membros, não a campos. Enquanto `init` os campos foram rejeitados pelo LDM, eles ainda são interessantes para considerar a discussão sobre modreq versus atributo. O `init` recurso para campos é um relaxamento da restrição existente do `readonly`. Isso significa que,

se emitirmos os campos como `readonly` + um atributo, não haverá risco de os compiladores mais antigos desaparecerem usando o campo porque eles já reconheceram `readonly`. Portanto, usar um modreq aqui não adiciona nenhuma proteção extra.

**Resolução** O recurso usará um modreq para codificar o setter da propriedade `init`. Os fatores convincentes eram (sem uma ordem específica):

- Desejo desencorajar os compiladores mais antigos de violar a `init` semântica
- Desejo de fazer a adição ou remoção `init` em uma `virtual` declaração ou `interface` uma alteração de quebra de fonte e binária.

Considerando que não havia nenhum suporte significativo para `init` a remoção para ser uma alteração compatível com binário, ela fez a opção de usar modreq straightforward.

## init vs. `InitOnly`

Havia três formas de sintaxe que tiveram uma consideração significativa durante nossa reunião LDM:

```
C#  
  
// 1. Use init  
int Option1 { get; init; }  
// 2. Use init set  
int Option2 { get; init set; }  
// 3. Use initonly  
int Option3 { get; initonly; }
```

**Resolução** Não havia nenhuma sintaxe que fosse intensamente favorecida no LDM.

Um ponto que tem uma atenção significativa era como a escolha da sintaxe afetaria nossa capacidade de fazer `init` Membros como um recurso geral no futuro. Escolher a opção 1 significaria que seria difícil definir uma propriedade que tinha um `init` `get` método de estilo no futuro. Eventualmente, foi decidido que, se decidirmos avançar com membros gerais `init` no futuro, poderíamos permitir que `init` fosse um modificador na lista de acessadores de propriedade, bem como um pouco curto para o `init set`. Essencialmente, as duas declarações a seguir seriam idênticas.

```
C#  
  
int Property1 { get; init; }  
int Property1 { get; init set; }
```

A decisão foi feita para avançar com `init` como um acessador autônomo na lista de acessadores de propriedade.

## Avisar sobre falha na inicialização

Considere este cenário. Um tipo declara um `init` único membro que não está definido no construtor. O código que constrói o objeto receberá um aviso se ele não conseguir inicializar o valor?

Nesse ponto, fica claro que o campo nunca será definido e, portanto, tem muitas semelhanças com o aviso sobre falha na inicialização dos `private` dados. Portanto, um aviso aparentemente teria algum valor aqui?

No entanto, há desvantagens significativas para esse aviso:

1. Isso complica a história de compatibilidade de alteração `readonly` para o `init`.
2. Ele requer a realização de metadados adicionais para indicar os membros que precisam ser inicializados pelo chamador.

Mais detalhes se acreditarmos que há um valor aqui no cenário geral de forçar os criadores de objetos a serem avisados/erros sobre campos específicos, isso provavelmente faz sentido como um recurso geral. Não há motivo para ele se limitar apenas a `init` Membros.

**Resolução** Não haverá nenhum aviso sobre o consumo de `init` campos e propriedades.

O LDM quer ter uma discussão mais ampla sobre a ideia de campos e propriedades obrigatórios. Isso pode fazer com que possamos voltar e reconsiderar nossa posição em `init` Membros e validação.

## Permitir `init` como um modificador de campo

Da mesma forma `init` pode servir como um acessador de propriedade, ele também pode servir como uma designação em campos para dar a eles comportamentos semelhantes como `init` Propriedades. Isso permitiria que o campo fosse atribuído antes que a construção fosse concluída pelo tipo, tipos derivados ou inicializadores de objeto.

```

class Student
{
    public init string FirstName;
    public init string LastName;
}

var s = new Student()
{
    FirstName = "Jarde",
    LastName = "Parsons",
}

s.FirstName = "Jared"; // Error FirstName is readonly

```

Em metadados, esses campos seriam marcados da mesma maneira que os `readonly` campos, mas com um atributo adicional ou modreq para indicar que são `init` campos de estilo.

**Resolução** O LDM concorda que esta proposta é um som, mas geral o cenário parecia não ser separado das propriedades. A decisão era continuar apenas com `init` as propriedades por enquanto. Isso tem um nível adequado de flexibilidade, pois uma `init` propriedade pode mutar um `readonly` campo no tipo declarativo da propriedade. Isso será reconsiderado se houver comentários significativos do cliente que justificam o cenário.

## Permitir `init` como um modificador de tipo

Da mesma forma que o `readonly` modificador pode ser aplicado a um `struct` para declarar automaticamente todos os campos como `readonly`, o `init` único modificador pode ser declarado em um `struct` ou `class` para marcar automaticamente todos os campos como `init`. Isso significa que as duas declarações de tipo a seguir são equivalentes:

C#

```

struct Point
{
    public init int X;
    public init int Y;
}

// vs.

init struct Point
{
    public int X;
}

```

```
    public int Y;  
}
```

**Resolução** Este recurso é muito *graciosos* aqui e está em conflito com o `readonly struct` recurso no qual ele se baseia. O `readonly struct` recurso é simples no que se aplica `readonly` a todos os membros: campos, métodos, etc... O `init struct` recurso só se aplicaria a propriedades. Isso realmente acaba tornando confuso para os usuários.

Dado que `init` só é válido em determinados aspectos de um tipo, rejeitamos a ideia de tê-lo como um modificador de tipo.

## Considerações

### Compatibilidade

O `init` recurso foi projetado para ser compatível com `get` as propriedades somente existentes. Especificamente, ela é destinada a ser uma alteração completamente aditiva para uma propriedade que é `get` apenas hoje, mas deseja mais semântica de criação de objeto flexível.

Por exemplo, considere o seguinte tipo:

```
C#  
  
class Name  
{  
    public string First { get; }  
    public string Last { get; }  
  
    public Name(string first, string last)  
    {  
        First = first;  
        Last = last;  
    }  
}
```

`init` A adição a essas propriedades é uma alteração não significativa:

```
C#  
  
class Name  
{  
    public string First { get; init; }  
    public string Last { get; init; }
```

```
public Name(string first, string last)
{
    First = first;
    Last = last;
}
```

## Verificação de IL

Quando o .NET Core decidir reimplementar a verificação de IL, as regras precisarão ser ajustadas para a conta dos `init` Membros. Isso precisará ser incluído nas alterações de regra para acesso não mutado aos `readonly` dados.

As regras de verificação de IL precisarão ser divididas em duas partes:

1. Permitindo que `init` os Membros definam um `readonly` campo.
2. Determinando quando um `init` membro pode ser legalmente chamado.

O primeiro é um ajuste simples para as regras existentes. O verificador de IL pode ser ensinado a reconhecer `init` Membros e, a partir daí, só precisa considerar um `readonly` campo para ser configurável nesse `this` membro.

A segunda regra é mais complicada. No caso simples de inicializadores de objeto, a regra é direta. Deve ser legal chamar `init` Membros quando o resultado de uma `new` expressão ainda estiver na pilha. Ou seja, até que o valor tenha sido armazenado em um campo ou elemento de matriz local ou passado como um argumento para outro método, ainda será legal chamar `init` Membros. Isso garante que, uma vez que o resultado da `new` expressão seja publicado em um identificador nomeado (diferente de `this`), não será mais legal chamar `init` Membros.

No entanto, o caso mais complicado é quando misturamos `init` Membros, inicializadores de objeto e `await`. Isso pode fazer com que o objeto recém-criado seja temporariamente dividido em um computador de estado e, portanto, colocado em um campo.

C#

```
var student = new Student()
{
    Name = await SomeMethod()
};
```

Aqui, o resultado de `new Student()` será hossed em um computador de estado como um campo antes do conjunto de `Name` ocorrerem. O compilador precisará marcar tais campos de guindaste de forma que o verificador de IL entenda que não estão acessíveis ao usuário e, portanto, não viola a semântica pretendida de `init`.

## Membros de `init`

O `init` modificador pode ser estendido para ser aplicado a todos os membros da instância. Isso generalizaria o conceito de `init` durante a construção do objeto e permitiria que os tipos declarassem métodos auxiliares que poderiam partipate no processo de construção para inicializar `init` campos e propriedades.

Esses membros teriam todos os restricions que um `init` acessador faz nesse design. No entanto, a necessidade é questionável, e isso pode ser adicionado com segurança em uma versão futura do idioma de maneira compatível.

## Gerar três acessadores

Uma implementação potencial das `init` Propriedades é deixar `init` completamente separada do `set`. Isso significa que uma propriedade pode potencialmente ter três acessadores diferentes: `get`, `set` e `init`.

Isso tem a possível vantagem de permitir o uso de modreq para impor a exatidão enquanto mantém a compatibilidade binária. A implementação seria basicamente o seguinte:

1. Um `init` acessador sempre será emitido se houver um `set`. Quando não definido pelo desenvolvedor, ele é simplesmente uma referência a `set`.
2. O conjunto de uma propriedade em um inicializador de objeto sempre será usado `init` se estiver presente, mas retorne para `set` se ele estiver ausente.

Isso significa que um desenvolvedor sempre pode excluir `init` de uma propriedade com segurança.

A desvantagem desse design é que só será útil se `init` o for **sempre** emitido quando houver um `set`. A linguagem não pode saber se `init` foi excluída no passado, ela precisa pressupor que foi e, portanto, a `init` sempre deve ser emitida. Isso causaria uma expansão de metadados significativa e simplesmente não vale o custo da compatibilidade aqui.

# Expressões com tipo de destino new

Artigo • 16/09/2021

- [x] proposta
- [x] protótipo
- [] Implementação
- [] Especificação

## Resumo

Não exigir especificação de tipo para construtores quando o tipo é conhecido.

## Motivação

Permitir inicialização de campo sem duplicar o tipo.

C#

```
Dictionary<string, List<int>> field = new() {  
    { "item1", new() { 1, 2, 3 } }  
};
```

Permitir a omissão do tipo quando ele puder ser inferido do uso.

C#

```
XmlReader.Create(reader, new() { IgnoreWhitespace = true });
```

Crie uma instância de um objeto sem soletrar o tipo.

C#

```
private readonly static object s_syncObj = new();
```

## Especificação

Um novo formulário sintático, *target\_typed\_new* do *object\_creation\_expression* é aceito, no qual o *tipo* é opcional.

antlr

```

object_creation_expression
: 'new' type '(' argument_list? ')' object_or_collection_initializer?
| 'new' type object_or_collection_initializer
| target_typed_new
;
target_typed_new
: 'new' '(' argument_list? ')' object_or_collection_initializer?
;

```

Uma expressão de *target\_typed\_new* não tem um tipo. No entanto, há uma nova conversão de criação de objeto que é uma conversão implícita de expressão, que existe de um *target\_typed\_new* a cada tipo.

Dado um tipo de destino  $T$ , o tipo  $T_0$  é o  $T$  tipo subjacente se  $T$  for uma instância do `System.Nullable`. Caso contrário,  $T_0$  é  $T$ . O significado de uma *target\_typed\_new* expressão que é convertida no tipo  $T$  é igual ao significado de um *object\_creation\_expression* correspondente que especifica  $T_0$  como o tipo.

É um erro de tempo de compilação se um *target\_typed\_new* for usado como um operando de um operador unário ou binário, ou se for usado onde ele não está sujeito a uma conversão de criação de objeto.

**Problema aberto:** devemos permitir delegações e tuplas como o tipo de destino?

As regras acima incluem delegados (um tipo de referência) e tuplas (um tipo de struct). Embora ambos os tipos sejam constructible, se o tipo for inferência, uma função anônima ou um literal de tupla já poderá ser usado.

C#

```

(int a, int b) t = new(1, 2); // "new" is redundant
Action a = new(() => {}); // "new" is redundant

(int a, int b) t = new(); // OK; same as (0, 0)
Action a = new(); // no constructor found

```

## Diversos

As seguintes são as consequências da especificação:

- `throw new()` é permitido (o tipo de destino é `System.Exception`)
- O tipo de destino `new` não é permitido com operadores binários.

- Não é permitido quando não há tipo para destino: operadores unários, coleção de um `foreach`, em um, em uma `using` declaração, em uma `await` expressão, como uma propriedade de tipo anônimo (), em uma instrução, em um `new { Prop = new() }` `lock sizeof`, em uma `fixed` instrução, em um acesso de membro (`new().field`), em uma operação expedida dinamicamente (`someDynamic.Method(new())`), em uma consulta LINQ, como o operando do `is` operador, como o operando esquerdo do `??` operador,...
- Ele também não é permitido como um `ref`.
- Os tipos de tipos a seguir não são permitidos como destinos da conversão
  - **Tipos de enumeração:** `new()` funcionará (como `new Enum()` funciona para fornecer o valor padrão), mas `new(1)` não funcionará, pois os tipos de enumeração não têm um construtor.
  - **Tipos de interface:** Isso funcionaria da mesma forma que a expressão de criação correspondente para tipos COM.
  - **Tipos de matriz:** as matrizes precisam de uma sintaxe especial para fornecer o comprimento.
  - **dinâmico:** não permitimos `new dynamic()`, portanto, não permitimos `new()` with `dynamic` como um tipo de destino.
  - **tuplas:** Eles têm o mesmo significado de uma criação de objeto usando o tipo subjacente.
  - Todos os outros tipos que não são permitidos no `object_creation_expression` também são excluídos, por exemplo, tipos de ponteiro.

## Desvantagens

Houve algumas preocupações com o tipo de destino `new` criando novas categorias de alterações significativas, mas já temos isso com `null` e `default`, e isso não foi um problema significativo.

## Alternativas

A maioria das reclamações sobre os tipos muito longos para duplicar na inicialização de campo é sobre os *argumentos de tipo*, não o próprio tipo, poderíamos inferir apenas argumentos de tipo como `new Dictionary(...)` (ou semelhantes) e inferir argumentos de tipo localmente a partir de argumentos ou o inicializador de coleção.

## Perguntas

- Devemos proibir usos em árvores de expressão? foi
- Como o recurso interage com `dynamic` argumentos? (sem tratamento especial)
- Como o IntelliSense deve funcionar `new()`? (somente quando há um único tipo de destino)

## Criar reuniões

- [LDM-2017-10-18 ↗](#)
- [LDM-2018-05-21 ↗](#)
- [LDM-2018-06-25 ↗](#)
- [LDM-2018-08-22 ↗](#)
- [LDM-2018-10-17 ↗](#)
- [LDM-2020-03-25 ↗](#)

# Inicializadores de módulo

Artigo • 16/09/2021

- [x] proposta
- [] Protótipo: [em andamento ↗](#)
- [] Implementação: em andamento
- [] Especificação: [não iniciada](#)

## Resumo

Embora a plataforma .NET tenha um [recurso ↗](#) que dá suporte diretamente à gravação do código de inicialização para o assembly (teoricamente, o módulo), ele não é exposto em C#. Esse é um cenário bastante nicho, mas quando você se depara com ele, as soluções parecem ser bem problemáticas. Há relatórios de [vários clientes ↗](#) (dentro e fora da Microsoft) que se esforçam com o problema e não há dúvida em casos mais não documentados.

## Motivação

- 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 `static` abordagens do Construtor 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

## Design detalhado

Um método pode ser designado como um inicializador de módulo decorando-o com um `[ModuleInitializer]` atributo.

C#

```
using System;
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
```

```
public sealed class ModuleInitializerAttribute : Attribute { }
```

O atributo pode ser usado da seguinte maneira:

C#

```
using System.Runtime.CompilerServices;
class C
{
    [ModuleInitializer]
    internal static void M1()
    {
        // ...
    }
}
```

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

1. O método deve ser `static`.
2. O método deve ser sem parâmetros.
3. O método deve retornar `void`.
4. O método não deve ser genérico ou estar contido em um tipo genérico.
5. 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.

## Desvantagens

Por que *não* fazemos isso?

- Talvez as ferramentas de terceiros existentes para inicializadores de módulo "injetando" sejam suficientes para os usuários que já solicitaram esse recurso.

## Criar reuniões

8 de abril de 2020 ↗

# Estendendo métodos parciais

Artigo • 16/09/2021

## Resumo

Essa proposta visa remover todas as restrições em relação às assinaturas de `partial` métodos em C#. O objetivo é expandir o conjunto de cenários nos quais esses métodos possam trabalhar com geradores de origem, bem como um formulário de declaração mais geral para métodos C#.

Consulte também a [especificação de métodos parciais originais](#).

## Motivação

O C# tem suporte limitado para desenvolvedores que dividem métodos em declarações e definições/implementações.

```
C#  
  
partial class C  
{  
    // The declaration of C.M  
    partial void M(string message);  
}  
  
partial class C  
{  
    // The definition of C.M  
    partial void M(string message) => Console.WriteLine(message);  
}
```

Um comportamento dos `partial` métodos é que, quando a definição estiver ausente, a linguagem simplesmente apagará todas as chamadas para o `partial` método.

Essencialmente, ele se comporta como uma chamada para um `[Conditional]` método em que a condição foi avaliada como false.

```
C#  
  
partial class D  
{  
    partial void M(string message);  
  
    void Example()  
    {
```

```
    M(GetIt()); // Call to M and GetIt erased at compile time
}

string GetIt() => "Hello World";
}
```

A motivação original para esse recurso era a geração de origem na forma de código gerado pelo designer. Os usuários estavam constantemente editando o código gerado porque desejavam vincular algum aspecto do código gerado. As partes mais notáveis do processo de inicialização Windows Forms, depois que os componentes foram inicializados.

A edição do código gerado foi propenso a erros porque qualquer ação que fez com que o designer regenerasse o código faria com que a edição do usuário fosse apagada. O `partial` método de funcionalidade facilitou essa tensão porque permitia que os designers emitissem ganchos na forma de `partial` métodos.

Os designers podem emitir ganchos como `partial void OnComponentInit()` e os desenvolvedores poderiam definir declarações para eles ou não defini-los. Em ambos os casos, embora o código gerado seja compilado e os desenvolvedores que estavam interessados no processo pudessem se conectar conforme necessário.

Isso significa que os métodos parciais têm várias restrições:

1. Deve ter um `void` tipo de retorno.
2. Não é possível ter `out` parâmetros.
3. Não é possível ter nenhuma acessibilidade (implicitamente `private`).

Essas restrições existem porque o idioma deve ser capaz de emitir código quando o site de chamada é apagado. Dado que eles podem ser apagados `private` é a única acessibilidade possível, pois o membro não pode ser exposto nos metadados do assembly. Essas restrições também servem para limitar o conjunto de cenários nos quais os `partial` métodos podem ser aplicados.

A proposta aqui é remover todas as restrições existentes em relação a `partial` métodos. Essencialmente, permita que eles tenham `out` tipos de retorno não nulos ou qualquer tipo de acessibilidade. Essas `partial` declarações teriam o requisito adicionado de que uma definição deve existir. Isso significa que o idioma não precisa considerar o impacto de apagar os sites de chamada.

Isso expandiria o conjunto de cenários de gerador nos quais os `partial` métodos poderiam participar e, portanto, vinculo perfeitamente com nosso recurso de geradores de origem. Por exemplo, um Regex poderia ser definido usando o seguinte padrão:

C#

```
[RegexGenerated("(dog|cat|fish)")]
partial bool IsPetMatch(string input);
```

Isso dá ao desenvolvedor um modo declarativo simples de se optar por geradores, além de fornecer aos geradores um conjunto muito fácil de declarações a serem examinadas no código-fonte para orientar a saída gerada.

Compare isso com a dificuldade de que um gerador tenha conectado o trecho de código a seguir.

C#

```
var regex = new RegularExpression("(dog|cat|fish)");
if (regex.IsMatch(someInput))
{}
```

Considerando que o compilador não permite que os geradores modifiquem o código, a conexão desse padrão seria praticamente impossível para os geradores. Eles precisariam recorrer à reflexão na `IsMatch` implementação ou pedir que os usuários alterem seus sites de chamada para um novo método + Refactor o Regex para passar o literal da cadeia de caracteres como um argumento. É bem confuso.

## Design detalhado

O idioma será alterado para permitir que os `partial` métodos sejam anotados com um modificador de acessibilidade explícito. Isso significa que eles podem ser rotulados como `private` , `public` etc...

Quando um `partial` método tem um modificador de acessibilidade explícito, embora a linguagem exija que a declaração tenha uma definição correspondente mesmo quando a acessibilidade é `private` :

C#

```
partial class C
{
    // Okay because no definition is required here
    partial void M1();

    // Okay because M2 has a definition
```

```

    private partial void M2();

    // Error: partial method M3 must have a definition
    private partial void M3();
}

partial class C
{
    private partial void M2() { }
}

```

Além disso, a linguagem removerá todas as restrições sobre o que pode aparecer em um `partial` método que tem uma acessibilidade explícita. Essas declarações podem conter tipos de retorno não nulos, `out` parâmetros, `extern` modificadores, etc... Essas assinaturas terão o expressividade completo da linguagem C#.

```

C#

partial class D
{
    // Okay
    internal partial bool TryParse(string s, out int i);
}

partial class D
{
    internal partial bool TryParse(string s, out int i) { }
}

```

Isso permite explicitamente que os `partial` métodos participem `overrides` e `interface` implementações:

```

C#

interface IStudent
{
    string GetName();
}

partial class C : IStudent
{
    public virtual partial string GetName();
}

partial class C
{
    public virtual partial string GetName() => "Jarde";
}

```

O compilador irá alterar o erro emitido quando um `partial` método contiver um elemento ilegal para, essencialmente, dizer:

Não é possível usar `ref` em um `partial` método que não possui acessibilidade explícita

Isso ajudará a indicar os desenvolvedores na direção certa ao usar esse recurso.

Restrições:

- `partial` declarações com acessibilidade explícita devem ter uma definição
- `partial` as declarações e as assinaturas de definição devem corresponder a todos os modificadores de método e parâmetro. Os únicos aspectos que podem ser diferentes são nomes de parâmetro e listas de atributos (isso não é novo, mas sim um requisito existente de `partial` métodos).

## Perguntas

### parcial em todos os membros

Considerando que estamos expandindo `partial` para ser mais amigável aos geradores de origem, também podemos expandi-lo para trabalhar em todos os membros da classe? Por exemplo, devemos ser capazes de declarar `partial` construtores, operadores, etc...

**Resolução** A ideia é soar mas, neste ponto da agenda do C# 9, estamos tentando evitar estouros de recursos desnecessários. Deseja resolver o problema imediato de expandir o recurso para trabalhar com os geradores de origem modernos.

A extensão `partial` para dar suporte a outros membros será considerada para a versão C# 10. Parece que vamos considerar essa extensão.

### Usar `abstract` em vez de `partial`

O crux dessa proposta é, essencialmente, garantir que uma declaração tenha uma definição/implementação correspondente. Considerando que devemos usar `abstract`, pois ela já é uma palavra-chave de linguagem que força o desenvolvedor a pensar em ter uma implementação?

**Resolução** Houve uma discussão íntegra sobre isso, mas, eventualmente, foi decidida. Sim, os requisitos estão familiarizados, mas os conceitos são significativamente

diferentes. Pode levar facilmente o desenvolvedor a acreditar que estava criando Slots virtuais quando eles não faziam isso.

# Funções anônimas static

Artigo • 16/09/2021

## Resumo

Permite um modificador ' static ' em lambdas e em métodos anônimos, o que não permite a captura de locais ou o estado da instância de conter escopos.

## Motivação

Evite capturar involuntariamente o estado do contexto delimitador, o que pode resultar em uma retenção inesperada de objetos capturados ou alocações adicionais inesperadas.

## Design detalhado

Um método lambda ou anônimo pode ter um `static` modificador. O `static` modificador indica que o método lambda ou anônimo é uma *função anônima estática*.

Uma *função anônima estática* não pode capturar o estado do escopo delimitador. Como resultado, locais, parâmetros e `this` do escopo de delimitação não estão disponíveis em uma *função anônima estática*.

Uma *função anônima estática* não pode referenciar membros de instância de um implícito ou explícito `this` ou de `base` referência.

Uma *função anônima estática* pode referenciar `static` membros do escopo delimitador.

Uma *função anônima estática* pode referenciar `constant` definições do escopo delimitador.

`nameof()` em uma *função anônima estática* pode referenciar locais, parâmetros ou `this` ou `base` do escopo delimitador.

As regras de acessibilidade para `private` Membros no escopo delimitador são as mesmas para `static` `static` funções não anônimas.

Nenhuma garantia é feita como se uma definição de *função anônima estática* seja emitida como um `static` método nos metadados. Isso é deixado para a implementação do compilador para otimizar.

Uma função não `static` local ou uma função anônima pode capturar o estado de uma *função anônima estática* delimitadora, mas não pode capturar o estado fora da *função anônima estática* delimitadora.

A remoção do `static` modificador de uma função anônima em um programa válido não altera o significado do programa.

# Target-Typed expressão condicional

Artigo • 16/09/2021

## Conversão de expressão condicional

Para uma expressão condicional  $c ? e_1 : e_2$ , quando

1. Não há nenhum tipo comum para  $e_1$  and  $e_2$ , or
2. para o qual existe um tipo comum, mas uma das expressões  $e_1$  ou  $e_2$  não tem conversão implícita para esse tipo

definimos uma nova *conversão de expressão condicional* implícita que permite uma conversão implícita da expressão condicional para qualquer tipo  $T$  para o qual haja uma conversão-de-expressão de  $e_1$  para  $T$  e também de  $e_2$  para  $T$ . Erro se uma expressão condicional não tiver um tipo comum entre  $e_1$  e  $e_2$  nem estiver sujeito a uma *conversão de expressão condicional*.

## Melhor conversão de expressão

Alteramos

### Melhor conversão de expressão

Dada uma conversão implícita  $c_1$  que converte de uma expressão  $E$  em um tipo  $T_1$  e uma conversão implícita  $c_2$  que converte de uma expressão  $E$  em um tipo  $T_2$   $c_1$  é uma **conversão melhor** do que  $c_2$  se não  $E$  corresponder exatamente  $T_2$  e pelo menos uma das seguintes isenções:

- $E$  correspondências exatas  $T_1$  ([expressão exatamente correspondente](#))
- $T_1$  é um destino de conversão melhor do que  $T_2$  ([melhor destino de conversão](#))

como

### Melhor conversão de expressão

Dada uma conversão implícita  $c_1$  que converte de uma expressão  $E$  em um tipo  $T_1$  e uma conversão implícita  $c_2$  que converte de uma expressão  $E$  em um tipo  $T_2$   $c_1$

é uma *conversão melhor* do que  $c_2$  se não  $E$  corresponder exatamente  $T_2$  e pelo menos uma das seguintes isenções:

- $E$  correspondências exatas  $T_1$  ([expressão exatamente correspondente](#))
- \*\* $c_1$  Não é uma *conversão de expressão condicional* e  $c_2$  é uma conversão de expressão condicional \* \* \*.
- $T_1$  é um destino de conversão melhor do que  $T_2$  ([melhor destino de conversão](#)) \* \* e  $c_1$  e  $c_2$  são *conversões de expressões condicionais* ou nenhuma conversão de expressão condicional \* \* \*.

## Expressão CAST

A especificação de linguagem C# atual diz

Uma *cast\_expression* do formulário  $(T)E$ , em que  $T$  é um *tipo* e  $E$  é um *unary\_expression*, executa uma conversão explícita ([conversões explícitas](#)) do valor de  $E$  para tipo  $T$ .

Na presença da conversão de *expressão condicional*, pode haver mais de uma conversão possível de  $E$  para  $T$ . Com a adição de *conversão de expressão condicional*, preferimos qualquer outra conversão para uma *conversão de expressão condicional* e uso a *conversão de expressão condicional* somente como último recurso.

## Observações do design

O motivo da alteração para *uma melhor conversão da expressão* é manipular um caso como este:

```
C#  
  
M(b ? 1 : 2);  
  
void M(short);  
void M(long);
```

Essa abordagem tem duas desvantagens pequenas. Primeiro, não é exatamente o mesmo que a expressão switch:

```
C#  
  
M(b ? 1 : 2); // calls M(long)  
M(b switch { true => 1, false => 2 }); // calls M(short)
```

Isso ainda é uma alteração significativa, mas seu escopo é menos provável de afetar os programas reais:

C#

```
M(b ? 1 : 2, 1); // calls M(long, long) without this feature; ambiguous with  
this feature.  
  
M(short, short);  
M(long, long);
```

Isso se torna ambíguo porque a conversão para `long` é melhor para o primeiro argumento (porque ele não usa a *conversão de expressão condicional*), mas a conversão para `short` é melhor para o segundo argumento (porque `short` é um destino de *conversão melhor* do que o `long`). Essa alteração significativa parece menos séria porque não altera silenciosamente o comportamento de um programa existente.

O motivo para as observações sobre a expressão de conversão é manipular um caso como este:

C#

```
_ = (short)(b ? 1 : 2);
```

Este programa atualmente usa a conversão explícita de `int` para `short` queremos preservar o significado do idioma atual deste programa. A alteração não seria possível em tempo de execução, mas com o seguinte programa a alteração seria observável:

C#

```
_ = (A)(b ? c : d);
```

onde `c` é do tipo `C`, `d` é do tipo `D`, e há uma conversão implícita definida pelo usuário de `C` para `D`, e uma conversão implícita definida pelo usuário de `A` para `D` e `A` uma conversão implícita definida pelo usuário de `C` para `A`. Se esse código for compilado antes do C# 9,0, quando `b` for verdadeiro, converteremos de `c` para `D` `A`. Se usarmos a *conversão de expressão condicional*, quando `b` for verdadeiro, converteremos de `c` para `A` diretamente, que executa uma sequência diferente de código de usuário. Portanto, tratamos a *conversão de expressão condicional* como um último recurso em uma conversão, para preservar o comportamento existente.

# Retornos covariantes

Artigo • 16/09/2021

## Resumo

Suprime a *tipos de retorno covariantes*. Especificamente, permite que a substituição de um método declare um tipo de retorno mais derivado do que o método que ele substitui e, da mesma forma, permite a substituição de uma propriedade somente leitura para declarar um tipo mais derivado. As declarações de substituição exibidas em tipos mais derivados seriam necessárias para fornecer um tipo de retorno pelo menos tão específico quanto o que aparece em substituições em seus tipos base. Os chamadores do método ou da propriedade receberão estaticamente o tipo de retorno mais refinado de uma invocação.

## Motivação

É um padrão comum no código que nomes de métodos diferentes precisam ser inventados para contornar a restrição de idioma que as substituições devem retornar o mesmo tipo do método substituído.

Isso seria útil no padrão de fábrica. Por exemplo, na base de código Roslyn, teríamos

C#

```
class Compilation ...
{
    public virtual Compilation WithOptions(Options options)...
}
```

C#

```
class CSharpCompilation : Compilation
{
    public override CSharpCompilation WithOptions(Options options)...
}
```

## Design detalhado

Esta é uma especificação para [tipos de retorno covariantes ↗](#) em C#. Nossa intenção é permitir que a substituição de um método retorne um tipo de retorno mais derivado do

que o método que ele substitui e, da mesma forma, para permitir que a substituição de uma propriedade somente leitura retorne um tipo de retorno mais derivado. Os chamadores do método ou da propriedade receberão estaticamente o tipo de retorno mais refinado de uma invocação, e as substituições que aparecem em tipos mais derivados seriam necessárias para fornecer um tipo de retorno pelo menos tão específico quanto o que aparecesse em substituições em seus tipos base.

---

## Substituição do método de classe

A [restrição existente nos métodos de substituição de classe](#)

- O método override e o método base substituído têm o mesmo tipo de retorno.

é modificado para

- O método override deve ter um tipo de retorno que seja conversível por uma conversão de identidade ou (se o método tiver uma conversão de referência implícita de retorno de valor, não uma [ref](#)) para o tipo de retorno do método base substituído.

E os seguintes requisitos adicionais são acrescentados a essa lista:

- O método override deve ter um tipo de retorno que seja conversível por uma conversão de identidade ou (se o método tiver uma conversão de referência implícita de retorno de valor, não uma [ref](#)) para o tipo de retorno de cada substituição do método base substituído que é declarado em um tipo base (direto ou indireto) do método override.
- O tipo de retorno do método de substituição deve ser pelo menos acessível como o método de substituição ([domínios de acessibilidade](#)).

Essa restrição permite que um método override em uma `private` classe tenha um `private` tipo de retorno. No entanto, ele requer um `public` método override em um `public` tipo para ter um `public` tipo de retorno.

## Propriedade de classe e substituição de indexador

A [restrição existente nas propriedades de substituição de classe](#)

Uma declaração de propriedade de substituição deve especificar exatamente os mesmos modificadores de acessibilidade e nome que a propriedade herdada, e deve haver uma conversão de identidade entre o tipo de substituição e a propriedade herdada. Se a propriedade Inherited tiver apenas um único acessador (ou seja, se a propriedade herdada for somente leitura ou somente gravação), a propriedade de substituição deverá incluir somente esse acessador. Se a propriedade Inherited incluir os acessadores (ou seja, se a propriedade herdada for Read-Write), a propriedade de substituição poderá incluir um único acessador ou ambos os acessadores.

é modificado para

Uma declaração de propriedade de substituição deve especificar exatamente os mesmos modificadores de acessibilidade e o nome que a propriedade herdada, e deve haver uma conversão de identidade ou (se a propriedade herdada for somente leitura e tiver uma [conversão de referência implícita de valor de retorno-não uma referência](#)) do tipo da propriedade de substituição para o tipo da propriedade herdada. Se a propriedade Inherited tiver apenas um único acessador (ou seja, se a propriedade herdada for somente leitura ou somente gravação), a propriedade de substituição deverá incluir somente esse acessador. Se a propriedade Inherited incluir os acessadores (ou seja, se a propriedade herdada for Read-Write), a propriedade de substituição poderá incluir um único acessador ou ambos os acessadores. O tipo da propriedade de substituição deve ser pelo menos acessível como a propriedade de substituição ([domínios de acessibilidade](#)).

---

*O restante da especificação de rascunho abaixo propõe uma extensão adicional para retornos covariantes de métodos de interface a serem considerados posteriormente.*

## Método de interface, propriedade e substituição de indexador

Adicionando aos tipos de membros que são permitidos em uma interface com a adição do recurso DIM no C# 8,0, adicionamos mais suporte para `override` Membros juntamente com retornos covariantes. Eles seguem as regras de `override` Membros conforme especificado para classes, com as seguintes diferenças:

O texto a seguir em classes:

O método substituído por uma declaração de substituição é conhecido como **método base substituído**. Para um método de substituição `M` declarado em uma classe `C`, o método base substituído é determinado examinando cada classe base de `C`, começando pela classe base direta de `C` e continuando com cada classe base com sucesso, até que em um determinado tipo de classe base, pelo menos um método acessível esteja localizado, que tenha a mesma assinatura `M` de após a substituição dos argumentos de tipo.

recebe a especificação correspondente para interfaces:

O método substituído por uma declaração de substituição é conhecido como o **\*método base substituído** `_`. Para um método de substituição `M` declarado em uma interface `I`, o método base substituído é determinado examinando cada interface base direta ou indireta do `I`, coletando o conjunto de interfaces declarando um método acessível que tem a mesma assinatura de `M` após a substituição dos argumentos de tipo. Se esse conjunto de interfaces tiver um `_most` tipo derivado `*`, ao qual há uma conversão de identidade ou de referência implícita de cada tipo nesse conjunto, e esse tipo contiver uma declaração de método exclusivo, esse será o *método base substituído*.

De forma semelhante, permitimos `override` Propriedades e indexadores em interfaces conforme especificado para classes em *acessadores virtuais, lacrados, substituídos e abstratos do 15.7.6*.

## Pesquisa de nome

Pesquisa de nome na presença de declarações de classe `override` atualmente modifica o resultado da pesquisa de nome, impondo os detalhes do membro encontrado da declaração mais derivada `override` na hierarquia de classe, começando do tipo de qualificador do identificador (ou quando não há `this` nenhum qualificador). Por exemplo, em *12.6.2.2 parâmetros correspondentes* que temos

Para métodos virtuais e indexadores definidos em classes, a lista de parâmetros é escolhida na primeira declaração ou substituição do membro da função encontrado ao iniciar com o tipo estático do receptor e Pesquisar por meio de suas classes base.

para isso, adicionamos

Para métodos virtuais e indexadores definidos em interfaces, a lista de parâmetros é escolhida na declaração ou substituição do membro da função encontrado no tipo mais derivado entre os tipos que contêm a declaração de substituição do membro da função. É um erro de tempo de compilação se nenhum tipo exclusivo existir.

Para o tipo de resultado de um acesso de propriedade ou indexador, o texto existente

- Se eu identificar uma propriedade de instância, o resultado será um acesso de propriedade com uma expressão de instância associada de e e um tipo associado que é o tipo da propriedade. Se T for um tipo de classe, o tipo associado será escolhido da primeira declaração ou substituição da propriedade encontrada ao começar com T e pesquisando suas classes base.

é aumentado com

Se T for um tipo de interface, o tipo associado será escolhido da declaração ou substituição da propriedade encontrada no mais derivado de T ou suas interfaces base diretas ou indiretas. É um erro de tempo de compilação se nenhum tipo exclusivo existir.

Uma alteração semelhante deve ser feita no *acesso ao indexador 12.7.7.3*

Em *expressões de invocação 12.7.6*, aumentamos o texto existente

- Caso contrário, o resultado será um valor, com um tipo associado do tipo de retorno do método ou delegado. Se a invocação for de um método de instância e o receptor for de um tipo de classe T, o tipo associado será escolhido da primeira declaração ou substituição do método encontrado ao começar com T e pesquisando suas classes base.

por

Se a invocação for de um método de instância e o receptor for de um tipo de interface T, o tipo associado será escolhido da declaração ou da substituição do método encontrado na interface mais derivada entre T e suas interfaces base diretas e indiretas. É um erro de tempo de compilação se nenhum tipo exclusivo existir.

## Implementações de interface implícita

Esta seção da especificação

Para fins de mapeamento de interface, um membro de classe A corresponde a um membro de interface B quando:

- A e B são métodos, e as listas de parâmetro Name, Type e formal de A e B são idênticas.
- A e B são propriedades, o nome e o tipo de A e B são idênticos e A têm os mesmos acessadores que (tem B A permissão para ter acessadores adicionais se não for uma implementação de membro de interface explícita).
- A e B são eventos, e o nome e o tipo de A e B são idênticos.
- A e B são indexadores, o tipo e as listas de parâmetros formais de A e B são idênticos e A têm os mesmos acessadores como B (A é permitido ter acessadores adicionais se não for uma implementação de membro de interface explícita).

é modificado da seguinte maneira:

Para fins de mapeamento de interface, um membro de classe A corresponde a um membro de interface B quando:

- A e B são métodos, e o nome e as listas de parâmetros formais de A e B são idênticos, e o tipo de retorno A é conversível para o tipo de retorno B por meio de uma identidade de conversão de referência implícita para o tipo de retorno de B .
- A e B são propriedades, o nome de A e B são idênticos, A têm os mesmos acessadores que B (tem A permissão para ter acessadores adicionais se não for uma implementação de membro de interface explícita) e o tipo de A é conversível para o tipo de retorno B por meio de uma conversão de identidade ou, se A for uma propriedade ReadOnly, uma conversão de referência implícita.
- A e B são eventos, e o nome e o tipo de A e B são idênticos.
- A e B são indexadores, as listas de parâmetros formais de A e B são idênticas, A têm os mesmos acessadores que B (tem A permissão para ter acessadores adicionais se não for uma implementação de membro de interface explícita) e o tipo de A é conversível para o tipo de retorno B por meio de uma conversão de identidade ou, se A for um indexador ReadOnly, uma conversão de referência implícita.

Essa é tecnicamente uma alteração significativa, pois o programa abaixo imprime "C1. M "hoje, mas imprimiria" C2. M "na revisão proposta.

```

C#  
  

using System;  
  

interface I1 { object M(); }  

class C1 : I1 { public object M() { return "C1.M"; } }  

class C2 : C1, I1 { public new string M() { return "C2.M"; } }  

class Program  

{  

    static void Main()  

    {  

        I1 i = new C2();  

        Console.WriteLine(i.M());  

    }
}

```

Devido a essa alteração significativa, poderemos considerar não oferecer suporte a tipos de retorno covariantes em implementações implícitas.

## Restrições na implementação da interface

Precisaremos de uma regra de que uma implementação de interface explícita deva declarar um tipo de retorno não menos derivado do que o tipo de retorno declarado em qualquer substituição em suas interfaces base.

## Implicações de compatibilidade de API

*TBD*

## Problemas Abertos

A especificação não diz como o chamador obtém o tipo de retorno mais refinado. Supostamente, isso seria feito de forma semelhante à forma como os chamadores obtêm as especificações de parâmetro da substituição mais derivada.

Se tivermos as seguintes interfaces:

```

C#  
  

interface I1 { I1 M(); }  

interface I2 { I2 M(); }  

interface I3: I1, I2 { override I3 M(); }

```

Observe que `I3`, em, os métodos `I1.M()` e `I2.M()` foram "mesclados". Ao implementar `I3`, é necessário implementá-los juntos.

Em geral, exigimos uma implementação explícita para fazer referência ao método original. A pergunta é, em uma classe

```
C#  
  
class C : I1, I2, I3  
{  
    C IN.M();  
}
```

O que isso significa aqui? O que deveria ser  $N$ ?

Sugiro que possamos implementar um `I1.M` ou `I2.M` (mas não ambos) e tratá-lo como uma implementação de ambos.

## Desvantagens

- [] Cada alteração de idioma deve pagar por si mesma.
- [] Devemos garantir que o desempenho seja razoável, mesmo no caso de hierarquias de herança profundas
- [] Devemos garantir que os artefatos da estratégia de tradução não afetem a semântica da linguagem, mesmo ao consumir o novo IL de compiladores antigos.

## Alternativas

Poderíamos relaxar ligeiramente as regras de linguagem para permitir, na origem,

```
C#  
  
abstract class Cloneable  
{  
    public abstract Cloneable Clone();  
}  
  
class Digit : Cloneable  
{  
    public override Cloneable Clone()  
    {  
        return this.Clone();  
    }  
  
    public new Digit Clone() // Error: 'Digit' already defines a member  
    called 'Clone' with the same parameter types  
    {  
        return this;  
    }  
}
```

```
    }  
}
```

## Perguntas não resolvidas

- [] Como as APIs que foram compiladas para usar esse recurso funcionam em versões mais antigas do idioma?

## Criar reuniões

- algumas discussões em <https://github.com/dotnet/roslyn/issues/357> .
- <https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-01-08.md>
- Discussão offline em direção a uma decisão de dar suporte à substituição de métodos de classe somente em C# 9,0.

# GetEnumerator Suporte de extensão para foreach loops.

Artigo • 16/09/2021

## Resumo

Permita que loops foreach reconheçam um método GetEnumerator do método de extensão que, de outra forma, satisfaça o padrão ForEach e execute o loop sobre a expressão quando, de outra forma, fosse um erro.

## Motivação

Isso trará foreach embutido com o modo como outros recursos em C# são implementados, incluindo a desconstrução assíncrona e baseada em padrões.

## Design detalhado

A alteração de especificação é relativamente simples. Modificamos `The foreach statement` a seção para este texto:

O processamento em tempo de compilação de uma instrução foreach determina primeiro o *tipo de coleção\_*, *tipo de \_enumerador\_ e \_tipo de elemento* da expressão. Essa determinação continua da seguinte maneira:

- Se o tipo `x` de *expressão* for um tipo de matriz, haverá uma conversão de referência implícita de `x` para a `IEnumerable` interface (desde que `System.Array` implementa essa interface). O *tipo de coleção\_* é a `IEnumerable interface`, o *tipo de enumerador\_* é a `IEnumerator` interface e o *tipo \_elemento* é o tipo de elemento do tipo de matriz `X`.
- Se o tipo `x` de *expressão* for `dynamic`, haverá uma conversão implícita de *expression* para a `IEnumerable` interface ([conversões dinâmicas implícitas](#)). O *tipo de coleção\_* é a `IEnumerable` interface e o *tipo de enumerador\** é a `IEnumerator` interface. Se o `var` identificador for fornecido como o `_local_variable_type` \*, o *tipo de elemento* será `dynamic`, caso contrário, será `object`.

- Caso contrário, determine se o tipo `x` tem um `GetEnumerator` método apropriado:
  - Executar pesquisa de membro no tipo `x` com identificador `GetEnumerator` e nenhum argumento de tipo. Se a pesquisa de membro não produzir uma correspondência ou se gerar uma ambiguidade ou produzir uma correspondência que não seja um grupo de métodos, verifique se há uma interface enumerável, conforme descrito abaixo. É recomendável que um aviso seja emitido se a pesquisa de membros produzir qualquer coisa, exceto um grupo de métodos ou nenhuma correspondência.
  - Execute a resolução de sobrecarga usando o grupo de métodos resultante e uma lista de argumentos vazia. Se a resolução de sobrecarga não resultar em métodos aplicáveis, resultar em uma ambiguidade ou resultar em um único método melhor, mas o método for estático ou não público, verifique se há uma interface enumerável, conforme descrito abaixo. É recomendável que um aviso seja emitido se a resolução de sobrecarga produzir algo, exceto um método de instância pública não ambígua, ou nenhum dos métodos aplicáveis.
  - Se o tipo `E` de retorno do `GetEnumerator` método não for uma classe, struct ou tipo de interface, um erro será produzido e nenhuma etapa adicional será executada.
  - A pesquisa de membros é executada em `E` com o identificador `Current` e nenhum argumento de tipo. Se a pesquisa de membro não produzir nenhuma correspondência, o resultado é um erro ou o resultado é qualquer coisa, exceto uma propriedade de instância pública que permite a leitura, um erro é produzido e nenhuma etapa adicional é executada.
  - A pesquisa de membros é executada em `E` com o identificador `MoveNext` e nenhum argumento de tipo. Se a pesquisa de membro não produzir nenhuma correspondência, o resultado é um erro ou o resultado é qualquer coisa, exceto um grupo de métodos, um erro é produzido e nenhuma etapa adicional é executada.
  - A resolução de sobrecarga é executada no grupo de métodos com uma lista de argumentos vazia. Se a resolução de sobrecarga não resultar em métodos aplicáveis, resultar em uma ambiguidade ou resultar em um único método melhor, mas esse método for estático ou não público, ou seu tipo de retorno não for `bool`, um erro será produzido e nenhuma etapa adicional será executada.
  - O *tipo de coleção\_* é `x`, o *tipo de enumerador\_* é `E` e o *tipo \_ Element* é o tipo da `current` propriedade.
- Caso contrário, verifique se há uma interface enumerável:

- Se entre todos os tipos `Ti` para os quais há uma conversão implícita de `x` para `IEnumerable<Ti>`, há um tipo exclusivo de `T` que `T` não é `dynamic` e para todos os outros `Ti` há uma conversão implícita de `IEnumerable<T>` para `IEnumerable<Ti>`, então o \* tipo de coleção \_ é a interface `IEnumerable<T>`, o tipo de enumerador é a interface `IEnumerator<T>` e o \_ tipo de elemento\* é `T` .
- Caso contrário, se houver mais de um desses tipos `T`, um erro será produzido e nenhuma etapa adicional será executada.
- Caso contrário, se houver uma conversão implícita de na `x` `System.Collections.IEnumerable` interface, o \* tipo de coleção \_ é essa interface, o tipo de enumerador será a interface `System.Collections.IEnumerator` e o tipo de elemento \_ \* será `object` .
- Caso contrário, determine se o tipo ' X ' tem um `GetEnumerator` método de extensão apropriado:
  - Execute a pesquisa de método de extensão no tipo `x` com identificador `GetEnumerator` . Se a pesquisa de membro não produzir uma correspondência ou se gerar uma ambiguidade ou produzir uma correspondência que não seja um grupo de métodos, um erro será produzido e nenhuma outra etapa será executada. É recomendável que um aviso seja emitido se a pesquisa de membros produzir qualquer coisa, exceto um grupo de métodos ou nenhuma correspondência.
  - Execute a resolução de sobrecarga usando o grupo de métodos resultante e um único argumento do tipo `x` . Se a resolução de sobrecarga não produzir nenhum método aplicável, resultar em uma ambiguidade ou resultar em um único método melhor, mas esse método não estiver acessível, um erro será produzido quando não for feita nenhuma etapa adicional.
    - Essa resolução permite que o primeiro argumento seja passado por ref se `x` for um tipo struct e o tipo REF for `in` .
  - Se o tipo `E` de retorno do `GetEnumerator` método não for uma classe, struct ou tipo de interface, um erro será produzido e nenhuma etapa adicional será executada.
  - A pesquisa de membros é executada em `E` com o identificador `Current` e nenhum argumento de tipo. Se a pesquisa de membro não produzir nenhuma correspondência, o resultado é um erro ou o resultado é qualquer coisa, exceto uma propriedade de instância pública que permite a leitura, um erro é produzido e nenhuma etapa adicional é executada.
  - A pesquisa de membros é executada em `E` com o identificador `MoveNext` e nenhum argumento de tipo. Se a pesquisa de membro não produzir

nenhuma correspondência, o resultado é um erro ou o resultado é qualquer coisa, exceto um grupo de métodos, um erro é produzido e nenhuma etapa adicional é executada.

- A resolução de sobrecarga é executada no grupo de métodos com uma lista de argumentos vazia. Se a resolução de sobrecarga não resultar em métodos aplicáveis, resultar em uma ambiguidade ou resultar em um único método melhor, mas esse método for estático ou não público, ou seu tipo de retorno não for `bool`, um erro será produzido e nenhuma etapa adicional será executada.
- O *tipo de coleção\_* é `x`, o *tipo de enumerador\_* é `E` e o *tipo \_ Element* é o tipo da `current` propriedade.
- Caso contrário, um erro é produzido e nenhuma etapa adicional é executada.

Para `await foreach` o, as regras são modificadas da mesma forma. A única alteração necessária para essa especificação é remover a `Extension methods do not contribute.` linha da descrição, pois o restante dessa especificação é baseado nas regras acima com nomes diferentes substituídos pelos métodos de padrão.

## Desvantagens

Cada alteração adiciona mais complexidade à linguagem, e isso potencialmente permite que as coisas que não foram projetadas para serem Ed `foreach foreach`, como `Range`.

## Alternativas

Não fazendo nada.

## Perguntas não resolvidas

Nenhum neste ponto.

# Parâmetros discard de lambda

Artigo • 16/09/2021

## Resumo

Permitir que os Descartes (`_`) sejam usados como parâmetros de lambdas e métodos anônimos. Por exemplo:

- Lambdas: `(_, _) => 0`, `(int _, int _) => 0`
- métodos anônimos: `delegate(int _, int _) { return 0; }`

## Motivação

Os parâmetros não utilizados não precisam ser nomeados. A intenção de Descartes é clara, ou seja, não são usadas/descartadas.

## Design detalhado

[Parâmetros do método ↗](#) Na lista de parâmetros de um método lambda ou anônimo com mais de um parâmetro chamado `_`, esses parâmetros são parâmetros de descarte. Observação: se um único parâmetro for nomeado `_`, ele será um parâmetro regular para motivos de compatibilidade com versões anteriores.

Os parâmetros de descarte não introduzem nenhum nome para nenhum escopo. Observe que isso significa que eles não fazem com que os `_` nomes (sublinhado) fiquem ocultos.

[Nomes simples](#) Se `k` for zero e o *Simple\_name* aparecer dentro de um *bloco* e se o espaço de declaração local ([declarações](#)) do *bloco*(ou um *bloco* delimitador) contiver uma variável local, parâmetro (com exceção de parâmetros de descarte) ou constante com nome `I`, o *Simple\_name* se refere a essa variável local, parâmetro ou constante e é classificado como uma variável ou valor.

[Escopos](#) Com exceção dos parâmetros de descarte, o escopo de um parâmetro declarado em uma *lambda\_expression* ([expressões de função anônimas](#)) é a *anonymous\_function\_body* do *lambda\_expression* com a exceção de parâmetros de descarte, o escopo de um parâmetro declarado em uma *anonymous\_method\_expression* ([expressões de função anônimas](#)) é o *bloco* desse *anonymous\_method\_expression*.

# Seções de especificações relacionadas

- Parâmetros correspondentes

# Atributos em funções locais

Artigo • 16/09/2021

## Atributos

Agora, as declarações de função local têm permissão para ter [atributos](#). Parâmetros e parâmetros de tipo em funções locais também podem ter atributos.

Atributos com um significado especificado quando aplicados a um método, seus parâmetros ou seus parâmetros de tipo terão o mesmo significado quando aplicados a uma função local, seus parâmetros ou seus parâmetros de tipo, respectivamente.

Uma função local pode se tornar condicional no mesmo sentido que um [método condicional](#), decorando-a com um `[ConditionalAttribute]`. Uma função local condicional também deve ser `static`. Todas as restrições em métodos condicionais também se aplicam a funções locais condicionais, incluindo que o tipo de retorno deve ser `void`.

## Externo

O `extern` modificador agora é permitido em funções locais. Isso torna a função local externa no mesmo sentido que um [método externo](#).

Da mesma forma que um método externo, o *corpo da função local* de uma função local externa deve ser um ponto-e-vírgula. Um *corpo de função local* de ponto e vírgula só é permitido em uma função local externa.

Uma função local externa também deve ser `static`.

## Syntax

A [gramática de funções locais](#) é modificada da seguinte maneira:

```
local-function-header
  : attributes? local-function-modifiers? return-type identifier type-
parameter-list?
    ( formal-parameter-list? ) type-parameter-constraints-clauses
  ;
```

```
local-function-modifiers
  : (async | unsafe | static | extern)*
  ;

local-function-body
  : block
  | arrow-expression-body
  | ';'*
  ;
```

# Inteiros de tamanho nativo

Artigo • 16/09/2021

## Resumo

Suporte a idiomas para tipos inteiros assinados e sem sinal de tamanho nativo.

A motivação é para cenários de interoperabilidade e bibliotecas de nível baixo.

## Design

Os identificadores `nint` e `nuint` são novas palavras-chave contextuais que representam tipos de inteiros assinados e não assinados nativos. Os identificadores são tratados apenas como palavras-chave quando a pesquisa de nome não encontra um resultado viável no local do programa.

C#

```
nint x = 3;
string y = nameof(nuint);
_ = nint.Equals(x, 3);
```

Os tipos `nint` e `nuint` são representados pelos tipos subjacentes `System.IntPtr` e `System.UIntPtr` com o compilador identificando as conversões e operações adicionais para esses tipos como ints nativos.

## Constantes

Expressões constantes podem ser do tipo `nint` ou `nuint`. Não há sintaxe direta para literais native int. Conversões implícitas ou explícitas de outros valores de constante integral podem ser usadas em vez disso: `const nint i = (nint)42;`.

`nint` as constantes estão no intervalo [ `int.MinValue` , `int.MaxValue` ].

`nuint` as constantes estão no intervalo [ `uint.MinValue` , `uint.MaxValue` ].

Não há nenhum `MinValue` `MaxValue` campo ou ou `nint` `nuint` porque, a não `nuint.MinValue` ser, esses valores não podem ser emitidos como constantes.

O dobramento de constante tem suporte para todos os operadores unários { `+` , `-` , `~` } e operadores binários { `+` , `...` , `*` , `/` , `%` , `==` , `!=` , `<` , `<=` , `>` , `>=` , `&` , `|` , `^` , `<<` , `>>` }. As operações de dobra de constantes são avaliadas com os `Int32` `UInt32` operandos e em vez de ints nativas para um comportamento consistente, independentemente da plataforma do compilador. Se a operação resultar em um valor constante em 32 bits, o dobramento constante será executado em tempo de compilação. Caso contrário, a operação será executada em tempo de execução e não será considerada uma constante.

## Conversões

Há uma conversão de identidade entre `nint` e `IntPtr` entre `nuint` e `UIntPtr`. Há uma conversão de identidade entre os tipos compostos que diferem somente por ints nativos e tipos subjacentes: matrizes, `Nullable<>` tipos construídos e tuplas.

As tabelas a seguir abrangem as conversões entre tipos especiais. (O IL para cada conversão inclui as variantes para `unchecked` e `checked` contextos, se for diferente.)

<b>Operando</b>	<b>Destino</b>	<b>Conversão</b>	<b>IL</b>
<code>object</code>	<code>nint</code>	Conversão unboxing	<code>unbox</code>
<code>void*</code>	<code>nint</code>	<code>PointerToVoid</code>	<code>conv.i</code>
<code>sbyte</code>	<code>nint</code>	<code>ImplicitNumeric</code>	<code>conv.i</code>
<code>byte</code>	<code>nint</code>	<code>ImplicitNumeric</code>	<code>conv.u</code>
<code>short</code>	<code>nint</code>	<code>ImplicitNumeric</code>	<code>conv.i</code>
<code>ushort</code>	<code>nint</code>	<code>ImplicitNumeric</code>	<code>conv.u</code>
<code>int</code>	<code>nint</code>	<code>ImplicitNumeric</code>	<code>conv.i</code>
<code>uint</code>	<code>nint</code>	<code>ExplicitNumeric</code>	<code>conv.u / conv.ovf.u</code>
<code>long</code>	<code>nint</code>	<code>ExplicitNumeric</code>	<code>conv.i / conv.ovf.i</code>
<code>ulong</code>	<code>nint</code>	<code>ExplicitNumeric</code>	<code>conv.i / conv.ovf.i</code>
<code>char</code>	<code>nint</code>	<code>ImplicitNumeric</code>	<code>conv.i</code>
<code>float</code>	<code>nint</code>	<code>ExplicitNumeric</code>	<code>conv.i / conv.ovf.i</code>
<code>double</code>	<code>nint</code>	<code>ExplicitNumeric</code>	<code>conv.i / conv.ovf.i</code>

<b>Operando</b>	<b>Destino</b>	<b>Conversão</b>	<b>IL</b>
decimal	nint	ExplicitNumeric	long decimal.op_Explicit(decimal) conv.i / ... conv.ovf.i
IntPtr	nint	Identidade	
UIntPtr	nint	Nenhum	
object	nuint	Conversão unboxing	unbox
void*	nuint	PointerToVoid	conv.u
sbyte	nuint	ExplicitNumeric	conv.u / conv.ovf.u
byte	nuint	ImplicitNumeric	conv.u
short	nuint	ExplicitNumeric	conv.u / conv.ovf.u
ushort	nuint	ImplicitNumeric	conv.u
int	nuint	ExplicitNumeric	conv.u / conv.ovf.u
uint	nuint	ImplicitNumeric	conv.u
long	nuint	ExplicitNumeric	conv.u / conv.ovf.u
ulong	nuint	ExplicitNumeric	conv.u / conv.ovf.u
char	nuint	ImplicitNumeric	conv.u
float	nuint	ExplicitNumeric	conv.u / conv.ovf.u
double	nuint	ExplicitNumeric	conv.u / conv.ovf.u
decimal	nuint	ExplicitNumeric	ulong decimal.op_Explicit(decimal) conv.u / ... conv.ovf.u.un
IntPtr	nuint	Nenhum	
UIntPtr	nuint	Identidade	
Enumeração	nint	ExplicitEnumeration	
Enumeração	nuint	ExplicitEnumeration	

<b>Operando</b>	<b>Destino</b>	<b>Conversão</b>	<b>IL</b>
nint	object	Conversão boxing	box

Operando	Destino	Conversão	IL
nint	void*	PointerToVoid	conv.i
nint	nuint	ExplicitNumeric	conv.u / conv.ovf.u
nint	sbyte	ExplicitNumeric	conv.i1 / conv.ovf.i1
nint	byte	ExplicitNumeric	conv.u1 / conv.ovf.u1
nint	short	ExplicitNumeric	conv.i2 / conv.ovf.i2
nint	ushort	ExplicitNumeric	conv.u2 / conv.ovf.u2
nint	int	ExplicitNumeric	conv.i4 / conv.ovf.i4
nint	uint	ExplicitNumeric	conv.u4 / conv.ovf.u4
nint	long	ImplicitNumeric	conv.i8 / conv.ovf.i8
nint	ulong	ExplicitNumeric	conv.i8 / conv.ovf.i8
nint	char	ExplicitNumeric	conv.u2 / conv.ovf.u2
nint	float	ImplicitNumeric	conv.r4
nint	double	ImplicitNumeric	conv.r8
nint	decimal	ImplicitNumeric	conv.i8 decimal decimal.op_Implicit(long)
nint	IntPtr	Identidade	
nint	UIntPtr	Nenhum	
nint	Enumeração	ExplicitEnumeration	
nuint	object	Conversão boxing	box
nuint	void*	PointerToVoid	conv.u
nuint	nint	ExplicitNumeric	conv.i / conv.ovf.i
nuint	sbyte	ExplicitNumeric	conv.i1 / conv.ovf.i1
nuint	byte	ExplicitNumeric	conv.u1 / conv.ovf.u1
nuint	short	ExplicitNumeric	conv.i2 / conv.ovf.i2
nuint	ushort	ExplicitNumeric	conv.u2 / conv.ovf.u2
nuint	int	ExplicitNumeric	conv.i4 / conv.ovf.i4

Operando	Destino	Conversão	IL
<code>nuint</code>	<code>uint</code>	ExplicitNumeric	<code>conv.u4 / conv.ovf.u4</code>
<code>nuint</code>	<code>long</code>	ExplicitNumeric	<code>conv.i8 / conv.ovf.i8</code>
<code>nuint</code>	<code>ulong</code>	ImplicitNumeric	<code>conv.u8 / conv.ovf.u8</code>
<code>nuint</code>	<code>char</code>	ExplicitNumeric	<code>conv.u2 / conv.ovf.u2.un</code>
<code>nuint</code>	<code>float</code>	ImplicitNumeric	<code>conv.r.un conv.r4</code>
<code>nuint</code>	<code>double</code>	ImplicitNumeric	<code>conv.r.un conv.r8</code>
<code>nuint</code>	<code>decimal</code>	ImplicitNumeric	<code>conv.u8 decimal decimal.op_Implicit(ulong)</code>
<code>nuint</code>	<code>IntPtr</code>	Nenhum	
<code>nuint</code>	<code>UIntPtr</code>	Identidade	
<code>nuint</code>	Enumeração	ExplicitEnumeration	

A conversão de `A` para `Nullable<B>` é:

- uma conversão anulável implícita se houver uma conversão de identidade ou conversão implícita de `A` para `B` ;
- uma conversão anulável explícita se houver uma conversão explícita de `A` para `B` ;
- caso contrário, inválido.

A conversão de `Nullable<A>` para `B` é:

- uma conversão anulável explícita se houver uma conversão de identidade ou conversão numérica implícita ou explícita de `A` para `B` ;
- caso contrário, inválido.

A conversão de `Nullable<A>` para `Nullable<B>` é:

- uma conversão de identidade se houver uma conversão de identidade de `A` para `B` ;
- uma conversão anulável explícita se houver uma conversão numérica implícita ou explícita de `A` para `B` ;
- caso contrário, inválido.

## Operadores

Os operadores predefinidos são os seguintes. Esses operadores são considerados durante a resolução de sobrecarga com base em regras normais para conversões implícitas se pelo menos um dos operandos for do tipo `nint` ou `nuint`.

(O IL para cada operador inclui as variantes para `unchecked` e `checked` contextos, se for diferente.)

Unário	Assinatura de operador	IL
<code>+</code>	<code>nint operator +(nint value)</code>	<code>nop</code>
<code>+</code>	<code>nuint operator +(nuint value)</code>	<code>nop</code>
<code>-</code>	<code>nint operator -(nint value)</code>	<code>neg</code>
<code>~</code>	<code>nint operator ~(nint value)</code>	<code>not</code>
<code>~</code>	<code>nuint operator ~(nuint value)</code>	<code>not</code>

Binário	Assinatura de operador	IL
<code>+</code>	<code>nint operator +(nint left, nint right)</code>	<code>add / add.ovf</code>
<code>+</code>	<code>nuint operator +(nuint left, nuint right)</code>	<code>add / add.ovf.un</code>
<code>-</code>	<code>nint operator -(nint left, nint right)</code>	<code>sub / sub.ovf</code>
<code>-</code>	<code>nuint operator -(nuint left, nuint right)</code>	<code>sub / sub.ovf.un</code>
<code>*</code>	<code>nint operator *(nint left, nint right)</code>	<code>mul / mul.ovf</code>
<code>*</code>	<code>nuint operator *(nuint left, nuint right)</code>	<code>mul / mul.ovf.un</code>
<code>/</code>	<code>nint operator /(nint left, nint right)</code>	<code>div</code>
<code>/</code>	<code>nuint operator /(nuint left, nuint right)</code>	<code>div.un</code>
<code>%</code>	<code>nint operator %(nint left, nint right)</code>	<code>rem</code>
<code>%</code>	<code>nuint operator %(nuint left, nuint right)</code>	<code>rem.un</code>
<code>==</code>	<code>bool operator ==(nint left, nint right)</code>	<code>beq / ceq</code>
<code>==</code>	<code>bool operator ==(nuint left, nuint right)</code>	<code>beq / ceq</code>
<code>!=</code>	<code>bool operator !=(nint left, nint right)</code>	<code>bne</code>
<code>!=</code>	<code>bool operator !=(nuint left, nuint right)</code>	<code>bne</code>
<code>&lt;</code>	<code>bool operator &lt;(nint left, nint right)</code>	<code>blt / clt</code>

Binário	Assinatura de operador	IL
<	bool operator <(nuint left, nuint right)	blt.un / clt.un
≤	bool operator <=(nint left, nint right)	ble
≤	bool operator <=(nuint left, nuint right)	ble.un
>	bool operator >(nint left, nint right)	bgt / cgt
>	bool operator >(nuint left, nuint right)	bgt.un / cgt.un
≥	bool operator >=(nint left, nint right)	bge
≥	bool operator >=(nuint left, nuint right)	bge.un
&	nint operator &(nint left, nint right)	and
&	nuint operator &(nuint left, nuint right)	and
	nint operator  (nint left, nint right)	or
	nuint operator  (nuint left, nuint right)	or
^	nint operator ^(nint left, nint right)	xor
^	nuint operator ^(nuint left, nuint right)	xor
≪	nint operator <<(nint left, int right)	shl
≪	nuint operator <<(nuint left, int right)	shl
≫	nint operator >>(nint left, int right)	shr
≫	nuint operator >>(nuint left, int right)	shr.un

Para alguns operadores binários, os operadores de IL dão suporte a tipos de operando adicionais (consulte a [tabela de tipos de operando ECMA-335](#) III. 1.5). Mas o conjunto de tipos de operando com suporte do C# é limitado para simplificar e para a consistência com os operadores existentes no idioma.

Versões levantadas dos operadores, em que os argumentos e os tipos de retorno são `nint?` e têm `nuint?` suporte.

As operações de atribuição composta `x op= y` em que `x` ou `y` são ints nativas seguem as mesmas regras que com outros tipos primitivos com operadores predefinidos. Especificamente, a expressão é associada como `x = (T)(x op y)` onde `T` é o tipo de `x` e onde `x` é avaliado apenas uma vez.

Os operadores Shift devem mascarar o número de bits para SHIFT-to 5 bits se `sizeof(nint)` for 4 e para 6 bits se `sizeof(nint)` for 8. (consulte [alternar operadores](#) na especificação C#).

O compilador C# 9 relatará erros de associação a operadores inteiros nativos predefinidos ao compilar com uma versão de idioma anterior, mas permitirá o uso de conversões predefinidas de e para inteiros nativos.

```
csc -langversion:9 -t:library A.cs
```

```
C#
```

```
public class A
{
    public static nint F;
}
```

```
csc -langversion:8 -r:A.dll B.cs
```

```
C#
```

```
class B : A
{
    static void Main()
    {
        F = F + 1; // error: nint operator+ not available with -
langversion:8
        F = (System.IntPtr)F + 1; // ok
    }
}
```

## Dinâmico

As conversões e os operadores são sintetizados pelo compilador e não fazem parte dos `IntPtr` tipos e subjacentes `UIntPtr`. Como resultado, essas conversões e operadores *não estão disponíveis* no associador de tempo de execução para o `dynamic`.

```
C#
```

```
nint x = 2;
nint y = x + x; // ok
dynamic d = x;
nint z = d + x; // RuntimeBinderException: '+' cannot be applied
'System.IntPtr' and 'System.IntPtr'
```

## Membros de tipos

O único Construtor para `nint` ou `nuint` é o construtor sem parâmetros.

Os seguintes membros de `System.IntPtr` e `System.UIntPtr` *são explicitamente excluídos* de `nint` ou `nuint`:

C#

```
// constructors
// arithmetic operators
// implicit and explicit conversions
public static readonly IntPtr Zero; // use 0 instead
public static int Size { get; } // use sizeof() instead
public static IntPtr Add(IntPtr pointer, int offset);
public static IntPtr Subtract(IntPtr pointer, int offset);
public intToInt32();
public longToInt64();
public void* ToPointer();
```

Os membros restantes de `System.IntPtr` e `System.UIntPtr` *são incluídos implicitamente* no `nint` e no `nuint`. Para .NET Framework 4.7.2:

C#

```
public override bool Equals(object obj);
public override int GetHashCode();
public override string ToString();
public string ToString(string format);
```

Interfaces implementadas pelo `System.IntPtr` e `System.UIntPtr` *são incluídas implicitamente* no `nint` e `nuint`, com ocorrências dos tipos subjacentes substituídos pelos tipos inteiros nativos correspondentes. Por exemplo `IntPtr`, se implementa `ISerializable`, `IEquatable<IntPtr>`, `IComparable<IntPtr>`, `nint` implementa `ISerializable`, `IEquatable<nint>`, `IComparable<nint>`.

## Substituindo, ocultando e implementando

`nint` e `System.IntPtr` `nuint` e `System.UIntPtr` são considerados equivalentes para substituir, ocultar e implementar.

Sobrecargas não podem diferir de `nint` e `System.IntPtr`, e `nuint` e `System.UIntPtr`, apenas. Substituições e implementações podem diferir de `nint` e `System.IntPtr`, ou

`nuint` e `System.UIntPtr`, sozinhas. Os métodos ocultam outros métodos que diferem de `nint` e `System.IntPtr`, ou `nuint` e `System.UIntPtr`, apenas.

## Diversos

`nint` e `nuint` as expressões usadas como índices de matriz são emitidas sem conversão.

C#

```
static object GetItem(object[] array, nint index)
{
    return array[index]; // ok
}
```

`nint` e `nuint` pode ser usado como um `enum` tipo base.

C#

```
enum E : nint // ok
{
}
```

Leituras e gravações são atômicas para tipos `nint`, `nuint` e `enum` com tipo base `nint` ou `nuint`.

Os campos podem ser marcados `volatile` para tipos `nint` e `nuint`. O [ECMA-334](#) 15.5.4 não inclui `enum` com o tipo base `System.IntPtr` ou, `System.UIntPtr` no entanto.

`default(nint)` e `new nint()` são equivalentes a `(nint)0`.

`typeof(nint)` é `typeof(IntPtr)`.

`sizeof(nint)` tem suporte, mas requer a compilação em um contexto não seguro (como o faz `sizeof(IntPtr)`). O valor não é uma constante de tempo de compilação.

`sizeof(nint)` é implementado como `sizeof(IntPtr)` em vez de `IntPtr.Size`.

Diagnóstico de compilador para referências de tipo envolvendo `nint` ou `nuint` relatório `nint` ou `nuint` em vez de `IntPtr` ou `UIntPtr`.

## Metadados

`nint` e `nuint` são representados em metadados como `System.IntPtr` e `System.UIntPtr`

Referências de tipo que incluem `nint` ou `nuint` são emitidas com um `System.Runtime.CompilerServices.NativeIntegerAttribute` para indicar quais partes da referência de tipo são ints nativas.

C#

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(
        AttributeTargets.Class | 
        AttributeTargets.Event | 
        AttributeTargets.Field | 
        AttributeTargets.GenericParameter | 
        AttributeTargets.Parameter | 
        AttributeTargets.Property | 
        AttributeTargets.ReturnValue,
        AllowMultiple = false,
        Inherited = false)]
    public sealed class NativeIntegerAttribute : Attribute
    {
        public NativeIntegerAttribute()
        {
            TransformFlags = new[] { true };
        }
        public NativeIntegerAttribute(bool[] flags)
        {
            TransformFlags = flags;
        }
        public readonly bool[] TransformFlags;
    }
}
```

A codificação de referências de tipo com `NativeIntegerAttribute` é abordada em [NativeIntegerAttribute.MD](#).

## Alternativas

Uma alternativa à abordagem de "eliminação de tipo" acima é apresentar novos tipos: `System.NativeInt` e `System.NativeUInt`.

C#

```
public readonly struct NativeInt
{
    public IntPtr Value;
}
```

Tipos distintos permitiriam sobrecarregamentos distintos de `IntPtr` e permitiriam a análise distinta e `ToString()`. Mas haveria mais trabalho para o CLR lidar com esses tipos com eficiência, o que anula a finalidade principal da eficiência dos recursos. E a interoperabilidade com o código int nativo existente que usa `IntPtr` seria mais difícil.

Outra alternativa é adicionar mais suporte int nativo para `IntPtr` na estrutura, mas sem nenhum suporte de compilador específico. Todas as novas conversões e operações aritméticas seriam compatíveis com o compilador automaticamente. Mas o idioma não forneceria palavras-chave, constantes ou `checked` operações.

## Criar reuniões

- [https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-05-26.md ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-05-26.md)
- [https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-06-13.md ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-06-13.md)
- [https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-07-05.md#native-int-and-intptr-operators ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-07-05.md#native-int-and-intptr-operators)
- [https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-10-23.md ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-10-23.md)
- [https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-03-25.md ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-03-25.md)

# Ponteiros de função

Artigo • 16/09/2021

## Resumo

Esta proposta fornece construções de linguagem que expõem opcodes IL que atualmente não podem ser acessados com eficiência ou, em C#, hoje: `ldftn` e `calli`. Esses opcodes de IL podem ser importantes em código de alto desempenho e os desenvolvedores precisam de uma maneira eficiente para acessá-los.

## Motivação

As motivações e o plano de fundo desse recurso são descritos no seguinte problema (como é uma implementação potencial do recurso):

[https://github.com/dotnet/csharplang/issues/191 ↗](https://github.com/dotnet/csharplang/issues/191)

Esta é uma proposta de design alternativa para [intrínsecos do compilador ↗](#)

## Design detalhado

### Ponteiros de função

O idioma permitirá a declaração de ponteiros de função usando a `delegate*` sintaxe. A sintaxe completa é descrita detalhadamente na próxima seção, mas ela se destina a se assemelhar à sintaxe usada `Func` pelas `Action` declarações de tipo e.

C#

```
unsafe class Example {
    void Example(Action<int> a, delegate*<int, void> f) {
        a(42);
        f(42);
    }
}
```

Esses tipos são representados usando o tipo de ponteiro de função, conforme descrito em ECMA-335. Isso significa que a invocação de um `delegate*` usará `calli` onde a invocação de um `delegate` será usada `callvirt` no `Invoke` método. Sintaticamente, embora a invocação seja idêntica para ambas as construções.

A definição ECMA-335 dos ponteiros de método inclui a Convenção de chamada como parte da assinatura de tipo (seção 7,1). A Convenção de chamada padrão será `managed`. As convenções de chamada não gerenciadas podem ser especificadas por meio da inserção de uma `unmanaged` palavra-chave após a `delegate*` sintaxe, que usará o padrão da plataforma de tempo de execução. Convenções não gerenciadas específicas podem então ser especificadas entre colchetes para a `unmanaged` palavra-chave, especificando qualquer tipo começando com `CallConv` no `System.Runtime.CompilerServices` namespace, deixando o `CallConv` prefixo. Esses tipos devem vir da biblioteca principal do programa, e o conjunto de combinações válidas é dependente da plataforma.

C#

```
//This method has a managed calling convention. This is the same as leaving
the managed keyword off.
delegate* managed<int, int>;
```

```
// This method will be invoked using whatever the default unmanaged calling
convention on the runtime
// platform is. This is platform and architecture dependent and is
determined by the CLR at runtime.
delegate* unmanaged<int, int>;
```

```
// This method will be invoked using the cdecl calling convention
// Cdecl maps to System.Runtime.CompilerServices.CallConvCdecl
delegate* unmanaged[Cdecl] <int, int>;
```

```
// This method will be invoked using the stdcall calling convention, and
suppresses GC transition
// Stdcall maps to System.Runtime.CompilerServices.CallConvStdcall
// SuppressGCTransition maps to
System.Runtime.CompilerServices.CallConvSuppressGCTransition
delegate* unmanaged[Stdcall, SuppressGCTransition] <int, int>;
```

As conversões entre `delegate*` os tipos são feitas com base em sua assinatura, incluindo a Convenção de chamada.

C#

```
unsafe class Example {
    void Conversions() {
        delegate*<int, int, int> p1 = ...;
        delegate* managed<int, int, int> p2 = ...;
        delegate* unmanaged<int, int, int> p3 = ...;

        p1 = p2; // okay p1 and p2 have compatible signatures
        Console.WriteLine(p2 == p1); // True
        p2 = p3; // error: calling conventions are incompatible
    }
}
```

Um `delegate*` tipo é um tipo de ponteiro que significa que ele tem todos os recursos e restrições de um tipo de ponteiro padrão:

- Válido somente em um `unsafe` contexto.
- Os métodos que contêm um `delegate*` parâmetro ou tipo de retorno só podem ser chamados de um `unsafe` contexto.
- Não pode ser convertido em `object`.
- Não pode ser usado como um argumento genérico.
- Pode converter implicitamente `delegate*` em `void*`.
- Pode converter explicitamente de `void*` para `delegate*`.

Restrições:

- Atributos personalizados não podem ser aplicados a um `delegate*` ou a qualquer um de seus elementos.
- Um `delegate*` parâmetro não pode ser marcado como `params`
- Um `delegate*` tipo tem todas as restrições de um tipo de ponteiro normal.
- A aritmética de ponteiro não pode ser executada diretamente em tipos de ponteiro de função.

## Sintaxe de ponteiro de função

A sintaxe de ponteiro de função completa é representada pela seguinte gramática:

```
antlr

pointer_type
: ...
| funcptr_type
;

funcptr_type
: 'delegate' '*' calling_convention_specifier? '<' 
funcptr_parameter_list funcptr_return_type '>'
;

calling_convention_specifier
: 'managed'
| 'unmanaged' ('[' unmanaged_calling_convention ']')?
;

unmanaged_calling_convention
: 'Cdecl'
| 'Stdcall'
| 'Thiscall'
;
```

```

| 'Fastcall'
| identifier (',' identifier)*
;

funptr_parameter_list
: (funcptr_parameter ',')*
;

funcptr_parameter
: funcptr_parameter_modifier? type
;

funcptr_return_type
: funcptr_return_modifier? return_type
;

funcptr_parameter_modifier
: 'ref'
| 'out'
| 'in'
;

funcptr_return_modifier
: 'ref'
| 'ref readonly'
;

```

Se não `calling_conventionSpecifier` for fornecido, o padrão será `managed`. A codificação de metadados precisa dos `calling_conventionSpecifier` e quais `identifier`s são válidos no `unmanaged_calling_convention` é abordada na representação de metadados das convenções de chamada.

C#

```

delegate int Func1(string s);
delegate Func1 Func2(Func1 f);

// Function pointer equivalent without calling convention
delegate*<string, int>;
delegate*<delegate*<string, int>, delegate*<string, int>>;

// Function pointer equivalent with calling convention
delegate* managed<string, int>;
delegate*<delegate* managed<string, int>, delegate*<string, int>>;

```

## Conversões de ponteiro de função

Em um contexto sem segurança, o conjunto de conversões implícitas disponíveis (conversões implícitas) é estendido para incluir as seguintes conversões de ponteiro

implícitas:

- *Conversões existentes* ↗
- De \_ tipo funcptr  $F_0$  para outro \_ tipo de funcptr  $F_1$  , desde que todos os seguintes itens sejam verdadeiros:
  - $F_0$  e  $F_1$  têm o mesmo número de parâmetros, e cada parâmetro  $D_{0n}$  no  $F_0$  tem os mesmos `ref` `out` modificadores,, ou `in` como o parâmetro correspondente  $D_{1n}$  no  $F_1$  .
  - Para cada parâmetro de valor (um parâmetro sem nenhum `ref` , `out` ou `in` Modificador), uma conversão de identidade, conversão de referência implícita ou conversão de ponteiro implícita existe do tipo de parâmetro em  $F_0$  para o tipo de parâmetro correspondente no  $F_1$  .
  - Para cada `ref` `out` parâmetro,, ou `in` , o tipo de parâmetro no  $F_0$  é o mesmo que o tipo de parâmetro correspondente no  $F_1$  .
  - Se o tipo de retorno for por valor (não `ref` ou `ref readonly` ), uma identidade, referência implícita ou conversão implícita do ponteiro existirá do tipo de retorno de  $F_1$  para o tipo de retorno de  $F_0$  .
  - Se o tipo de retorno for por referência ( `ref` ou `ref readonly` ), o tipo de retorno e os `ref` modificadores de  $F_1$  serão iguais aos do tipo de retorno e `ref` dos modificadores de  $F_0$  .
  - A Convenção de chamada do  $F_0$  é igual à Convenção de chamada do  $F_1$  .

## Permitir endereço para métodos de destino

Os grupos de métodos agora serão permitidos como argumentos para uma expressão de endereço. O tipo de tal expressão será um `delegate*` que tem a assinatura equivalente do método de destino e uma Convenção de chamada gerenciada:

C#

```
unsafe class Util {
    public static void Log() { }

    void Use() {
        delegate*<void> ptr1 = &Util.Log;

        // Error: type "delegate*<void>" not compatible with "delegate*<int>";
        delegate*<int> ptr2 = &Util.Log;
    }
}
```

Em um contexto sem segurança, um método `M` é compatível com um tipo de ponteiro `F` de função se todas as seguintes opções forem verdadeiras:

- `M` e `F` têm o mesmo número de parâmetros, e cada parâmetro no `M` tem os mesmos `ref` `out` modificadores,, ou `in` como o parâmetro correspondente no `F` .
- Para cada parâmetro de valor (um parâmetro sem nenhum `ref` , `out` ou `in` Modificador), uma conversão de identidade, conversão de referência implícita ou conversão de ponteiro implícita existe do tipo de parâmetro em `M` para o tipo de parâmetro correspondente no `F` .
- Para cada `ref` `out` parâmetro,, ou `in` , o tipo de parâmetro no `M` é o mesmo que o tipo de parâmetro correspondente no `F` .
- Se o tipo de retorno for por valor (não `ref` ou `ref readonly` ), uma identidade, referência implícita ou conversão implícita do ponteiro existirá do tipo de retorno de `F` para o tipo de retorno de `M` .
- Se o tipo de retorno for por referência ( `ref` ou `ref readonly` ), o tipo de retorno e os `ref` modificadores de `F` serão iguais aos do tipo de retorno e `ref` dos modificadores de `M` .
- A Convenção de chamada do `M` é igual à Convenção de chamada do `F` . Isso inclui o bit da Convenção de chamada, bem como quaisquer sinalizadores de Convenção de chamada especificados no identificador não gerenciado.
- `M` é um método estático.

Em um contexto não seguro, existe uma conversão implícita de uma expressão de endereço cujo destino é um grupo de métodos `E` para um tipo de ponteiro de função compatível `F` se `E` contiver pelo menos um método que seja aplicável em seu formato normal a uma lista de argumentos construída pelo uso dos tipos de parâmetro e modificadores de `F` , conforme descrito no seguinte.

- Um único método `M` é selecionado, correspondendo a uma invocação de método do formulário `E(A)` com as seguintes modificações:
  - A lista de argumentos `A` é uma lista de expressões, cada uma classificada como variável e com o tipo e o modificador ( `ref` , `out` ou `in` ) da *\_ lista de parâmetros funcptr* correspondente de `F` .
  - Os métodos candidatos são apenas os métodos que são aplicáveis em seu formato normal, não aqueles aplicáveis em sua forma expandida.
  - Os métodos candidatos são apenas os métodos que são estáticos.
- Se o algoritmo de resolução de sobrecarga produzir um erro, ocorrerá um erro de tempo de compilação. Caso contrário, o algoritmo produz um único método melhor `M` com o mesmo número de parâmetros que `F` e a conversão é considerada como existente.

- O método selecionado `M` deve ser compatível (conforme definido acima) com o tipo de ponteiro de função `F`. Caso contrário, ocorrerá um erro de tempo de compilação.
- O resultado da conversão é um ponteiro de função do tipo `F`.

Isso significa que os desenvolvedores podem depender das regras de resolução de sobrecarga para trabalhar em conjunto com o operador address-of:

C#

```
unsafe class Util {
    public static void Log() { }
    public static void Log(string p1) { }
    public static void Log(int i) { };

    void Use() {
        delegate*<void> a1 = &Log; // Log()
        delegate*<int, void> a2 = &Log; // Log(int i)

        // Error: ambiguous conversion from method group Log to "void*"
        void* v = &Log;
    }
}
```

O operador address-of será implementado usando a `ldftn` instrução.

Restrições deste recurso:

- Aplica-se somente a métodos marcados como `static`.
- Funções não `static` locais não podem ser usadas no `&`. Os detalhes de implementação desses métodos são deliberadamente não especificados pelo idioma. Isso inclui se eles são estáticos versus instância ou exatamente em qual assinatura eles são emitidos.

## Operadores em tipos de ponteiro de função

A seção em código não seguro em operadores é modificada da seguinte maneira:

Em um contexto não seguro, várias construções estão disponíveis para operar em todos os `type_s` de `_pointer_` que não são `_funcptr_type_s`:

- O `*` operador pode ser usado para executar o direcionamento de ponteiro ([indireção de ponteiro](#)).
- O `->` operador pode ser usado para acessar um membro de uma struct por meio de um ponteiro ([acesso de membro de ponteiro](#)).

- O `[]` operador pode ser usado para indexar um ponteiro ([acesso de elemento de ponteiro](#)).
- O `&` operador pode ser usado para obter o endereço de uma variável ([o operador address-of](#)).
- Os `++` `--` operadores e podem ser usados para incrementar e decrementar ponteiros ([incremento de ponteiro e decréscimo](#)).
- Os `+ -` operadores e podem ser usados para executar aritmética de ponteiro ([aritmética de ponteiro](#)).
- Os `==` operadores, `!=`, `< , , >`, `<=` e `=>` podem ser usados para comparar ponteiros ([comparação de ponteiros](#)).
- O `stackalloc` operador pode ser usado para alocar memória da pilha de chamadas ([buffers de tamanho fixo](#)).
- A `fixed` instrução pode ser usada para corrigir temporariamente uma variável para que seu endereço possa ser obtido ([a instrução Fixed](#)).

Em um contexto não seguro, várias construções estão disponíveis para operar em todos os `_funcptr _ type_s`:

- O `&` operador pode ser usado para obter o endereço de métodos estáticos ([permitir endereço-de para métodos de destino](#))
- Os `==` operadores, `!=`, `< , , >`, `<=` e `=>` podem ser usados para comparar ponteiros ([comparação de ponteiros](#)).

Além disso, modificamos todas as seções em `Pointers in expressions` para proibir tipos de ponteiro de função, exceto `Pointer comparison` e `The sizeof operator`.

## Melhor membro da função

A melhor especificação de membro de função será alterada para incluir a seguinte linha:

Um `delegate*` é mais específico do que `void*`

Isso significa que é possível sobrepor `void*` e a `delegate*` e ainda facilmente usar o operador `address-of`.

## Inferência de tipos

Em código não seguro, as seguintes alterações são feitas nos algoritmos de inferência de tipos:

## Tipos de entrada

[https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#input-types ↗](https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#input-types)

O seguinte é adicionado:

Se  $E$  for um grupo de métodos de endereço e  $T$  for um tipo de ponteiro de função, todos os tipos de parâmetro de  $T$  são tipos de entrada de  $E$  com o tipo  $T$ .

## Tipos de saída

[https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#output-types ↗](https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#output-types)

O seguinte é adicionado:

Se  $E$  for um grupo de métodos de endereço e  $T$  for um tipo de ponteiro de função, o tipo de retorno de  $T$  será um tipo de saída  $E$  com o tipo  $T$ .

## Inferências de tipo de saída

[https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#output-type-inferences ↗](https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#output-type-inferences)

O marcador a seguir é adicionado entre os marcadores 2 e 3:

- Se  $E$  é um grupo de métodos de endereço e  $T$  é um tipo de ponteiro de função com tipos de parâmetro  $T_1 \dots T_k$  e tipo de retorno  $T_b$ , e a resolução de sobrecarga de  $E$  com os tipos  $T_1 \dots T_k$  gera um único método com tipo de retorno  $U$ , uma *inferência de limite inferior* é feita de  $U$  para  $T_b$ .

## Inferências exatas

[https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#exact-inferences ↗](https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#exact-inferences)

O submarcador a seguir é adicionado como um caso no marcador 2:

- $V$  é um tipo de ponteiro de função  $\text{delegate}^* < V_2 \dots V_k, V_1 >$  e  $U$  é um tipo de ponteiro de função  $\text{delegate}^* < U_2 \dots U_k, U_1 >$ , e a Convenção de chamada de  $V$  é idêntica a  $U$  e o refness de  $V_i$  é idêntico a  $U_i$ .

## Inferências de limite inferior

[https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#lower-bound-inferences ↗](https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#lower-bound-inferences)

O seguinte caso é adicionado ao marcador 3:

- `v` é um tipo de ponteiro de função `delegate*<V2..Vk, V1>` e há um tipo de ponteiro de função `delegate*<U2..Uk, U1>` que `U` é idêntico a `delegate*<U2..Uk, U1>`, e a Convenção de chamada de `V` é idêntica a `U` e o refness de `Vi` é idêntico a `Ui`.

O primeiro marcador de inferência de `Ui` para `Vi` é modificado para:

- Se `U` não for um tipo de ponteiro de função e `Ui` não for conhecido como um tipo de referência, ou se `U` for um tipo de ponteiro de função e `Ui` não for conhecido como um tipo de ponteiro de função ou um tipo de referência, uma *inferência exata* será feita

Depois, adicionado após o submarcador de inferência de `Ui` para `Vi`:

- Caso contrário, se `V` for `delegate*<V2..Vk, V1>`, a inferência dependerá do parâmetro i-th de `delegate*<V2..Vk, V1>`:
  - Se `V1`:
    - Se o retorno for por valor, será feita uma *inferência de limite inferior*.
    - Se o retorno for por referência, uma *inferência exata* será feita.
  - Se `V2..Vk`:
    - Se o parâmetro for por valor, uma *inferência de limite superior* será feita.
    - Se o parâmetro for por referência, uma *inferência exata* será feita.

## Inferências de limite superior

[https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#upper-bound-inferences ↗](https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#upper-bound-inferences)

O seguinte caso é adicionado ao marcador 2:

- `U` é um tipo de ponteiro de função `delegate*<U2..Uk, U1>` e `V` é um tipo de ponteiro de função que é idêntico a `delegate*<V2..Vk, V1>`, e a Convenção de chamada de `U` é idêntica a `V` e o refness de `Ui` é idêntico a `Vi`.

O primeiro marcador de inferência de `Ui` para `Vi` é modificado para:

- Se `U` não for um tipo de ponteiro de função e `Ui` não for conhecido como um tipo de referência, ou se `U` for um tipo de ponteiro de função e `Ui` não for conhecido como um tipo de ponteiro de função ou um tipo de referência, uma *inferência exata* será feita

Depois, adicionado após o submarcador de inferência de `Ui` para `Vi`:

- Caso contrário, se `U` for `delegate*<U2..Uk, U1>`, a inferência dependerá do parâmetro i-th de `delegate*<U2..Uk, U1>`:
  - Se `U1`:
    - Se o retorno for por valor, uma *inferência de limite superior* será feita.
    - Se o retorno for por referência, uma *inferência exata* será feita.
  - Se `U2.. Britânico`
    - Se o parâmetro for por valor, será feita uma *inferência de limite inferior*.
    - Se o parâmetro for por referência, uma *inferência exata* será feita.

## Representação de metadados `in` de `out` parâmetros,, e `ref readonly` tipos de retorno

As assinaturas de ponteiro de função não têm nenhum local de sinalizadores de parâmetro, portanto, devemos codificar se os parâmetros e o tipo de retorno são `in`, `out` ou `ref readonly` usando modreqs.

### `in`

Reutilizamos `System.Runtime.InteropServices.InAttribute`, aplicada como um `modreq` para o especificador de referência em um parâmetro ou tipo de retorno, para significar o seguinte:

- Se aplicado a um especificador de referência de parâmetro, esse parâmetro será tratado como `in`.
- Se aplicado ao especificador de referência de tipo de retorno, o tipo de retorno será tratado como `ref readonly`.

### `out`

Usamos `System.Runtime.InteropServices.OutAttribute`, aplicados como um `modreq` para o especificador de referência em um tipo de parâmetro, para significar que o parâmetro é um `out` parâmetro.

## Errors

- É um erro a ser aplicado `OutAttribute` como um modreq a um tipo de retorno.
- É um erro aplicar `InAttribute` e `OutAttribute` como um modreq a um tipo de parâmetro.
- Se ambos forem especificados via modopt, eles serão ignorados.

## Representação de metadados de convenções de chamada

As convenções de chamada são codificadas em uma assinatura de método nos metadados por uma combinação do `CallKind` sinalizador na assinatura e zero ou mais `modopt`s no início da assinatura. O ECMA-335 declara atualmente os seguintes elementos no `CallKind` sinalizador:

```
antlr

CallKind
: default
| unmanaged cdecl
| unmanaged fastcall
| unmanaged thiscall
| unmanaged stdcall
| varargs
;
```

Desses, os ponteiros de função no C# oferecerão suporte a todos, exceto `varargs`.

Além disso, o tempo de execução (e, eventualmente, 335) será atualizado para incluir um novo `CallKind` nas novas plataformas. Isso não tem um nome formal no momento, mas este documento será usado `unmanaged ext` como um espaço reservado para destacar o novo formato de Convenção de chamada extensível. Sem `modopt`s, `unmanaged ext` é a Convenção de chamada padrão da plataforma, com `unmanaged` colchetes.

## Mapeando o `calling_convention_specifier` para um `CallKind`

Um `calling_convention_specifier` que é omitido ou especificado como `managed`, é mapeado para o `default CallKind`. Isso é o padrão `CallKind` de qualquer método não

atribuído com `UnmanagedCallersOnly` .

O C# reconhece 4 identificadores especiais que são mapeados para os não gerenciados específicos existentes `CallKind` do ECMA 335. Para que esse mapeamento ocorra, esses identificadores devem ser especificados por conta própria, sem nenhum outro identificador, e esse requisito é codificado na especificação para `unmanaged_calling_convention`s. Esses identificadores são `Cdecl`, `Thiscall`, `Stdcall` e `Fastcall`, que correspondem a `unmanaged cdecl`, `unmanaged thiscall`, `unmanaged stdcall` e `unmanaged fastcall`, respectivamente. Se mais de um `identifier` for especificado ou o único `identifier` não for dos identificadores reconhecidos especialmente, executaremos uma pesquisa de nome especial no identificador com as seguintes regras:

- Precedemos o `identifier` com a cadeia de caracteres `CallConv`
- Examinamos apenas os tipos definidos no `System.Runtime.CompilerServices` namespace.
- Examinamos apenas os tipos definidos na biblioteca principal do aplicativo, que é a biblioteca que define `System.Object` e não tem dependências.
- Estamos procurando apenas em tipos públicos.

Se a pesquisa for realizada com sucesso em todos os `identifier`s especificados em um `unmanaged_calling_convention`, codificaremos o `CallKind unmanaged ext` e codificaremos cada um dos tipos resolvidos no conjunto de `modopt`s no início da assinatura do ponteiro de função. Como uma observação, essas regras significam que os usuários não podem prefixar esses `identifier`s com `CallConv`, pois isso resultará na pesquisa `CallConvCallConvVectorCall` .

Ao interpretar metadados, primeiro vamos examinar o `CallKind`. Se for algo diferente de `unmanaged ext`, ignoramos todos os `modopt`s no tipo de retorno para fins de determinação da Convenção de chamada e usam apenas o `CallKind`. Se `callKind` for, veremos `unmanaged ext` o modopts no início do tipo de ponteiro de função, assumindo a União de todos os tipos que atendem aos seguintes requisitos:

- O é definido na biblioteca principal, que é a biblioteca que faz referência a nenhuma outra biblioteca e define `System.Object` .
- O tipo é definido no `System.Runtime.CompilerServices` namespace.
- O tipo começa com o prefixo `CallConv` .
- O tipo é público.

Eles representam os tipos que devem ser encontrados ao executar a pesquisa nos `identifier`s em um `unmanaged_calling_convention` ao definir um tipo de ponteiro de

função na origem.

É um erro tentar usar um ponteiro de função com um `CallKind` de `unmanaged ext` se o tempo de execução de destino não oferecer suporte ao recurso. Isso será determinado procurando a presença da

`System.Runtime.CompilerServices.RuntimeFeature.UnmanagedCallKind` constante. Se essa constante estiver presente, o tempo de execução será considerado para dar suporte ao recurso.

## `System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute`

`System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute` é um atributo usado pelo CLR para indicar que um método deve ser chamado com uma Convenção de chamada específica. Por isso, apresentamos o seguinte suporte para trabalhar com o atributo:

- É um erro chamar diretamente um método anotado com esse atributo em C#. Os usuários devem obter um ponteiro de função para o método e, em seguida, invocar esse ponteiro.
- É um erro aplicar o atributo a qualquer coisa que não seja um método estático comum ou uma função local estática comum. O compilador C# marcará quaisquer métodos não estáticos ou estáticos não-comuns importados de metadados com esse atributo como sem suporte no idioma.
- É um erro para um método marcado com o atributo para ter um parâmetro ou tipo de retorno que não seja um `unmanaged_type`.
- É um erro para um método marcado com o atributo para ter parâmetros de tipo, mesmo se esses parâmetros de tipo forem restritos a `unmanaged`.
- É um erro para um método em um tipo genérico ser marcado com o atributo.
- É um erro converter um método marcado com o atributo para um tipo delegado.
- É um erro especificar quaisquer tipos para `UnmanagedCallersOnly.CallConvs` os quais não atendem aos requisitos para chamar a Convenção `modopt`s nos metadados.

Ao determinar a Convenção de chamada de um método marcado com um `UnmanagedCallersOnly` atributo válido, o compilador executa as seguintes verificações nos tipos especificados na `CallConvs` propriedade para determinar o efetivo `CallKind` e os `modopt`s que devem ser usados para determinar a Convenção de chamada:

- Se nenhum tipo for especificado, o `CallKind` será tratado como `unmanaged ext`, sem Convenção de chamada `modopt`s no início do tipo de ponteiro de função.

- Se houver um tipo especificado, e esse tipo for chamado `CallConvCdecl` , `CallConvThiscall` , `CallConvStdcall` ou `CallConvFastcall` , o `CallKind` será tratado como `unmanaged cdecl` , `unmanaged thiscall` , `unmanaged stdcall` ou `unmanaged fastcall` , respectivamente, sem nenhuma convenção `modopt` de chamada s no início do tipo de ponteiro de função.
- Se vários tipos forem especificados ou o tipo único não for nomeado um dos tipos especialmente chamados acima, o `CallKind` será tratado como `unmanaged ext` , com a União dos tipos especificados tratados como `modopt` s no início do tipo de ponteiro de função.

O compilador então examina esse efetivo `CallKind` e a `modopt` coleção e usa as regras de metadados normais para determinar a Convenção de chamada final do tipo de ponteiro de função.

## Perguntas em aberto

### Detectando suporte de tempo de execução para `unmanaged ext`

<https://github.com/dotnet/runtime/issues/38135> controla a adição deste sinalizador. Dependendo dos comentários da revisão, usaremos a propriedade especificada no problema ou usamos a presença de `UnmanagedCallersOnlyAttribute` como o sinalizador que determina se os tempos de execução são compatíveis com o `unmanaged ext` .

## Considerações

### Permitir métodos de instância

A proposta pode ser estendida para dar suporte a métodos de instância aproveitando a `EXPLICITTHIS` Convenção de chamada da CLI (chamada `instance` em código C#). Essa forma de ponteiros de função da CLI coloca o `this` parâmetro como um primeiro parâmetro explícito da sintaxe do ponteiro de função.

C#

```
unsafe class Instance {
    void Use() {
        delegate* instance<Instance, string> f = &ToString;
        f(this);
```

```
    }  
}
```

Isso é bom, mas adiciona certa complicação à proposta. Particularmente porque os ponteiros de função que diferem pela Convenção de chamada `instance` e `managed` seriam incompatíveis mesmo que ambos os casos sejam usados para invocar métodos gerenciados com a mesma assinatura C#. Além disso, em todos os casos, consideramos que isso seria valioso ter uma solução simples: usar uma `static` função local.

```
C#
```

```
unsafe class Instance {  
    void Use() {  
        static string toString(Instance i) => i.ToString();  
        delegate*<Instance, string> f = &toString;  
        f(this);  
    }  
}
```

## Não requer segurança na declaração

Em vez de exigir `unsafe` em cada uso de um `delegate*`, só é necessário no ponto em que um grupo de métodos é convertido em um `delegate*`. É aí que os problemas de segurança principal entram em cena (sabendo que o assembly recipiente não pode ser descarregado enquanto o valor estiver ativo). Exigir `unsafe` em outros locais pode ser visto como excessivo.

É assim que o design foi originalmente planejado. Mas as regras de linguagem resultantes pareciam muito estranhas. É impossível ocultar o fato de que esse é um valor de ponteiro e que ele continua exibindo mesmo sem a `unsafe` palavra-chave. Por exemplo, a conversão para `object` não pode ser permitida, não pode ser membro de um `class`, etc... O design do C# é exigir `unsafe` para todos os usos de ponteiro e, portanto, esse design é o seguinte.

Os desenvolvedores ainda poderão apresentar um wrapper *seguro* sobre `delegate*` os valores da mesma maneira que fazem para tipos de ponteiros normais hoje em dia. Considere:

```
C#
```

```
unsafe struct Action {  
    delegate*<void> _ptr;  
  
    Action(delegate*<void> ptr) => _ptr = ptr;
```

```
    public void Invoke() => _ptr();  
}
```

## Usando delegados

Em vez de usar um novo elemento Syntax, `delegate*` basta usar `delegate` tipos existentes com `*` o seguinte tipo:

C#

```
Func<object, object, bool>* ptr = &object.ReferenceEquals;
```

A manipulação de Convenção de chamada pode ser feita anotando os `delegate` tipos com um atributo que especifica um  `CallingConvention`  valor. A falta de um atributo significaria a Convenção de chamada gerenciada.

Codificar isso no IL é problemático. O valor subjacente precisa ser representado como um ponteiro, ainda que ele também deva:

1. Ter um tipo exclusivo para permitir sobrecargas com tipos diferentes de ponteiro de função.
2. Ser equivalente para fins de OHI em limites de assembly.

O último ponto é particularmente problemático. Isso significa que cada assembly que usa  `Func<int>*` deve codificar um tipo equivalente em metadados, embora  `Func<int>*` esteja definido em um assembly, embora não controle. Além disso, qualquer outro tipo que é definido com o nome  `System.Func<T>` em um assembly que não seja mscorelib deve ser diferente da versão definida em mscorelib.

Uma opção que foi explorada estava emitindo tal ponteiro como  `mod_req(Func<int>)`  `void*`. Isso não funciona, embora  `mod_req` não seja possível associar a a  `TypeSpec` e, portanto, não pode direcionar instâncias genéricas.

## Ponteiros de função nomeados

A sintaxe do ponteiro de função pode ser complicada, particularmente em casos complexos, como ponteiros de funções aninhadas. Em vez de os desenvolvedores digitarem a assinatura sempre que o idioma puder permitir declarações nomeadas de ponteiros de função como é feito com  `delegate` .

C#

```
func* void Action();

unsafe class NamedExample {
    void M(Action a) {
        a();
    }
}
```

Parte do problema aqui é que o primitivo de CLI subjacente não tem nomes, portanto, essa seria apenas uma invenção de C# e requer um pouco de trabalho de metadados para habilitar. Isso é factível, mas é um grande respeito do trabalho. Basicamente, ele requer que o C# tenha um complemento para a tabela de tipo def puramente para esses nomes.

Além disso, quando os argumentos dos ponteiros de função nomeados foram examinados, eles poderiam ser aplicados igualmente bem a vários outros cenários. Por exemplo, seria conveniente declarar tuplas nomeadas para reduzir a necessidade de digitar a assinatura completa em todos os casos.

```
C#

(int x, int y) Point;

class NamedTupleExample {
    void M(Point p) {
        Console.WriteLine(p.x);
    }
}
```

Após a discussão, decidimos não permitir a declaração nomeada de `delegate*` tipos. Se encontrarmos uma necessidade significativa para isso com base nos comentários de uso do cliente, investigaremos uma solução de nomenclatura que funciona para ponteiros de função, tuplas, genéricos, etc... Isso provavelmente será semelhante em forma de outras sugestões, como suporte completo `typedef` no idioma.

## Considerações futuras

### delegados estáticos

Isso se refere à [proposta ↗](#) para permitir a declaração de `delegate` tipos que só podem se referir a `static` Membros. A vantagem é que essas `delegate` instâncias podem ser de alocação gratuita e melhor em cenários de desempenho.

Se o recurso de ponteiro de função for implementado `static delegate`, a proposta provavelmente será fechada. A vantagem proposta desse recurso é a natureza livre de alocação. No entanto, foram encontradas investigações recentes que não podem ser obtidas devido ao descarregamento do assembly. Deve haver um identificador forte do `static delegate` para o método ao qual se refere para impedir que o assembly seja descarregado de dentro dele.

Para manter cada `static delegate` instância, seria necessário alocar um novo identificador que executa um contador para os objetivos da proposta. Havia alguns designs em que a alocação poderia ser amortizada para uma única alocação por site de chamada, mas isso era um pouco complexo e não parece que vale a pena compensar.

Isso significa que os desenvolvedores têm, essencialmente, decidir entre as seguintes compensações:

1. Segurança diante do descarregamento do assembly: isso requer alocações e, portanto, `delegate` já é uma opção suficiente.
2. Não há segurança em face de descarregamento de assembly: Use um `delegate*`.  
Isso pode ser encapsulado em um `struct` para permitir o uso fora de um `unsafe` contexto no restante do código.

# Suprimir emissão de `localsinit` sinalizador.

Artigo • 16/09/2021

- [x] proposta
- [] Protótipo: não iniciado
- [] Implementação: não iniciada
- [] Especificação: não iniciada

## Resumo

Permitir supressão de emissão de `localsinit` sinalizador via `SkipLocalsInitAttribute` atributo.

## Motivação

### Tela de fundo

Por variáveis locais de especificação CLR que não contêm referências não são inicializadas para um valor específico pela VM/JIT. A leitura dessas variáveis sem inicialização é de tipo seguro, mas, caso contrário, o comportamento é indefinido e específico da implementação. Normalmente, os locais não inicializados contêm quaisquer valores que foram deixados na memória que agora estão ocupados pelo quadro de pilha. Isso poderia levar a um comportamento não determinístico e difícil reproduzir bugs.

Há duas maneiras de "atribuir" uma variável local:

- ao armazenar um valor ou
- ao especificar `localsinit` o sinalizador que força tudo que está alocado para o pool de memória local ser inicializado com zero Observação: isso inclui variáveis locais e `stackalloc` dados.

O uso de dados não inicializados é desencorajado e não é permitido no código verificável. Embora seja possível provar que, por meio da análise de fluxo, é permitido que o algoritmo de verificação seja conservador e simplesmente exija que `localsinit` esteja definido.

Historicamente, o compilador C# emite um `localsinit` sinalizador em todos os métodos que declaram locais.

Embora o C# empregue uma análise de atribuição definitiva, que é mais estrita do que a especificação CLR exigida (C# também precisa considerar o escopo de locais), não é estritamente garantido que o código resultante seria formalmente verificável:

- As regras CLR e C# podem não concordar se a passagem de um `out` argumento as locais é a `use`.
- As regras do CLR e do C# podem não concordar com o tratamento de ramificações condicionais quando as condições são conhecidas (propagação constante).
- O CLR também poderia simplesmente exigir `localinit`, uma vez que isso é permitido.

## Problema

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.

## Design detalhado

Permite especificar `System.Runtime.CompilerServices.SkipLocalsInitAttribute` como uma maneira de informar ao compilador para não emitir o `localsinit` sinalizador.

O resultado final disso será que os locais podem não ser inicializados com zero pelo JIT, que está na maioria dos casos inobservados em C#.

Além disso `stackalloc`, os dados não serão inicializados com zero. Isso é definitivamente observável, mas também é o cenário mais motivador.

Os destinos de atributo permitidos e reconhecidos são: `Method` , `Property` , `Module` `Class` `Struct` `Interface` , `Constructor` . No entanto, o compilador não exigirá que esse atributo seja definido com os destinos listados, nem irá se preocupar em qual assembly o atributo está definido.

Quando o atributo é especificado em um contêiner ( `class` , `module` , que contém o método para um método aninhado,...), o sinalizador afeta todos os métodos contidos no contêiner.

Os métodos sintetizados "herdam" o sinalizador do contêiner/proprietário lógico.

O sinalizador afeta apenas a estratégia CodeGen para corpos de métodos reais. ,. o sinalizador não tem efeito sobre métodos abstratos e não é propagado para substituir/implementar métodos.

Isso é explicitamente um ***recurso de compilador*** e ***não um recurso de linguagem***.

Da mesma forma que as opções de linha de comando do compilador, o recurso controla os detalhes de implementação de uma estratégia CodeGen específica e não precisa ser exigido pela especificação do C#.

## Desvantagens

- Os compiladores antigos/outros podem não honrar o atributo. Ignorar o atributo é um comportamento compatível. Pode resultar apenas em uma leve queda de desempenho.
- O código sem `localinit` 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.

## Alternativas

- omitir `localinit` sinalizador quando o método é declarado no `unsafe` contexto. Isso poderia causar uma alteração de comportamento silenciosa e perigosa de determinístico para não determinístico em um caso de `stackalloc` .
- omitir `localinit` sempre sinalizador. Ainda pior do que acima.

- Omita `localinit` o sinalizador, a menos que `stackalloc` seja usado no corpo do método. Não aborda o cenário mais solicitado e pode desativar o código não verificável sem a opção de revertê-lo de volta.

## Perguntas não resolvidas

- O atributo deve ser realmente emitido para os metadados?

## Criar reuniões

Nenhum ainda.

# Anotações de parâmetro de tipo sem restrição

Artigo • 16/09/2021

## Resumo

Permitir anotações anuláveis para parâmetros de tipo que não estão restritos a tipos de valor ou tipos de referência: `T?`.

C#

```
static T? FirstOrDefault<T>(this IEnumerable<T> collection) { ... }
```

## ? anotação

No C# 8, as `?` anotações só podiam ser aplicadas a parâmetros de tipo que foram explicitamente restritos a tipos de valores ou tipos de referência. No C# 9, `?` as anotações podem ser aplicadas a qualquer parâmetro de tipo, independentemente das restrições.

A menos que um parâmetro de tipo seja restrito explicitamente a tipos de valor, as anotações só podem ser aplicadas dentro de um `#nullable enable` contexto.

Se um parâmetro de tipo `T` for substituído por um tipo de referência, `T?` representará uma instância anulável desse tipo de referência.

C#

```
var s1 = new string[0].FirstOrDefault(); // string? s1
var s2 = new string?[0].FirstOrDefault(); // string? s2
```

Se `T` é substituído por um tipo de valor, `T?` representa uma instância de `T`.

C#

```
var i1 = new int[0].FirstOrDefault(); // int i1
var i2 = new int?[0].FirstOrDefault(); // int? i2
```

Se `T` é substituído por um tipo anotado `U?`, `T?` representa o tipo anotado `U?` em vez de `U??`.

C#

```
var u1 = new U[0].FirstOrDefault(); // U? u1
var u2 = new U?[0].FirstOrDefault(); // U? u2
```

Se `T` é substituído por um tipo `U`, `T?` representa `U?`, mesmo dentro de um `#nullable disable` contexto.

C#

```
#nullable disable
var u3 = new U[0].FirstOrDefault(); // U? u3
```

Para valores de retorno, `T?` é equivalente a `[MaybeNull]T`; para valores de argumento, `T?` é equivalente a `[AllowNull]T`. A equivalência é importante ao substituir ou implementar interfaces de um assembly compilado com C# 8.

C#

```
public abstract class A
{
    [return: MaybeNull] public abstract T F1<T>();
    public abstract void F2<T>([AllowNull] T t);
}

public class B : A
{
    public override T? F1<T>() where T : default { ... }           // matches
A.F1<T>()
    public override void F2<T>(T? t) where T : default { ... } // matches
A.F2<T>()
}
```

## default Constraint

Para compatibilidade com o código existente onde os métodos genéricos substituídos e explicitamente implementados não podiam incluir cláusulas de restrição explícitas, `T?` em um método substituído ou explicitamente implementado é tratado como `Nullable<T>` onde `T` é um tipo de valor.

Para permitir anotações para parâmetros de tipo restritos a tipos de referência, o C# 8 permitia explicitar `where T : class` e `where T : struct` restrições no método substituído ou explicitamente implementado.

C#

```
class A1
{
    public virtual void F1<T>(T? t) where T : struct { }
    public virtual void F1<T>(T? t) where T : class { }
}

class B1 : A1
{
    public override void F1<T>(T? t) /*where T : struct*/ { }
    public override void F1<T>(T? t) where T : class { }
}
```

Para permitir anotações para parâmetros de tipo que não são restritos a tipos de referência ou tipos de valor, o C# 9 permite uma nova `where T : default` restrição.

C#

```
class A2
{
    public virtual void F2<T>(T? t) where T : struct { }
    public virtual void F2<T>(T? t) { }
}

class B2 : A2
{
    public override void F2<T>(T? t) /*where T : struct*/ { }
    public override void F2<T>(T? t) where T : default { }
}
```

É um erro usar uma `default` restrição diferente de em uma substituição de método ou implementação explícita. É um erro usar uma `default` restrição quando o parâmetro de tipo correspondente no método substituído ou de interface é restrito a um tipo de referência ou tipo de valor.

## Criar reuniões

- [https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-11-25.md ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-11-25.md)
- [https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-06-17.md#t ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-06-17.md#t)



# Record structs

Article • 06/23/2023

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

The syntax for a record struct is as follows:

```
antlr

record_struct_declarati
  : attributes? struct_modifier* 'partial'? 'record' 'struct' identifier
  type_parameter_list?
    parameter_list? struct_interfaces? type_parameter_constraints_clause*
  record_struct_body
  ;

record_struct_body
  : struct_body
  | ';'
```

Record struct types are value types, like other struct types. They implicitly inherit from the class `System.ValueType`. The modifiers and members of a record struct are subject to the same restrictions as those of structs (accessibility on type, modifiers on members, `base(...)` instance constructor initializers, definite assignment for `this` in constructor, destructors, ...). Record structs will also follow the same rules as structs for parameterless instance constructors and field initializers, but this document assumes that we will lift those restrictions for structs generally.

See §15.4.9 See [parameterless struct constructors](#) spec.

Record structs cannot use `ref` modifier.

At most one partial type declaration of a partial record struct may provide a `parameter_list`. The `parameter_list` may be empty.

Record struct parameters cannot use `ref`, `out` or `this` modifiers (but `in` and `params` are allowed).

## Members of a record struct

In addition to the members declared in the record struct body, a record struct type has additional synthesized members. Members are synthesized unless a member with a "matching" signature is declared in the record struct body or an accessible concrete non-virtual member with a "matching" signature is inherited. Two members are considered matching if they have the same signature or would be considered "hiding" in an inheritance scenario. See Signatures and overloading [§7.6](#). It is an error for a member of a record struct to be named "Clone".

It is an error for an instance field of a record struct to have an unsafe type.

A record struct is not permitted to declare a destructor.

The synthesized members are as follows:

## Equality members

The synthesized equality members are similar as in a record class (`Equals` for this type, `Equals` for `object` type, `==` and `!=` operators for this type), except for the lack of `EqualityComparer`, null checks or inheritance.

The record struct implements `System.IEquatable<R>` and includes a synthesized strongly-typed overload of `Equals(R other)` where `R` is the record struct. The method is `public`. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility.

If `Equals(R other)` is user-defined (not synthesized) but `GetHashCode` is not, a warning is produced.

C#

```
public readonly bool Equals(R other);
```

The synthesized `Equals(R)` returns `true` if and only if for each instance field `fieldN` in the record struct the value of

`System.Collections.Generic.EqualityComparer<TN>.Default.Equals(fieldN, other.fieldN)` where `TN` is the field type is `true`.

The record struct includes synthesized `==` and `!=` operators equivalent to operators declared as follows:

```
C#  
  
public static bool operator==(R r1, R r2)  
    => r1.Equals(r2);  
public static bool operator!=(R r1, R r2)  
    => !(r1 == r2);
```

The `Equals` method called by the `==` operator is the `Equals(R other)` method specified above. The `!=` operator delegates to the `==` operator. It is an error if the operators are declared explicitly.

The record struct includes a synthesized override equivalent to a method declared as follows:

```
C#  
  
public override readonly bool Equals(object? obj);
```

It is an error if the override is declared explicitly. The synthesized override returns `other is R temp && Equals(temp)` where `R` is the record struct.

The record struct includes a synthesized override equivalent to a method declared as follows:

```
C#  
  
public override readonly int GetHashCode();
```

The method can be declared explicitly.

A warning is reported if one of `Equals(R)` and `GetHashCode()` is explicitly declared but the other method is not explicit.

The synthesized override of `GetHashCode()` returns an `int` result of combining the values of

`System.Collections.Generic.EqualityComparer<TN>.Default.GetHashCode(fieldN)` for each instance field `fieldN` with `TN` being the type of `fieldN`.

For example, consider the following record struct:

```
C#
```

```
record struct R1(T1 P1, T2 P2);
```

For this record struct, the synthesized equality members would be something like:

C#

```
struct R1 : IEquatable<R1>
{
    public T1 P1 { get; set; }
    public T2 P2 { get; set; }
    public override bool Equals(object? obj) => obj is R1 temp &&
Equals(temp);
    public bool Equals(R1 other)
    {
        return
            EqualityComparer<T1>.Default.Equals(P1, other.P1) &&
            EqualityComparer<T2>.Default.Equals(P2, other.P2);
    }
    public static bool operator==(R1 r1, R1 r2)
        => r1.Equals(r2);
    public static bool operator!=(R1 r1, R1 r2)
        => !(r1 == r2);
    public override int GetHashCode()
    {
        return Combine(
            EqualityComparer<T1>.Default.GetHashCode(P1),
            EqualityComparer<T2>.Default.GetHashCode(P2));
    }
}
```

## Printing members: PrintMembers and ToString methods

The record struct includes a synthesized method equivalent to a method declared as follows:

C#

```
private bool PrintMembers(System.Text.StringBuilder builder);
```

The method does the following:

1. for each of the record struct's printable members (non-static public field and readable property members), appends that member's name followed by " = " followed by the member's value separated with ", ",
2. return true if the record struct has printable members.

For a member that has a value type, we will convert its value to a string representation using the most efficient method available to the target platform. At present that means calling `ToString` before passing to `StringBuilder.Append`.

If the record's printable members do not include a readable property with a non-`readonly` `get` accessor, then the synthesized `PrintMembers` is `readonly`. There is no requirement for the record's fields to be `readonly` for the `PrintMembers` method to be `readonly`.

The `PrintMembers` method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility.

The record struct includes a synthesized method equivalent to a method declared as follows:

C#

```
public override string ToString();
```

If the record struct's `PrintMembers` method is `readonly`, then the synthesized `ToString()` method is `readonly`.

The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility.

The synthesized method:

1. creates a `StringBuilder` instance,
2. appends the record struct name to the builder, followed by " { ",
3. invokes the record struct's `PrintMembers` method giving it the builder, followed by " " if it returned true,
4. appends "}",
5. returns the builder's contents with `builder.ToString()`.

For example, consider the following record struct:

C#

```
record struct R1(T1 P1, T2 P2);
```

For this record struct, the synthesized printing members would be something like:

C#

```

struct R1 : IEquatable<R1>
{
    public T1 P1 { get; set; }
    public T2 P2 { get; set; }

    private bool PrintMembers(StringBuilder builder)
    {
        builder.Append(nameof(P1));
        builder.Append(" = ");
        builder.Append(this.P1); // or builder.Append(this.P1.ToString());
    if P1 has a value type
        builder.Append(", ");

        builder.Append(nameof(P2));
        builder.Append(" = ");
        builder.Append(this.P2); // or builder.Append(this.P2.ToString());
    if P2 has a value type

        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R1));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}

```

## Positional record struct members

In addition to the above members, record structs with a parameter list ("positional records") synthesize additional members with the same conditions as the members above.

## Primary Constructor

A record struct has a public constructor whose signature corresponds to the value parameters of the type declaration. This is called the primary constructor for the type. It is an error to have a primary constructor and a constructor with the same signature

already present in the struct. If the type declaration does not include a parameter list, no primary constructor is generated.

C#

```
record struct R1
{
    public R1() { } // ok
}

record struct R2()
{
    public R2() { } // error: 'R2' already defines constructor with same
parameter types
}
```

Instance field declarations for a record struct are permitted to include variable initializers. If there is no primary constructor, the instance initializers execute as part of the parameterless constructor. Otherwise, at runtime the primary constructor executes the instance initializers appearing in the record-struct-body.

If a record struct has a primary constructor, any user-defined constructor must have an explicit `this` constructor initializer that calls the primary constructor or an explicitly declared constructor.

Parameters of the primary constructor as well as members of the record struct are in scope within initializers of instance fields or properties. Instance members would be an error in these locations (similar to how instance members are in scope in regular constructor initializers today, but an error to use), but the parameters of the primary constructor would be in scope and useable and would shadow members. Static members would also be useable.

A warning is produced if a parameter of the primary constructor is not read.

The definite assignment rules for struct instance constructors apply to the primary constructor of record structs. For instance, the following is an error:

C#

```
record struct Pos(int X) // definite assignment error in primary constructor
{
    private int x;
    public int X { get { return x; } set { x = value; } } = X;
}
```

## Properties

For each record struct parameter of a record struct declaration there is a corresponding public property member whose name and type are taken from the value parameter declaration.

For a record struct:

- A public `get` and `init` auto-property is created if the record struct has `readonly` modifier, `get` and `set` otherwise. Both kinds of set accessors (`set` and `init`) are considered "matching". So the user may declare an init-only property in place of a synthesized mutable one. An inherited `abstract` property with matching type is overridden. No auto-property is created if the record struct has an instance field with expected name and type. It is an error if the inherited property does not have `public get` and `set/init` accessors. It is an error if the inherited property or field is hidden.

The auto-property is initialized to the value of the corresponding primary constructor parameter. Attributes can be applied to the synthesized auto-property and its backing field by using `property:` or `field:` targets for attributes syntactically applied to the corresponding record struct parameter.

## Deconstruct

A positional record struct with at least one parameter synthesizes a public void-returning instance method called `Deconstruct` with an `out` parameter declaration for each parameter of the primary constructor declaration. Each parameter of the `Deconstruct` method has the same type as the corresponding parameter of the primary constructor declaration. The body of the method assigns each parameter of the `Deconstruct` method to the value from an instance member access to a member of the same name. If the instance members accessed in the body do not include a property with a non-`readonly` `get` accessor, then the synthesized `Deconstruct` method is `readonly`. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or is static.

## Allow `with` expression on structs

It is now valid for the receiver in a `with` expression to have a struct type.

On the right hand side of the `with` expression is a `member_initializer_list` with a sequence of assignments to `identifier`, which must be an accessible instance field or

property of the receiver's type.

For a receiver with struct type, the receiver is first copied, then each `member_initializer` is processed the same way as an assignment to a field or property access of the result of the conversion. Assignments are processed in lexical order.

## Improvements on records

### Allow `record class`

The existing syntax for record types allows `record class` with the same meaning as

`record:`

antlr

```
record_declaration
    : attributes? class_modifier* 'partial'? 'record' 'class'? identifier
      type_parameter_list?
        parameter_list? record_base? type_parameter_constraints_clause*
      record_body
    ;
```

### Allow user-defined positional members to be fields

See [https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-10-05.md#changing-the-member-type-of-a-primary-constructor-parameter ↗](https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-10-05.md#changing-the-member-type-of-a-primary-constructor-parameter)

No auto-property is created if the record has or inherits an instance field with expected name and type.

### Allow parameterless constructors and member initializers in structs

See [parameterless struct constructors](#) spec.

## Open questions

- how to recognize record structs in metadata? (we don't have an unspeakable clone method to leverage...)

## Answered

- confirm that we want to keep PrintMembers design (separate method returning `bool`) (answer: yes)
- confirm we won't allow `record ref struct` (issue with `IEquatable<RefStruct>` and `ref` fields) (answer: yes)
- confirm implementation of equality members. Alternative is that synthesized `bool Equals(R other)`, `bool Equals(object? other)` and operators all just delegate to `ValueType.Equals`. (answer: yes)
- confirm that we want to allow field initializers when there is a primary constructor. Do we also want to allow parameterless struct constructors while we're at it (the Activator issue was apparently fixed)? (answer: yes, updated spec should be reviewed in LDM)
- how much do we want to say about `combine` method? (answer: as little as possible)
- should we disallow a user-defined constructor with a copy constructor signature? (answer: no, there is no notion of copy constructor in the record structs spec)
- confirm that we want to disallow members named "Clone". (answer: correct)
- double-check that synthesized `Equals` logic is functionally equivalent to runtime implementation (e.g. `float.NaN`) (answer: confirmed in LDM)
- could field- or property-targeting attributes be placed in the positional parameter list? (answer: yes, same as for record class)
- `with` on generics? (answer: out of scope for C# 10)
- should `GetHashCode` include a hash of the type itself, to get different values between `record struct S1`; and `record struct S2`;? (answer: no)

# Parameterless struct constructors

Article • 06/23/2023

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

## Summary

Support parameterless constructors and instance field initializers for struct types.

## Motivation

Explicit parameterless constructors would give more control over minimally constructed instances of the struct type. Instance field initializers would allow simplified initialization across multiple constructors. Together these would close an obvious gap between `struct` and `class` declarations.

Support for field initializers would also allow initialization of fields in `record struct` declarations without explicitly implementing the primary constructor.

C#

```
record struct Person(string Name)
{
    public object Id { get; init; } = GetNextId();
```

If struct field initializers are supported for constructors with parameters, it seems natural to extend that to parameterless constructors as well.

C#

```
record struct Person()
{
    public string Name { get; init; }
    public object Id { get; init; } = GetNextId();
}
```

# Proposal

## Instance field initializers

Instance field declarations for a struct may include initializers.

As with class field initializers [§14.5.6.3](#):

A variable initializer for an instance field cannot reference the instance being created.

An error is reported if a struct has field initializers and no declared instance constructors since the field initializers will not be run.

C#

```
struct S { int F = 42; } // error: 'struct' with field initializers must
include an explicitly declared constructor
```

## Constructors

A struct may declare a parameterless instance constructor.

A parameterless instance constructor is valid for all struct kinds including `struct`, `readonly struct`, `ref struct`, and `record struct`.

If no parameterless instance constructor is declared, the struct (see [§15.4.9](#)) ...

A struct implicitly has a parameterless instance constructor which always returns the value that results from setting all value type fields to their default value and all reference type fields to null.

## Modifiers

A parameterless instance struct constructor must be declared `public`.

C#

```
struct S0 { } // ok
struct S1 { public S1() { } } // ok
struct S2 { internal S2() { } } // error: parameterless constructor must be
'public'
```

Non-public constructors are ignored when importing types from metadata.

Constructors can be declared `extern` or `unsafe`. Constructors cannot be `partial`.

## Executing field initializers

*Instance variable initializers* ([§14.11.3](#)) is modified as follows:

When a class instance constructor has no constructor initializer, or it has a constructor initializer of the form `base(...)`, that constructor implicitly performs the initializations specified by the *variable\_initializers* of the instance fields declared in its class. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor and before the implicit invocation of the direct base class constructor.

When a struct instance constructor has no constructor initializer, that constructor implicitly performs the initializations specified by the *variable\_initializers* of the instance fields declared in its struct. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor.

When a struct instance constructor has a `this()` constructor initializer that represents the *default parameterless constructor*, the declared constructor implicitly clears all instance fields and performs the initializations specified by the *variable\_initializers* of the instance fields declared in its struct. Immediately upon entry to the constructor, all value type fields are set to their default value and all reference type fields are set to `null`. Immediately after that, a sequence of assignments corresponding to the *variable\_initializers* are executed.

## Definite assignment

Instance fields (other than `fixed` fields) must be definitely assigned in struct instance constructors that do not have a `this()` initializer (see [§15.4.9](#)).

C#

```

struct S0 // ok
{
    int x;
    object y;
}

struct S1 // error: 'struct' with field initializers must include an
explicitly declared constructor
{
    int x = 1;
    object y;
}

struct S2
{
    int x = 1;
    object y;
    public S2() { } // error: field 'y' must be assigned
}

struct S3 // ok
{
    int x = 1;
    object y;
    public S3() { y = 2; }
}

```

## No `base()` initializer

A `base()` initializer is disallowed in struct constructors.

The compiler will not emit a call to the base `System.ValueType` constructor from struct instance constructors.

## `record struct`

An error is reported if a `record struct` has field initializers and does not contain a primary constructor nor any instance constructors since the field initializers will not be run.

C#

```

record struct R0;                      // ok
record struct R1 { int F = 42; }    // error: 'struct' with field
initializers must include an explicitly declared constructor
record struct R2() { int F = 42; } // ok
record struct R3(int F);           // ok

```

A `record struct` with an empty parameter list will have a parameterless primary constructor.

C#

```
record struct R3();           // primary .ctor: public R3() { }
record struct R4() { int F = 42; } // primary .ctor: public R4() { F = 42; }
```

An explicit parameterless constructor in a `record struct` must have a `this` initializer that calls the primary constructor or an explicitly declared constructor.

C#

```
record struct R5(int F)
{
    public R5() {}           // error: must have 'this' initializer
    that calls explicit .ctor
    public R5(object o) : this() {} // ok
    public int F = F;
}
```

## Fields

The implicitly-defined parameterless constructor will zero fields rather than calling any parameterless constructors for the field types. No warnings are reported that field constructors are ignored. *No change from C#9.*

C#

```
struct S0
{
    public S0() {}
}

struct S1
{
    S0 F; // S0 constructor ignored
}

struct S<T> where T : struct
{
    T F; // constructor (if any) ignored
}
```

## default expression

`default` ignores the parameterless constructor and generates a zeroed instance. *No change from C#9.*

```
C#  
  
// struct S { public S() { } }  
  
_ = default(S); // constructor ignored, no warning
```

## new()

Object creation invokes the parameterless constructor if public; otherwise the instance is zeroed. *No change from C#9.*

```
C#  
  
// public struct PublicConstructor { public PublicConstructor() { } }  
// public struct PrivateConstructor { private PrivateConstructor() { } }  
  
_ = new PublicConstructor(); // call PublicConstructor::ctor()  
_ = new PrivateConstructor(); // initobj PrivateConstructor
```

A warning wave may report a warning for use of `new()` with a struct type that has constructors but no parameterless constructor. No warning will be reported when using substituting such a struct type for a type parameter with a `new()` or `struct` constraint.

```
C#  
  
struct S { public S(int i) { } }  
static T CreateNew<T>() where T : new() => new T();  
  
_ = new S();           // warning: no constructor called  
_ = CreateNew<S>(); // ok
```

## Uninitialized values

A local or field of a struct type that is not explicitly initialized is zeroed. The compiler reports a definite assignment error for an uninitialized struct that is not empty. *No change from C#9.*

```
C#  
  
NoConstructor s1;  
PublicConstructor s2;
```

```
s1.ToString(); // error: use of unassigned local (unless type is empty)
s2.ToString(); // error: use of unassigned local (unless type is empty)
```

## Array allocation

Array allocation ignores any parameterless constructor and generates zeroed elements.  
*No change from C#9.*

C#

```
// struct S { public S() { } }
var a = new S[1]; // constructor ignored, no warning
```

## Parameter default value `new()`

A parameter default value of `new()` binds to the parameterless constructor if public (and reports an error that the value is not constant); otherwise the instance is zeroed. *No change from C#9.*

C#

```
// public struct PublicConstructor { public PublicConstructor() { } }
// public struct PrivateConstructor { private PrivateConstructor() { } }

static void F1(PublicConstructor s1 = new()) { } // error: default value
must be constant
static void F2(PrivateConstructor s2 = new()) { } // ok: initobj
```

## Type parameter constraints: `new()` and `struct`

The `new()` and `struct` type parameter constraints require the parameterless constructor to be `public` if defined (see Satisfying constraints - §8.4.5 ↴).

The compiler assumes all structs satisfy `new()` and `struct` constraints. *No change from C#9.*

C#

```
// public struct PublicConstructor { public PublicConstructor() { } }
// public struct InternalConstructor { internal InternalConstructor() { } }

static T CreateNew<T>() where T : new() => new T();
static T CreateStruct<T>() where T : struct => new T();
```

```
_ = CreateNew<PublicConstructor>();      // ok
_ = CreateStruct<PublicConstructor>();    // ok

_ = CreateNew<InternalConstructor>();    // compiles; may fail at runtime
_ = CreateStruct<InternalConstructor>(); // compiles; may fail at runtime
```

`new T()` is emitted as a call to `System.Activator.CreateInstance<T>()`, and the compiler assumes the implementation of `CreateInstance<T>()` invokes the `public` parameterless constructor if defined.

*With .NET Framework, `Activator.CreateInstance<T>()` invokes the parameterless constructor if the constraint is `where T : new()` but appears to ignore the parameterless constructor if the constraint is `where T : struct`.*

## Optional parameters

Constructors with optional parameters are not considered parameterless constructors.  
*No change from C#9.*

C#

```
struct S1 { public S1(string s = "") { } }
struct S2 { public S2(params object[] args) { } }

_ = new S1(); // ok: ignores constructor
_ = new S2(); // ok: ignores constructor
```

## Metadata

Explicit parameterless struct instance constructors will be emitted to metadata.

Public parameterless struct instance constructors will be imported from metadata; non-public struct instance constructors will be ignored. *No change from C#9.*

## See also

- [https://github.com/dotnet/roslyn/issues/1029 ↗](https://github.com/dotnet/roslyn/issues/1029)

## Design meetings

- [https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-04-28.md#open-questions-in-record-and-parameterless-structs ↗](https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-04-28.md#open-questions-in-record-and-parameterless-structs)
- [https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-03-10.md#parameterless-struct-constructors ↗](https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-03-10.md#parameterless-struct-constructors)
- [https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-01-27.md#field-initializers ↗](https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-01-27.md#field-initializers)

# Global Using Directive

Article • 03/24/2022

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

Syntax for a using directive is extended with an optional `global` keyword that can precede the `using` keyword:

```
antlr

compilation_unit
    : extern_alias_directive* global_using_directive* using_directive*
global_attributes? namespace_member_declaration*
;

global_using_directive
: global_using_alias_directive
| global_using_namespace_directive
| global_using_static_directive
;

global_using_alias_directive
: 'global' 'using' identifier '=' namespace_or_type_name ';'
;

global_using_namespace_directive
: 'global' 'using' namespace_name ';'
;

global_using_static_directive
: 'global' 'using' 'static' type_name ';'
;
```

- The `global_using_directives` are allowed only on the Compilation Unit level (cannot be used inside a `namespace_declaration`).
- The `global_using_directives`, if any, must precede any `using_directives`.

- The scope of a *global\_using\_directives* extends over the *namespace\_member\_declarations* of all compilation units within the program. The scope of a *global\_using\_directive* specifically does not include other *global\_using\_directives*. Thus, peer *global\_using\_directives* or those from a different compilation unit do not affect each other, and the order in which they are written is insignificant. The scope of a *global\_using\_directive* specifically does not include *using\_directives* immediately contained in any compilation unit of the program.

The effect of adding a *global\_using\_directive* to a program can be thought of as the effect of adding a similar *using\_directive* that resolves to the same target namespace or type to every compilation unit of the program. However, the target of a *global\_using\_directive* is resolved in context of the compilation unit that contains it.

## §7.7 Scopes

These are the relevant bullet points with proposed additions (which are **in bold**):

- The scope of name defined by an *extern\_alias\_directive* extends over the ***global\_using\_directives*, *using\_directives*, *global\_attributes*** and *namespace\_member\_declarations* of its immediately containing compilation unit or namespace body. An *extern\_alias\_directive* does not contribute any new members to the underlying declaration space. In other words, an *extern\_alias\_directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.
- The scope of a name defined or imported by a *global\_using\_directive* extends over the ***global\_attributes* and *namespace\_member\_declarations*** of all the *compilation\_units* in the program.

## §7.8 Namespace and type names

Changes are made to the algorithm determining the meaning of a *namespace\_or\_type\_name* as follows.

This is the relevant bullet point with proposed additions (which are **in bold**):

- If the *namespace\_or\_type\_name* is of the form **I** or of the form **I<A<sub>1</sub>, ..., A<sub>k</sub>>**:
  - If **K** is zero and the *namespace\_or\_type\_name* appears within a generic method declaration ([§14.6](#)) and if that declaration includes a type parameter ([§14.2.3](#)) with name **I**, then the *namespace\_or\_type\_name* refers to that type parameter.

- Otherwise, if the *namespace\_or\_type\_name* appears within a type declaration, then for each instance type  $T$  ([§14.3.2](#)), starting with the instance type of that type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
  - If  $K$  is zero and the declaration of  $T$  includes a type parameter with name  $I$ , then the *namespace\_or\_type\_name* refers to that type parameter.
  - Otherwise, if the *namespace\_or\_type\_name* appears within the body of the type declaration, and  $T$  or any of its base types contain a nested accessible type having name  $I$  and  $K$  type parameters, then the *namespace\_or\_type\_name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that non-type members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) and type members with a different number of type parameters are ignored when determining the meaning of the *namespace\_or\_type\_name*.
- If the previous steps were unsuccessful then, for each namespace  $N$ , starting with the namespace in which the *namespace\_or\_type\_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
  - If  $K$  is zero and  $I$  is the name of a namespace in  $N$ , then:
    - If the location where the *namespace\_or\_type\_name* occurs is enclosed by a namespace declaration for  $N$  and the namespace declaration contains an *extern\_alias\_directive* or *using\_alias\_directive* that associates the name  $I$  with a namespace or type, **or any namespace declaration for  $N$  in the program contains a *global\_using\_alias\_directive*** that associates the name  $I$  with a namespace or type, then the *namespace\_or\_type\_name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *namespace\_or\_type\_name* refers to the namespace named  $I$  in  $N$ .
  - Otherwise, if  $N$  contains an accessible type having name  $I$  and  $K$  type parameters, then:
    - If  $K$  is zero and the location where the *namespace\_or\_type\_name* occurs is enclosed by a namespace declaration for  $N$  and the namespace declaration contains an *extern\_alias\_directive* or *using\_alias\_directive* that associates the name  $I$  with a namespace or type, **or any namespace declaration for  $N$  in the program contains a *global\_using\_alias\_directive*** that associates the name  $I$  with a namespace or type, then the *namespace\_or\_type\_name* is ambiguous and a compile-time error occurs.

- Otherwise, the *namespace\_or\_type\_name* refers to the type constructed with the given type arguments.
- Otherwise, if the location where the *namespace\_or\_type\_name* occurs is enclosed by a namespace declaration for **N**:
  - If **K** is zero and the namespace declaration contains an *extern\_alias\_directive* or *using\_alias\_directive* that associates the name **I** with an imported namespace or type, **or any namespace declaration for N in the program contains a global\_using\_alias\_directive that associates the name I with an imported namespace or type**, then the *namespace\_or\_type\_name* refers to that namespace or type.
  - Otherwise, if the namespaces and type declarations imported by the *using\_namespace\_directives* and *using\_alias\_directives* of the namespace declaration **and the namespaces and type declarations imported by the global\_using\_namespace\_directives and global\_using\_static\_directives of any namespace declaration for N in the program** contain exactly one accessible type having name **I** and **K** type parameters, then the *namespace\_or\_type\_name* refers to that type constructed with the given type arguments.
  - Otherwise, if the namespaces and type declarations imported by the *using\_namespace\_directives* and *using\_alias\_directives* of the namespace declaration **and the namespaces and type declarations imported by the global\_using\_namespace\_directives and global\_using\_static\_directives of any namespace declaration for N in the program** contain more than one accessible type having name **I** and **K** type parameters, then the *namespace\_or\_type\_name* is ambiguous and an error occurs.
- Otherwise, the *namespace\_or\_type\_name* is undefined and a compile-time error occurs.

## Simple names §11.7.4 ↗

Changes are made to the *simple\_name* evaluation rules as follows.

This is the relevant bullet point with proposed additions (which are in **bold**):

- Otherwise, for each namespace **N**, starting with the namespace in which the *simple\_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
  - If **K** is zero and **I** is the name of a namespace in **N**, then:

- If the location where the *simple\_name* occurs is enclosed by a namespace declaration for *N* and the namespace declaration contains an *extern\_alias\_directive* or *using\_alias\_directive* that associates the name *I* with a namespace or type, **or any namespace declaration for *N* in the program contains a *global\_using\_alias\_directive*** that associates the name *I* with a namespace or type, then the *simple\_name* is ambiguous and a compile-time error occurs.
- Otherwise, the *simple\_name* refers to the namespace named *I* in *N*.
- Otherwise, if *N* contains an accessible type having name *I* and *K* type parameters, then:
  - If *K* is zero and the location where the *simple\_name* occurs is enclosed by a namespace declaration for *N* and the namespace declaration contains an *extern\_alias\_directive* or *using\_alias\_directive* that associates the name *I* with a namespace or type, **or any namespace declaration for *N* in the program contains a *global\_using\_alias\_directive*** that associates the name *I* with a namespace or type, then the *simple\_name* is ambiguous and a compile-time error occurs.
  - Otherwise, the *namespace\_or\_type\_name* refers to the type constructed with the given type arguments.
- Otherwise, if the location where the *simple\_name* occurs is enclosed by a namespace declaration for *N*:
  - If *K* is zero and the namespace declaration contains an *extern\_alias\_directive* or *using\_alias\_directive* that associates the name *I* with an imported namespace or type, **or any namespace declaration for *N* in the program contains a *global\_using\_alias\_directive*** that associates the name *I* with an imported namespace or type, then the *simple\_name* refers to that namespace or type.
  - Otherwise, if the namespaces and type declarations imported by the *using\_namespace\_directives* and *using\_static\_directives* of the namespace declaration **and the namespaces and type declarations imported by the *global\_using\_namespace\_directives* and *global\_using\_static\_directives* of any namespace declaration for *N* in the program** contain exactly one accessible type or non-extension static member having name *I* and *K* type parameters, then the *simple\_name* refers to that type or member constructed with the given type arguments.
  - Otherwise, if the namespaces and types imported by the *using\_namespace\_directives* of the namespace declaration **and the namespaces and type declarations imported by the *global\_using\_namespace\_directives* and *global\_using\_static\_directives* of**

any namespace declaration for `N` in the program contain more than one accessible type or non-extension-method static member having name `I` and `K` type parameters, then the *simple\_name* is ambiguous and an error occurs.

## Extension method invocations §11.7.8.3 ↗

Changes are made to the algorithm to find the best *type\_name* `C` as follows. This is the relevant bullet point with proposed additions (which are **in bold**):

- Starting with the closest enclosing namespace declaration, continuing with each enclosing namespace declaration, and ending with the containing compilation unit, successive attempts are made to find a candidate set of extension methods:
  - If the given namespace or compilation unit directly contains non-generic type declarations `Ci` with eligible extension methods `Mj`, then the set of those extension methods is the candidate set.
  - If types `Ci` imported by *using\_static\_declarations* and directly declared in namespaces imported by *using\_namespace\_directives* in the given namespace or compilation unit **and, if containing compilation unit is reached, imported by global\_using\_static\_declarations and directly declared in namespaces imported by global\_using\_namespace\_directives in the program** directly contain eligible extension methods `Mj`, then the set of those extension methods is the candidate set.

## Compilation units §13.2 ↗

A *compilation\_unit* defines the overall structure of a source file. A compilation unit consists of **zero or more global\_using\_directives followed by zero or more using\_directives** followed by zero or more *global\_attributes* followed by zero or more *namespace\_member\_declarations*.

```
antlr
```

```
compilation_unit
    : extern_alias_directive* global_using_directive* using_directive*
      global_attributes? namespace_member_declaration*
    ;
```

A C# program consists of one or more compilation units, each contained in a separate source file. When a C# program is compiled, all of the compilation units are processed together. Thus, compilation units can depend on each other, possibly in a circular fashion.

The *global\_using\_directives* of a compilation unit affect the *global\_attributes* and *namespace\_member\_declarations* of all compilation units in the program.

## Extern aliases §13.4 ↗

The scope of an *extern\_alias\_directive* extends over the *global\_using\_directives*, *using\_directives*, *global\_attributes* and *namespace\_member\_declarations* of its immediately containing compilation unit or namespace body.

## Using alias directives §13.5.2 ↗

The order in which *using\_alias\_directives* are written has no significance, and resolution of the *namespace\_or\_type\_name* referenced by a *using\_alias\_directive* is not affected by the *using\_alias\_directive* itself or by other *using\_directives* in the immediately containing compilation unit or namespace body, **and, if the *using\_alias\_directive* is immediately contained in a compilation unit, is not affected by the *global\_using\_directives* in the program**. In other words, the *namespace\_or\_type\_name* of a *using\_alias\_directive* is resolved as if the immediately containing compilation unit or namespace body had no *using\_directives* **and, if the *using\_alias\_directive* is immediately contained in a compilation unit, the program had no *global\_using\_directives***. A *using\_alias\_directive* may however be affected by *extern\_alias\_directives* in the immediately containing compilation unit or namespace body.

## Global Using alias directives

A *global\_using\_alias\_directive* introduces an identifier that serves as an alias for a namespace or type within the program.

antlr

```
global_using_alias_directive
    : 'global' 'using' identifier '=' namespace_or_type_name ';'
    ;
```

Within member declarations in any compilation unit of a program that contains a *global\_using\_alias\_directive*, the identifier introduced by the *global\_using\_alias\_directive* can be used to reference the given namespace or type.

The *identifier* of a *global\_using\_alias\_directive* must be unique within the declaration space of any compilation unit of a program that contains the *global\_using\_alias\_directive*.

Just like regular members, names introduced by *global\_using\_alias\_directives* are hidden by similarly named members in nested scopes.

The order in which *global\_using\_alias\_directives* are written has no significance, and resolution of the *namespace\_or\_type\_name* referenced by a *global\_using\_alias\_directive* is not affected by the *global\_using\_alias\_directive* itself or by other *global\_using\_directives* or *using\_directives* in the program. In other words, the *namespace\_or\_type\_name* of a *global\_using\_alias\_directive* is resolved as if the immediately containing compilation unit had no *using\_directives* and the entire containing program had no *global\_using\_directives*. A *global\_using\_alias\_directive* may however be affected by *extern\_alias\_directives* in the immediately containing compilation unit.

A *global\_using\_alias\_directive* can create an alias for any namespace or type.

Accessing a namespace or type through an alias yields exactly the same result as accessing that namespace or type through its declared name.

Using aliases can name a closed constructed type, but cannot name an unbound generic type declaration without supplying type arguments.

## Global Using namespace directives

A *global\_using\_namespace\_directive* imports the types contained in a namespace into the program, enabling the identifier of each type to be used without qualification.

```
antlr

global_using_namespace_directive
: 'global' 'using' namespace_name ';'
;
```

Within member declarations in a program that contains a *global\_using\_namespace\_directive*, the types contained in the given namespace can be referenced directly.

A *global\_using\_namespace\_directive* imports the types contained in the given namespace, but specifically does not import nested namespaces.

Unlike a *global\_using\_alias\_directive*, a *global\_using\_namespace\_directive* may import types whose identifiers are already defined within a compilation unit of the program. In effect, in a given compilation unit, names imported by any

*global\_using\_namespace\_directive* in the program are hidden by similarly named members in the compilation unit.

When more than one namespace or type imported by *global\_using\_namespace\_directives* or *global\_using\_static\_directives* in the same program contain types by the same name, references to that name as a *type\_name* are considered ambiguous.

Furthermore, when more than one namespace or type imported by *global\_using\_namespace\_directives* or *global\_using\_static\_directives* in the same program contain types or members by the same name, references to that name as a *simple\_name* are considered ambiguous.

The *namespace\_name* referenced by a *global\_using\_namespace\_directive* is resolved in the same way as the *namespace\_or\_type\_name* referenced by a *global\_using\_alias\_directive*. Thus, *global\_using\_namespace\_directives* in the same program do not affect each other and can be written in any order.

## Global Using static directives

A *global\_using\_static\_directive* imports the nested types and static members contained directly in a type declaration into the containing program, enabling the identifier of each member and type to be used without qualification.

```
antlr

global_using_static_directive
    : 'global' 'using' 'static' type_name ';'
;
```

Within member declarations in a program that contains a *global\_using\_static\_directive*, the accessible nested types and static members (except extension methods) contained directly in the declaration of the given type can be referenced directly.

A *global\_using\_static\_directive* specifically does not import extension methods directly as static methods, but makes them available for extension method invocation.

A *global\_using\_static\_directive* only imports members and types declared directly in the given type, not members and types declared in base classes.

Ambiguities between multiple *global\_using\_namespace\_directives* and *global\_using\_static\_directives* are discussed in the section for *global\_using\_namespace\_directives* (above).

## Qualified alias member §13.8 ↗

Changes are made to the algorithm determining the meaning of a *qualified\_alias\_member* as follows.

This is the relevant bullet point with proposed additions (which are **in bold**):

- Otherwise, starting with the namespace declaration (§13.3 ↗) immediately containing the *qualified\_alias\_member* (if any), continuing with each enclosing namespace declaration (if any), and ending with the compilation unit containing the *qualified\_alias\_member*, the following steps are evaluated until an entity is located:
  - If the namespace declaration or compilation unit contains a *using\_alias\_directive* that associates **N** with a type, **or, when a compilation unit is reached, the program contains a global\_using\_alias\_directive that associates N with a type**, then the *qualified\_alias\_member* is undefined and a compile-time error occurs.
  - Otherwise, if the namespace declaration or compilation unit contains an *extern\_alias\_directive* or *using\_alias\_directive* that associates **N** with a namespace, **\*or, when a compilation unit is reached, the program contains a global\_using\_alias\_directive that associates N with a namespace**, then:
    - If the namespace associated with **N** contains a namespace named **I** and **K** is zero, then the *qualified\_alias\_member* refers to that namespace.
    - Otherwise, if the namespace associated with **N** contains a non-generic type named **I** and **K** is zero, then the *qualified\_alias\_member* refers to that type.
    - Otherwise, if the namespace associated with **N** contains a type named **I** that has **K** type parameters, then the *qualified\_alias\_member* refers to that type constructed with the given type arguments.
    - Otherwise, the *qualified\_alias\_member* is undefined and a compile-time error occurs.

# File Scoped Namespaces

Article • 06/23/2023

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

## Summary

File scoped namespaces use a less verbose format for the typical case of files containing only one namespace. The file scoped namespace format is `namespace X.Y.Z;` (note the semicolon and lack of braces). This allows for files like the following:

```
c#  
  
namespace X.Y.Z;  
  
using System;  
  
class X  
{  
}
```

The semantics are that using the `namespace X.Y.Z;` form is equivalent to writing `namespace X.Y.Z { ... }` where the remainder of the file following the file-scoped namespace is in the `...` section of a standard namespace declaration.

## Motivation

Analysis of the C# ecosystem shows that approximately 99.7% files are all of either one of these forms:

```
c#
```

```
namespace X.Y.Z
{
    // usings

    // types
}
```

or

```
c#

// usings

namespace X.Y.Z
{
    // types
}
```

However, both these forms force the user to indent the majority of their code and add a fair amount of ceremony for what is effectively a very basic concept. This affects clarity, uses horizontal and vertical space, and is often unsatisfying for users both used to C# and coming from other languages (which commonly have less ceremony here).

The primary goal of the feature therefore is to meet the needs of the majority of the ecosystem with less unnecessary boilerplate.

## Detailed design

This proposal takes the form of a diff to the existing compilation units ([§13.2 ↴](#)) section of the specification.

### Diff

~~A *compilation\_unit* defines the overall structure of a source file. A *compilation\_unit* consists of zero or more *using\_directives* followed by zero or more *global\_attributes* followed by zero or more *namespace\_member\_declarations*.~~

A *compilation\_unit* defines the overall structure of a source file. A *compilation\_unit* consists of zero or more *using\_directives* followed by zero or more *global\_attributes* followed by a *compilation\_unit\_body*. A *compilation\_unit\_body* can either be a *file\_scoped\_namespace\_declaration* or zero or more *statements* and *namespace\_member\_declarations*.

```
antlr
```

```
compilation_unit
~~   : extern_alias_directive* using_directive* global_attributes?
namespace_member_declaration*~~
      : extern_alias_directive* using_directive* global_attributes?
compilation_unit_body
;

compilation_unit_body
: statement* namespace_member_declaration*
| file_scoped_namespace_declaration
;
```

... unchanged ...

A *file\_scoped\_namespace\_declaration* will contribute members corresponding to the *namespace\_declaration* it is semantically equivalent to. See ([Namespace Declarations](#)) for more details.

## Namespace declarations

A *namespace\_declaration* consists of the keyword `namespace`, followed by a namespace name and body, optionally followed by a semicolon. A *file\_scoped\_namespace\_declaration* consists of the keyword `namespace`, followed by a namespace name, a semicolon and an optional list of *extern\_alias\_directives*, *using\_directives* and *type\_declarations*.

```
antlr
```

```
namespace_declaration
: 'namespace' qualified_identifier namespace_body ';'?
;

file_scoped_namespace_declaration
: 'namespace' qualified_identifier ';' extern_alias_directive*
using_directive* type_declaration*
;

... unchanged ...
```

... unchanged ...

the two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names `N1.N2.A` and `N1.N2.B`. Because

the two declarations contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

A *file\_scoped\_namespace\_declaration* permits a namespace declaration to be written without the { ... } block. For example:

```
C#  
  
extern alias A;  
namespace Name;  
using B;  
class C  
{  
}
```

is semantically equivalent to

```
C#  
  
extern alias A;  
namespace Name  
{  
    using B;  
    class C  
    {  
    }  
}
```

Specifically, a *file\_scoped\_namespace\_declaration* is treated the same as a *namespace\_declaration* at the same location in the *compilation\_unit* with the same *qualified\_identifier*. The *extern\_alias\_directives*, *using\_directives* and *type\_declarations* of that *file\_scoped\_namespace\_declaration* act as if they were declared in the same order inside the *namespace\_body* of that *namespace\_declaration*.

A source file cannot contain both a *file\_scoped\_namespace\_declaration* and a *namespace\_declaration*. A source file cannot contain multiple *file\_scoped\_namespace\_declarations*. A *compilation\_unit* cannot contain both a *file\_scoped\_namespace\_declaration* and any top level *statements*. *type\_declarations* cannot precede a *file\_scoped\_namespace\_declaration*.

## Extern aliases

... unchanged ...

# Extended property patterns

Article • 06/23/2023

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

## Summary

Allow property subpatterns to reference nested members, for instance:

C#

```
if (e is MethodCallExpression { Method.Name: "MethodName" })
```

Instead of:

C#

```
if (e is MethodCallExpression { Method: { Name: "MethodName" } })
```

## Motivation

When you want to match a child property, nesting another recursive pattern adds too much noise which will hurt readability with no real advantage.

## Detailed design

The [property\\_pattern](#) syntax is modified as follow:

diff

```

property_pattern
  : type? property_pattern_clause simple_designation?
  ;

property_pattern_clause
  : '{' (subpattern (',' subpattern)* ',' '?')? '}'
  ;

subpattern
- : identifier ':' pattern
+ : subpattern_name ':' pattern
  ;

+subpattern_name
+ : identifier
+ | subpattern_name '.' identifier
+ ;

```

The receiver for each name lookup is the type of the previous member  $T_0$ , starting from the *input type* of the *property\_pattern*. if  $T$  is a nullable type,  $T_0$  is its underlying type, otherwise  $T_0$  is equal to  $T$ .

For example, a pattern of the form `{ Prop1.Prop2: pattern }` is exactly equivalent to `{ Prop1: { Prop2: pattern } }`.

Note that this will include the null check when  $T$  is a nullable value type or a reference type. This null check means that the nested properties available will be the properties of  $T_0$ , not of  $T$ .

Repeated member paths are allowed. The compilation of pattern matching can take advantage of common parts of patterns.

# Improved Interpolated Strings

Article • 03/21/2022

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

## Summary

We introduce a new pattern for creating and using interpolated string expressions to allow for efficient formatting and use in both general `string` scenarios and more specialized scenarios such as logging frameworks, without incurring unnecessary allocations from formatting the string in the framework.

## Motivation

Today, string interpolation mainly lowers down to a call to `string.Format`. This, while general purpose, can be inefficient for a number of reasons:

1. It boxes any struct arguments, unless the runtime has happened to introduce an overload of `string.Format` that takes exactly the correct types of arguments in exactly the correct order.
  - This ordering is why the runtime is hesitant to introduce generic versions of the method, as it would lead to combinatoric explosion of generic instantiations of a very common method.
2. It has to allocate an array for the arguments in most cases.
3. There is no opportunity to avoid instantiating the instance if it's not needed.  
Logging frameworks, for example, will recommend avoiding string interpolation because it will cause a string to be realized that may not be needed, depending on the current log-level of the application.

4. It can never use `Span` or other ref struct types today, because ref structs are not allowed as generic type parameters, meaning that if a user wants to avoid copying to intermediate locations they have to manually format strings.

Internally, the runtime has a type called `ValueStringBuilder` to help deal with the first 2 of these scenarios. They pass a stackalloc'd buffer to the builder, repeatedly call `AppendFormat` with every part, and then get a final string out. If the resulting string goes past the bounds of the stack buffer, they can then move to an array on the heap. However, this type is dangerous to expose directly, as incorrect usage could lead to a rented array to be double-disposed, which then will cause all sorts of undefined behavior in the program as two locations think they have sole access to the rented array. This proposal creates a way to use this type safely from native C# code by just writing an interpolated string literal, leaving written code unchanged while improving every interpolated string that a user writes. It also extends this pattern to allow for interpolated strings passed as arguments to other methods to use a handler pattern, defined by receiver of the method, that will allow things like logging frameworks to avoid allocating strings that will never be needed, and giving C# users familiar, convenient interpolation syntax.

## Detailed Design

### The handler pattern

We introduce a new handler pattern that can represent an interpolated string passed as an argument to a method. The simple English of the pattern is as follows:

When an *interpolated\_string\_expression* is passed as an argument to a method, we look at the type of the parameter. If the parameter type has a constructor that can be invoked with 2 int parameters, `literalLength` and `formattedCount`, optionally takes additional parameters specified by an attribute on the original parameter, optionally has an out boolean trailing parameter, and the type of the original parameter has instance `AppendLiteral` and `AppendFormatted` methods that can be invoked for every part of the interpolated string, then we lower the interpolation using that, instead of into a traditional call to `string.Format(formatStr, args)`. A more concrete example is helpful for picturing this:

C#

```
// The handler that will actually "build" the interpolated string
[InterpolatedStringHandler]
public ref struct TraceLoggerParamsInterpolatedStringHandler
```

```

{
    // Storage for the built-up string

    private bool _logLevelEnabled;

    public TraceLoggerParamsInterpolatedStringHandler(int literalLength, int
formattedCount, Logger logger, out bool handlerIsValid)
    {
        if (!logger._logLevelEnabled)
        {
            handlerIsValid = false;
            return;
        }

        handlerIsValid = true;
        _logLevelEnabled = logger.EnabledLevel;
    }

    public void AppendLiteral(string s)
    {
        // Store and format part as required
    }

    public void AppendFormatted<T>(T t)
    {
        // Store and format part as required
    }
}

// The logger class. The user has an instance of this, accesses it via
static state, or some other access
// mechanism
public class Logger
{
    // Initialization code omitted
    public LogLevel EnabledLevel;

    public void
LogTrace([InterpolatedStringHandlerArguments("")]TraceLoggerParamsInterpolat
edStringHandler handler)
    {
        // Impl of logging
    }
}

Logger logger = GetLogger(LogLevel.Info);

// Given the above definitions, usage looks like this:
var name = "Fred Silberberg";
logger.LogTrace($"{name} will never be printed because info is < trace!");

// This is converted to:
var name = "Fred Silberberg";
var receiverTemp = logger;
var handler = new TraceLoggerParamsInterpolatedStringHandler(literalLength:

```

```

47, formattedCount: 1, receiverTemp, out var handlerIsValid);
if (handlerIsValid)
{
    handler.AppendFormatted(name);
    handler.AppendLiteral(" will never be printed because info is <
trace!");
}
receiverTemp.LogTrace(handler);

```

Here, because `TraceLoggerParamsInterpolatedStringHandler` has a constructor with the correct parameters, we say that the interpolated string has an implicit handler conversion to that parameter, and it lowers to the pattern shown above. The species needed for this is a bit complicated, and is expanded below.

The rest of this proposal will use `Append...` to refer to either of `AppendLiteral` or `AppendFormatted` in cases when both are applicable.

## New attributes

The compiler recognizes the

`System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute`:

C#

```

using System;
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
AllowMultiple = false, Inherited = false)]
    public sealed class InterpolatedStringHandlerAttribute : Attribute
    {
        public InterpolatedStringHandlerAttribute()
        {
        }
    }
}

```

This attribute is used by the compiler to determine if a type is a valid interpolated string handler type.

The compiler also recognizes the

`System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute`:

C#

```

namespace System.Runtime.CompilerServices
{

```

```

[AttributeUsage(AttributeTargets.Parameter, AllowMultiple = false,
Inherited = false)]
public sealed class InterpolatedStringHandlerArgumentAttribute : Attribute
{
    public InterpolatedHandlerArgumentAttribute(string argument);
    public InterpolatedHandlerArgumentAttribute(params string[]
arguments);

    public string[] Arguments { get; }
}
}

```

This attribute is used on parameters, to inform the compiler how to lower an interpolated string handler pattern used in a parameter position.

## Interpolated string handler conversion

Type `T` is said to be an *applicable\_interpolated\_string\_handler\_type* if it is attributed with `System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute`. There exists an implicit *interpolated\_string\_handler\_conversion* to `T` from an *interpolated\_string\_expression*, or an *additive\_expression* composed entirely of *\_interpolated\_string\_expression\_s* and using only `+` operators.

For simplicity in the rest of this speclet, *interpolated\_string\_expression* refers to both a simple *interpolated\_string\_expression*, and to an *additive\_expression* composed entirely of *\_interpolated\_string\_expression\_s* and using only `+` operators.

Note that this conversion always exists, regardless of whether there will be later errors when actually attempting to lower the interpolation using the handler pattern. This is done to help ensure that there are predictable and useful errors and that runtime behavior doesn't change based on the content of an interpolated string.

## Applicable function member adjustments

We adjust the wording of the applicable function member algorithm ([§11.6.4.2](#)) as follows (a new sub-bullet is added to each section, in bold):

A function member is said to be an ***applicable function member*** with respect to an argument list `A` when all of the following are true:

- Each argument in `A` corresponds to a parameter in the function member declaration as described in Corresponding parameters ([§11.6.2.2](#)), and any parameter to which no argument corresponds is an optional parameter.

- For each argument in `A`, the parameter passing mode of the argument (i.e., value, `ref`, or `out`) is identical to the parameter passing mode of the corresponding parameter, and
  - for a value parameter or a parameter array, an implicit conversion ([§10.2 ↴](#)) exists from the argument to the type of the corresponding parameter, or
  - for a `ref` parameter whose type is a struct type, an implicit *interpolated\_string\_handler\_conversion* exists from the argument to the type of the corresponding parameter, or
  - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter. After all, a `ref` or `out` parameter is an alias for the argument passed.

For a function member that includes a parameter array, if the function member is applicable by the above rules, it is said to be applicable in its *normal form*. If a function member that includes a parameter array is not applicable in its normal form, the function member may instead be applicable in its *expanded form*:

- The expanded form is constructed by replacing the parameter array in the function member declaration with zero or more value parameters of the element type of the parameter array such that the number of arguments in the argument list `A` matches the total number of parameters. If `A` has fewer arguments than the number of fixed parameters in the function member declaration, the expanded form of the function member cannot be constructed and is thus not applicable.
- Otherwise, the expanded form is applicable if for each argument in `A` the parameter passing mode of the argument is identical to the parameter passing mode of the corresponding parameter, and
  - for a fixed value parameter or a value parameter created by the expansion, an implicit conversion ([§10.2 ↴](#)) exists from the type of the argument to the type of the corresponding parameter, or
  - for a `ref` parameter whose type is a struct type, an implicit *interpolated\_string\_handler\_conversion* exists from the argument to the type of the corresponding parameter, or
  - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter.

Important note: this means that if there are 2 otherwise equivalent overloads, that only differ by the type of the *applicable\_interpolated\_string\_handler\_type*, these overloads will be considered ambiguous. Further, because we do not see through explicit casts, it is possible that there could arise an unresolvable scenario where both applicable overloads use `InterpolatedStringHandlerArguments` and are totally uncallable without manually performing the handler lowering pattern. We could potentially make changes

to the better function member algorithm to resolve this if we so choose, but this scenario unlikely to occur and isn't a priority to address.

## Better conversion from expression adjustments

We change the better conversion from expression ([§11.6.4.4](#)) section to the following:

Given an implicit conversion `c1` that converts from an expression `E` to a type `T1`, and an implicit conversion `c2` that converts from an expression `E` to a type `T2`, `c1` is a **better conversion** than `c2` if:

1. `E` is a non-constant *interpolated\_string\_expression*, `c1` is an *implicit\_string\_handler\_conversion*, `T1` is an *applicable\_interpolated\_string\_handler\_type*, and `c2` is not an *implicit\_string\_handler\_conversion*, or
2. `E` does not exactly match `T2` and at least one of the following holds:
  - `E` exactly matches `T1` ([§11.6.4.4](#))
  - `T1` is a better conversion target than `T2` ([§11.6.4.6](#))

This does mean that there are some potentially non-obvious overload resolution rules, depending on whether the interpolated string in question is a constant-expression or not. For example:

C#

```
void Log(string s) { ... }
void Log(TraceLoggerParamsInterpolatedStringHandler p) { ... }

Log(""); // Calls Log(string s), because "" is a constant expression
Log($"{{test}}"); // Calls Log(string s), because $"{{test}}" is a constant
expression
Log($"{1}"); // Calls Log(TraceLoggerParamsInterpolatedStringHandler p),
because $"{1}" is not a constant expression
```

This is introduced so that things that can simply be emitted as constants do so, and don't incur any overhead, while things that cannot be constant use the handler pattern.

## InterpolatedStringHandler and Usage

We introduce a new type in `System.Runtime.CompilerServices`:

`DefaultInterpolatedStringHandler`. This is a ref struct with many of the same semantics

as `ValueStringBuilder`, intended for direct use by the C# compiler. This struct would look approximately like this:

C#

```
// API Proposal issue: https://github.com/dotnet/runtime/issues/50601
namespace System.Runtime.CompilerServices
{
    [InterpolatedStringHandler]
    public ref struct DefaultInterpolatedStringHandler
    {
        public DefaultInterpolatedStringHandler(int literalLength, int
formattedCount);
        public string ToStringAndClear();

        public void AppendLiteral(string value);

        public void AppendFormatted<T>(T value);
        public void AppendFormatted<T>(T value, string? format);
        public void AppendFormatted<T>(T value, int alignment);
        public void AppendFormatted<T>(T value, int alignment, string?
format);

        public void AppendFormatted(ReadOnlySpan<char> value);
        public void AppendFormatted(ReadOnlySpan<char> value, int alignment
= 0, string? format = null);

        public void AppendFormatted(string? value);
        public void AppendFormatted(string? value, int alignment = 0,
string? format = null);

        public void AppendFormatted(object? value, int alignment = 0,
string? format = null);
    }
}
```

We make a slight change to the rules for the meaning of an *interpolated\_string\_expression* ([§11.7.3 ↴](#)):

If the type of an interpolated string is `string` and the type `System.Runtime.CompilerServices.DefaultInterpolatedStringHandler` exists, and the current context supports using that type, the `string` is lowered using the handler pattern. The final `string` value is then obtained by calling `ToStringAndClear()` on the handler type. Otherwise, if the type of an interpolated string is `System.IFormattable` or `System.FormattableString` [the rest is unchanged]

The "and the current context supports using that type" rule is intentionally vague to give the compiler leeway in optimizing usage of this pattern. The handler type is likely to be a ref struct type, and ref struct types are normally not permitted in async methods. For

this particular case, the compiler would be allowed to make use the handler if none of the interpolation holes contain an `await` expression, as we can statically determine that the handler type is safely used without additional complicated analysis because the handler will be dropped after the interpolated string expression is evaluated.

#### Open Question:

Do we want to instead just make the compiler know about `DefaultInterpolatedStringHandler` and skip the `string.Format` call entirely? It would allow us to hide a method that we don't necessarily want to put in people's faces when they manually call `string.Format`.

Answer: Yes.

#### Open Question:

Do we want to have handlers for `System.IFormattable` and `System.FormattableString` as well?

Answer: No.

## Handler pattern codegen

In this section, method invocation resolution refers to the steps listed in §11.7.8.2 ↗.

### Constructor resolution

Given an *applicable\_interpolated\_string\_handler\_type* `T` and an *interpolated\_string\_expression* `i`, method invocation resolution and validation for a valid constructor on `T` is performed as follows:

1. Member lookup for instance constructors is performed on `T`. The resulting method group is called `M`.
2. The argument list `A` is constructed as follows:
  - a. The first two arguments are integer constants, representing the literal length of `i`, and the number of *interpolation* components in `i`, respectively.
  - b. If `i` is used as an argument to some parameter `pi` in method `M1`, and parameter `pi` is attributed with `System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute`, then for every name `Argx` in the `Arguments` array of that attribute the compiler matches it to a parameter `px` that has the same name. The empty string is matched to the receiver of `M1`.

- If any `Argx` is not able to be matched to a parameter of `M1`, or an `Argx` requests the receiver of `M1` and `M1` is a static method, an error is produced and no further steps are taken.
  - Otherwise, the type of every resolved `px` is added to the argument list, in the order specified by the `Arguments` array. Each `px` is passed with the same `ref` semantics as is specified in `M1`.
- c. The final argument is a `bool`, passed as an `out` parameter.
3. Traditional method invocation resolution is performed with method group `M` and argument list `A`. For the purposes of method invocation final validation, the context of `M` is treated as a *member\_access* through type `T`.
- If a single-best constructor `F` was found, the result of overload resolution is `F`.
  - If no applicable constructors were found, step 3 is retried, removing the final `bool` parameter from `A`. If this retry also finds no applicable members, an error is produced and no further steps are taken.
  - If no single-best method was found, the result of overload resolution is ambiguous, an error is produced, and no further steps are taken.
4. Final validation on `F` is performed.
- If any element of `A` occurred lexically after `i`, an error is produced and no further steps are taken.
  - If any `A` requests the receiver of `F`, and `F` is an indexer being used as an *initializer\_target* in a *member\_initializer*, then an error is reported and no further steps are taken.
- Note: the resolution here intentionally do *not* use the actual expressions passed as other arguments for `Argx` elements. We only consider the types post-conversion. This makes sure that we don't have double-conversion issues, or unexpected cases where a lambda is bound to one delegate type when passed to `M1` and bound to a different delegate type when passed to `M`.
- Note: We report an error for indexers uses as member initializers because of the order of evaluation for nested member initializers. Consider this code snippet:

C#

```
var x1 = new C1 { C2 = { [GetString()] = { A = 2, B = 4 } } };

/* Lowering:
```

```

__c1 = new C1();
string argTemp = GetString();
__c1.C2[argTemp][1] = 2;
__c1.C2[argTemp][3] = 4;

Prints:
GetString
get_C2
get_C2
*/
}

string GetString()
{
    Console.WriteLine("GetString");
    return "";
}

class C1
{
    private C2 c2 = new C2();
    public C2 C2 { get { Console.WriteLine("get_C2"); return c2; } set { } }
}

class C2
{
    public C3 this[string s]
    {
        get => new C3();
        set { }
    }
}

class C3
{
    public int A
    {
        get => 0;
        set { }
    }
    public int B
    {
        get => 0;
        set { }
    }
}

```

The arguments to `__c1.C2[]` are evaluated *before* the receiver of the indexer. While we could come up with a lowering that works for this scenario (either by creating a temp for `__c1.C2` and sharing it across both indexer invocations, or only using it for the first indexer invocation and sharing the argument across both invocations) we think that any lowering would be confusing for what we believe is a pathological scenario. Therefore, we forbid the scenario entirely.

## ~~Open Question:~~

If we use a constructor instead of `create`, we'd improve runtime codegen, at the expense of narrowing the pattern a bit.

*Answer:* We will restrict to constructors for now. We can revisit adding a general `Create` method later if the scenario arises.

## Append... method overload resolution

Given an `applicable_interpolated_string_handler_type T` and an `interpolated_string_expression i`, overload resolution for a set of valid `Append...` methods on `T` is performed as follows:

1. If there are any `interpolated_regular_string_character` components in `i`:
  - a. Member lookup on `T` with the name `AppendLiteral` is performed. The resulting method group is called `M1`.
  - b. The argument list `A1` is constructed with one value parameter of type `string`.
  - c. Traditional method invocation resolution is performed with method group `M1` and argument list `A1`. For the purposes of method invocation final validation, the context of `M1` is treated as a *member\_access* through an instance of `T`.
    - If a single-best method `Fi` is found and no errors were produced, the result of method invocation resolution is `Fi`.
    - Otherwise, an error is reported.
2. For every `interpolation ix` component of `i`:
  - a. Member lookup on `T` with the name `AppendFormatted` is performed. The resulting method group is called `Mf`.
  - b. The argument list `Af` is constructed:
    - i. The first parameter is the `expression` of `ix`, passed by value.
    - ii. If `ix` directly contains a `constant_expression` component, then an integer value parameter is added, with the name `alignment` specified.
    - iii. If `ix` is directly followed by an `interpolation_format`, then a string value parameter is added, with the name `format` specified.
  - c. Traditional method invocation resolution is performed with method group `Mf` and argument list `Af`. For the purposes of method invocation final validation, the context of `Mf` is treated as a *member\_access* through an instance of `T`.

- If a single-best method `Fi` is found, the result of method invocation resolution is `Fi`.
- Otherwise, an error is reported.

3. Finally, for every `Fi` discovered in steps 1 and 2, final validation is performed:

- If any `Fi` does not return `bool` by value or `void`, an error is reported.
- If all `Fi` do not return the same type, an error is reported.

Note that these rules do not permit extension methods for the `Append...` calls. We could consider enabling that if we choose, but this is analogous to the enumerator pattern, where we allow `GetEnumerator` to be an extension method, but not `Current` or `MoveNext()`.

These rules *do* permit default parameters for the `Append...` calls, which will work with things like `CallerLineNumber` or `CallerArgumentExpression` (when supported by the language).

We have separate overload lookup rules for base elements vs interpolation holes because some handlers will want to be able to understand the difference between the components that were interpolated and the components that were part of the base string.

### **Open Question**

Some scenarios, like structured logging, want to be able to provide names for interpolation elements. For example, today a logging call might look like `Log("name") bought {itemCount} items", name, items.Count);`. The names inside the `{}` provide important structure information for loggers that help with ensuring output is consistent and uniform. Some cases might be able to reuse the `:format` component of an interpolation hole for this, but many loggers already understand format specifiers and have existing behavior for output formatting based on this info. Is there some syntax we can use to enable putting these named specifiers in?

Some cases may be able to get away with `CallerArgumentExpression`, provided that support does land in C# 10. But for cases that invoke a method/property, that may not be sufficient.

*Answer:*

While there are some interesting parts to templated strings we could explore in an orthogonal language feature, we don't think a specific syntax here has much benefit over solutions such as using a tuple: `$(("StructuredCategory", myExpression))`.

## Performing the conversion

Given an *applicable\_interpolated\_string\_handler\_type* `T` and an *interpolated\_string\_expression* `i` that had a valid constructor `Fc` and `Append...` methods `Fa` resolved, lowering for `i` is performed as follows:

1. Any arguments to `Fc` that occur lexically before `i` are evaluated and stored into temporary variables in lexical order. In order to preserve lexical ordering, if `i` occurred as part of a larger expression `e`, any components of `e` that occurred before `i` will be evaluated as well, again in lexical order.
2. `Fc` is called with the length of the interpolated string literal components, the number of *interpolation* holes, any previously evaluated arguments, and a `bool` out argument (if `Fc` was resolved with one as the last parameter). The result is stored into a temporary value `ib`.
  - a. The length of the literal components is calculated after replacing any *open\_brace\_escape\_sequence* with a single `{`, and any *close\_brace\_escape\_sequence* with a single `}`.
3. If `Fc` ended with a `bool` out argument, a check on that `bool` value is generated. If true, the methods in `Fa` will be called. Otherwise, they will not be called.
4. For every `Fax` in `Fa`, `Fax` is called on `ib` with either the current literal component or *interpolation* expression, as appropriate. If `Fax` returns a `bool`, the result is logically anded with all preceding `Fax` calls.
  - a. If `Fax` is a call to `AppendLiteral`, the literal component is unescaped by replacing any *open\_brace\_escape\_sequence* with a single `{`, and any *close\_brace\_escape\_sequence* with a single `}`.
5. The result of the conversion is `ib`.

Again, note that arguments passed to `Fc` and arguments passed to `e` are the same temp. Conversions may occur on top of the temp to convert to a form that `Fc` requires, but for example lambdas cannot be bound to a different delegate type between `Fc` and `e`.

### Open Question

This lowering means that subsequent parts of the interpolated string after a false-returning `Append...` call don't get evaluated. This could potentially be very confusing, particularly if the format hole is side-effecting. We could instead evaluate all format holes first, then repeatedly call `Append...` with the results, stopping if it returns false. This would ensure that all expressions get evaluated as one might expect, but we call as

few methods as we need to. While the partial evaluation might be desirable for some more advanced cases, it is perhaps non-intuitive for the general case.

Another alternative, if we want to always evaluate all format holes, is to remove the `Append...` version of the API and just do repeated `Format` calls. The handler can track whether it should just be dropping the argument and immediately returning for this version.

*Answer:* We will have conditional evaluation of the holes.

### Open Question

Do we need to dispose of disposable handler types, and wrap calls with try/finally to ensure that Dispose is called? For example, the interpolated string handler in the bcl might have a rented array inside it, and if one of the interpolation holes throws an exception during evaluation, that rented array could be leaked if it wasn't disposed.

*Answer:* No. handlers can be assigned to locals (such as `MyHandler handler = $" {MyCode()}";`), and the lifetime of such handlers is unclear. Unlike foreach enumerators, where the lifetime is obvious and no user-defined local is created for the enumerator.

## Impact on nullable reference types

To minimize complexity of the implementation, we have a few limitations on how we perform nullable analysis on interpolated string handler constructors used as arguments to a method or indexer. In particular, we do not flow information from the constructor back through to the original slots of parameters or arguments from the original context, and we do not use constructor parameter types to inform generic type inference for type parameters in the containing method. An example of where this can have an impact is:

C#

```
string s = "";
C c = new C();
c.M(s, $$", c.ToString(), s.ToString()); // No warnings on c.ToString() or
s.ToString(), as the `MaybeNull` does not flow back.

public class C
{
    public void M(string s1, [InterpolatedStringHandlerArgument("", "s1")]
CustomHandler c1, string s2, string s3) { }

[InterpolatedStringHandler]
public partial struct CustomHandler
```

```
{  
    public CustomHandler(int literalLength, int formattedCount, [MaybeNull]  
C c, [MaybeNull] string s) : this()  
    {  
    }  
}
```

C#

```
string? s = null;  
M(s, ""); // Infers `string` for `T` because of the `T?` parameter, not  
`string?`, as flow analysis does not consider the unannotated `T` parameter  
of the constructor  
  
void M<T>(T? t, [InterpolatedStringHandlerArgument("s1")] CustomHandler<T>  
c) {}  
  
[InterpolatedStringHandler]  
public partial struct CustomHandler<T>  
{  
    public CustomHandler(int literalLength, int formattedCount, T t) :  
this()  
    {  
    }  
}
```

## Other considerations

### Allow `string` types to be convertible to handlers as well

For type author simplicity, we could consider allowing expressions of type `string` to be implicitly-convertible to *applicable\_interpolated\_string\_handler\_types*. As proposed today, authors will likely need to overload on both that handler type and regular `string` types, so their users don't have to understand the difference. This may be an annoying and non-obvious overhead, as a `string` expression can be viewed as an interpolation with `expression.Length` prefilled length and 0 holes to be filled.

This would allow new APIs to only expose a handler, without also having to expose a `string`-accepting overload. However, it won't get around the need for changes to better conversion from expression, so while it would work it may be unnecessary overhead.

*Answer:*

We think that this could end up being confusing, and there's an easy workaround for custom handler types: add a user-defined conversion from string.

## Incorporating spans for heap-less strings

`ValueStringBuilder` as it exists today has 2 constructors: one that takes a count, and allocates on the heap eagerly, and one that takes a `Span<char>`. That `Span<char>` is usually a fixed size in the runtime codebase, around 250 elements on average. To truly replace that type, we should consider an extension to this where we also recognize `GetInterpolatedString` methods that take a `Span<char>`, instead of just the count version. However, we see a few potential thorny cases to resolve here:

- We don't want to `stackalloc` repeatedly in a hot loop. If we were to do this extension to the feature, we'd likely want to share the `stackalloc`'d span between loop iterations. We know this is safe, as `Span<T>` is a ref struct that can't be stored on the heap, and users would have to be pretty devious to manage to extract a reference to that `Span` (such as creating a method that accepts such a handler then deliberately retrieving the `Span` from the handler and returning it to the caller). However, allocating ahead of time produces other questions:
  - Should we eagerly `stackalloc`? What if the loop is never entered, or exits before it needs the space?
  - If we don't eagerly `stackalloc`, does that mean we introduce a hidden branch on every loop? Most loops likely won't care about this, but it could affect some tight loops that don't want to pay the cost.
- Some strings can be quite big, and the appropriate amount to `stackalloc` is dependent on a number of factors, including runtime factors. We don't really want the C# compiler and specification to have to determine this ahead of time, so we'd want to resolve <https://github.com/dotnet/runtime/issues/25423> and add an API for the compiler to call in these cases. It also adds more pros and cons to the points from the previous loop, where we don't want to potentially allocate large arrays on the heap many times or before one is needed.

*Answer:*

This is out of scope for C# 10. We can look at this in general when we look at the more general `params Span<T>` feature.

## Non-try version of the API

For simplicity, this spec currently just proposes recognizing a `Append...` method, and things that always succeed (like `InterpolatedStringHandler`) would always return true

from the method. This was done to support partial formatting scenarios where the user wants to stop formatting if an error occurs or if it's unnecessary, such as the logging case, but could potentially introduce a bunch of unnecessary branches in standard interpolated string usage. We could consider an addendum where we use just `FormatX` methods if no `Append...` method is present, but it does present questions about what we do if there's a mix of both `Append...` and `FormatX` calls.

*Answer:*

We want the non-try version of the API. The proposal has been updated to reflect this.

## Passing previous arguments to the handler

There is unfortunate lack of symmetry in the proposal at it currently exists: invoking an extension method in reduced form produces different semantics than invoking the extension method in normal form. This is different from most other locations in the language, where reduced form is just a sugar. We propose adding an attribute to the framework that we will recognize when binding a method, that informs the compiler that certain parameters should be passed to the constructor on the handler. Usage looks like this:

C#

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Parameter, AllowMultiple = false,
    Inherited = false)]
    public sealed class InterpolatedStringHandlerArgumentAttribute : Attribute
    {
        public InterpolatedStringHandlerArgumentAttribute(string argument);
        public InterpolatedStringHandlerArgumentAttribute(params string[]
    arguments);

        public string[] Arguments { get; }
    }
}
```

Usage of this is then:

C#

```
namespace System
{
    public sealed class String
    {
```

```

        public static string Format(IFormatProvider? provider,
[InterpolatedStringHandlerArgument("provider")] ref
DefaultInterpolatedStringHandler handler);
        ...
    }
}

namespace System.Runtime.CompilerServices
{
    public ref struct DefaultInterpolatedStringHandler
    {
        public DefaultInterpolatedStringHandler(int baseLength, int
holeCount, IFormatProvider? provider); // additional factory
        ...
    }
}

var formatted = string.Format(CultureInfo.InvariantCulture, $"{X} = {Y}");

// Is lowered to

var tmp1 = CultureInfo.InvariantCulture;
var handler = new DefaultInterpolatedStringHandler(3, 2, tmp1);
handler.AppendFormatted(X);
handler.AppendLiteral(" = ");
handler.AppendFormatted(Y);
var formatted = string.Format(tmp1, handler);

```

The questions we need to answer:

1. Do we like this pattern in general?
2. Do we want to allow these arguments to come from after the handler parameter?  
Some existing patterns in the BCL, such as `Utf8Formatter`, put the value to be formatted *before* the thing needed to format into. To fit in best with these patterns, we'd likely want to allow this, but we need to decide if this out-of-order evaluate is ok.

*Answer:*

We want to support this. The spec has been updated to reflect this. Arguments will be required to be specified in lexical order at the call site, and if a needed argument to the create method is specified after the interpolated string literal, an error is produced.

## await usage in interpolation holes

Because `($"{await A()}"` is a valid expression today, we need to rationalize how interpolation holes with await. We could solve this with a few rules:

1. If an interpolated string used as a `string`, `IFormattable`, or `FormattableString` has an `await` in an interpolation hole, fall back to old-style formatter.
2. If an interpolated string is subject to an *implicit\_string\_handler\_conversion* and *applicable\_interpolated\_string\_handler\_type* is a `ref struct`, `await` is not allowed to be used in the format holes.

Fundamentally, this desugaring could use a ref struct in an async method as long as we guarantee that the `ref struct` will not need to be saved to the heap, which should be possible if we forbid `awaits` in the interpolation holes.

Alternatively, we could simply make all handler types non-ref structs, including the framework handler for interpolated strings. This would, however, preclude us from someday recognizing a `Span` version that does not need to allocate any scratch space at all.

*Answer:*

We will treat interpolated string handlers the same as any other type: this means that if the handler type is a ref struct and the current context doesn't allow the usage of ref structs, it is illegal to use handler here. The spec around lowering of string literals used as strings is intentionally vague to allow the compiler to decide on what rules it deems appropriate, but for custom handler types they will have to follow the same rules as the rest of the language.

## Handlers as ref parameters

Some handlers might want to be passed as ref parameters (either `in` or `ref`). Should we allow either? And if so, what will a `ref` handler look like? `ref $""` is confusing, as you're not actually passing the string by ref, you're passing the handler that is created from the ref by ref, and has similar potential issues with async methods.

*Answer:*

We want to support this. The spec has been updated to reflect this. The rules should reflect the same rules that apply to extension methods on value types.

## Interpolated strings through binary expressions and conversions

Because this proposal makes interpolated strings context sensitive, we would like to allow the compiler to treat a binary expression composed entirely of interpolated

strings, or an interpolated string subjected to a cast, as an interpolated string literal for the purposes of overload resolution. For example, take the following scenario:

```
C#  
  
struct Handler1  
{  
    public Handler1(int literalLength, int formattedCount, C c) => ...;  
    // AppendX... methods as necessary  
}  
struct Handler2  
{  
    public Handler2(int literalLength, int formattedCount, C c) => ...;  
    // AppendX... methods as necessary  
}  
  
class C  
{  
    void M(Handler1 handler) => ...;  
    void M(Handler2 handler) => ...;  
}  
  
c.M(${"X"}); // Ambiguous between the M overloads
```

This would be ambiguous, necessitating a cast to either `Handler1` or `Handler2` in order to resolve. However, in making that cast, we would potentially throw away the information that there is context from the method receiver, meaning that the cast would fail because there is nothing to fill in the information of `c`. A similar issue arises with binary concatenation of strings: the user could want to format the literal across several lines to avoid line wrapping, but would not be able to because that would no longer be an interpolated string literal convertible to the handler type.

To resolve these cases, we make the following changes:

- An *additive\_expression* composed entirely of *interpolated\_string\_expressions* and using only `+` operators is considered to be an *interpolated\_string\_literal* for the purposes of conversions and overload resolution. The final interpolated string is created by logically concatenating all individual *interpolated\_string\_expression* components, from left to right.
- A *cast\_expression* or a *relational\_expression* with operator `as` whose operand is an *interpolated\_string\_expressions* is considered an *interpolated\_string\_expressions* for the purposes of conversions and overload resolution.

Open Questions:

Do we want to do this? We don't do this for `System.FormattableString`, for example, but that can be broken out onto a different line, whereas this can be context-dependent and therefore not able to be broken out into a different line. There are also no overload resolution concerns with `FormattableString` and `IFormattable`.

*Answer:*

We think that this is a valid use case for additive expressions, but that the cast version is not compelling enough at this time. We can add it later if necessary. The spec has been updated to reflect this decision.

## Other use cases

See <https://github.com/dotnet/runtime/issues/50635> for examples of proposed handler APIs using this pattern.

# Constant Interpolated Strings

Article • 06/23/2023

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

## Summary

Enables constants to be generated from interpolated strings of type string constant.

## Motivation

The following code is already legal:

```
public class C
{
    const string S1 = "Hello world";
    const string S2 = "Hello" + " " + "World";
    const string S3 = S1 + " Kevin, welcome to the team!";
}
```

However, there have been many community requests to make the following also legal:

```
public class C
{
    const string S1 = $"Hello world";
    const string S2 = $"Hello{ " " }World";
    const string S3 = $"{S1} Kevin, welcome to the team!";
}
```

This proposal represents the next logical step for constant string generation, where existing string syntax that works in other situations is made to work for constants.

## Detailed design

The following represent the updated specifications for constant expressions under this new proposal. Current specifications from which this was directly based on can be found in §11.20 ↴.

## Constant Expressions

A *constant\_expression* is an expression that can be fully evaluated at compile-time.

```
antlr

constant_expression
    : expression
    ;
```

A constant expression must be the `null` literal or a value with one of the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `object`, `string`, or any enumeration type. Only the following constructs are permitted in constant expressions:

- Literals (including the `null` literal).
- References to `const` members of class and struct types.
- References to members of enumeration types.
- References to `const` parameters or local variables
- Parenthesized sub-expressions, which are themselves constant expressions.
- Cast expressions, provided the target type is one of the types listed above.
- `checked` and `unchecked` expressions
- Default value expressions
- Nameof expressions
- The predefined `+`, `-`, `!`, and `~` unary operators.
- The predefined `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, and `>=` binary operators, provided each operand is of a type listed above.
- The `?:` conditional operator.
- *Interpolated strings*  `${ }` , provided that all components are constant expressions of type `string` and all interpolated components lack alignment and format specifiers.

The following conversions are permitted in constant expressions:

- Identity conversions
- Numeric conversions
- Enumeration conversions
- Constant expression conversions
- Implicit and explicit reference conversions, provided that the source of the conversions is a constant expression that evaluates to the null value.

Other conversions including boxing, unboxing and implicit reference conversions of non-null values are not permitted in constant expressions. For example:

C#

```
class C
{
    const object i = 5;           // error: boxing conversion not permitted
    const object str = "hello"; // error: implicit reference conversion
}
```

the initialization of `i` is an error because a boxing conversion is required. The initialization of `str` is an error because an implicit reference conversion from a non-null value is required.

Whenever an expression fulfills the requirements listed above, the expression is evaluated at compile-time. This is true even if the expression is a sub-expression of a larger expression that contains non-constant constructs.

The compile-time evaluation of constant expressions uses the same rules as run-time evaluation of non-constant expressions, except that where run-time evaluation would have thrown an exception, compile-time evaluation causes a compile-time error to occur.

Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur in integral-type arithmetic operations and conversions during the compile-time evaluation of the expression always cause compile-time errors ([§11.20](#)).

Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile-time.

- Constant declarations ([§14.4](#)).
- Enumeration member declarations ([§18.4](#)).
- Default arguments of formal parameter lists ([§14.6.2](#))
- `case` labels of a `switch` statement ([§12.8.3](#)).
- `goto case` statements ([§12.10.4](#)).

- Dimension lengths in an array creation expression ([§11.7.15.5](#)) that includes an initializer.
- Attributes ([§21](#)).

An implicit constant expression conversion ([§10.2.11](#)) permits a constant expression of type `int` to be converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the constant expression is within the range of the destination type.

## Drawbacks

This proposal adds additional complexity to the compiler in exchange for broader applicability of interpolated strings. As these strings are fully evaluated at compile time, the valuable automatic formatting features of interpolated strings are less necessary. Most use cases can be largely replicated through the alternatives below.

## Alternatives

The current `+` operator for string concatenation can combine strings in a similar manner to the current proposal.

## Unresolved questions

What parts of the design are still undecided?

## Design meetings

Link to design notes that affect this proposal, and describe in one sentence for each what changes they led to.

# Lambda improvements

Article • 06/23/2023

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

## Summary

Proposed changes:

1. Allow lambdas with attributes
2. Allow lambdas with explicit return type
3. Infer a natural delegate type for lambdas and method groups

## Motivation

Support for attributes on lambdas would provide parity with methods and local functions.

Support for explicit return types would provide symmetry with lambda parameters where explicit types can be specified. Allowing explicit return types would also provide control over compiler performance in nested lambdas where overload resolution must bind the lambda body currently to determine the signature.

A natural type for lambda expressions and method groups will allow more scenarios where lambdas and method groups may be used without an explicit delegate type, including as initializers in `var` declarations.

Requiring explicit delegate types for lambdas and method groups has been a friction point for customers, and has become an impediment to progress in ASP.NET with recent work on [MapAction](#).

ASP.NET MapAction [↗](#) without proposed changes (`MapAction()` takes a `System.Delegate` argument):

```
C#  
  
[HttpGet("/")] Todo GetTodo() => new(Id: 0, Name: "Name");  
app.MapAction((Func<Todo>)GetTodo);  
  
[HttpPost("/")] Todo PostTodo([FromBody] Todo todo) => todo;  
app.MapAction((Func<Todo, Todo>)PostTodo);
```

ASP.NET MapAction [↗](#) with natural types for method groups:

```
C#  
  
[HttpGet("/")] Todo GetTodo() => new(Id: 0, Name: "Name");  
app.MapAction(GetTodo);  
  
[HttpPost("/")] Todo PostTodo([FromBody] Todo todo) => todo;  
app.MapAction(PostTodo);
```

ASP.NET MapAction [↗](#) with attributes and natural types for lambda expressions:

```
C#  
  
app.MapAction([HttpGet("/")]) () => new Todo(Id: 0, Name: "Name"));  
app.MapAction([HttpPost("/")]) ([FromBody] Todo todo) => todo);
```

## Attributes

Attributes may be added to lambda expressions and lambda parameters. To avoid ambiguity between method attributes and parameter attributes, a lambda expression with attributes must use a parenthesized parameter list. Parameter types are not required.

```
C#  
  
f = [A] () => { };           // [A] lambda  
f = [return:A] x => x;       // syntax error at '=>'  
f = [return:A] (x) => x;     // [A] lambda  
f = [A] static x => x;      // syntax error at '=>'  
  
f = ([A] x) => x;           // [A] x  
f = ([A] ref int x) => x;   // [A] x
```

Multiple attributes may be specified, either comma-separated within the same attribute list or as separate attribute lists.

```
C#
```

```
var f = [A1, A2][A3] () => { }; // ok
var g = ([A1][A2, A3] int x) => x; // ok
```

Attributes are not supported for *anonymous methods* declared with `delegate {}` syntax.

```
C#
```

```
f = [A] delegate { return 1; }; // syntax error at 'delegate'
f = delegate ([A] int x) { return x; }; // syntax error at '['
```

The parser will look ahead to differentiate a collection initializer with an element assignment from a collection initializer with a lambda expression.

```
C#
```

```
var y = new C { [A] = x }; // ok: y[A] = x
var z = new C { [A] x => x }; // ok: z[0] = [A] x => x
```

The parser will treat `?[]` as the start of a conditional element access.

```
C#
```

```
x = b ? [A]; // ok
y = b ? [A] () => { } : z; // syntax error at '('
```

Attributes on the lambda expression or lambda parameters will be emitted to metadata on the method that maps to the lambda.

In general, customers should not depend on how lambda expressions and local functions map from source to metadata. How lambdas and local functions are emitted can, and has, changed between compiler versions.

The changes proposed here are targeted at the `Delegate` driven scenario. It should be valid to inspect the `MethodInfo` associated with a `Delegate` instance to determine the signature of the lambda expression or local function including any explicit attributes and additional metadata emitted by the compiler such as default parameters. This allows teams such as ASP.NET to make available the same behaviors for lambdas and local functions as ordinary methods.

# Explicit return type

An explicit return type may be specified before the parenthesized parameter list.

C#

```
f = T () => default;           // ok
f = short x => 1;             // syntax error at '='
f = ref int (ref int x) => ref x; // ok
f = static void (_) => { };     // ok
f = async await (await await) => await; // ok?
```

The parser will look ahead to differentiate a method call `T()` from a lambda expression `T () => e.`

Explicit return types are not supported for anonymous methods declared with `delegate { }` syntax.

C#

```
f = delegate int { return 1; };      // syntax error
f = delegate int (int x) { return x; }; // syntax error
```

Method type inference should make an exact inference from an explicit lambda return type.

C#

```
static void F<T>(Func<T, T> f) { ... }
F(int (i) => i); // Func<int, int>
```

Variance conversions are not allowed from lambda return type to delegate return type (matching similar behavior for parameter types).

C#

```
Func<Object> f1 = string () => null; // error
Func<Object?> f2 = object () => x;    // warning
```

The parser allows lambda expressions with `ref` return types within expressions without additional parentheses.

C#

```
d = ref int () => x; // d = (ref int () => x)
F(ref int () => x); // F((ref int () => x))
```

`var` cannot be used as an explicit return type for lambda expressions.

C#

```
class var { }

d = var (var v) => v;           // error: contextual keyword 'var' cannot
be used as explicit lambda return type
d = @var (var v) => v;           // ok
d = ref var (ref var v) => ref v; // error: contextual keyword 'var' cannot
be used as explicit lambda return type
d = ref @var (ref var v) => ref v; // ok
```

## Natural (function) type

An *anonymous function* expression ([§11.16](#)) (a *lambda expression* or an *anonymous method*) has a natural type if the parameters types are explicit and the return type is either explicit or can be inferred (see [§11.6.3.13](#)).

A *method group* has a natural type if all candidate methods in the method group have a common signature. (If the method group may include extension methods, the candidates include the containing type and all extension method scopes.)

The natural type of an anonymous function expression or method group is a *function\_type*. A *function\_type* represents a method signature: the parameter types and ref kinds, and return type and ref kind. Anonymous function expressions or method groups with the same signature have the same *function\_type*.

*Function\_types* are used in a few specific contexts only:

- implicit and explicit conversions
- method type inference ([§11.6.3](#)) and best common type ([§11.6.3.15](#))
- `var` initializers

A *function\_type* exists at compile time only: *function\_types* do not appear in source or metadata.

## Conversions

From a *function\_type* `F` there are implicit *function\_type* conversions:

- To a *function\_type*  $G$  if the parameters and return types of  $F$  are variance-convertible to the parameters and return type of  $G$
- To `System.MulticastDelegate` or base classes or interfaces of `System.MulticastDelegate`
- To `System.Linq.Expressions.Expression` or `System.Linq.Expressions.LambdaExpression`

Anonymous function expressions and method groups already have *conversions from expression* to delegate types and expression tree types (see anonymous function conversions [§10.7](#) and method group conversions [§10.8](#)). Those conversions are sufficient for converting to strongly-typed delegate types and expression tree types. The *function\_type* conversions above add *conversions from type* to the base types only:

`System.MulticastDelegate`, `System.Linq.Expressions.Expression`, etc.

There are no conversions to a *function\_type* from a type other than a *function\_type*. There are no explicit conversions for *function\_types* since *function\_types* cannot be referenced in source.

A conversion to `System.MulticastDelegate` or base type or interface realizes the anonymous function or method group as an instance of an appropriate delegate type. A conversion to `System.Linq.Expressions.Expression<TDelegate>` or base type realizes the lambda expression as an expression tree with an appropriate delegate type.

C#

```
Delegate d = delegate (object obj) { }; // Action<object>
Expression e = () => ""; // Expression<Func<string>>
object o = "".Clone(); // Func<object>
```

*Function\_type* conversions are not implicit or explicit standard conversions [§10.4](#) and are not considered when determining whether a user-defined conversion operator is applicable to an anonymous function or method group. From evaluation of user defined conversions [§10.5.3](#):

For a conversion operator to be applicable, it must be possible to perform a standard conversion ([§10.4](#)) from the source type to the operand type of the operator, and it must be possible to perform a standard conversion from the result type of the operator to the target type.

C#

```

class C
{
    public static implicit operator C(Delegate d) { ... }
}

C c;
c = () => 1;      // error: cannot convert lambda expression to type 'C'
c = (C)((() => 2)); // error: cannot convert lambda expression to type 'C'

```

A warning is reported for an implicit conversion of a method group to `object`, since the conversion is valid but perhaps unintentional.

C#

```

Random r = new Random();
object obj;
obj = r.NextDouble();           // warning: Converting method group to 'object'.
Did you intend to invoke the method?
obj = (object)r.NextDouble(); // ok

```

## Type inference

The existing rules for type inference are mostly unchanged (see [§11.6.3](#)). There are however a couple of changes below to specific phases of type inference.

### First phase

The first phase ([§11.6.3.2](#)) allows an anonymous function to bind to `Ti` even if `Ti` is not a delegate or expression tree type (perhaps a type parameter constrained to `System.Delegate` for instance).

For each of the method arguments `Ei`:

- If `Ei` is an anonymous function and `Ti` is a delegate type or expression tree type, an *explicit parameter type inference* is made from `Ei` to `Ti` and an *explicit return type inference* is made from `Ei` to `Ti`.
- Otherwise, if `Ei` has a type `U` and `xi` is a value parameter then a *lower-bound inference* is made from `U` to `Ti`.
- Otherwise, if `Ei` has a type `U` and `xi` is a `ref` or `out` parameter then an *exact inference* is made from `U` to `Ti`.
- Otherwise, no inference is made for this argument.

## Explicit return type inference

An *explicit return type inference* is made *from* an expression  $E$  to a type  $T$  in the following way:

- If  $E$  is an anonymous function with explicit return type  $Ur$  and  $T$  is a delegate type or expression tree type with return type  $Vr$  then an *exact inference* ([§11.6.3.9](#)) is made *from*  $Ur$  to  $Vr$ .

## Fixing

Fixing ([§11.6.3.12](#)) ensures other conversions are preferred over *function\_type* conversions. (Lambda expressions and method group expressions only contribute to lower bounds so handling of *function\_types* is needed for lower bounds only.)

An *unfixed* type variable  $xi$  with a set of bounds is *fixed* as follows:

- The set of *candidate types*  $Uj$  starts out as the set of all types in the set of bounds for  $xi$  where **function types are ignored in lower bounds if there are any types that are not function types.**
- We then examine each bound for  $xi$  in turn: For each exact bound  $U$  of  $xi$  all types  $Uj$  which are not identical to  $U$  are removed from the candidate set. For each lower bound  $U$  of  $xi$  all types  $Uj$  to which there is *not* an implicit conversion from  $U$  are removed from the candidate set. For each upper bound  $U$  of  $xi$  all types  $Uj$  from which there is *not* an implicit conversion to  $U$  are removed from the candidate set.
- If among the remaining candidate types  $Uj$  there is a unique type  $V$  from which there is an implicit conversion to all the other candidate types, then  $xi$  is fixed to  $V$ .
- Otherwise, type inference fails.

## Best common type

Best common type ([§11.6.3.15](#)) is defined in terms of type inference so the type inference changes above apply to best common type as well.

C#

```
var fs = new[] { (string s) => s.Length; (string s) => int.Parse(s) } //  
Func<string, int>[]
```

## var

Anonymous functions and method groups with function types can be used as initializers in `var` declarations.

C#

```
var f1 = () => default;           // error: cannot infer type
var f2 = x => x;                 // error: cannot infer type
var f3 = () => 1;                  // System.Func<int>
var f4 = string () => null;       // System.Func<string>
var f5 = delegate (object o) { }; // System.Action<object>

static void F1() { }
static void F1<T>(this T t) { }
static void F2(this string s) { }

var f6 = F1;      // error: multiple methods
var f7 = "".F1; // System.Action
var f8 = F2;      // System.Action<string>
```

Function types are not used in assignments to discards.

C#

```
d = () => 0; // ok
_ = () => 1; // error
```

## Delegate types

The delegate type for the anonymous function or method group with parameter types  $P_1, \dots, P_n$  and return type  $R$  is:

- if any parameter or return value is not by value, or there are more than 16 parameters, or any of the parameter types or return are not valid type arguments (say, `(int* p) => {}`), then the delegate is a synthesized `internal` anonymous delegate type with signature that matches the anonymous function or method group, and with parameter names `arg1, ..., argn` or `arg` if a single parameter;
- if  $R$  is `void`, then the delegate type is `System.Action<P1, ..., Pn>`;
- otherwise the delegate type is `System.Func<P1, ..., Pn, R>`.

The compiler may allow more signatures to bind to `System.Action<>` and `System.Func<>` types in the future (if `ref struct` types are allowed type arguments for instance).

`modopt()` or `modreq()` in the method group signature are ignored in the corresponding delegate type.

If two anonymous functions or method groups in the same compilation require synthesized delegate types with the same parameter types and modifiers and the same return type and modifiers, the compiler will use the same synthesized delegate type.

## Overload resolution

Better function member ([§11.6.4.3 ↴](#)) is updated to prefer members where none of the conversions and none of the type arguments involved inferred types from lambda expressions or method groups.

### Better function member

... Given an argument list `A` with a set of argument expressions `{E1, E2, ..., En}` and two applicable function members `Mp` and `Mq` with parameter types `{P1, P2, ..., Pn}` and `{Q1, Q2, ..., Qn}`, `Mp` is defined to be a **better function member** than `Mq` if

1. for each argument, the implicit conversion from `Ex` to `Px` is not a *function\_type\_conversion*, and
  - `Mp` is a non-generic method or `Mp` is a generic method with type parameters `{X1, X2, ..., Xp}` and for each type parameter `Xi` the type argument is inferred from an expression or from a type other than a *function\_type*, and
  - for at least one argument, the implicit conversion from `Ex` to `Qx` is a *function\_type\_conversion*, or `Mq` is a generic method with type parameters `{Y1, Y2, ..., Yq}` and for at least one type parameter `Yi` the type argument is inferred from a *function\_type*, or
2. for each argument, the implicit conversion from `Ex` to `Qx` is not better than the implicit conversion from `Ex` to `Px`, and for at least one argument, the conversion from `Ex` to `Px` is better than the conversion from `Ex` to `Qx`.

Better conversion from expression ([§11.6.4.4 ↴](#)) is updated to prefer conversions that did not involve inferred types from lambda expressions or method groups.

### Better conversion from expression

Given an implicit conversion `c1` that converts from an expression `E` to a type `T1`, and an implicit conversion `c2` that converts from an expression `E` to a type `T2`, `c1` is a ***better conversion*** than `c2` if:

1. `c1` is not a *function\_type\_conversion* and `c2` is a *function\_type\_conversion*, or
2. `E` is a non-constant *interpolated\_string\_expression*, `c1` is an *implicit\_string\_handler\_conversion*, `T1` is an *applicable\_interpolated\_string\_handler\_type*, and `c2` is not an *implicit\_string\_handler\_conversion*, or
3. `E` does not exactly match `T2` and at least one of the following holds:
  - `E` exactly matches `T1` ([§11.6.4.4](#))
  - `T1` is a better conversion target than `T2` ([§11.6.4.6](#))

## Syntax

antlr

```
lambda_expression
: modifier* identifier '>=' (block | expression)
| attribute_list* modifier* type? lambda_parameters '>=' (block |
expression)
;

lambda_parameters
: lambda_parameter
| '(' (lambda_parameter (',' lambda_parameter)*)? ')'
;

lambda_parameter
: identifier
| attribute_list* modifier* type? identifier equals_value_clause?
;
```

## Open issues

Should default values be supported for lambda expression parameters for completeness?

Should `System.Diagnostics.ConditionalAttribute` be disallowed on lambda expressions since there are few scenarios where a lambda expression could be used conditionally?

C#

```
([Conditional("DEBUG")] static (x, y) => Assert(x == y))(a, b); // ok?
```

Should the *function\_type* be available from the compiler API, in addition to the resulting delegate type?

Currently, the inferred delegate type uses `System.Action<>` or `System.Func<>` when parameter and return types are valid type arguments *and* there are no more than 16 parameters, and if the expected `Action<>` or `Func<>` type is missing, an error is reported. Instead, should the compiler use `System.Action<>` or `System.Func<>` regardless of arity? And if the expected type is missing, synthesize a delegate type otherwise?

# CallerArgumentExpression

Article • 06/23/2023

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

## Summary

Allow developers to capture the expressions passed to a method, to enable better error messages in diagnostic/testing APIs and reduce keystrokes.

## Motivation

When an assertion or argument validation fails, the developer wants to know as much as possible about where and why it failed. However, today's diagnostic APIs do not fully facilitate this. Consider the following method:

```
C#  
  
T Single<T>(this T[] array)  
{  
    Debug.Assert(array != null);  
    Debug.Assert(array.Length == 1);  
  
    return array[0];  
}
```

When one of the asserts fail, only the filename, line number, and method name will be provided in the stack trace. The developer will not be able to tell which assert failed from this information-- they will have to open the file and navigate to the provided line number to see what went wrong.

This is also the reason testing frameworks have to provide a variety of assert methods. With xUnit, `Assert.True` and `Assert.False` are not frequently used because they do not provide enough context about what failed.

While the situation is a bit better for argument validation because the names of invalid arguments are shown to the developer, the developer must pass these names to exceptions manually. If the above example were rewritten to use traditional argument validation instead of `Debug.Assert`, it would look like

```
C#  
  
T Single<T>(this T[] array)  
{  
    if (array == null)  
    {  
        throw new ArgumentNullException(nameof(array));  
    }  
  
    if (array.Length != 1)  
    {  
        throw new ArgumentException("Array must contain a single element.",  
            nameof(array));  
    }  
  
    return array[0];  
}
```

Notice that `nameof(array)` must be passed to each exception, although it's already clear from context which argument is invalid.

## Detailed design

In the above examples, including the string `"array != null"` or `"array.Length == 1"` in the assert message would help the developer determine what failed. Enter `CallerArgumentExpression`: it's an attribute the framework can use to obtain the string associated with a particular method argument. We would add it to `Debug.Assert` like so

```
C#  
  
public static class Debug  
{  
    public static void Assert(bool condition,  
        [CallerArgumentExpression("condition")] string message = null);  
}
```

The source code in the above example would stay the same. However, the code the compiler actually emits would correspond to

```
C#  
  
T Single<T>(this T[] array)  
{  
    Debug.Assert(array != null, "array != null");  
    Debug.Assert(array.Length == 1, "array.Length == 1");  
  
    return array[0];  
}
```

The compiler specially recognizes the attribute on `Debug.Assert`. It passes the string associated with the argument referred to in the attribute's constructor (in this case, `condition`) at the call site. When either assert fails, the developer will be shown the condition that was false and will know which one failed.

For argument validation, the attribute cannot be used directly, but can be made use of through a helper class:

```
C#  
  
public static class Verify  
{  
    public static void Argument(bool condition, string message,  
[CallerArgumentExpression("condition")] string conditionExpression = null)  
    {  
        if (!condition) throw new ArgumentException(message: message,  
paramName: conditionExpression);  
    }  
  
    public static void InRange(int argument, int low, int high,  
[CallerArgumentExpression("argument")] string argumentExpression =  
null,  
[CallerArgumentExpression("low")] string lowExpression = null,  
[CallerArgumentExpression("high")] string highExpression = null)  
    {  
        if (argument < low)  
        {  
            throw new ArgumentOutOfRangeException(paramName:  
argumentExpression,  
message: $"{argumentExpression} ({argument}) cannot be less  
than {lowExpression} ({low}).");  
        }  
  
        if (argument > high)  
        {  
            throw new ArgumentOutOfRangeException(paramName:  
argumentExpression,
```

```

        message: "${argumentExpression} ({argument}) cannot be
greater than {highExpression} ({high}).");
    }
}

public static void NotNull<T>(T argument,
[CallerArgumentExpression("argument")] string argumentExpression = null)
    where T : class
{
    if (argument == null) throw new ArgumentNullException(paramName:
argumentExpression);
}
}

T Single<T>(this T[] array)
{
    Verify.NotNull(array); // paramName: "array"
    Verify.Argument(array.Length == 1, "Array must contain a single
element."); // paramName: "array.Length == 1"

    return array[0];
}

T ElementAt(this T[] array, int index)
{
    Verify.NotNull(array); // paramName: "array"
    // paramName: "index"
    // message: "index (-1) cannot be less than 0 (0).", or
    //           "index (6) cannot be greater than array.Length - 1 (5)."
    Verify.InRange(index, 0, array.Length - 1);

    return array[index];
}

```

A proposal to add such a helper class to the framework is underway at <https://github.com/dotnet/corefx/issues/17068>. If this language feature was implemented, the proposal could be updated to take advantage of this feature.

## Extension methods

The `this` parameter in an extension method may be referenced by `CallerArgumentExpression`. For example:

C#

```

public static void ShouldBe<T>(this T @this, T expected,
[CallerArgumentExpression("this")] string thisExpression = null) {}

contestant.Points.ShouldBe(1337); // thisExpression: "contestant.Points"

```

`thisExpression` will receive the expression corresponding to the object before the dot. If it's called with static method syntax, e.g. `Ext.ShouldBe(contestant.Points, 1337)`, it will behave as if first parameter wasn't marked `this`.

There should always be an expression corresponding to the `this` parameter. Even if an instance of a class calls an extension method on itself, e.g. `this.Single()` from inside a collection type, the `this` is mandated by the compiler so "this" will get passed. If this rule is changed in the future, we can consider passing `null` or the empty string.

## Extra details

- Like the other `Caller*` attributes, such as `CallerMemberName`, this attribute may only be used on parameters with default values.
- Multiple parameters marked with `CallerArgumentExpression` are permitted, as shown above.
- The attribute's namespace will be `System.Runtime.CompilerServices`.
- If `null` or a string that is not a parameter name (e.g. `"notAParameterName"`) is provided, the compiler will pass in an empty string.
- The type of the parameter `CallerArgumentExpressionAttribute` is applied to must have a standard conversion from `string`. This means no user-defined conversions from `string` are allowed, and in practice means the type of such a parameter must be `string`, `object`, or an interface implemented by `string`.

## Drawbacks

- People who know how to use decompilers will be able to see some of the source code at call sites for methods marked with this attribute. This may be undesirable/unexpected for closed-source software.
- Although this is not a flaw in the feature itself, a source of concern may be that there exists a `Debug.Assert` API today that only takes a `bool`. Even if the overload taking a message had its second parameter marked with this attribute and made optional, the compiler would still pick the no-message one in overload resolution. Therefore, the no-message overload would have to be removed to take advantage of this feature, which would be a binary (although not source) breaking change.

## Alternatives

- If being able to see source code at call sites for methods that use this attribute proves to be a problem, we can make the attribute's effects opt-in. Developers will enable it through an assembly-wide `[assembly: EnableCallerArgumentExpression]` attribute they put in `AssemblyInfo.cs`.
  - In the case the attribute's effects are not enabled, calling methods marked with the attribute would not be an error, to allow existing methods to use the attribute and maintain source compatibility. However, the attribute would be ignored and the method would be called with whatever default value was provided.

C#

```
// Assembly1

void Foo(string bar); // V1
void Foo(string bar, string barExpression = "not provided"); // V2
void Foo(string bar, [CallerArgumentExpression("bar")] string barExpression
= "not provided"); // V3

// Assembly2

Foo(a); // V1: Compiles to Foo(a), V2, V3: Compiles to Foo(a, "not
provided")
Foo(a, "provided"); // V2, V3: Compiles to Foo(a, "provided")

// Assembly3

[assembly: EnableCallerArgumentExpression]

Foo(a); // V1: Compiles to Foo(a), V2: Compiles to Foo(a, "not provided"),
V3: Compiles to Foo(a, "a")
Foo(a, "provided"); // V2, V3: Compiles to Foo(a, "provided")
```

- To prevent the [binary compatibility problem](#) from occurring every time we want to add new caller info to `Debug.Assert`, an alternative solution would be to add a `CallerInfo` struct to the framework that contains all the necessary information about the caller.

C#

```
struct CallerInfo
{
    public string MemberName { get; set; }
    public string TypeName { get; set; }
    public string Namespace { get; set; }
    public string FullTypeName { get; set; }
    public string FilePath { get; set; }
    public int LineNumber { get; set; }
```

```

    public int ColumnNumber { get; set; }
    public Type Type { get; set; }
    public MethodBase Method { get; set; }
    public string[] ArgumentExpressions { get; set; }
}

[Flags]
enum CallerInfoOptions
{
    MemberName = 1, TypeName = 2, ...
}

public static class Debug
{
    public static void Assert(bool condition,
        // If a flag is not set here, the corresponding CallerInfo member is
        // not populated by the caller, so it's
        // pay-for-play friendly.
        [CallerInfo(CallerInfoOptions.FilePath | CallerInfoOptions.Method |
        CallerInfoOptions.ArgumentExpressions)] CallerInfo callerInfo =
        default(CallerInfo))
    {
        string filePath = callerInfo.FilePath;
        MethodBase method = callerInfo.Method;
        string conditionExpression = callerInfo.ArgumentExpressions[0];
        //...
    }
}

class Bar
{
    void Foo()
    {
        Debug.Assert(false);

        // Translates to:

        var callerInfo = new CallerInfo();
        callerInfo.FilePath = @"C:\Bar.cs";
        callerInfo.Method = MethodBase.GetCurrentMethod();
        callerInfo.ArgumentExpressions = new string[] { "false" };
        Debug.Assert(false, callerInfo);
    }
}

```

This was originally proposed at [https://github.com/dotnet/csharplang/issues/87 ↗](https://github.com/dotnet/csharplang/issues/87).

There are a few disadvantages of this approach:

- Despite being pay-for-play friendly by allowing you to specify which properties you need, it could still hurt perf significantly by allocating an array for the expressions/calling `MethodBase.GetCurrentMethod()` even when the assert passes.

- Additionally, while passing a new flag to the `CallerInfo` attribute won't be a breaking change, `Debug.Assert` won't be guaranteed to actually receive that new parameter from call sites that compiled against an old version of the method.

## Unresolved questions

TBD

## Design meetings

N/A

# Enhanced #line directives

Article • 01/24/2023

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

## Summary

The compiler applies the mapping defined by `#line` directives to diagnostic locations and sequence points emitted to the PDB.

Currently only the line number and file path can be mapped while the starting character is inferred from the source code. The proposal is to allow specifying full span mapping.

## Motivation

DSLs that generate C# source code (such as ASP.NET Razor) can't currently produce precise source mapping using `#line` directives. This results in degraded debugging experience in some cases as the sequence points emitted to the PDB can't map to the precise location in the original source code.

For example, the following Razor code

```
@page "/"
Time: @DateTime.Now
```

generates code like so (simplified):

```
C#
```

```
#line hidden
void Render()
{
    _builder.Add("Time:");
#line 2 "page.razor"
    _builder.Add(DateTime.Now);
#line hidden
}
```

The above directive would map the sequence point emitted by the compiler for the `_builder.Add(DateTime.Now);` statement to the line 2, but the column would be off (16 instead of 7).

The Razor source generator actually incorrectly generates the following code:

C#

```
#line hidden
void Render()
{
    _builder.Add("Time:");
    _builder.Add(
#line 2 "page.razor"
        DateTime.Now
#line hidden
    );
}
```

The intent was to preserve the starting character and it works for diagnostic location mapping. However, this does not work for sequence points since `#line` directive only applies to the sequence points that follow it. There is no sequence point in the middle of the `_builder.Add(DateTime.Now);` statement (sequence points can only be emitted at IL instructions with empty evaluation stack). The `#line 2` directive in above code thus has no effect on the generated PDB and the debugger won't place a breakpoint or stop on the `@DateTime.Now` snippet in the Razor page.

Issues addressed by this proposal: [https://github.com/dotnet/roslyn/issues/43432 ↗](https://github.com/dotnet/roslyn/issues/43432)  
[https://github.com/dotnet/roslyn/issues/46526 ↗](https://github.com/dotnet/roslyn/issues/46526)

## Detailed design

We amend the syntax of `line_indicator` used in `pp_line` directive like so:

Current:

```

line_indicator
  : decimal_digit+ whitespace file_name
  | decimal_digit+
  | 'default'
  | 'hidden'
;

```

Proposed:

```

line_indicator
  : '(' decimal_digit+ ',' decimal_digit+ ')' '-' '(' decimal_digit+ ',' decimal_digit+ ')' whitespace decimal_digit+ whitespace file_name
    | '(' decimal_digit+ ',' decimal_digit+ ')' '-' '(' decimal_digit+ ',' decimal_digit+ ')' whitespace file_name
      | decimal_digit+ whitespace file_name
      | decimal_digit+
      | 'default'
      | 'hidden'
;

```

That is, the `#line` directive would accept either 5 decimal numbers (*start line, start character, end line, end character, character offset*), 4 decimal numbers (*start line, start character, end line, end character*), or a single one (*line*).

If *character offset* is not specified its default value is 0, otherwise it specifies the number of UTF-16 characters. The number must be non-negative and less than length of the line following the `#line` directive in the unmapped file.

*(start line, start character)-(end line, end character)* specifies a span in the mapped file. *start line* and *end line* are positive integers that specify line numbers. *start character, end character* are positive integers that specify UTF-16 character numbers. *start line, start character, end line, end character* are 1-based, meaning that the first line of the file and the first UTF-16 character on each line is assigned number 1.

The implementation would constraint these numbers so that they specify a valid sequence point source span [↗](#):

- *start line - 1* is within range [0, 0x20000000) and not equal to 0xfffffe.
- *end line - 1* is within range [0, 0x20000000) and not equal to 0xfffffe.
- *start character - 1* is within range [0, 0x10000)
- *end character - 1* is within range [0, 0x10000)
- *end line* is greater or equal to *start line*.

- *start line* is equal to *end line* then *end character* is greater than *start character*.

Note that the numbers specified in the directive syntax are 1-based numbers but the actual spans in the PDB are zero-based. Hence the -1 adjustments above.

The mapped spans of sequence points and the locations of diagnostics that `#line` directive applies to are calculated as follows.

Let  $d$  be the zero-based number of the unmapped line containing the `#line` directive.

Let span  $L = (\text{start: } (\text{start line} - 1, \text{start character} - 1), \text{end: } (\text{end line} - 1, \text{end character} - 1))$  be zero-based span specified by the directive.

Function  $M$  that maps a position (line, character) within the scope of the `#line` directive in the source file containing the `#line` directive to a mapped position (mapped line, mapped character) is defined as follows:

$$M(l, c) =$$

$$l == d + 1 \Rightarrow (L.start.line + l - d - 1, L.start.character + \max(c - character offset, 0))$$

$$l > d + 1 \Rightarrow (L.start.line + l - d - 1, c)$$

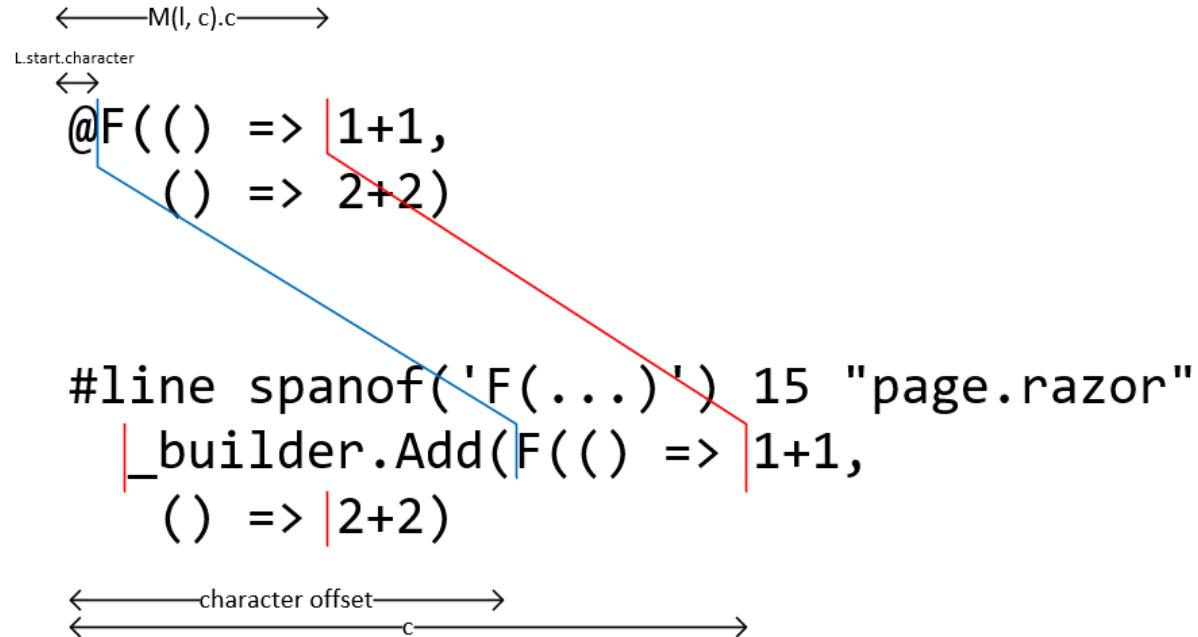
The syntax constructs that sequence points are associated with are determined by the compiler implementation and not covered by this specification. The compiler also decides for each sequence point its unmapped span. This span may partially or fully cover the associated syntax construct.

Once the unmapped spans are determined by the compiler the function  $M$  defined above is applied to their starting and ending positions, with the exception of the ending position of all sequence points within the scope of the `#line` directive whose unmapped location is at line  $d + 1$  and character less than character offset. The end position of all these sequence points is  $L.end$ .

Example [5.i] demonstrates why it is necessary to provide the ability to specify the end position of the first sequence point span.

The above definition allows the generator of the unmapped source code to avoid intimate knowledge of which exact source constructs of the C# language produce sequence points. The mapped spans of the sequence points in the scope of the `#line` directive are derived from the relative position of the corresponding unmapped spans to the first unmapped span.

Specifying the *character offset* allows the generator to insert any single-line prefix on the first line. This prefix is generated code that is not present in the mapped file. Such inserted prefix affects the value of the first unmapped sequence point span. Therefore the starting character of subsequent sequence point spans need to be offset by the length of the prefix (*character offset*). See example [2].



## Examples

For clarity the examples use `spanof('...')` and `lineof('...')` pseudo-syntax to express the mapped span start position and line number, respectively, of the specified code snippet.

### 1. First and subsequent spans

Consider the following code with unmapped zero-based line numbers listed on the right:

```

#line (1,10)-(1,15) "a" // 3
A();B(                  // 4
);C();                  // 5
D();                   // 6

```

$$d = 3 \quad L = (0, 9)..(0, 14)$$

There are 4 sequence point spans the directive applies to with following unmapped and mapped spans: (4, 2)..(4, 5) => (0, 9)..(0, 14) (4, 6)..(5, 1) => (0, 15)..(1, 1) (5, 2)..(5, 5) => (1, 2)..(1, 5) (6, 4)..(6, 7) => (2, 4)..(2, 7)

## 2. Character offset

Razor generates `_builder.Add()` prefix of length 15 (including two leading spaces).

Razor:

```
@page "/"
@F(() => 1+1,
    () => 2+2
)
```

Generated C#:

```
C#
#line hidden
void Render()
{
#line spanof('F(...)') 15 "page.razor" // 4
    _builder.Add(F(() => 1+1,           // 5
                  () => 2+2           // 6
                ));                 // 7
#line hidden
}
);
}
```

$d = 4 \ L = (1, 1)..(3,0)$  character offset = 15

Spans:

- `_builder.Add(F(...));` => `F(...):` (5, 2)..(7, 2) => (1, 1)..(3, 0)
- `1+1` => `1+1:` (5, 23)..(5, 25) => (1, 9)..(1, 11)
- `2+2` => `2+2:` (6, 7)..(6, 9) => (2, 7)..(2, 9)

## 3. Razor: Single-line span

Razor:

```
@page "/"
Time: @DateTime.Now
```

Generated C#:

```
C#

#line hidden
void Render()
{
    _builder.Add("Time:");
#line spanof('DateTime.Now') 15 "page.razor"
    _builder.Add(DateTime.Now);
#line hidden
);
}
```

## 4. Razor: Multi-line span

Razor:

```
@page "/"
@JsonObject(@"
{
    ""key1"": "value1",
    ""key2"": "value2"
}")
```

Generated C#:

```
C#

#line hidden
void Render()
{
    _builder.Add("Time:");
#line spanof('JsonObject(@"..."') 15 "page.razor"
    _builder.Add(JsonObject(@"
{
    ""key1"": "value1",
    ""key2"": "value2"
}"));
#line hidden
}
);
}
```

## 5. Razor: block constructs

### i. block containing expressions

In this example, the mapped span of the first sequence point that is associated with the IL instruction that is emitted for the `_builder.Add(Html.Helper(() => ...))` statement needs to cover the whole expression of `Html.Helper(...)` in the generated file `a.razor`. This is achieved by application of rule [1] to the end position of the sequence point.

```
@HtmlHelper(() =>
{
    <p>Hello World</p>
    @DateTime.Now
})
```

C#

```
#line spanof('HtmlHelper(() => { ... })') 13 "a.razor"
_builder.Add(HtmlHelper(() =>
#line lineof('{') "a.razor"
{
#line spanof('DateTime.Now') 13 "a.razor"
_builder.Add(DateTime.Now);
#line lineof('}') "a.razor"
}
#line hidden
)
```

### ii. block containing statements

Uses existing `#line line file` form since

- a) Razor does not add any prefix, b) `{` is not present in the generated file and there can't be a sequence point placed on it, therefore the span of the first unmapped sequence point is unknown to Razor.

The starting character of `Console` in the generated file must be aligned with the Razor file.

```
@{Console.WriteLine(1);Console.WriteLine(2);}
```

C#

```
#line lineof('@') "a.razor"
Console.WriteLine(1);Console.WriteLine(2);
#line hidden
```

### iii. block containing top-level code (@code, @functions)

Uses existing `#line line file` form since

a) Razor does not add any prefix, b) `{` is not present in the generated file and there can't be a sequence point placed on it, therefore the span of the first unmapped sequence point is unknown to Razor.

The starting character of `[Parameter]` in the generated file must be aligned with the Razor file.

```
@code {
    [Parameter]
    public int IncrementAmount { get; set; }
}
```

C#

```
#line lineof('[') "a.razor"
[Parameter]
public int IncrementAmount { get; set; }
#line hidden
```

## 6. Razor: `@for`, `@foreach`, `@while`, `@do`, `@if`, `@switch`, `@using`, `@try`, `@lock`

Uses existing `#line line file` form since a) Razor does not add any prefix. b) the span of the first unmapped sequence point may not be known to Razor (or shouldn't need to know).

The starting character of the keyword in the generated file must be aligned with the Razor file.

```
@for (var i = 0; i < 10; i++)
{
}
@if (condition)
{
}
else
{
}
```

C#

```
#line lineof('for') "a.razor"
for (var i = 0; i < 10; i++)
{
}
#line lineof('if') "a.razor"
if (condition)
{
}
else
{
}
#line hidden
```

# Improved Definite Assignment Analysis

Article • 05/29/2022

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

## Summary

Definite assignment [§9.4](#) as specified has a few gaps which have caused users inconvenience. In particular, scenarios involving comparison to boolean constants, conditional-access, and null coalescing.

## Related discussions and issues

csharplang discussion of this proposal:

<https://github.com/dotnet/csharplang/discussions/4240>

Probably a dozen or so user reports can be found via this or similar queries (i.e. search for "definite assignment" instead of "CS0165", or search in csharplang).

[https://github.com/dotnet/roslyn/issues?  
q=is%3Aclosed+is%3Aissue+label%3A%22Resolution-By+Design%22+cs0165](https://github.com/dotnet/roslyn/issues?q=is%3Aclosed+is%3Aissue+label%3A%22Resolution-By+Design%22+cs0165)

I have included related issues in the scenarios below to give a sense of the relative impact of each scenario.

## Scenarios

As a point of reference, let's start with a well-known "happy case" that does work in definite assignment and in nullable.

```
#nullable enable

C c = new C();
if (c != null && c.M(out object obj0))
{
    obj0.ToString(); // ok
}

public class C
{
    public bool M(out object obj)
    {
        obj = new object();
        return true;
    }
}
```

## Comparison to bool constant

- [https://github.com/dotnet/csharplang/discussions/801 ↗](https://github.com/dotnet/csharplang/discussions/801)
- [https://github.com/dotnet/roslyn/issues/45582 ↗](https://github.com/dotnet/roslyn/issues/45582)
  - Links to 4 other issues where people were affected by this.

C#

```
if ((c != null && c.M(out object obj1)) == true)
{
    obj1.ToString(); // undesired error
}

if ((c != null && c.M(out object obj2)) is true)
{
    obj2.ToString(); // undesired error
}
```

## Comparison between a conditional access and a constant value

- [https://github.com/dotnet/roslyn/issues/33559 ↗](https://github.com/dotnet/roslyn/issues/33559)
- [https://github.com/dotnet/csharplang/discussions/4214 ↗](https://github.com/dotnet/csharplang/discussions/4214)
- [https://github.com/dotnet/csharplang/issues/3659 ↗](https://github.com/dotnet/csharplang/issues/3659)
- [https://github.com/dotnet/csharplang/issues/3485 ↗](https://github.com/dotnet/csharplang/issues/3485)
- [https://github.com/dotnet/csharplang/issues/3659 ↗](https://github.com/dotnet/csharplang/issues/3659)

This scenario is probably the biggest one. We do support this in nullable but not in definite assignment.

```
C#  
  
if (c?.M(out object obj3) == true)  
{  
    obj3.ToString(); // undesired error  
}
```

## Conditional access coalesced to a bool constant

- [https://github.com/dotnet/csharplang/discussions/916 ↗](https://github.com/dotnet/csharplang/discussions/916)
- [https://github.com/dotnet/csharplang/issues/3365 ↗](https://github.com/dotnet/csharplang/issues/3365)

This scenario is very similar to the previous one. This is also supported in nullable but not in definite assignment.

```
C#  
  
if (c?.M(out object obj4) ?? false)  
{  
    obj4.ToString(); // undesired error  
}
```

## Conditional expressions where one arm is a bool constant

- [https://github.com/dotnet/roslyn/issues/4272 ↗](https://github.com/dotnet/roslyn/issues/4272)

It's worth pointing out that we already have special behavior for when the condition expression is constant (i.e. `true ? a : b`). We just unconditionally visit the arm indicated by the constant condition and ignore the other arm.

Also note that we haven't handled this scenario in nullable.

```
C#  
  
if (c != null ? c.M(out object obj4) : false)  
{  
    obj4.ToString(); // undesired error  
}
```

# Specification

## ? . (null-conditional operator) expressions

We introduce a new section [?. \(null-conditional operator\) expressions](#). See the null-conditional operator specification ([§11.7.7](#)) and Precise rules for determining definite assignment [§9.4.4](#) for context.

As in the definite assignment rules linked above, we refer to a given initially unassigned variable as *v*.

We introduce the concept of "directly contains". An expression *E* is said to "directly contain" a subexpression *E<sub>1</sub>* if it is not subject to a user-defined conversion [§10.5](#) whose parameter is not of a non-nullable value type, and one of the following conditions holds:

- *E* is *E<sub>1</sub>*. For example, `a?.b()` directly contains the expression `a?.b()`.
- If *E* is a parenthesized expression `(E2)`, and *E<sub>2</sub>* directly contains *E<sub>1</sub>*.
- If *E* is a null-forgiving operator expression `E2!`, and *E<sub>2</sub>* directly contains *E<sub>1</sub>*.
- If *E* is a cast expression `(T)E2`, and the cast does not subject *E<sub>2</sub>* to a non-lifted user-defined conversion whose parameter is not of a non-nullable value type, and *E<sub>2</sub>* directly contains *E<sub>1</sub>*.

For an expression *E* of the form `primary_expression null_conditional_operations`, let *E<sub>0</sub>* be the expression obtained by textually removing the leading `?` from each of the *null\_conditional\_operations* of *E* that have one, as in the linked specification above.

In subsequent sections we will refer to *E<sub>0</sub>* as the *non-conditional counterpart* to the null-conditional expression. Note that some expressions in subsequent sections are subject to additional rules that only apply when one of the operands directly contains a null-conditional expression.

- The definite assignment state of *v* at any point within *E* is the same as the definite assignment state at the corresponding point within *E<sub>0</sub>*.
- The definite assignment state of *v* after *E* is the same as the definite assignment state of *v* after *primary\_expression*.

## Remarks

We use the concept of "directly contains" to allow us to skip over relatively simple "wrapper" expressions when analyzing conditional accesses that are compared to other values. For example, `((a?.b(out x))!) == true` is expected to result in the same flow state as `a?.b == true` in general.

We also want to allow analysis to function in the presence of a number of possible conversions on a conditional access. Propagating out "state when not null" is not possible when the conversion is user-defined, though, since we can't count on user-defined conversions to honor the constraint that the output is non-null only if the input is non-null. The only exception to this is when the user-defined conversion's input is a non-nullable value type. For example:

```
C#
```

```
public struct S1 { }
public struct S2 { public static implicit operator S2?(S1 s1) => null; }
```

This also includes lifted conversions like the following:

```
C#
```

```
string x;

S1? s1 = null;
_ = s1?.M1(x = "a") ?? s1.Value.M2(x = "a");

x.ToString(); // ok

public struct S1
{
    public S1 M1(object obj) => this;
    public S2 M2(object obj) => new S2();
}
public struct S2
{
    public static implicit operator S2(S1 s1) => null;
}
```

When we consider whether a variable is assigned at a given point within a null-conditional expression, we simply assume that any preceding null-conditional operations within the same null-conditional expression succeeded.

For example, given a conditional expression `a?.b(out x)?.c(x)`, the non-conditional counterpart is `a.b(out x).c(x)`. If we want to know the definite assignment state of `x` before `? .c(x)`, for example, then we perform a "hypothetical" analysis of `a.b(out x)` and use the resulting state as an input to `? .c(x)`.

## Boolean constant expressions

We introduce a new section "Boolean constant expressions":

For an expression *expr* where *expr* is a constant expression with a bool value:

- The definite assignment state of *v* after *expr* is determined by:
  - If *expr* is a constant expression with value *true*, and the state of *v* before *expr* is "not definitely assigned", then the state of *v* after *expr* is "definitely assigned when false".
  - If *expr* is a constant expression with value *false*, and the state of *v* before *expr* is "not definitely assigned", then the state of *v* after *expr* is "definitely assigned when true".

## Remarks

We assume that if an expression has a constant value `bool false`, for example, it's impossible to reach any branch that requires the expression to return `true`. Therefore variables are assumed to be definitely assigned in such branches. This ends up combining nicely with the spec changes for expressions like `??` and `?:` and enabling a lot of useful scenarios.

It's also worth noting that we never expect to be in a conditional state *before* visiting a constant expression. That's why we do not account for scenarios such as "*expr* is a constant expression with value *true*, and the state of *v* before *expr* is "definitely assigned when true".

## `??` (null-coalescing expressions) augment

We augment section §9.4.4.29 as follows:

For an expression *expr* of the form `expr_first ?? expr_second`:

- ...
- The definite assignment state of *v* after *expr* is determined by:
  - ...
  - If *expr\_first* directly contains a null-conditional expression *E*, and *v* is definitely assigned after the non-conditional counterpart *E<sub>0</sub>*, then the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after *expr\_second*.

## Remarks

The above rule formalizes that for an expression like `a?.M(out x) ?? (x = false)`, either the `a?.M(out x)` was fully evaluated and produced a non-null value, in which case `x` was

assigned, or the `x = false` was evaluated, in which case `x` was also assigned. Therefore `x` is always assigned after this expression.

This also handles the `dict?.TryGetValue(key, out var value) ?? false` scenario, by observing that `v` is definitely assigned after `dict.TryGetValue(key, out var value)`, and `v` is "definitely assigned when true" after `false`, and concluding that `v` must be "definitely assigned when true".

The more general formulation also allows us to handle some more unusual scenarios, such as:

- `if (x?.M(out y) ?? (b && z.M(out y))) y.ToString();`
- `if (x?.M(out y) ?? z?.M(out y) ?? false) y.ToString();`

## ?: (conditional) expressions

We augment section [§9.4.4.30](#) as follows:

For an expression `expr` of the form `expr_cond ? expr_true : expr_false`:

- ...
- The definite assignment state of `v` after `expr` is determined by:
  - ...
  - If the state of `v` after `expr_true` is "definitely assigned when true", and the state of `v` after `expr_false` is "definitely assigned when true", then the state of `v` after `expr` is "definitely assigned when true".
  - If the state of `v` after `expr_true` is "definitely assigned when false", and the state of `v` after `expr_false` is "definitely assigned when false", then the state of `v` after `expr` is "definitely assigned when false".

## Remarks

This makes it so when both arms of a conditional expression result in a conditional state, we join the corresponding conditional states and propagate it out instead of unsplitting the state and allowing the final state to be non-conditional. This enables scenarios like the following:

C#

```
bool b = true;
object x = null;
int y;
if (b ? x != null && Set(out y) : x != null && Set(out y))
```

```

{
    y.ToString();
}

bool Set(out int x) { x = 0; return true; }

```

This is an admittedly niche scenario, that compiles without error in the native compiler, but was broken in Roslyn in order to match the specification at the time. See [internal issue](#).

## `==/!=` (relational equality operator) expressions

We introduce a new section `==/!=` (relational equality operator) expressions.

The general rules for expressions with embedded expressions [§9.4.4.23](#) apply, except for the scenarios described below.

For an expression *expr* of the form `expr_first == expr_second`, where `==` is a predefined comparison operator ([§11.11](#)) or a lifted operator ([§11.4.8](#)), the definite assignment state of *v* after *expr* is determined by:

- If *expr\_first* directly contains a null-conditional expression *E* and *expr\_second* is a constant expression with value *null*, and the state of *v* after the non-conditional counterpart *E<sub>0</sub>* is "definitely assigned", then the state of *v* after *expr* is "definitely assigned when false".
- If *expr\_first* directly contains a null-conditional expression *E* and *expr\_second* is an expression of a non-nullable value type, or a constant expression with a non-null value, and the state of *v* after the non-conditional counterpart *E<sub>0</sub>* is "definitely assigned", then the state of *v* after *expr* is "definitely assigned when true".
- If *expr\_first* is of type *boolean*, and *expr\_second* is a constant expression with value *true*, then the definite assignment state after *expr* is the same as the definite assignment state after *expr\_first*.
- If *expr\_first* is of type *boolean*, and *expr\_second* is a constant expression with value *false*, then the definite assignment state after *expr* is the same as the definite assignment state of *v* after the logical negation expression `!expr_first`.

For an expression *expr* of the form `expr_first != expr_second`, where `!=` is a predefined comparison operator ([§11.11](#)) or a lifted operator ([§11.4.8](#)), the definite assignment state of *v* after *expr* is determined by:

- If *expr\_first* directly contains a null-conditional expression *E* and *expr\_second* is a constant expression with value *null*, and the state of *v* after the non-conditional

counterpart  $E_0$  is "definitely assigned", then the state of  $v$  after  $expr$  is "definitely assigned when true".

- If  $expr\_first$  directly contains a null-conditional expression  $E$  and  $expr\_second$  is an expression of a non-nullable value type, or a constant expression with a non-null value, and the state of  $v$  after the non-conditional counterpart  $E_0$  is "definitely assigned", then the state of  $v$  after  $expr$  is "definitely assigned when false".
- If  $expr\_first$  is of type `boolean`, and  $expr\_second$  is a constant expression with value `true`, then the definite assignment state after  $expr$  is the same as the definite assignment state of  $v$  after the logical negation expression `!expr_first`.
- If  $expr\_first$  is of type `boolean`, and  $expr\_second$  is a constant expression with value `false`, then the definite assignment state after  $expr$  is the same as the definite assignment state after  $expr\_first$ .

All of the above rules in this section are commutative, meaning that if a rule applies when evaluated in the form `expr_second op expr_first`, it also applies in the form `expr_first op expr_second`.

## Remarks

The general idea expressed by these rules is:

- if a conditional access is compared to `null`, then we know the operations definitely occurred if the result of the comparison is `false`
- if a conditional access is compared to a non-nullable value type or a non-null constant, then we know the operations definitely occurred if the result of the comparison is `true`.
- since we can't trust user-defined operators to provide reliable answers where initialization safety is concerned, the new rules only apply when a predefined `==/!=` operator is in use.

We may eventually want to refine these rules to thread through conditional state which is present at the end of a member access or call. Such scenarios don't really happen in definite assignment, but they do happen in nullable in the presence of `[NotNullWhen(true)]` and similar attributes. This would require special handling for `bool` constants in addition to just handling for `null/non-null` constants.

Some consequences of these rules:

- `if (a?.b(out var x) == true)) x() else x();` will error in the 'else' branch
- `if (a?.b(out var x) == 42)) x() else x();` will error in the 'else' branch
- `if (a?.b(out var x) == false)) x() else x();` will error in the 'else' branch

- `if (a?.b(out var x) == null)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != true)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != 42)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != false)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != null)) x() else x();` will error in the 'else' branch

## `is` operator and `is` pattern expressions

We introduce a new section `is` operator and `is` pattern expressions.

For an expression `expr` of the form `E is T`, where `T` is any type or pattern

- The definite assignment state of `v` before `E` is the same as the definite assignment state of `v` before `expr`.
- The definite assignment state of `v` after `expr` is determined by:
  - If `E` directly contains a null-conditional expression, and the state of `v` after the non-conditional counterpart `E0` is "definitely assigned", and `T` is any type or a pattern that does not match a `null` input, then the state of `v` after `expr` is "definitely assigned when true".
  - If `E` directly contains a null-conditional expression, and the state of `v` after the non-conditional counterpart `E0` is "definitely assigned", and `T` is a pattern that matches a `null` input, then the state of `v` after `expr` is "definitely assigned when false".
  - If `E` is of type boolean and `T` is a pattern which only matches a `true` input, then the definite assignment state of `v` after `expr` is the same as the definite assignment state of `v` after `E`.
  - If `E` is of type boolean and `T` is a pattern which only matches a `false` input, then the definite assignment state of `v` after `expr` is the same as the definite assignment state of `v` after the logical negation expression `!expr`.
  - Otherwise, if the definite assignment state of `v` after `E` is "definitely assigned", then the definite assignment state of `v` after `expr` is "definitely assigned".

## Remarks

This section is meant to address similar scenarios as in the `==/!=` section above. This specification does not address recursive patterns, e.g. `(a?.b(out x), c?.d(out y)) is (object, object)`. Such support may come later if time permits.

## Additional scenarios

This specification doesn't currently address scenarios involving pattern switch expressions and switch statements. For example:

```
C#  
  
_ = c?.M(out object obj4) switch  
{  
    not null => obj4.ToString() // undesired error  
};
```

It seems like support for this could come later if time permits.

There have been several categories of bugs filed for nullable which require we essentially increase the sophistication of pattern analysis. It is likely that any ruling we make which improves definite assignment would also be carried over to nullable.

[https://github.com/dotnet/roslyn/issues/49353 ↗](https://github.com/dotnet/roslyn/issues/49353)

[https://github.com/dotnet/roslyn/issues/46819 ↗](https://github.com/dotnet/roslyn/issues/46819)

[https://github.com/dotnet/roslyn/issues/44127 ↗](https://github.com/dotnet/roslyn/issues/44127)

## Drawbacks

It feels odd to have the analysis "reach down" and have special recognition of conditional accesses, when typically flow analysis state is supposed to propagate upward. We are concerned about how a solution like this could intersect painfully with possible future language features that do null checks.

## Alternatives

Two alternatives to this proposal:

1. Introduce "state when null" and "state when not null" to the language and compiler. This has been judged to be too much effort for the scenarios we are trying to solve, but that we could potentially implement the above proposal and then move to a "state when null/not null" model later on without breaking people.
2. Do nothing.

## Unresolved questions

There are impacts on switch expressions that should be specified:

[https://github.com/dotnet/csharplang/discussions/4240#discussioncomment-343395 ↗](https://github.com/dotnet/csharplang/discussions/4240#discussioncomment-343395)

# Design meetings

[https://github.com/dotnet/csharplang/discussions/4243 ↗](https://github.com/dotnet/csharplang/discussions/4243)

# AsyncMethodBuilder override

Article • 09/21/2021

## ⓘ Note

This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent [language design meeting \(LDM\) notes](#). Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the [specifications](#).

## Summary

Allow per-method override of the async method builder to use. For some async methods we want to customize the invocation of `Builder.Create()` to use a different *builder type*.

C#

```
[AsyncMethodBuilderAttribute(typeof(PoolingAsyncValueTaskMethodBuilder<>))]
// new usage of AsyncMethodBuilderAttribute type
static async ValueTask<int> ExampleAsync() { ... }
```

## Motivation

Today, async method builders are tied to a given type used as a return type of an async method. For example, any method that's declared as `async Task` uses `AsyncTaskMethodBuilder`, and any method that's declared as `async ValueTask<T>` uses `AsyncValueTaskMethodBuilder<T>`. This is due to the `[AsyncMethodBuilder(Type)]` attribute on the type used as a return type, e.g. `ValueTask<T>` is attributed as `[AsyncMethodBuilder(typeof(AsyncValueTaskMethodBuilder<>))]`. This addresses the majority common case, but it leaves a few notable holes for advanced scenarios.

In .NET 5, an experimental feature was shipped that provides two modes in which `AsyncValueTaskMethodBuilder` and `AsyncValueTaskMethodBuilder<T>` operate. The on-by-default mode is the same as has been there since the functionality was introduced: when

the state machine needs to be lifted to the heap, an object is allocated to store the state, and the async method returns a `ValueTask<T>` backed by a `Task<T>`. However, if an environment variable is set, all builders in the process switch to a mode where, instead, the `ValueTask<T>` instances are backed by reusable `IValueTaskSource<T>` implementations that are pooled. Each async method has its own pool with a fixed maximum number of instances allowed to be pooled, and as long as no more than that number are ever returned to the pool to be pooled at the same time, `async ValueTask<T>` methods effectively become free of any GC allocation overhead.

There are several problems with this experimental mode, however, which is both why a) it's off by default and b) we're likely to remove it in a future release unless very compelling new information emerges (<https://github.com/dotnet/runtime/issues/13633>).

- It introduces a behavioral difference for consumers of the returned `ValueTask<T>` if that `ValueTask` isn't being consumed according to spec. When it's backed by a `Task`, you can do with the `ValueTask` things you can do with a `Task`, like await it multiple times, await it concurrently, block waiting for it to complete, etc. But when it's backed by an arbitrary `IValueTaskSource`, such operations are prohibited, and automatically switching from the former to the latter can lead to bugs. With the switch at the process level and affecting all `async ValueTask` methods in the process, whether you control them or not, it's too big a hammer.
- It's not necessarily a performance win, and could represent a regression in some situations. The implementation is trading the cost of pooling (accessing a pool isn't free) with the cost of GC, and in various situations the GC can win. Again, applying the pooling to all `async ValueTask` methods in the process rather than being selective about the ones it would most benefit is too big a hammer.
- It adds to the IL size of a trimmed application, even if the flag isn't set, and then to the resulting asm size. It's possible that can be worked around with improvements to the implementation to teach it that for a given deployment the environment variable will always be false, but as it stands today, every `async ValueTask` method saw for example an ~2K binary footprint increase in aot images due to this option, and, again, that applies to all `async ValueTask` methods in the whole application closure.
- Different methods may benefit from differing levels of control, e.g. the size of the pool employed because of knowledge of the method and how it's used, but the same setting is applied to all uses of the builder. One could imagine working around that by having the builder code use reflection at runtime to look for some attribute, but that adds significant run-time expense, and likely on the startup path.

On top of all of these issues with the existing pooling, it's also the case that developers are prevented from writing their own customized builders for types they don't own. If, for example, a developer wants to implement their own pooling support, they also have to introduce a brand new task-like type, rather than just being able to use `{Value}Task{<T>}`, because the attribute specifying the builder is only specifiable on the type declaration of the return type.

We need a way to have an individual async method opt-in to a specific builder.

## Detailed design

### Using AsyncMethodBuilderAttribute on methods

In `dotnet/runtime`, add `AttributeTargets.Method` to the targets for `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute`:

C#

```
namespace System.Runtime.CompilerServices
{
    /// <summary>
    /// Indicates the type of the async method builder that should be used
    /// by a language compiler:
    /// - to build the return type of an async method that is attributed,
    /// - to build the attributed type when used as the return type of an
    ///   async method.
    /// </summary>
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class |
    AttributeTargets.Struct | AttributeTargets.Interface |
    AttributeTargets.Delegate | AttributeTargets.Enum, Inherited = false,
    AllowMultiple = false)]
    public sealed class AsyncMethodBuilderAttribute : Attribute
    {
        /// <summary>Initializes the <see
        cref="AsyncMethodBuilderAttribute"/>. </summary>
        /// <param name="builderType">The <see cref="Type"/> of the
        /// associated builder. </param>
        public AsyncMethodBuilderAttribute(Type builderType) => BuilderType
        = builderType;

        /// <summary>Gets the <see cref="Type"/> of the associated builder.
        </summary>
        public Type BuilderType { get; }
    }
}
```

This allows the attribute to be applied on methods or local functions or lambdas.

Example of usage on a method:

C#

```
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))] // new  
usage, referring to some custom builder type  
static async ValueTask<int> ExampleAsync() { ... }
```

It is an error to apply the attribute multiple times on a given method.

It is an error to apply the attribute to a lambda with an implicit return type.

A developer who wants to use a specific custom builder for all of their methods can do so by putting the relevant attribute on each method.

## Determining the builder type for an async method

When compiling an async method, the builder type is determined by:

1. using the builder type from the `AsyncMethodBuilder` attribute if one is present,
2. otherwise, falling back to the builder type determined by previous approach. (see [spec for task-like types ↗](#)).

If an `AsyncMethodBuilder` attribute is present, we take the builder type specified by the attribute and construct it if necessary.

If the override type is an open generic type, take the single type argument of the async method's return type and substitute it into the override type.

If the override type is a bound generic type, then we produce an error.

If the async method's return type does not have a single type argument, then we produce an error.

We verify that the builder type is compatible with the return type of the async method:

1. look for the public `create` method with no type parameters and no parameters on the constructed builder type.  
It is an error if the method is not found. It is an error if the method returns a type other than the constructed builder type.
2. look for the public `Task` property.  
It is an error if the property is not found.
3. consider the type of that `Task` property (a task-like type):  
It is an error if the task-like type does not matches the return type of the async method.

Note that it is not necessary for the return type of the method to be a task-like type.

## Execution

The builder type determined above is used as part of the existing async method design.

For example, today if a method is defined as:

```
C#
```

```
public async ValueTask<T> ExampleAsync() { ... }
```

the compiler will generate code akin to:

```
C#
```

```
[AsyncStateMachine(typeof(<ExampleAsync>d__29))]
[CompilerGenerated]
static ValueTask<int> ExampleAsync()
{
    <ExampleAsync>d__29 stateMachine;
    stateMachine.<>t__builder = AsyncValueTaskMethodBuilder<int>.Create();
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}
```

With this change, if the developer wrote:

```
C#
```

```
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))] // new
usage, referring to some custom builder type
static async ValueTask<int> ExampleAsync() { ... }
```

it would instead be compiled to:

```
C#
```

```
[AsyncStateMachine(typeof(<ExampleAsync>d__29))]
[CompilerGenerated]
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))] // retained but not necessary anymore
static ValueTask<int> ExampleAsync()
{
    <ExampleAsync>d__29 stateMachine;
    stateMachine.<>t__builder =
    PoolingAsyncValueTaskMethodBuilder<int>.Create(); // <>t__builder now a
    different type
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
```

```
    return stateMachine.<>t__builder.Task;
}
```

Just those small additions enable:

- Anyone to write their own builder that can be applied to async methods that return `Task<T>` and `ValueTask<T>`
- As "anyone", the runtime to ship the experimental builder support as new public builder types that can be opted into on a method-by-method basis; the existing support would be removed from the existing builders. Methods (including some we care about in the core libraries) can then be attributed on a case-by-case basis to use the pooling support, without impacting any other unattributed methods.

and with minimal surface area changes or feature work in the compiler.

Note that we need the emitted code to allow a different type being returned from `Create` method:

```
AsyncPooledBuilder _builder = AsyncPooledBuilderWithSize4.Create();
```

Note that this mechanism to change the the builder type cannot be used when the synthesized entry-point for top-level statements is `async`. An explicit entry-point should be used instead.

## Drawbacks

- The syntax for applying such an attribute to a method is verbose. The impact of this is lessened if a developer can apply it to multiple methods en masse, e.g. at the type or module level.

## Alternatives

- Implement a different task-like type and expose that difference to consumers. `ValueTask` was made extensible via the `IValueTaskSource` interface to avoid that need, however.
- Address just the `ValueTask` pooling part of the issue by enabling the experiment as the on-by-default-and-only implementation. That doesn't address other aspects, such as configuring the pooling, or enabling someone else to provide their own builder.

- Earlier versions of this document allowed for scoped override of builder types.

## Unresolved questions

1. **Replace or also create.** All of the examples in this proposal are about replacing a buildable task-like's builder. Should the feature be scoped to just that? Or should you be able to use this attribute on a method with a return type that doesn't already have a builder (e.g. some common interface)? That could impact overload resolution.
2. **Private Builders.** Should the compiler support non-public async method builders? This is not spec'd today, but experimentally we only support public ones. That makes some sense when the attribute is applied to a type to control what builder is used with that type, since anyone writing an async method with that type as the return type would need access to the builder. However, with this new feature, when that attribute is applied to a method, it only impacts the implementation of that method, and thus could reasonably reference a non-public builder. Likely we will want to support library authors who have non-public ones they want to use.